UNIVERSIDADE FEDERAL FLUMINENSE

**EDUARDO CHARLES VASCONCELLOS**

# ACELERAÇÃO DE MODELOS DA ELETROFISIOLOGIA CARDÍACA COM GPUs

NITERÓI

2019

UNIVERSIDADE FEDERAL FLUMINENSE

**EDUARDO CHARLES VASCONCELLOS**

# ACELERAÇÃO DE MODELOS DA ELETROFISIOLOGIA CARDÍACA COM GPUs

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito para a obtenção do Grau de Doutor em Computação. Área de concentração: COMPUTAÇÃO VISUAL

Orientador:
ESTEBAN WALTER GONZALEZ CLUA

Co-orientador:
MARCELO PANARO DE MORAES ZAMITH

NITERÓI

2019

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

V331a Vasconcellos, Eduardo Charles
Aceleração de modelos da eletrofisiologia cardíaca com GPUs / Eduardo Charles Vasconcellos ; Esteban Walter Gonzalez Clua, orientador ; Marcelo Panaro de Moraes Zamith, coorientador. Niterói, 2019.
113 f. : il.

Tese (doutorado)-Universidade Federal Fluminense, Niterói, 2019.

DOI: http://dx.doi.org/10.22409/PGC.2019.d.05330117798

1. Computação paralela e distribuída. 2. Método de diferenças finitas. 3. Eletrofisiologia do coração. 4. Unidade de processamento gráfico. 5. Produção intelectual. I. Clua, Esteban Walter Gonzalez, orientador. II. Zamith, Marcelo Panaro de Moraes, coorientador. III. Universidade Federal Fluminense. Instituto de Computação. IV. Título.

CDD -

Bibliotecária responsável: Fabiana Menezes Santos da Silva - CRB7/5274

EDUARDO CHARLES VASCONCELLOS

# ACELERAÇÃO DE MODELOS DA ELETROFISIOLOGIA CARDÍACA COM GPUs

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito para a obtenção do Grau de Doutor em Computação. Área de concentração: COMPUTAÇÃO VISUAL

Aprovada em 28 de junho de 2019.

BANCA EXAMINADORA

Prof. Esteban W. G. Clua - Orientador, UFF

Prof. Marcelo P. M. Zamith - Coorientador, UFRRJ

Dr. Flavio Fenton - GATECH

Dr. Reinaldo R. Rosa, INPE

Prof. Lúcia M. A. Drummond, UFF

Prof. Anselmo Antunes Montenegro, UFF

Prof. Rodrigo Weber dos Santos, UFJF

Dr. Abouzar Kaboudian, GATECH

Niterói

2019

*Dedicatória(s): Dedico esta tese a minha amada esposa Fernanda e a memória de minha querida avó, Luiza.*

# Agradecimentos

Gostaria de agradecer a minha esposa, por toda dedicação e suporte, sem os quais eu não teria conseguido terminar esta tese. Aos meus orientadores, pelo apoio, oportunidades e amizade. A CAPES e a NVIDIA pelo apoio financeiro à pesquisa desenvolvida.

# Resumo

Segundo a organização mundial da saúde, problemas cardíacos são uma das principais causas de morte no mundo. Problemas como arritmias são causadas por comportamentos anômalos na dinâmica elétrica do tecido cardíaco e podem ser estudados por meio de simulações computacionais, usando modelos matemáticos que descrevem a eletrofisiologia do tecido. Nesta tese foram desenvolvidas duas estratégias paralelas usando GPUs e CUDA para acelerar tais simulações. Dois modelos da eletrofisiologia cardíaca foram implementados para criar os simuladores e testar as estratégias paralelas propostas aqui: um modelo de duas variáveis e outro de 41 variáveis. Uma das estratégias propostas, chamada towerDS, foi baseada na abordagem de janela deslizante, desenvolvida para paralelizar o método de diferenças centradas em GPU usando o CUDA. Nossa estratégia propõe uma estrutura de armazenamento de dados projetada para otimizar os acessos a memória global da GPU quando executando a janela deslizante. A segunda estratégia propõe uma estrutura de dados que pode ser ajustada à geometria da parte ventricular do coração, otimizando o uso de recursos da GPU e o acesso a memória global da mesma pelas threads. Também foi desenvolvida uma estratégia de comunicação para execução de simulações com a janela deslizante e sua estrutura otimizada em ambientes com múltiplas GPUs. Os resultados atingidos demonstram ser possível simular 1 segundo de atividade elétrica com apenas 4 segundos de simulação. Este resultado indica que em breve, com arquiteturas de GPUs um pouco melhores, poderemos atingir o tempo real. Esta simulação em 4 segundos foi com o modelo de duas variáveis e oito GPUs, para um domínio discreto cúbico com $256^3$ células. Com uma única GPU, a estratégia paralela com a estrutura towerDS aplicada ao modelo de duas variáveis, foi capaz de reduzir o tempo de processamento da simulação entre 6% e 40%, quando comparada a implementação com a janela deslizante clássica. A variação no ganho de tempo depende do tamanho do domínio e da GPU utilizada. Já para o modelo de 41 variáveis, a implementação paralela com a estrutura adaptativa chega a ser até 10× mais rápida que a implementação com a janela deslizante.

**Palavras-chave**: computação em GPUs, simulações numéricas paralelas, modelos da eletrofisiologia cardíaca, método de diferenças finitas em GPU, otimização do uso de memória em GPU.

# Abstract

According to the World Health Organization, heart problems are one of the leading causes of death in the world. Issues such as arrhythmias are caused by abnormal behaviors in the electrical dynamics of cardiac tissue. They can be studied through computational simulations using mathematical models that describe tissue electrophysiology. In this thesis, we developed two parallel strategies using GPUs and CUDA to accelerate such simulations to achieve results close to real-time. Two models of cardiac electrophysiology were implemented to create the simulators and to test the parallel strategies proposed here: a two-variable model and a 41 variables model. One of the proposed strategies, called towerDS, was based on the sliding window approach, able to parallelize the method of GPU-centered differences using CUDA. This strategy proposes a data storage structure designed to optimize GPU's global memory accesses when running the sliding window. The second strategy proposes a data structure that can be adapted to the geometry of the ventricle of the heart, optimizing the use of GPU resources and access to the GPU's global memory by the threads. We also developed a communication strategy to execute simulations with the sliding window and its optimized structure in environments with multiple GPUs. The results achieved make possible to simulate 1 second of electrical activity in just 4 seconds. We believe that soon, with new GPU architectures, better results could be achieved. This 4 seconds simulation was achieved with the two-variable model and eight GPUs for a discrete cubic domain with $256^3$ cells. With a single GPU, our strategy based on a sliding window approach, applied to the two-variable model, was able to reduce the simulation processing time between 6% and 40%. The processing time reduction depends on the size of the domain and the GPU used. For the 41-variable model, the parallel implementation with the adaptive framework is up to $10\times$ faster than the sliding window implementation.

**Keywords**: GPU computing, Parallel numerical simulations, Cardiac electrophysiology models, Finite difference method in GPU, Memory use optimization in GPU.

# Lista de Figuras

# Lista de Tabelas

# Lista de Abreviaturas e Siglas

| | | |
|---|---|---|
| API | : | *Application Programming Interface* ou Interface de Programação de Aplicações; |
| CSR | : | *compressed sparse row*; |
| CUDA | : | *Compute Unified Device Architecture*; |
| EDP | : | Equação Diferencial Parcial; |
| EDO | : | Equação Diferencial Ordinária; |
| GPU | : | *Graphics Processing Unit* ou Unidade de Processamento Gráfico; |
| JDC | : | Janela Deslisante Clássica |
| LBM | : | Método de Lattice Boltzmann; |
| MDF | : | Método de Diferenças Finitas; |
| MPI | : | *Message Passing Interface*; |
| OMS | : | Organização Mundial da Saúde; |
| SM | : | *Stream multiprocessor* |
| SO | : | sistema operacional; |
| PCG | : | *preconditioned conjugate gradient*; |
| SIMD | : | *Single Instruction Multiple Data*; |
| SIMT | : | *Single Instruction Multiple Threads*; |

# Sumário

# Capítulo 1

# Introdução

Modelos matemáticos são amplamente utilizados em diversas áreas da ciência e da engenharia para possibilitar a simulação computacional de fenômenos físicos, químicos, econômicos e biológicos [21, 19, 14]. Previsão de tempo, propagação de ondas e migração de espécies são somente alguns fenômenos reais que podem ser simulados computacionalmente com auxílio dos modelos matemáticos. Muitos fenômenos são descritos por um conjunto de equações diferenciais cuja solução pode nos dizer o seu comportamento no tempo e/ou no espaço [21].

Um importante fenômeno bio-físico que pode ser estudado por meio de modelos matemáticos é a eletrofisiologia do tecido cardíaco. A dinâmica elétrica no coração pode ser modelada por um sistema de equações diferenciais que descrevem a variação de correntes elétricas pela membrana celular e do potencial elétrico ao longo do tecido.

Segundo dados da Organização Mundial da Saúde, cerca de $17,9$ milhões de pessoas morrem vítimas de doenças cardiovasculares por ano. Mais de 75% dessas mortes ocorrem em países subdesenvolvidos ou em desenvolvimento. Muitos problemas cardíacos são causados por variações no comportamento elétrico natural do coração. Tais variações podem causar diferentes tipos de arritmias (Figura 1.1), impedindo o funcionamento normal do mesmo e podendo causar a morte do indivíduo. Os modelos matemáticos da eletrofisiologia cardíaca permitem simular como o tecido cardíaco responde a diferentes estímulos, permitindo estudar comportamentos anômalos e possíveis tratamentos.

As simulações numéricas usando modelos realistas são de grande importância para o desenvolvimento de tratamentos direcionados ao paciente[47, 48, 32, 51]. Contudo, modelos matemáticos realistas podem usar dezenas ou até mesmo centenas de equações diferenciais para modelar uma célula do tecido cardíaco [16]. Portanto, simulações com

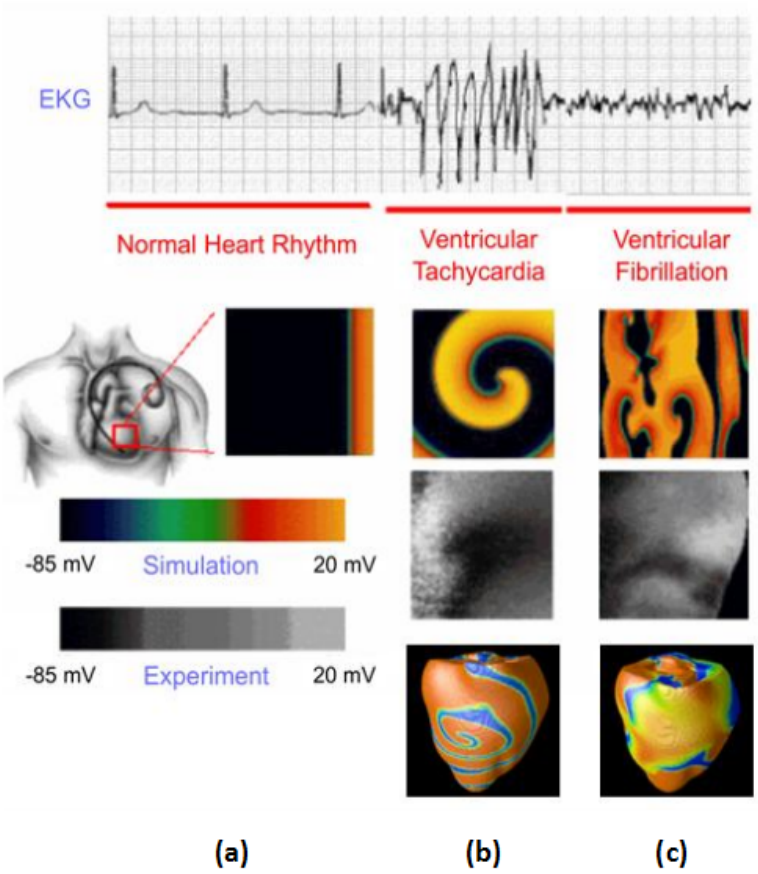



Figura 1.1: A figura mostra a transição do (a) comportamento normal do coração para a (b) taquicardia ventricular e por fim para o (c) estado de fibrilação. De cima para baixo são mostrados: um eletrocardiograma, uma simulação computacional 2D, o imageamento de um experimento `in-vitro` e uma simulação 3D.
**Fonte:** Adaptado de [2].

modelos realistas exigem grande capacidade computacional sendo de fundamental importância a paralelização das soluções numéricas [12, 34, 25]. Simulações próximas ao tempo real permitem simulações clínicas, ou seja, simular as condições de um paciente e a reação do mesmo a diversos tratamentos [47, 48, 2, 16].

Atualmente as GPUs (*Graphics Processing Unit* ou Unidade de Processamento Gráfico) oferecem uma alta capacidade de computação paralela a um custo relativamente baixo. Elas são arquiteturas baseadas em um único fluxo de instruções para múltiplos fluxos de threads - SIMT (*Single Instruction Multiple Threads*). Uma GPU moderna pode chegar a calcular a solução numérica de $2,0 \times 10^{10}$ à $3,0 \times 10^{10}$ equações por segundo. Muitos autores têm recorrido a GPUs para acelerar as simulações da dinâmica elétrica do coração [2, 39, 34, 50, 42, 25].

## 1.1 Objetivos

As GPUs oferecem uma grande capacidade de processamento paralelo, possuindo milhares de cores que podem executar concorrentemente um grande número de tarefas. Somado a essa capacidade, a API (*Application Programming Interface* ou Interface de Programação de Aplicações) CUDA (*Compute Unified Device Architecture*) possibilita a confecção de aplicações paralelas nas GPUs da Nvidia para programadores fora da área da computação gráfica. Contudo, o programador deve ficar atento ao ajuste da sua solução paralela ao modelo de execução da GPU. Acessos irregulares a memória e condicionais que causam divergência no fluxo de tarefas podem limitar severamente o desempenho de uma aplicação [13, 31, 44, 2, 35].

Visando a aceleração de modelos numéricos em GPU, com foco nos modelos da eletrofisiologia cardíaca, este trabalho tem como objetivo a implementação de alguns modelos de simulação numérica e propõe duas novas abordagens paralelas para o cálculo de soluções numéricas usando o método de diferenças finitas. As técnicas propostas têm por objetivo otimizar os acessos a memória durante o cálculo do passo de tempo numérico, reduzindo a latência causada por tais acessos. A pesquisa de teste foi desenvolvida cumprido-se os seguintes etapas:

- implementação de um modelo de duas variáveis da eletrofisiologia cardíaca, o modelo Karma [26];
- implementação de um modelo de 41 variáveis da eletrofisiologia cardíaca, o modelo OVVR [26];
- implementação do modelo Karma para ambientes com múltiplas GPUs;
- desenvolvimento de uma estrutura de memória e de um algoritmo de acesso para otimizar a estratégia de paralelização em GPU do método de diferenças centradas proposta por Micikevicius (2009)[31];
- desenvolvimento de uma estratégia de cópia de comunicação para a implementação multi-GPUs adequada a estrutura de dados;
- implementação e análise do comportamento da implementação multi-GPUs com diferentes tipo de alocação de memória e o uso de CUDA `streams`;
- desenvolvimento de uma estrutura de dados capaz de se ajustar ao domínio irregular definido pela geometria do coração;

- avaliação o desempenho das duas novas abordagens paralelas propostas e da abordagem clássica proposta por Micikevicius (2009)[31].

Até onde a pesquisa bibliográfica revelou, as estratégias de organização de memória apresentadas neste trabalho não aparecem na literatura. Trabalhos como o de Nimmagadda (2012)[34] propõe soluções com múltiplas GPUs, mas não tratam da organização de memória localmente em uma GPU nem da influência do uso da memória paginada (*page-lock memory*) na comunicação entre as GPUs.

### 1.1.1 Contribuições da Tese

O desenvolvimento desta pesquisa de tese gerou as seguintes contribuições pretendidas:

- um simulador tridimensional (3D) da propagação do potencial elétrico no tecido cardíaco com o modelo Karma em GPU;
- um simulador tridimensional (3D) da propagação do potencial elétrico no tecido cardíaco com o modelo OVVR em GPU;
- um simulador tridimensional (3D) da propagação do potencial elétrico no tecido cardíaco com o modelo Karma para ambientes com múltiplas GPUs;
- uma estrutura de dados otimizada para implementação de solução paralela do método de diferenças centradas, chamada towerDS;
- uma estrutura de dados otimizada para implementação de solução paralela do método de diferenças centradas e capaz de se adaptar a domínios irregulares que apresentem espaços vazios, chamada blockDS;
- uma estratégia de comunicação para utilizar a estrutura towerDS em ambientes com múltiplas GPUs;
- um estudo do impacto do tipo de alocação de memória e do uso de CUDA `streams` com a estratégia de comunicação aqui proposta ;
- premissas para trabalhos futuros que podem ajudar a acelerar ainda mais as simulações da eletrofisologia cardíaca usando GPUs.

Os desenvolvimentos feitos nessa tese e as implementações paralelas de modelos cardíacos estão sendo utilizados como base para o desenvolvimento, em andamento, de uma

ferramenta interativa para simulações 3D do coração humano. O desenvolvimento desta ferramenta está sendo conduzida em um projeto de iniciação científica e visa permitir a usuários interagirem com os simuladores, aqui implementados, de forma dinâmica.

### 1.1.2 Estrutura da tese

Esta tese está dividida em seis capítulos. No Capítulo 2 é fornecida uma pequena revisão da modelagem matemática da eletrofisiologia cardíaca e de ambos os modelos utilizados, bem como do método de diferenças finitas. No Capítulo 3 é feita uma revisão sobre CUDA e sobre a literatura, tanto referente a paralelização do método de diferenças centradas em GPU, quanto ao uso de GPUs nas simulações da dinâmica elétrica do tecido cardíaco. No Capítulo 4 descreve-se as diferentes estratégias paralelas abordadas durante a pesquisa de tese. O Capítulo 5 apresenta um conjunto de experimentos onde foram avaliadas as duas abordagens paralelas propostas com relação ao desempenho e a coerência nos resultados finais das simulações. O Capítulo 6 conclui a tese fazendo considerações sobre os resultados atingidos e deliberando sobre trabalhos futuros.

# Capítulo 2

# Modelagem Matemática da Eletrofisiologia Cardíaca

Um modelo matemático da eletrofisiologia do tecido cardíaco pode ser construído a partir da combinação de um conjunto de equações diferenciais ordinárias (EDOs) que representam a eletrofisiologia da célula cardíaca, juntamente com uma equação que descreva a propagação de um potencial elétrico [12]. A propagação do potencial elétrico no tecido é usualmente descrita por uma equação de reação-difusão da forma [2]:

$$\begin{aligned} \frac{\partial U}{\partial t} &= \nabla \cdot \mathbf{D} \nabla U - \frac{I_{ion}}{C_m} \, , \\ \nabla &= \left( \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right) . \end{aligned} \tag{2.1}$$

O primeiro termo do lado direito da Equação 2.1 representa o componente difusivo e descreve a variação no tempo e no espaço do potencial elétrico na membrana celular ($U$) ao longo do tecido cardíaco. O coeficiente $\mathbf{D}$ é um tensor que descreve como as células estão acopladas, podendo conter informações sobre a arquitetura do tecido, como, por exemplo, a orientação da fibra local. As informações contidas em $\mathbf{D}$ afetam a propagação das ondas elétricas e são importantes para determinar a velocidade da onda no tecido.

O segundo termo do lado direito representa o componente reativo e descreve o somatório das diversas correntes iônicas responsáveis por causar a variação no potencial da membrana celular [16]. Nessa parte $I_{ion}$ é a corrente total através da membrana celular e $C_m$, a capacitância constante da membrana celular. O termo reativo é composto por um sistema não linear de EDOs da forma:

$$\frac{d\mathbf{y}}{dt} = \mathbf{F}(\mathbf{y}, U(\mathbf{y}, t), t) \ , \tag{2.2}$$

sendo $\mathbf{F}(\mathbf{y}, U, t)$ uma função não-linear. Muitos modelos usam a aproximação de Hodgkin-Huxley [20] para a modelar as correntes iônicas nas células [12]. Nesta aproximação as correntes iônicas que fluem através da membrana celular são determinadas por uma ou mais equações do forma:

$$\frac{dy}{dt} = \frac{y_{\text{inf}}(U) - y}{\tau_y(U)} \ , \tag{2.3}$$

onde $y_{\text{inf}}$ é o valor estacionário da variável $y$ e $\tau_y$ é a constante de tempo de ativação/desativação da variável $y$. Ambas são dependentes do potencial da membrana $U$. Estas variáveis $y$ que definem o fluxo das correntes iônicas são chamadas variáveis de portão (do inglês *gate variables*).

O tamanho do sistema de equações diferenciais representado pela Equação 2.2 depende da complexidade do modelo, ou seja, do grau de detalhamento com o qual o modelo descreve as células cardíacas. O leitor pode encontrar uma discussão detalhada da modelagem numérica da eletrofisiologia do tecido cardíaco em [12, 16].

Em geral, a diferença entre os modelos de tecido cardíaco está no número de EDOs usadas para descrever a variação das concentrações iônicas dentro e fora das células (parte reativa na Equação 2.1), bem como no tensor $D$ que descreve algumas características do tecido [12, 39, 8]. A complexidade da computação da parte difusiva na Equação 2.1 é dada pela presença do operador divergente que impõe uma dependência geométrica na solução da equação diferencial. Neste trabalho utilizamos um modelos simples de duas variáveis [26] e um modelo complexo de 41 variáveis [36].

Segundo resultados apresentados por Nimmagadda em 2012 [34], a solução do termo difusivo da equação de propagação do pulso elétrico (Equação 2.1) apresenta a maior demanda em tempo de processamento ao solucionarmos numericamente a equação de reação-difusão. Uma vez que a complexidade na solução da parte difusiva é similar a todos os modelos, metodologias capazes de acelerar a solução da mesma em um modelo simples pode também ser utilizada para acelerar um modelo mais complexo.

## 2.1 Solução Numérica

Usar um modelo do tecido cardíaco para simular a dinâmica elétrica requer a aplicação de métodos numéricos para calcular uma solução aproximada para a Equação 2.1.

As EDOs usadas para modelar as correntes iônicas celulares podem ser resolvidas pelo métodos de diferenças finitas como o método de Euler, contudo um método muito utilizado é o proposto por Rush & Larsen em 1978 [40]. Nele a solução da Equação 2.3 é aproximada por:

$$y^{t+1} = y_{\infty}^{t}\left(U^{t}\right) + \left(y^{t} - y_{\infty}\left(U^{t}\right)\right) e^{-\delta t/\tau_{y}\left(U^{t}\right)} \tag{2.4}$$

O uso do método de Rush & Larsen permite que as EDOs do sistema possam ser resolvidas uma a uma, pois, para intervalos de tempo suficientemente pequenos, as equações do sistema podem ser desacopladas [49, 36].

No caso da EDP que modela a dinâmica do potencial no tecido, diversos métodos podem ser aplicados, tais como o método de elementos finitos[33, 39, 29] e o de volumes finitos[42]. Contudo, com as escolhas corretas dos passos espacial e de tempo, simulações com o método de diferenças finitas podem resultar na propagação de um estímulo com frentes de onda com curvaturas suaves [10, 11, 12]. Neste trabalho iremos focar no método de diferenças finitas aplicado a EDP 2.1.

### 2.1.1 O método de diferenças finitas

O método das diferenças finitas (MDF) é um método clássico e robusto para solução de equações diferenciais e é utilizado em diversas áreas da ciência e da engenharia [21]. Colocado de forma simplificada, o método substitui as derivadas da equação por diferenças entre valores das variáveis dependentes em diferentes pontos do espaço e do tempo. Tais diferenças podem ser obtidas através de aproximações em série de Taylor. Seja, por exemplo, uma função $f(x)$, expandindo $f(x+h)$ em série de Taylor teremos:

$$f(x+h) = f(x) + \frac{h}{1!}\frac{df(x)}{dx} + \frac{h^2}{2!}\frac{d^2f(x)}{dx^2} + \frac{h^3}{3!}\frac{d^3f(x)}{dx^3} + \frac{h^4}{4!}\frac{d^4f(x)}{dx^4} + \dots\ , \tag{2.5}$$

onde $h$ é uma constante. Manipulando a Equação 2.5, podemos escrever:

$$\begin{aligned}
\frac{h}{1!}\frac{df(x)}{dx} &= f(x+h) - f(x) - \frac{h^2}{2!}\frac{d^2f(x)}{dx^2} - \frac{h^3}{3!}\frac{d^3f(x)}{dx^3} - \frac{h^4}{4!}\frac{d^4f(x)}{dx^4} - \dots \\
\frac{df(x)}{dx} &= \frac{f(x+h) - f(x)}{h} - \frac{h}{2!}\frac{d^2f(x)}{dx^2} - \frac{h^2}{3!}\frac{d^3f(x)}{dx^3} - \frac{h^4}{4!}\frac{d^4f(x)}{dx^4} - \dots \\
\frac{df(x)}{dx} &= \frac{f(x+h) - f(x)}{h} + O(h) \ . \qquad (2.6)
\end{aligned}$$

Para valores de $h$ muito pequenos (próximos de zero), teremos que o termo $O(h)$ também fica muito pequeno, e a Equação 2.6 fica:

$$\lim_{h\to 0}\frac{df(x)}{dx} = \frac{f(x+h) - f(x)}{h} \ . \qquad (2.7)$$

O método de aproximação que faz uso da Equação 2.7 é conhecido como método de diferenças avançado ou como método de Euler. Quando aplicamos o método avançado a uma função dependente do tempo $g(t)$, diz-se que o método é explicito no tempo, pois para calcularmos a aproximação do instante de tempo $t+1$ precisamos somente conhecer o valor da função no instante $t$.

Existem equações diferenciais que possuem derivadas de ordem superior a 1, como as derivadas de ordem 2 na equação 2.1. Nestes casos, a aproximação dada pela Equação 2.7 não é aplicável. Contudo, podemos utilizar um raciocínio análogo ao usado no método avançado para achar uma aproximação para tais derivadas. Considere a expansão em série de Taylor exibida na Equação 2.8:

$$f(x-h) = f(x) - \frac{h}{1!}\frac{df(x)}{dx} + \frac{h^2}{2!}\frac{d^2f(x)}{dx^2} - \frac{h^3}{3!}\frac{d^3f(x)}{dx^3} + \frac{h^4}{4!}\frac{d^4f(x)}{dx^4} - \dots \ . \qquad (2.8)$$

Somando as Equações 2.5 e 2.8 teremos:

$$\begin{aligned}
f(x+h) + f(x-h) &= 2f(x) + h^2\frac{d^2f(x)}{dx^2} + 2\frac{h^4}{4!}\frac{d^4f(x)}{dx^4} + \dots \\
h^2\frac{d^2f(x)}{dx^2} &= f(x+h) + f(x-h) - 2f(x) - 2\frac{h^4}{4!}\frac{d^4f(x)}{dx^4} - \dots \\
\frac{d^2f(x)}{dx^2} &= \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - O(h^2) \qquad (2.9) \\
\lim_{h\to 0}\frac{d^2f(x)}{dx^2} &= \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \ . \qquad (2.10)
\end{aligned}$$

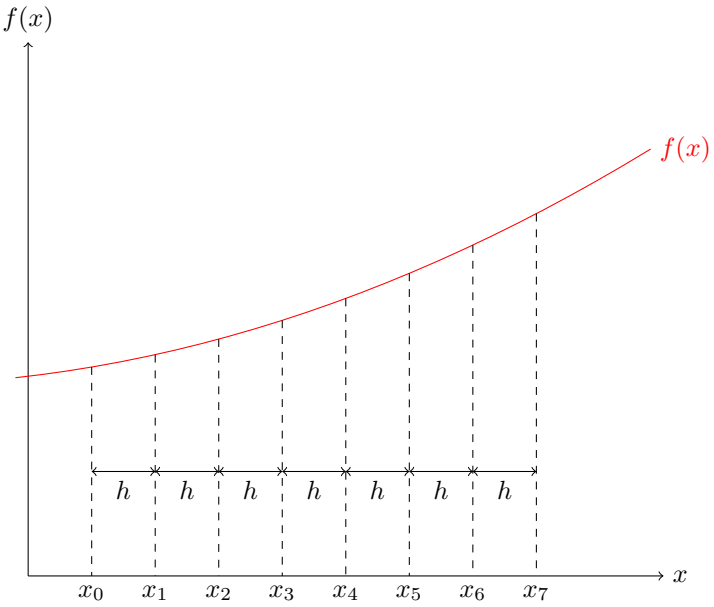



Figura 2.1: Exemplo da discretização do domínio de uma função contínua.

A aproximação é na Equação 2.10 é conhecida com diferenças centradas.

As aproximações de diferenças finitas apresentadas na forma das Equações 2.7 e 2.10 dependem de um domínio discreto com espaçamento constante $h$ (Figura 2.1). Neste caso existe um erro associado que é diretamente proporcional ao passo $h$, tendendo a zero quando $h$ assume valores muito pequenos e próximos de zero. Um método é dito de ordem $n$ quando o erro da aproximação, ou seja, o fator remanescente $O(h^n)$ é múltiplo da $n$-ésima potência de $h$. Assim sendo, o método avançado é de ordem 1 em $h$ enquanto que o método de diferenças centradas é de ordem 2 em $h$.

## 2.2 O modelo Karma

O modelo de Karma[26] é um modelo simplificado da eletrofisiologia cardíaca que reproduz algumas características básicas da sua dinâmica. O modelo é composto por duas variáveis: uma relacionada à tensão da membrana e outra variável modela a variação nas concentrações iônicas. Este modelo foi desenvolvido pela primeira vez para o estudo de ondas espirais, como as mostradas na Figura 1.1**b**, e descreve o comportamento da atividade elétrica no tecido cardíaco por meio das equações:

$$\frac{\partial U}{\partial t} = D\nabla^2 U - U + \left(\left[1 - \tanh(U-3)\right] U/2\right)\left[\gamma - \left(\frac{v}{v^*}\right)^{xm}\right] , \tag{2.11}$$

$$\frac{dv}{dt} = \epsilon\left[\Theta\left(U-1\right) - v\right] . \tag{2.12}$$

A variável $U$ representa o potencial elétrico na membrana celular, e a variável $v$ representa a diferença nas concentrações de íons de sódio e potássio dentro e fora da célula. $\theta(U-1)$ é uma função degrau e $\epsilon$ é a relação enter as escalas de tempo na qual o potencial de membrana se eleva e na qual ele permanece elevado até decair (veja Figura 2.2). O operador $\nabla^2$ é conhecido como operador de Laplace ou, simplesmente, Laplaciano e é definido como

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} . \tag{2.13}$$

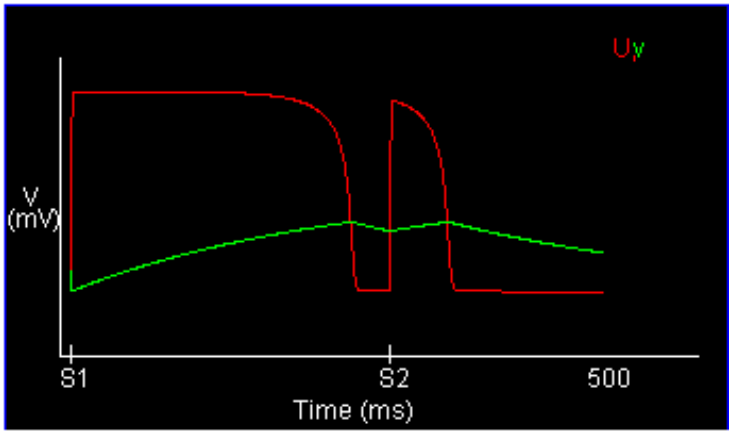



Figura 2.2: A figura mostra a evolução temporal das variáveis $U$ e $v$, que modelam, respectivamente o potencial de membrana e a variação nas concentrações iônicas.
**Fonte:** Adaptado de [16].

### 2.2.1 Solução numérica usando diferenças finitas

Através da solução das equações 2.11 e 2.12, buscamos conhecer o comportamento de $U$ e $v$ no espaço e no tempo. As variáveis $U$ e $v$ são chamadas variáveis dependentes, pois elas podem ser definidas em função das variáveis independentes $x$, $y$, $z$ e $t$:

$$U = U\left(x, y, z, t\right) \tag{2.14}$$

$$v = v\left(x, y, z, t\right) \tag{2.15}$$

Uma vez que iremos trabalhar com espaços discretos, por uma questão de simplicidade de notação, podemos reescrever as igualdades 2.14 e 2.15 como:

$$U\left(x_i, y_j, z_k, t_n\right) = U^n_{i,j,k} \tag{2.16}$$
$$v\left(x_i, y_j, z_k, t_n\right) = v^n_{i,j,k}\ . \tag{2.17}$$

Para calcular a solução numérica do termo difusivo do modelo Karma (Equação 2.11), aplicamos a aproximação por diferença avançada à derivada temporal e a diferença centrada às derivadas espaciais. Neste caso, usamos para a discretização no tempo $h = \Delta t$. Para discretização no espaço, usamos uma malha uniforme onde $h_x = h_y = h_z = \Delta s$.

$$\begin{aligned}
F(U,v) &= \left(\left[1-\tanh(U^n_{i,j,k}-3)\right]U/2\right)\left[\gamma-\left(\frac{v}{v^*}\right)^{xm}\right]\\
\frac{\partial U}{\partial t} &= D\left(\frac{\partial U}{\partial x}+\frac{\partial U}{\partial y}+\frac{\partial U}{\partial z}\right)-U+F(U,v)\\
\frac{U^{n+1}_{i,j,k}-U^n_{i,j,k}}{\Delta t} &= D\left(\frac{U^n_{i+1,j,k}-2U^n_{i,j,k}+U^n_{i-1,j,k}}{\Delta s}+\frac{U^n_{i,j+1,k}-2U^n_{i,j,k}+U^n_{i,j-1,k}}{\Delta s}+\right.\\
&\quad\left.+\frac{U^n_{i,j,k+1}-2U^n_{i,j,k}+U^n_{i,j,k-1}}{\Delta s}\right)-U^n_{i,j,k}+F(U^n_{i,j,k},v^n_{i,j,k})\\
U^{n+1}_{i,j,k} &= U^n_{i,j,k}+D\Delta t\left(\frac{U^n_{i+1,j,k}-2U^n_{i,j,k}+U^n_{i-1,j,k}}{\Delta s}+\right.\\
&\quad\left.+\frac{U^n_{i,j+1,k}-2U^n_{i,j,k}+U^n_{i,j-1,k}}{\Delta s}+\frac{U^n_{i,j,k+1}-2U^n_{i,j,k}+U^n_{i,j,k-1}}{\Delta s}\right)+\\
&\quad-\Delta t U^n_{i,j,k}+\Delta t F(U^n_{i,j,k},v^n_{i,j,k})\\
U^{n+1}_{i,j,k} &= (1-\Delta t)\,U^n_{i,j,k}+\frac{D\Delta t}{\Delta s}\left(U^n_{i+1,j,k}-2U^n_{i,j,k}+U^n_{i-1,j,k}+U^n_{i,j+1,k}+\right.\\
&\quad\left.-2U^n_{i,j,k}+U^n_{i,j-1,k}+U^n_{i,j,k+1}-2U^n_{i,j,k}+U^n_{i,j,k-1}\right)+\\
&\quad+\Delta t F(U^n_{i,j,k},v^n_{i,j,k})\\
U^{n+1}_{i,j,k} &= (1-\Delta t)\,U^n_{i,j,k}+\frac{D\Delta t}{\Delta s}\left(U^n_{i+1,j,k}+U^n_{i-1,j,k}+U^n_{i,j+1,k}+U^n_{i,j-1,k}+\right.\\
&\quad\left.+U^n_{i,j,k+1}+U^n_{i,j,k-1}-6U^n_{i,j,k}\right)+\Delta t F(U^n_{i,j,k},v^n_{i,j,k})\ .
\end{aligned} \tag{2.18}$$

Para simplificar a compreensão, podemos reescrever a Equação 2.18 como:

$$\begin{aligned} U_{i,j,k}^{n+1} &= (1-\Delta t)\, U_{i,j,k}^{n} + \Delta t \left( \frac{lap * D}{\Delta s} + F(U_{i,j,k}^{n}, v_{i,j,k}^{n}) \right) \\ lap &= U_{i+1,j,k}^{n} + U_{i-1,j,k}^{n} + U_{i,j+1,k}^{n} + U_{i,j-1,k}^{n} + U_{i,j,k+1}^{n} + U_{i,j,k-1}^{n} - 6U_{i,j,k}^{n} \end{aligned} \tag{2.19}$$

Para o termo reativo dado pela Equação 2.12, substituímos a derivada temporal $dv/dt$ pela aproximação de diferença avançada com $h = \Delta t$:

$$\begin{aligned} \frac{v_{i,j,k}^{n+1} - v_{i,j,k}^{n}}{\Delta t} &= \epsilon \left[ \Theta \left( U_{i,j,k}^{n} - 1 \right) - v_{i,j,k}^{n} \right] \\ v_{i,j,k}^{n+1} &= v_{i,j,k}^{n} + \epsilon \left[ \Theta \left( U_{i,j,k}^{n} - 1 \right) - v_{i,j,k}^{n} \right] \Delta t \ , \end{aligned} \tag{2.20}$$

As Equações 2.19 e 2.20 são as aproximações numéricas que nos permitem calcular as variáveis dependentes $U$ e $v$.

## 2.3 O modelo OVVR

O modelo OVVR foi proposto por O'Hara et al. em 2011 [36]. Os autores utilizaram dados do coração humano para desenvolver um modelo eletrofisiológico. Eles usam centenas de equações, entre variáveis e parâmetros, para modelar o comportamento das correntes iônicas com características do ventrículo do coração humano. A Figura 2.3 mostra o potencial elétrico na membrana celular simulado com o modelo OVVR. O comportamento do potencial elétrico e das correntes iônicas foram validados com dados observacionais. Os autores usaram simulações em um domínio espacial 1D para coletar os resultados.

### 2.3.1 Solução numérica usando diferenças finitas

Os autores propõe diferentes aproximações numéricas para as diferentes equações no modelo.

As variáveis de barreira são atualizadas no tempo por equações do tipo

$$y^{t} = y_{\infty} - \left( y_{\infty} - y^{t} \right) e^{-dt/\tau_{y}} \ . \tag{2.21}$$

A variável de barreira $n$ e as variáveis $J_{rel,NP}$ e $J_{rel,CaMK}$[1] são atualizadas pelas equa-

[1] As variáveis $J_{rel,NP}$ e $J_{rel,CaMK}$ estão relacionadas ao fluxo de liberação dos íons de cálcio via receptores Ryanodine [36].

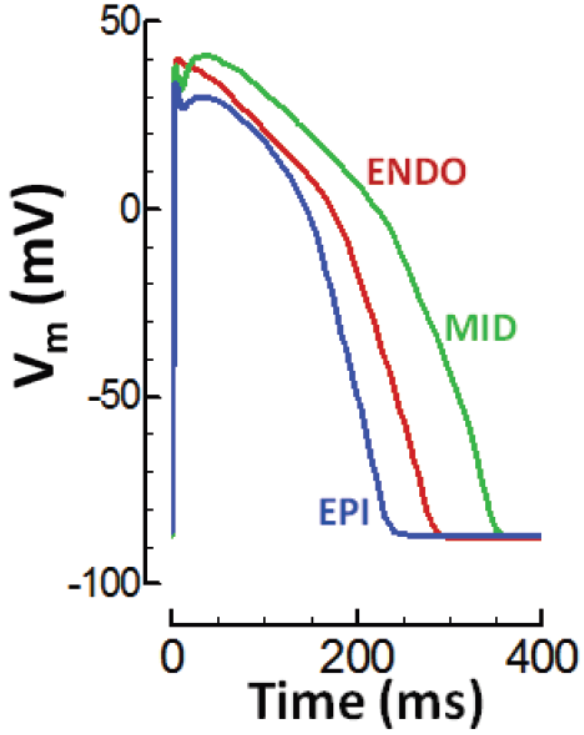



Figura 2.3: A figura mostra o potencial elétrico celular simulado com o modelo OVVR para três tipos de células diferentes.
**Fonte:** Adaptado de [36].

ções

$$n = \alpha_n \cdot \frac{k_{+2,n}}{k_{-2,n}} - \left(\alpha_n \cdot \frac{k_{+2,n}}{k_{-2,n}} - n\right) e^{(-k_{-2,n} \cdot dt)} , \tag{2.22}$$

$$J_{rel,NP} = J_{rel,NP,\infty} - (J_{rel,NP,\infty} - J_{rel,NP})\, e^{\left(-dt/\tau_{rel,NP}\right)} , \tag{2.23}$$

$$J_{rel,CaMK} = J_{rel,CaMK,\infty} - (J_{rel,CaMK,\infty} - J_{rel,CaMK})\, e^{\left(-dt/\tau_{rel,CaMK}\right)} . \tag{2.24}$$

O método de diferenças finitas avançado (método de Euler) é aplicado as concentrações iônicas e ao potencial elétrico na membrana celular. Para os autores a equação do potencial elétrico tem a forma

$$\begin{aligned} \frac{dU}{dt} = & -\frac{1}{C_m} \cdot (I_{Na} + I_{to} + I_{CaL} + I_{CaNa} + I_{CaK} + I_{Kr} + I_{Ks} + I_{K1} + \\ & + I_{NaCa} + I_{NaK} + I_{Nab} + I_{Cab} + I_{Kb} + I_{pCa} + I_{stim}) . \end{aligned} \tag{2.25}$$

Esta forma deve-se ao fato das simulações terem sido realizadas para um única célula. Como neste trabalho iremos considerar um volume de tecido, a equação precisa do termo

difusivo:

$$\begin{aligned}\frac{\partial U}{\partial t} &= D\nabla^2 U - \frac{1}{C_m}\cdot(I_{Na} + I_{to} + I_{CaL} + I_{CaNa} + I_{CaK} + I_{Kr} + I_{Ks} + I_{K1} + \\ &\quad + I_{NaCa} + I_{NaK} + I_{Nab} + I_{Cab} + I_{Kb} + I_{pCa} + I_{stim}) \ . \end{aligned} \tag{2.26}$$

A aproximação por diferenças centradas aplicada ao termo difusivo da Equação 2.26 é similar ao do modelo Karma (Equação 2.19), onde

$$\nabla^2 U^n_{i,j,k} \simeq \frac{U^n_{i+1,j,k} + U^n_{i-1,j,k} + U^n_{i,j+1,k} + U^n_{i,j-1,k} + U^n_{i,j,k+1} + U^n_{i,j,k-1} - 6U^n_{i,j,k}}{\Delta s} \ . \tag{2.27}$$

Neste trabalho iremos implementar com uma abordagem paralela e usando GPUs ambos os modelos citados. Implementações convencionais, tais como encontradas na literatura, estão longe de serem tempo real[2]. Após uma implementação convencional, iremos apresentar diversas estratégias de otimização, tanto para soluções baseadas em uma única GPU como para múltiplas GPUs. Embora os resultados ainda não possam ser considerados tempo real, iremos mostrar que estão próximos disto, podendo afirmar que em breve, com arquiteturas ligeiramente incrementadas e/ou pequenos incrementos neste trabalho, o tempo real será atingido. Tendo como objetivo permitir uma visualização de tal simulação, desenvolvemos um ambiente gráfico que interage CUDA com OpenGL.

[2]Por tempo real entende-se o tempo de atividade elétrica simulado. Uma simulação em tempo real significaria que uma simulação de 1 segundo de atividade elétrica é completada em apenas 1 segundo.

# Capítulo 3

# Paralelização de soluções numéricas para a eletrofisiologia cardíaca em GPU

A simulação computacional da atividade elétrica no tecido cardíaco pode ajudar médicos e pesquisadores a reproduzir as condições de um paciente e a resposta do tecido a estímulos, podendo ajudar no diagnóstico e tratamento de distúrbios como as arritmias cardíacas [43, 2]. Contudo, o tempo computacional de tais simulações torna proibitivo o uso clínico das mesmas. O modelo Karma, por exemplo, é um modelo simples, com apenas duas variáveis, e ainda assim sua execução serial em um processador Intel i7 com 4 *cores* físicos de $4,2GHz$ cada, gasta $39744,57ms$ (milissegundos) para simular $10ms$ da atividade elétrica do coração com um passo de tempo de $0,05ms$. Por esta razão, muitos autores vêm buscando meios de acelerar as simulações numéricas da atividade elétrica cardíaca para tempos próximos ao tempo real, sendo o uso de GPUs uma das tentativas mais promissoras.

Neste capítulo será fornecido o contexto necessário para compreensão dos esforços feitos no sentido da aceleração das simulações da eletrofisiologia cardíaca em GPUs. O capítulo é iniciado com uma breve descrição dos principais aspectos do modelo de programação paralela para GPUs usando o CUDA. Em seguida o leitor é apresentado a trabalhos publicados nos quais os autores propõe estratégias para paralelização do método de diferenças finitas em GPUs. Tais trabalhos não são voltados para o problema específico da eletrofisiologia cardíaca, mas para problemas similares de propagação nos quais o laplaciano cria uma dependência espacial entre as células do domínio discreto (vide Seção 2.1). Por fim, os trabalhos voltados para implementação em GPU de modelos da eletrofisiologia cardíaca são apresentados.

## 3.1 Programação de propósito geral em GPUs Nvidia

A arquitetura física de uma GPU consiste em um conjunto de processadores simétricos. Isso significa que todos os processadores GPU possuem características iguais. O número de processadores, núcleos e frequência em uma GPU depende da sua arquitetura [13].

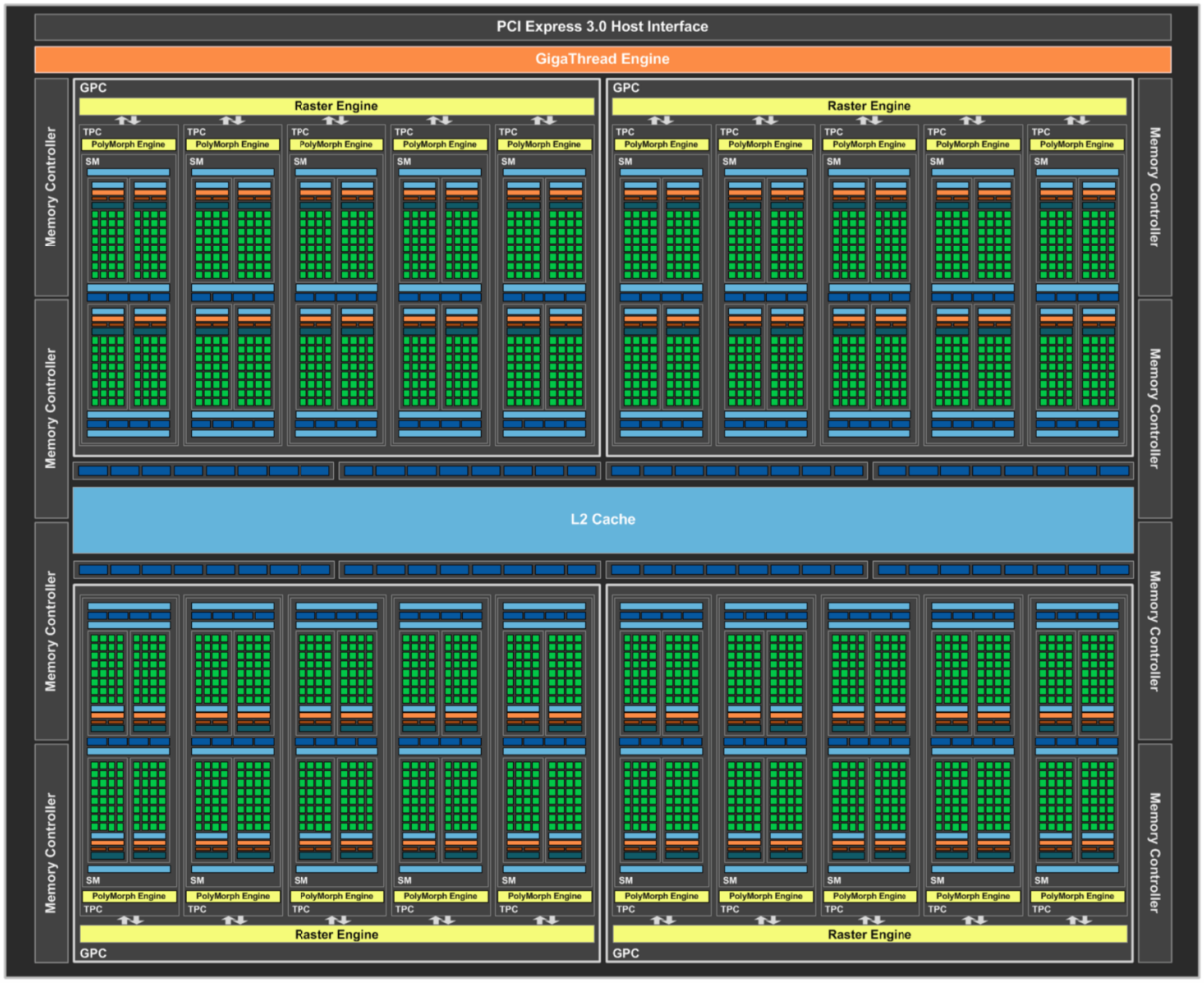



Figura 3.1: Arquitetura Pascal.
**Fonte:** `https://www.notebookcheck.net/Nvidia-Pascal-Architecture-Overview.165493.0.html`.

### 3.1.1 O modelo de programação paralela em CUDA

O CUDA é uma API (*Application Programming Interface*) lançada pela Nvidia para possibilitar programação de propósito geral em GPUs, colocando a disposição do programador toda capacidade de processamento massivamente paralelo de seus núcleos. O modelo de programação paralela do CUDA é baseado em uma grade de blocos de threads. Neste modelo, um conjunto de $n$ threads irá executar as mesmas tarefas ao mesmo tempo. Esse modelo paralelo ganha força quando precisa-se operar um mesmo conjunto de tarefas

sobre um grande volume de dados. A quantidade $n$ de threads executadas de forma concorrente depende, não só da arquitetura da GPU utilizada, como também da estratégia de paralelização usada pelo programador.

A computação feita com CUDA é chamada heterogênea, pois um programa CUDA usa recursos da CPU e da GPU. O programador pode escolher usar somente o poder computacional da GPU ou combinar em sua solução as capacidades de processamento da CPU e da GPU. Todo programa CUDA é composto por uma parte a ser executada na CPU e outra na GPU, sendo esta última organizada em blocos de código chamados `kernels`. A parte do programa CUDA executado pela CPU, usualmente chamada de `HOST`, é responsável por "informar" a GPU, usualmente referenciada com `DEVICE`, o `kernel` que deverá ser executado e qual configuração da grade de threads usada para tal. Quando existem dados na memória RAM do `HOST` que precisam ser processados no `DEVICE`, esses dados precisam ser copiados para a memória global do `DEVICE`. Nas arquiteturas atuais combinadas com as versões mais recentes do CUDA, existe um recurso chamado de memória compartilhada. Quando o programador aloca uma área de memória compartilhada, a própria API CUDA irá copiar os dados sobre demanda, eximindo o programador da responsabilidade de declarar a cópia dos dados.

Todo kernel submetido à GPU será executado por todas as threads da grade definida no momento do seu lançamento pelo código do `HOST`. As threads da grade são separadas em blocos com uma, duas ou três dimensões. Da mesma maneira a organização dos blocos na grade pode ser uni, bi ou tri-dimensional. A Figura 3.2 mostra dois kernels submetidos pelo `HOST`. Cada kernel será executado em sua própria grade, sendo o Kernel 1 executado em uma grade bi-dimensional (2D) de blocos tri-dimensionais (3D). A maleabilidade da grade permite modelar diferentes problemas na GPU. Se, por exemplo, deseja-se processar uma imagem usando CUDA, o programador podará utilizar uma grade 2D atribuindo a cada thread a tarefa de processar um único pixel da imagem.

Na etapa de execução cada bloco de threads é designado a um SM (*stream multiprocessor*) da GPU. As GPUs mais modernas podem executar oito ou mais blocos simultaneamente, desde que hajam recursos suficientes no SM. Dentro do SM as threads de um mesmo bloco são executadas em conjuntos de 32 threads, chamados de `warps`. As threads do bloco são indexadas de forma linear usando a ordenação linha majoritária, como exemplificado na Figura 3.3. Cada `warp` é composto por 32 threads com índices sequenciais. Assim sendo, em um bloco com $8 \times 8$ threads, teremos dois `warps`, o primeiro composto pelas threadas de 0 à 31 e o segundo pelas de 32 à 63.

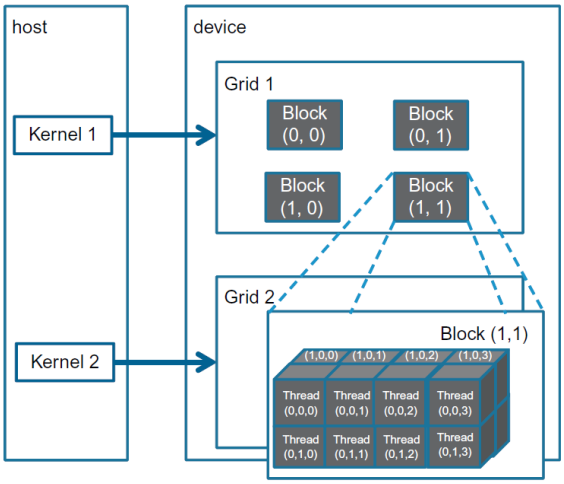



Figura 3.2: Submissão de kernels pelo `HOST` ao `DEVICE` para serem executados em grades 2D e 3D.

**Fonte:** Adaptado de [27].

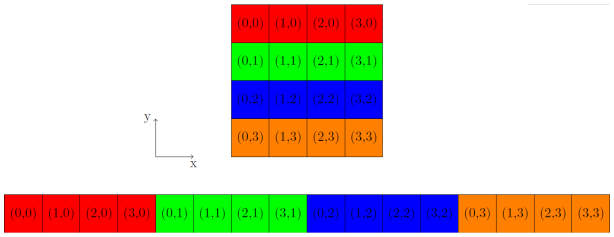


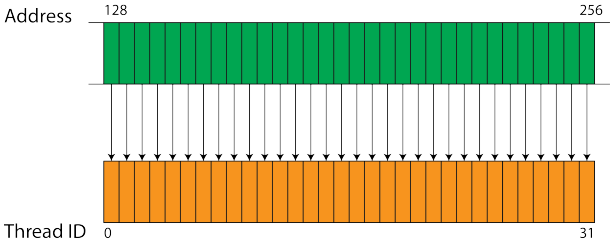

Figura 3.3: Ordenação linha majoritária aplicada a um pequeno domínio discreto 2D.

O `warp` é a unidade de processamento da GPU quando usamos o modelo de programação CUDA. Todas as threads de um mesmo `warp`executarão simultaneamente as mesmas instruções. Essa característica pode ser explorada para acelerar programas em CUDA, mas por outro lado, seu uso displicente pode causar graves prejuízos à paralelização de um código.

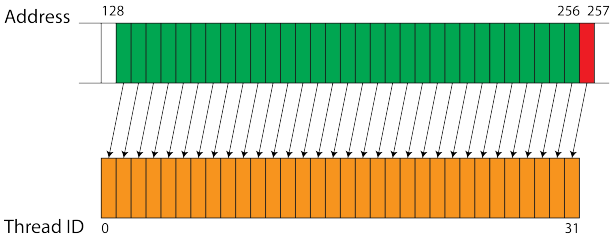
O grande destaque das GPUs jazem na sua capacidade de executar a mesma operação sobre uma grande quantidade de dados simultaneamente, funcionando como uma arquitetura do tipo SIMD (*Single Instruction Multiple Data*)[1]. O hardware da GPU foi desenvolvido para que um `warp` possa acessar dados adjacentes na memória global da GPU em uma única transação de memória. Esse tipo de transação é denominada de acesso coalescente e ocorre quando as 32 threads do `warp` operam sobre dados armazenados em uma palavra de 128-bytes, ou, em outras palavras, 32 valores de ponto flutuante consecutivos (Figura 3.4a). Qualquer outro padrão de acesso, como por exemplo os das Figuras 3.4b e 3.4c, causarão um aumento no número de transações com a memória global, que são

[1]Muitas vezes modelo de execução paralelo das GPUs é chamado de SIMT (*Single Instruction Multiple Threads*).

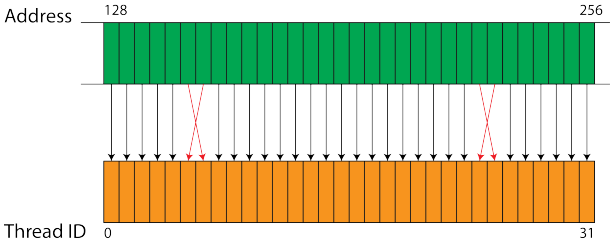
lentas e podem ser um gargalo na execução de um programa. Outro problema jaz no fato de que a cada transação são colocados no cache do SM 128 bytes de dados, portanto, transações que tragam dados desnecessários podem aumentar a chance de ocorrência de falhas cache (*cache miss*).




(a) Acesso alinhado e consecutivo (coalescente)




(b) Acesso não consecutivo (não coalescente)




(c) Acesso desalinhado (não coalescente)

Figura 3.4: Exemplos de acessos coalescente e não coalescentes.
**Fonte:** `https://cvw.cac.cornell.edu/gpu/coalesced?AspxAutoDetectCookieSupport=1`.

Além da questão da coalescência de dados, o modelo de execução em `warps` pode ainda sofrer de outro problema, a divergência de código. Como dito, todas as threads de um `warp` executarão a mesma instrução ao mesmo tempo. Portanto, quando um `warp` depara-se com uma instrução condicional, como um `if-else`, por exemplo, podemos ter diferentes threads do `warp` atendendo a diferentes condições. No trecho de código

```
if threadIdx.x < 29 then
     a[threadIdx.x] = 0
else
     a[threadIdx.x] = 2
```

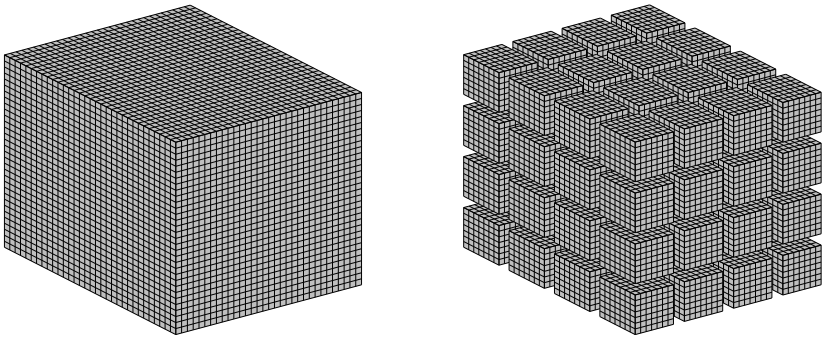
vemos que 29 das 32 threads de cada `warp` executarão `a[threadIdx.x] = 0`, e três executarão `a[threadIdx.x] = 2`. Nesse ponto a execução paralela das threads será interrompida e uma execução serial de cada condição será implementada pelo hardware. No exemplo acima teremos as 29 primeiras threads de cada `warp` executando sua tarefa. Durante este tempo, as demais 3 threads ficarão aguardando. Quando a tarefa condicionada das 29 primeiras threads for encerrada, as 3 que estavam aguardando começarão a trabalhar enquanto as outras 29 ficam paradas. Esse fenômeno é o que chama-se divergência de código.

### 3.1.2 A hierarquia de memória

As GPUs têm uma hierarquia de memória composta pela memória principal, dois níveis de cache, e as memórias compartilhada e de textura. Essas memórias apresentam diferentes capacidades de armazenamento, tempo de vida, escopo e latência. Podemos resumir suas características de acordo com o seu tipo: *i*) a **Memória Global** ou **DRAM**, é a memória mais lenta e cujo tempo de vida é definido pela CPU, que é responsável pelo seu gerenciamento. Essa memória reside *off chip* e todas as threads têm acesso a ela. A capacidade de armazenamento da memória global pode atingir muitos gigabytes, dependendo do modelo da GPU. *ii*) a **Cache L2** é compartilhada por todos os processadores da GPU, com latência e tamanho menores do que a memória global. Ela é também *off chip* e seu tempo de vida está ligado diretamente ao tempo de vida do kernel em execução na GPU. Depois de concluído o kernel, os dados da Cache L2 são perdidos. *iii*) A **memória compartilhada** tem um tempo de acesso próximo ao dos registradores. Ela é *on-chip* e está vinculada a execução de um bloco de threads, sendo seu tempo de vida igual ao do bloco. A memória compartilhada possibilita comunicação entre as threads do mesmo bloco. *iv*) A **Cache L1** é usada, nas arquiteturas mais recentes, apenas pelas memórias constantes e de textura. No primeiro caso, ela funciona com uma latência baixa e permite que todas as threads tenham acesso aos dados. Seu tempo de vida é o mesmo que o da Cache L2. A memória de textura tem o escopo e o tempo de vida semelhantes à memória global, mas com uma latência menor devido ao seu padrão somente de leitura. No entanto, o espaço alocado na memória de textura é restrito, sendo mais comumente utilizada no contexto gráfico. A Figura 3.1 ilustra a arquitetura das modernas GPUs Pascal da NVIDIA.

## 3.2 Métodos para paralelização de soluções de diferenças finitas em GPU

O método de diferenças centradas aparece na solução de problemas de propagação em diferentes áreas de estudo (biologia, sísmica, eletricidade,...). A solução numérica deste tipo de problema usando o MDF passa pelo processamento de um domínio discreto no espaço que pode ser facilmente ajustado ao modelo de paralelismo do CUDA. A Figura 3.5 mostra a representação de um domínio discreto 3D por uma grade de blocos de threads do CUDA. Podemos designar uma thread para calcular a solução numérica em cada célula do domínio discreto. Contudo, o método de diferenças centradas define uma dependência de dados entre células vizinhas e que pode ser representada pelo estêncil apresentado na Figura 3.6. Uma vez que acessos a memória global da GPU são lentos, a dependência mostrada na figura pode constituir um fator relevante de degradação da performance do uma implementação do MDF em CUDA. Nas últimas décadas vários autores apresentaram abordagens próprias para implementar de forma eficiente, em CUDA, cálculos dependentes de um estêncil como o da Figura 3.6.



(a) domínio discreto. (b) grade de blocos CUDA.

Figura 3.5: Discretização 3D para o MDF.

Micikevicius (2009) [31] define um estêncil espacial de ordem $k$ como a vizinhança de um ponto com $k$ vizinhos em cada direção. Podemos usar esse conceito de estêncil para ilustrar os elementos necessários ao processamento de cada célula de um domínio espacial discretizado com o MDF. Assim sendo, para a discretização de um MDF de ordem $k/2$ a computação de uma célula irá requerer um estêncil de ordem $k$, ou seja, $(d \cdot k + 1)$ elementos de entrada, onde $d$ é o número de dimensões espaciais do problema. O autor propõe o uso da memória compartilhada como forma de reduzir a redundância de acesso[2] aos dados na memória global. A Figura 3.7 mostra, para uma caso 2D, os elementos

[2]Micikevicius define o acesso redundante como a razão entre o número de elementos acessados e o número de elementos processados.

carregados na memória compartilhada para um bloco de $16 \times 16$ e um estêncil de ordem 8. No caso do exemplo da Figura 3.7, a redundância de acesso é 2. Em contrapartida, ao não usar a memória compartilhada, cada thread do bloco teria que carregar todo seu estêncil nos registradores, o que incorreria em uma redundância de 25.

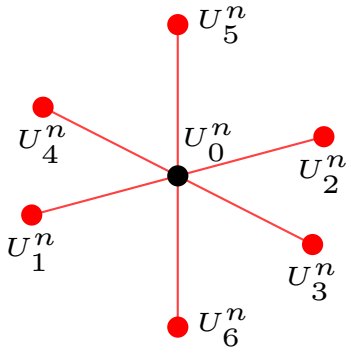



Figura 3.6: Estêncil representando os dados necessários para calcular $U_0^n$ ao usar diferenças centradas de $2^a$ ordem.

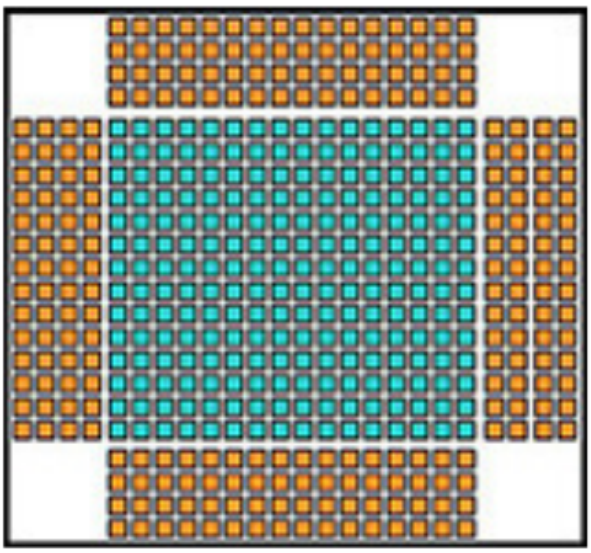


Figura 3.7: Elementos carregados na memória compartilhada para processar um bloco $16\times$ 16 com estêncil de ordem 2. Os elementos em azul claro serão efetivamente processados pelo bloco.

**Fonte:** [31]

Micikevicius (2009) também propõe que um domínio 3D seja processado por uma conjunto de blocos 2D que irão percorrer a direção $Z$. A memória compartilhada é usada para armazenar a vizinhança no plano $XY$, enquanto a vizinhança ao longo de $Z$ é armazenados nos registradores. A Figura 3.8 mostra as vizinhanças de um bloco $2 \times 2$. Os cubos brancos mostram elementos que são utilizados por mais de uma thread do bloco, enquanto os cubos coloridos indicam elementos acessados por somente uma

thread. A proposta do autor é que o bloco 2D percorra todas os planos $XY$ ao longo de $Z$ e para cada um, os valores na memória compartilhada e nos registradores sejam alterados de acordo. A Tabela 3.1 mostra o *throughput* em milhões de pontos por segundo obtidos pelo autor ao calcular uma equação do tipo:

$$D^{t+1}_{x,y,z} = c_0 D^t_{x,y,z} + \sum_{i=1}^{k/2} c_i \, ( \quad D^t_{x-i,y,z} + D^t_{x+i,y,z} + D^t_{x,y-i,z} + D^t_{x,y+i,z} + D^t_{x,y,z-i} + D^t_{x,y,z+i} \quad ) \ , \tag{3.1}$$

onde $D^t_{x,y,z}$ representa o valor de uma célula na posição $(x, y, z)$ calculada no passo de tempo $t$ e $c_i$ é o fator multiplicativo aplicado ao elemento a distância $i$ da célula processada. O autor também executou experimentos com a equação de onda com o FDM de $4^a$ ordem no espaço e $2^a$ no tempo. O autor relata que a GPU apresentou um desempenho de pelo menos uma ordem de grandeza superior ao um processador 4-core Haperton Xeon. Micikevicius também menciona o problema de domínios com tamanhos tais que não podem ser armazenados na memória de uma única GPU, propondo a divisão da computação do solução em múltiplas GPUs. Contudo, infelizmente o autor não comparou suas implementações com uso de memória compartilhada com outra na qual cada thread carregaria aos registradores seu próprio estêncil.

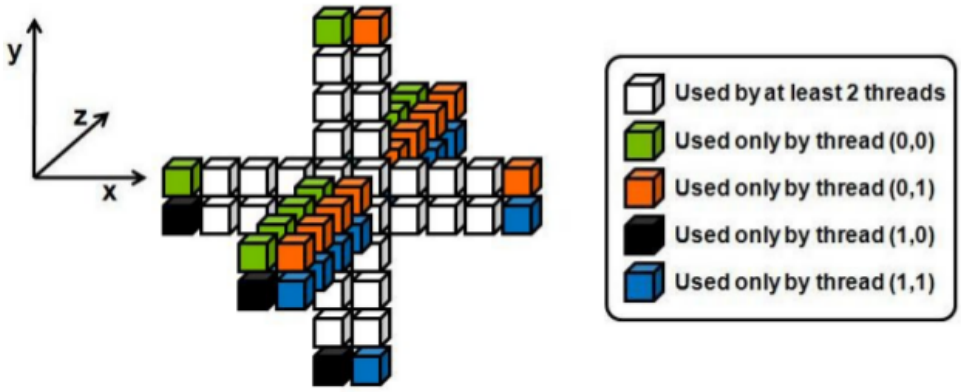



Figura 3.8: Reuso de elementos por diferentes threads de um bloco $2 \times 2$.
**Fonte:** [31]

Michea & Komatitsch (2010) [30] analisam aspectos da paralelização em GPU da equação da onda sísmica. Os autores relatam que suas implementações paralelas em GPU são de 20 a 60× mais rápidas que suas implementações seriais em CPU. Como Micikevicius (2009) [31], os autores endereçam a principal dificuldade da implementação em GPUs ao estêncil da Figura 3.6. Michea & Komatitsch chamam implementação natural em CUDA a representação do domínio discreto por blocos 3D como os da Figura 3.5b. Contudo, eles afirmam que, devido ao tamanho da memória compartilhada disponível nas GPUs (foram

| Dimensões | Ordem do estêncil | | | |
|---|---|---|---|---|
| | 6 | 8 | 10 | 12 |
| $480 \times 480 \times 400$ | 4.455 | 4.269 | 3.435 | 2.885 |
| $544 \times 544 \times 400$ | 4.389 | 4.214 | 3.347 | 2.816 |
| $640 \times 640 \times 400$ | 4.168 | 3.932 | 3.331 | 2.802 |
| $800 \times 800 \times 400$ | 3.807 | 3.717 | 3.236 | 2.752 |
| $800 \times 800 \times 800$ | 3.780 | 3.656 | 3.247 | 2.302 |

Tabela 3.1: Número de pontos de processados por segundo para estênceis 3D de diferentes ordens. O valores apresentados estão em $M$pontos$/s$.
**Fonte:** Adaptado de [31].

usadas GPUs GeForce 8800 GTX), essa estratégia não é tão eficiente quanto a de blocos 2D proposta por Micikevicius (2009)[31]. Infelizmente, os autores não apresentam um estudo comparativo para suportar esta afirmação, baseando-se apenas em cunho teórico. Sua base é que o uso da memória compartilhada reduz a redundância de acesso a memória global. Diferente de Micikevicius (2009) os autores discutem abertamente o uso das threads das bordas do bloco para acessar os halos de vizinhança (células laranjas na Figura 3.7). Os autores apontam que no acesso aos halos acarretaram em não coalescência de dados e divergência de código. Os autores demonstraram que o tempo de processamento na GPU escala linearmente com o aumento do tamanho do domínio em $Z$. Eles também observaram o desempenho da GPU, para um domínio de tamanho fixo, melhora com o aumento no número de blocos de threads utilizados (Figura 3.9). Segundo os autores, esse efeito deve-se ao fato de que, com mais blocos, o *scheduler*[3] pode esconder a latência de acesso a memória mesclando `warps` realizando processamento com `warps` requisitando dados da memória.

Abdelkhalek et al. (2012) [1] propõe uma implementação em GPU para a equação da onda acústica em três dimensões para uma aplicação real em modelagem sísmica. Os autores discutem desafios da implementação em GPU tais como o gerenciamento do uso de recursos. Segundo os autores, em uma GPU N80 a taxa de ocupação[4] atinge 66% com uma implementação direta, i.e., usando blocos 3D com cada thread processando uma célula do domínio e carregando seu estêncil associado. Contudo, eles consideraram o desempenho da implementação direta insatisfatório. Eles atribuem o desempenho ruim ao elevado número de acesso que, segundo eles, cada thread faz a memória global (25 acessos por thread). Ainda segundo os autores, para a mesma estratégia de blocos 3D, o uso da memória compartilhada para reduzir o número de acessos a memória global causa

[3] *scheduler* é o mecanismo de atribuição de tarefas da GPU.
[4] A taxa de ocupação é definida como a razão entre o número de `warps` ativas por multiprocessador e o valor máximo possível.

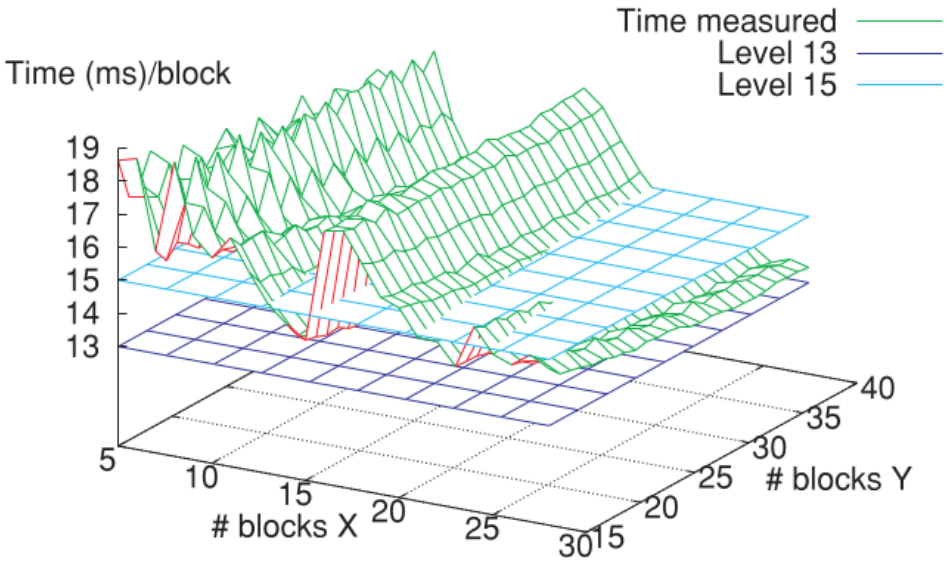



Figura 3.9: Tempo de processamento por bloco para um domínio de tamanho fixo e número (#) de blocos variável em $X$ e $Y$. As superfícies planas servem de bases para comparação.

**Fonte:** Adaptado de [30].

uma degradação na ocupação da GPU devido a limitação na quantidade de memória compartilhada por SM. Por fim, como Micikevicius [31], os autores buscaram reduzir o número da acessos a memória global aplicando a técnica de varrer a direção $Z$ com blocos 2D carregando as vizinhanças $XY$ na memória compartilhada para cada $z$ [31]. Os autores afirmam terem atingido uma ocupação de 100% usando memória compartilhada e blocos 2D com $16 \times 16$ threads. Infelizmente os autores somente relatam os resultados de desempenho para esta última estratégia. Usando GPUs Tesla T10 eles relatam ganhos de performance de 10x a 30x em relação a uma implementação em CPU.

Giles et al. (2014) [18] discute a implementação de soluções numéricas como o MDF para EDPs unidimensionais. Foram testadas implementações do MDF implícita e explícita no tempo. Os autores avaliaram duas estratégias de implementação do método explícito no tempo: (i) usando a memória compartilhada e cada thread processando uma única célula do domínio; (ii) usando o `warp` *shuffle*[5] do CUDA e cada thread processando oito células consecutivas do domínio. O caso de estudo utilizado pelos autores foi um domínio 1D com 256 células processado na GPU com um único bloco de 256 threads. Como este domínio é muito pequeno e pode ser armazenado facilmente na memória compartilhada, os autores são capazes de processar com um único kernel todos os passos de tempo desejados, sem precisar re-escrever dados na memória global. Na estratégia (i) os dados de vizinhança

[5]O `warp` *shuffle* é usado para comunicar variáveis entre threads do mesmo `warp` [35].

$x+1$ e $x-1$ atualizados a cada passo de tempo são compartilhados entre as threads com uso da memória compartilhada. Portando, barreiras de sincronização são necessárias. Na estratégia (ii) cada thread processa oito células consecutivas e as bordas destas linhas de células são comunicadas entre threads com o uso do `warp` *shuffle*. Nesse caso, o uso explícito de barreiras para todas as threads do bloco é desnecessário. Usando uma GPU Tesla K40 os autores relatam desempenhos de 700 GFlops para (i) e 3029 GFlops para (ii) usando precisão simples, e 610 GFlops para (i) e 1463 GFlops para (ii) usando precisão dupla. Os valores de pico da GPU utilizada relatado pelos autores são $5,04$ TFlops para precisão simples e $1,68$ TFlops para precisão dupla. Os autores também testaram três diferentes abordagens para soluções implícitas no tempo, obetendo perfromances em torno de 60% do valor de pico da GPU. Contudo, este trabalho não está voltado para métodos implícitos e a discussão desta parte foge ao escopo.

Porter-Sobieraj et al. (2015) [38] propõe o uso da memória de textura para acelerar o método proposto por Micikevicius (2009) [31], ao qual se refere como "janela deslisante" (*sliding window*). Os autores apontam duas possíveis causas de problemas na janela deslisante: (i) o uso das threads nas bordas de um bloco para carregar os halos de vizinhança podem causar aumento na complexidade do código e divergência de código quando o tamanho do domínio não for divisível pelo número de threads do bloco; (ii) uma simulação numérica pode envolver uma quantidade de variáveis que exceda o número de registradores disponíveis, sendo então necessário o uso da memória local que é consideravelmente mais lenta. A proposta do trabalho de Porter-Sobieraj et al. é armazenar o domínio discreto na memória de textura para aproveitar seu padrão de acesso que privilegia a localidade espacial. Usando uma GeForce GTX 660M os autores testaram o desempenho da solução numérica de equações hidrodinâmicas com diferenças finitas. As soluções numéricas utilizadas pelos autores envolvem estênceis de ordem entre 4, 6 e 8. Os autores declaram que nos testes realizados, blocos com $16 \times 32$ e $32 \times 32$ apresentam desempenhos consideravelmente inferiores a blocos menores devido a limitações no número de registradores. Os resultados dos autores mostram que, quando comparado ao método da janela deslizante, o uso da memória de textura melhora o desempenho de 2% a 23%. Os autores concluem declarando que a melhora no desempenho pelo uso da memória de textura está diretamente ligada a complexidade do algoritmo usado na solução numérica.

## 3.3 Em busca da simulação da eletrofisiologia cardíaca em tempo real

Alguns autores dedicaram esforços no desenvolvimento de implementações otimizadas para simulações da eletrofisiologia cardíaca em máquinas multicores e supercomputadores [23, 48, 33]. Niederer et al., por exemplo, publicaram em [33] os resultados de seus experimentos com simulações 3D da atividade elétrica cardíaca usando uma malha de elementos finitos com $26 \times 10^6$ nós e $153 \times 10^6$ elementos. Os resultados apresentados pelos autores mostram que o melhor resultado foi um tempo de simulação 240× mais lento do que o tempo real; para tal experimento foram utilizados 16.348 *cores*. O uso de supercomputadores pode auxiliar as pesquisas científicas na área da eletrofisiologia cardíaca, mas o acesso a um supercomputador constitui uma restrição ao uso clínico dos modelos de tecido cardíaco, pois este recurso não é largamente disponível.

Outros autores buscaram nas GPUs uma alternativa de menor custo ao uso de supercomputadores e *clusters*. O desenvolvimento atual da linguagem CUDA e das GPUs, com sua alta capacidade de processamento paralelo a um baixo custo, fizeram com que as GPUs venham ganhando destaque na busca pelas simulações em tempo real da atividade elétrica cardíaca [39, 2, 34, 50, 8]. Os resultados apresentados por Rocha et al. em [39] mostraram que uma simulação com mesmo modelo celular e domínio, demanda aproximadamente $78 \times 10^3$ segundos em uma CPU com 4 *cores* e aproximadamente $8 \times 10^3$ segundos numa GPU Gforce GTX 280 com 240 CUDA *cores*.

Bartocci et al. (2011) [2] descrevem uma análise de desempenho de implementações em CUDA para 5 modelos de tecido cardíaco. Os modelos variam em complexidade, tendo eles de 2 a 67 variáveis que descrevem os estados das células do tecido cardíaco (modelo Karma model, com duas variáveis [26]; o modelo Bueno-Orovio-Cherry-Fenton, com quatro variáveis [6]; o modelo Beeler-Reuter, com oito variáveis [3]; o modelo ten Tusscher-Panfilov, com dezenove variáveis [46]; o modelo Iyer-Mazhari-Winslow, com 67 variáveis [24]). Os autores tentaram diferentes abordagens para acelerar a computação da solução tanto do sistema de EDPs, quanto do sistema de EDOs. Todos os experimentos conduzidos por eles foram para um domínio 2D. Na solução das EDOs, eles utilizaram, sempre que possível, resultados tabelados para as funções não lineares envolvidas no cálculo da solução. Essas tabelas foram armazenadas na memória de textura. Além disso, todas as divisões que não envolviam variáveis foram substituídas por multiplicações equivalentes, e sempre que possível, as declarações `if` foram substituídas por expressões lógicas equivalentes. Na solução dos sistema de EDPs, o domínio foi dividido em pequenas

grades a serem processadas por blocos de threads da GPU, onde cada thread é responsável por calcular a solução numérica para um ponto do domínio. Contudo, a solução numérica por diferenças finitas empregada pelos autores demanda o conhecimento do estado da vizinhança a cada passo de tempo. Para acelerar o cálculo de um passo de tempo, os autores usaram a memória compartilhada da GPU e blocos com threads extras, cujo único propósito era carregar os dados da vizinhança do bloco na memória compartilhada. Os resultados publicados pelos autores demonstram que, com as otimizações nos modelos de 4, 8 e 19 variáveis, o tempo de simulação escala linearmente com o número de variáveis. As simulações com os modelos de 2 e 4 variáveis com uma grade 2D com $2^{18}$ pontos são executadas em tempos próximo ao tempo real. Todos os experimentos foram executados em uma GPU NVIDIA Tesla C2070.

Em [39], Rocha et al. (2011) discutem a implementação e avaliação de simulações 2D da atividade elétrica no tecido cardíaco em GPU. Os autores usaram dois modelos celulares diferentes, o Luo-Rudy e o ten Tusser-Panfilov. O desempenho das simulações foram avaliadas em uma GPU NVIDIA GTX 280. Na implementação em GPU, os autores lançam um kernel para resolver o sistema de EDOs e outro para o de EDPs. Cada um dos kernels é lançado com 256 threads por bloco, em cada um dos experimentos. As variáveis dos modelos são armazenadas em *arrays* 1D com $M * N_{eq}$ posições, onde $M$ é o número de células no domínio e $N_{eq}$ o número de equações, ou de variáveis de estado, do modelo. Segundo os autores, usando essa estratégia eles evitaram transações extra de memória entre a CPU e a GPU, além da redução nos acessos não coalescentes na memória global da GPU. Foram avaliadas duas diferentes estruturas para armazenamento dos dados: a *compressed sparse row* (CSR) e a ELLPACK [41]. A ELLPACK utiliza duas matrizes para armazenar os dados, sendo cada uma dessas matrizes armazenada como um *array* 1D. Para resolver o sistema de EDPs, os autores utilizaram o método *preconditioned conjugate gradient* (PCG). Nas simulações mais rápidas dos autores em GPU, usando o método de elementos finitos e discretizando o domínio em uma malha com 410.881 nós e 409.600 elementos, gastou-se: $648,78$ segundos, usando o modelo Luo-Rudy; $8394,53$ segundos, usando o modelo ten Tusscher-Panfilov.

Nimmaggada et al. descrevem em [34] simulações 3D em GPU da atividade elétrica no tecido cardíaco, usando o modelo celular ten Tusscher-Panfilov. Os autores testaram três diferentes abordagens na tentativa de obter uma melhor ocupação da GPU. As estratégias diferem no número de kernels lançados para calcular cada passo de tempo do método numérico. Na primeira estratégia, foram lançados quatro kernels, na segunda, cinco e na terceira, seis. Os autores apontam que a terceira estratégia obtém o melhor

desempenho. Eles avaliaram como o desempenho varia com o número de threads por bloco, concluindo que blocos com 68 threads fornecem um melhor desempenho. Foram usados *arrays* 1D para armazenar as variáveis e parâmetros do modelo na memória global da GPU. Com isso, os autores declaram reduzir o número de acessos não coalescente a memória global, permitindo que threads vizinhas acessem posições adjacentes da memória. Os autores também avaliaram o desempenho de uma implementação em um ambiente de múltiplas GPUs. Segundo os autores, ao usar múltiplas GPUs, a manipulação dos dados necessários para o processamento das células na fronteira dos subdomínios enviados a cada GPU, constituem um desafio. Eles declaram terem avaliado diversas técnicas de compartilhamento de dados, tendo escolhido copiar os dados da fronteira de volta para CPU após cada iteração do método numérico. Os resultados dos autores mostram que o cálculo do termo difusivo é o mais custoso, tomando quase 62% do tempo da simulação. Com a implementação dos autores, simular $350ms$ do tempo real gasta, para um domínio com $2^{24}$ células, 6.800 segundos em uma única GPU NVIDIA Tesla C1060. Com a implementação para múltiplas GPUs, a mesma simulação, usando quatro GPUs Tesla C1060, gasta 1.850 segundos.

Mena et al. (2015) [29] discutem a relevância do uso das GPUs nas simulações da dinâmica elétrica no coração. Os autores fazem um estudo comparativo entre uma implementação paralela em GPU e uma em CPU usando MPI (*Message Passing Interface*). Embora tenham utilizado o método de elementos finitos com os dados armazenados usando estruturas lineares sem nenhum tipo de ordenação especial, os autores relatam o uso de estruturas de matrizes esparsas. O desempenho da solução em GPU para o sistema de EDOs e para a EDP foram avaliados separadamente. A solução numérica da EDP foi calculada com a ajuda das bibliotecas CUSP e Thrust da NVIDIA. Neste caso, o *speedup* da implementação de método explícito no tempo em GPU com relação a implementação paralela em CPU, chega a ordem de grandeza de $10^2$ para malhas uni e bidimensionais, e $10^1$ para malhas tridimensionais. Esses resultados foram obtidos para o modelo de 19 variáveis proposto por [46] e grades com mais de $10^6$ pontos, independentemente do número de dimensões. Os autores utilizaram uma GPU Tesla E2090 e duas CPUS Xeon Quad-core E5620 $(2,4GHz)$.

Xia et al. discutem, em [50], estratégias paralelas e seus desempenhos para simular a atividade elétrica cardíaca em uma GPU. A estratégia dos autores é baseada em um modelo 3D do coração de uma ovelha [7]. Os autores buscaram otimizar os padrões de acesso à memória global da GPU, a fim de acelerar o processamento do seu código. Seu primeiro passo foi analisar o modelo, procurando por variáveis e parâmetros que não pre-

cisavam ser armazenados na memória. Eles descobriram que 40 variáveis precisam ser armazenadas na memória e outras 20 variáveis poderiam ser calculadas à cada passo de tempo. Seu segundo passo foi armazenar todas as variáveis em *arrays* 1D para reduzir os acessos de memória não-coalescentes. No entanto, devido à heterogeneidade das células, algumas delas poderiam representar locais vazios do domínio, não participando da computação. Designar threads para essas células poderia consistir em desperdício de recursos, pois essas threads ocupariam CUDA *cores* que não desempenhariam nenhuma computação. De acordo com os autores, esse problema foi resolvido indexando as threads de tal forma que nenhuma thread será designada para uma posição de célula vazia. Os autores também afirmam que a estrutura dos blocos de threads é diferente na solução dos termos reativo e difusivo da equação de propagação do pulso elétrico. Para resolver o termo reativo, foram usados blocos de threads 1D, enquanto que para resolver o termo difusivo foram usados blocos de threads 3D. De acordo com os autores, a estrutura 3D utilizada na solução de termo difusivo é mais conveniente porque a computação de uma célula está relacionada aos dados de 26 células adjacentes. Neste caso, uma tabela de mapeamento de uma dimensão para três dimensões precisou ser construída. Com todas essas estratégias, a implementação feita pelos autores em GPU leva 1.687 segundos para simular $0,6$ segundos do tempo real, considerando uma malha com aproximadamente $7 \times 10^6$ pontos. Neste caso, as simulações foram realizadas em NVIDIA K40 GPU, que pertence a arquitetura Kepler.

Campos et. al (2016) [8] apresentam uma implementação do Método de Lattice Boltzmann [9, 4] (LBM) aplicado à simulação da atividade elétrica cardíaca, em um domínio 3D discreto. Os autores implementaram o método para CPU e GPU. Em ambas as implementações, foi utilizada uma estrutura de dados esparsa [4] para tornar a utilização de memória mais eficiente ao lidar com geometrias irregulares. Cada conjunto de pontos homogêneos do domínio foi armazenado em uma matriz 1D diferente e, para cada um destes *arrays* 1D, uma estrutura de dados adicional (matriz de conectividade) foi criada para armazenar a lista de todas as posições de seus pontos vizinhos. A solução numérica do sistema de EDOs foi calculada usando o método de Rush-Larsen [40]. Com a implementação das otimizações, as simulações dos autores em GPU alcançaram velocidades de 230 e 480 vezes mais lentas que o tempo real. Os autores avaliaram simulações utilizando três modelos de células com diferentes níveis de complexidade: o modelo Luo-Rudy, o modelo ten Tusscher-Panfilov e o modelo Mahajan-Shiferaw [28]. Dependendo da complexidade do modelo celular e do número de pontos no domínio, o código GPU leva de 3 a 1.520 segundos para simular 1 segundo de atividade elétrica do coração. Para uma grade

de domínio regular com $224^3$ nós, e usando blocos com 256 threads e precisão simples, a simulação com o modelo Luo-Rudy levou 519 segundos, enquanto que com o modelo ten Tusscher-Panfilov levou 1.520 segundos. Para um domínio irregular (anisotrópico) com 3.760.560 nós, a simulação com o modelo Mahajan-Shiferaw levou 635 segundos. Todas as simulações foram executadas em uma GPU NVIDIA Tesla M2050.

Oliveira et al. (2018) [42] avaliou a performance da paralelização em GPU de um algoritmo adaptativo no espaço e no tempo aplicado à simulações da eletrofisiologia cardíaca. A derivada temporal na EDP foi aproximada com a discretização do MDF e as derivadas espaciais, com volumes finitos. Em todos os testes, o passo de tempo usado no cálculo da EDP foi de $0,05ms$. Os autores utilizaram os modelos da eletrofisiologia cardíaca [5] e [46] como casos de teste. O resultados mostraram que, em comparação com uma implementação OpenMP [37], somente a implementação paralela em GPU acelera a simulação em 33×. Com o uso do algoritmo adaptativo proposto por eles, os autores relatam uma aceleração de 498× no caso do modelo [46] e 165× no caso do [5].

Recentemente Kaboudian et al. (2019) [25] propuseram uma biblioteca desenvolvida em WebGL[6] para implementação de modelos da eletrofisiologia cardíaca. Usando como exemplo um modelo da eletrofisiologia do coração do porco, com três variáveis e um espaço bi-dimensional, os autores descrevem características da biblioteca e instruem como utilizá-la. Segundo os mesmos, para um domínio discreto com $512 \times 512$ células (pontos), sua implementação WebGL do modelo do coração do porco é capaz de processar 2.400 passos de tempo por segundo. Isso significa processar cerca de 629 milhões de pontos por segundo. Tal resultado é obtido com o uso do método de diferenças finitas explicito no tempo. Os autores também relatam ser possível, sob as mesmas condições anteriores, processar 38.000 passos de tempo por segundo (cerca de 9 milhões de pontos por segundo) ao usar uma GPU Titan X com arquitetura Pascal. Kaboudian et al. também explicam como adaptar sua solução para um domínio tri-dimensional e fornecem uma avaliação de seus resultados simulados. O autores exibem ondas expirais 3D geradas um sua implementação do modelo OVVR do coração humano, mas não fornecem informações de desempenho. Por fim, os autores mostram a aplicabilidade de sua biblioteca a outros problemas de propagação. A solução WebGL proposta pelos autores fornece uma solução não muito complexa de implementar um simulador da atividade elétrica do coração com recursos de visualização em tempo de execução. Contudo, a solução apresentada fica restrita a domínios com baixa resolução em número de pontos no domínio discreto.

[6]WebGL é uma API em JavaScript que oferece suporte para criação de gráficos 2D e 3D (`https://www.khronos.org/webgl/wiki/Main\_Page`).

As simulações com os modelos da eletrofisiologia cardíaca podem ajudar a compreender a dinâmica de diversos problemas do coração e testar possíveis tratamentos. Os trabalhos publicados demonstram que muitos autores buscam usar o poder de computação paralela das GPUs para executar simulações em tempos pequenos o suficiente para que tais simuladores tenham aplicabilidade no tratamento de pacientes reais. Contudo, o acesso a memória global é um dos maiores gargalos no processamento com GPUs e não encontramos até a presente data nenhuma publicações propondo otimizações neste contexto. Micikevicius (2009) [31] propôs uma solução paralela voltada para a implementação do método de diferenças centradas em GPU. Contudo, o método de Micikevicius sustenta um acesso ineficiente a memória ao acessar dados da vizinhança de um bloco. Nem o respectivo autor ou os demais autores que utilizaram esta estratégia, propuseram uma solução voltada para organização de dados na memória visando reduzir, ou mesmo eliminar, este problema. No Capítulo 4 deste documento, apresentaremos três soluções paralelas voltadas para a aceleração de simulações da eletrofisiologia cardíaca usando otimização de memória. Duas destas soluções usam como base o método de Micikevicius, propondo uma organização de dados na memória global de uma GPU para otimizar a cópia dos dados da vizinhança para a memória compartilhada antes do início da computação do passo de tempo numérico. A terceira solução é completamente nova e foi desenvolvida para otimizar o uso de memória e a carga de processamento na GPU, buscando assim, reduzir o tempo de uma simulação.

# Capítulo 4

# Soluções Paralelas Otimizadas para Simulações da Eletrofisiologia Cardíaca em GPUs

O objetivo principal deste trabalho foi propor e implementar soluções paralelas em GPU que pudessem ser empregadas para simular a eletrofisiologia do tecido cardíaco em tempos próximos ao tempo real. As soluções propostas aqui buscam otimizar o padrão de acesso a memória, acelerando o cálculo do estêncil ao utilizar a janela deslizante proposta por Micikevicius em 2009 [31](vide Capítulo 3).

Neste capítulo, iremos discutir a implementação da solução numérica por diferenças finitas de um modelo de tecido cardíaco em GPU, com foco na solução da EDP contendo o Laplaciano. A solução numérica com o MDF das EDOs é particularmente adequada para implementação em GPU, uma vez que o método é explícito no tempo. Propomos utilizar neste trabalho uma thread para processar cada célula do domínio espacial discreto, conforme ilustrado na Figura 3.5a, onde se pode observar a discretização do domínio espacial. Neste caso, em cada célula do domínio discreto teremos de resolver um sistema de EDOs. Usando o método avançado no tempo para discretização temporal, os valores das variáveis dependentes do tempo podem ser computados de maneira independente das demais células. Essa independência entre os cálculos executados em cada célula nos permite utilizar uma grade 3D de blocos de threads para representar o domínio discreto (Figura 3.5b) e graças ao modelo de paralelismo do CUDA, uma grande quantidade de células são atualizadas simultaneamente.

Mantendo o foco na EDP contendo o Laplaciano (vide Capítulo 2) e na otimização do método de janela deslizante, daremos ênfase as transações de memória, apontando algumas limitações da janela deslisante clássica e propondo alternativas para otimizar os

padrões de acesso a memória.

## 4.1 Estratégia de implementação

Antes de descrever os modelos paralelos é importante definir a estratégia de execução dos `kernels`, comum a todas as implementações em CUDA nesta pesquisa.

A inicialização das variáveis pode ser um gargalo na execução de uma aplicação CUDA. Em geral, as variáveis são inicializadas pela CPU e são depois copiadas da memória RAM da CPU para a memória global da GPU. Buscamos acelerar todas as nossas implementações, embutindo no cálculo do primeiro passo de tempo numérico a inicialização das variáveis $U$ e $v$ com suas respectivas condições iniciais. Portanto, cada implementação possui dois tipos de kernel, um que calcula o primeiro passo de tempo (kernel-A) e um que irá calcular todos os demais passos (kernel-B). A principal diferença entre eles reside na forma como os valores de $U$ são carregados na memória compartilhada e os de $v$ nos registradores. No caso do kernel-B, ambas as variáveis são lidas a partir da memória global. Já no caso do kernel-A, as variáveis são definidas por meio de uma função chamada pelo kernel, na qual estão definidas as condições iniciais como descritas nas Seções 5.3.1.1 e 5.4.1.1.

Outro problema de implementação foi a coerência na leitura e escrita na memória global. Como não há sincronia entre os blocos de threads, foram utilizados dois *buffers* de memória para armazenar a variável $U$. A cada passo de tempo, os *buffers* de entrada e saída são alternados, garantindo que no início do cálculo da solução numérica em $t + \Delta t$ todos os valores de $U$ correspondem a solução calculada em $t$. Esse cuidado somente é necessário para $U$, devido a dependência das vizinhanças.

## 4.2 Janela Deslizante Clássica

Proposta por Micikevivius em 2009 [31], essa técnica foi empregada por diversos autores (vide Seção 3.2). Nela, o domínio é processado por um conjunto de blocos 2D que percorrem a dimensão $Z$ processando as células de cada camada a seu tempo. A Figura 4.1 mostra a sequência de tarefas executadas por cada kernel implementando a janela deslizante para calcular o valor do potencial de membrana no passo de tempo $t + 1$ segundo uma equação do tipo da Equação 2.1. Ao processar a primeira camada ($z = 0$), três operações de memória são requeridas para cada thread: uma para armazenar nos

registradores a condição de contorno inferior $U^t_{x,y,z-1}$ ($U_{down}$); uma para carregar o valor $U^t_{x,y,z}$ da memória global para um *array* na memória compartilhada ($S_{i+1,j+1}$) e uma para carregar o valor $U^t_{x,y,z+1}$ da memória global para os registradores ($U_{up}$). Após a primeira iteração, $z$ é incrementado e apenas uma nova requisição a memória global é necessária ($U_{up} = U^t_{x,y,z+1}$). Os demais dados já estão nas memórias *in-chip*. Primeiro $U_{down}$ recebe o valor $S_{i+1,j+1}$ armazenado na memória compartilhada, então $S_{i+1,j+1}$ recebe o valor $U_{up}$ armazenado nos registradores e por fim $U_{up}$ é atualizado com o valor $U^t_{x,y,z+1}$ lido da memória global. Esse esquema reduz o número de acessos a memória ao longo da dimensão $Z$. A Figura 4.2 usa células coloridas para ressaltar as vizinhanças necessárias a um bloco 3D. Essas vizinhanças são necessárias ao cálculo de estêncil nas células das bordas do bloco. Cada um dos conjuntos de células destacados será usado por mais de um bloco resultando em acessos redundantes. O uso da janela deslizante elimina os acessos redundantes ao longo do eixo $Z$ (células em amarelo e vermelho).

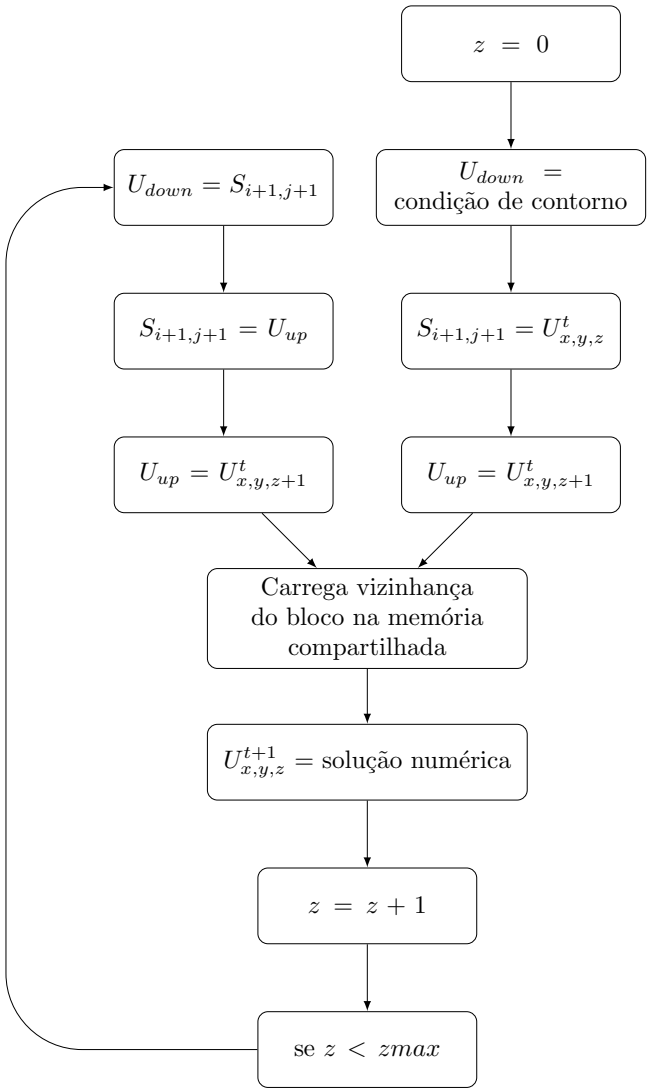



Figura 4.1: Tarefas executadas pelo kernel da abordagem de janela deslizante.

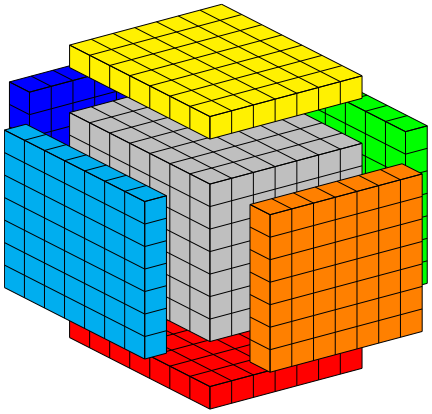


Figura 4.2: Dados necessários a um bloco de threads para calcular o Laplacianao usando o MDF.

O estêncil de diferenças centradas mostrado na Figura 3.6 implica em redundâncias de acesso não somente para células nas bordas do bloco, mas também células em seu interior. A janela deslisante busca reduzir o efeito da redundância no interior do bloco armazenando previamente os dados na memória compartilhada, cuja velocidade de acesso é mais de duas vezes superior ao da memória global. A Figura 4.3 ilustra o padrão de acesso para carregar os dados da memória global para a compartilhada ($S_{i+1,j+1} = U^t_{x,y,z}$). Nos exemplos daqui em diante consideraremos, por simplicidade, que um `warp` é representado por 8 threads e a palavra de 128-bytes (vide Seção 3.1) por oito posições consecutivas de memória. O mesmo padrão de acesso a memória global (parte inferior da figura) pode ser usado para ilustrar a carga dos dados nos registradores ($U_{up}$ e $U_{down}$). A ordenação em linha majoritária possibilita que dados requisitados por um `warp` sejam coalescidos em um único acesso. Os cubos brancos na Figura 4.3 representam a área de memória alocada para armazenar os dados de vizinhança necessários para a computação do passo de tempo pelas threads que tratam das bordas do bloco.

O acesso aos dados das vizinhanças pode ser dividido em duas partes: vizinhanças $Y$ e vizinhanças $X$. A Figura 4.4 ilustra o padrão de acesso usado para carregar as vizinhanças $Y$ na memória compartilhada. Podemos observar que, como no caso dos dados do núcleo (cubos cinza na figura), ao usarmos blocos com um número de threads em $X$ igual ao tamanho de um `warp` , os acessos a esses dados serão coalescentes. A Figura 4.5 ilustra o padrão de acesso adotado para carregar os dados das vizinhanças $X$ na memória compartilhada. Para ambas as vizinhanças utilizamos as threads das bordas do bloco para carregar as vizinhanças [30, 18]. Dois problemas chamam a atenção nesses padrões: ($i$) o posicionamento dos dados impõe acessos não coalescentes aos mesmos; ($ii$) utilizar as threads das bordas $X$ do bloco para acessar seus respectivos vizinhos acarreta

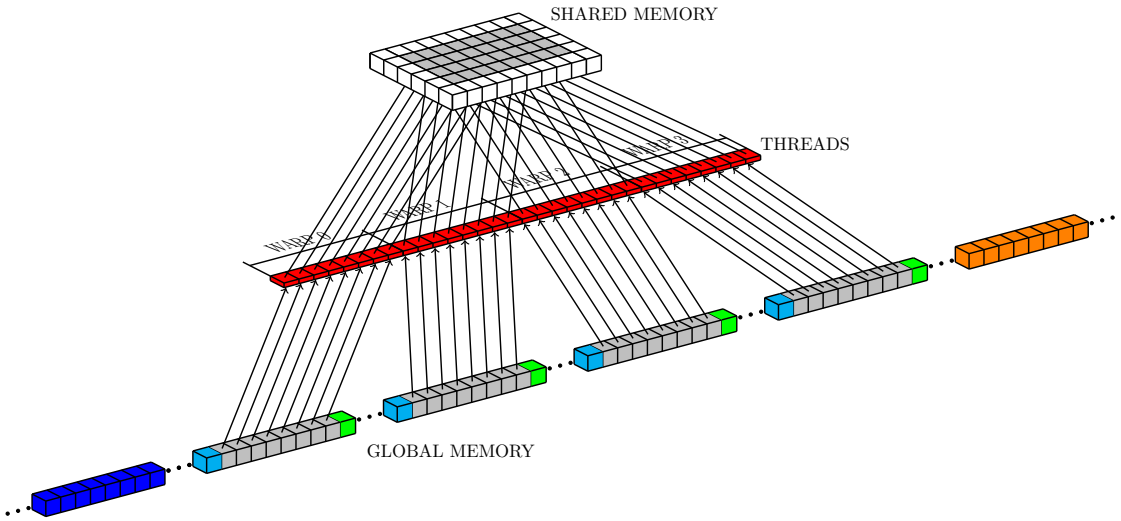



Figura 4.3: Carregamento de dados do núcleo do bloco, da memória global para a compartilhada.

divergência de código.

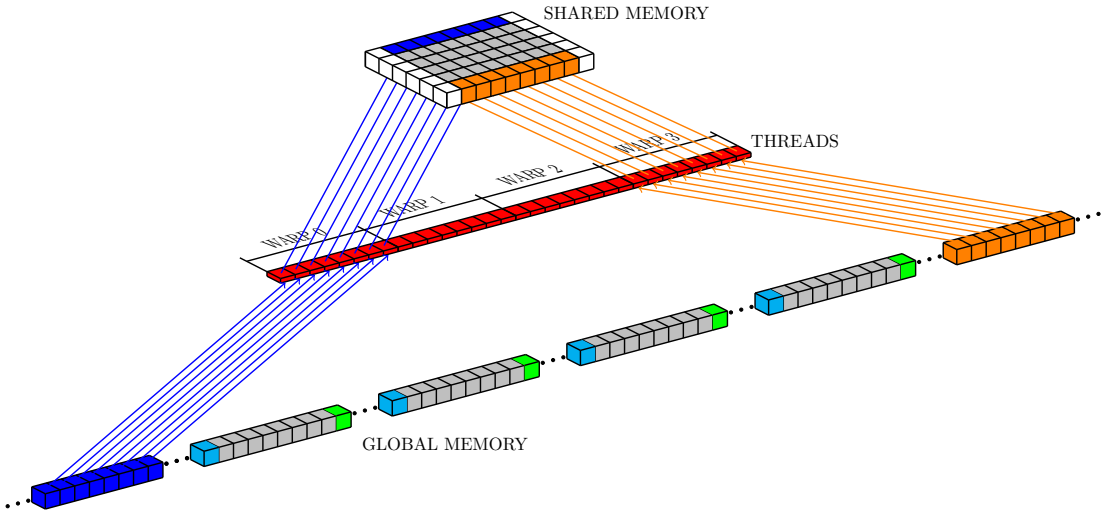



Figura 4.4: Carregamento de dados da vizinhança $Y$ do bloco, da memória global para a compartilhada.

## 4.3 Janela deslizante otimizada com a estrutura em torre

A janela deslisante clássica (JDC) discutida na seção anterior possui dois aspectos que comprometem o desempenho de uma simulação em GPU: *i*) o acúmulo de funções de leitura pelas threads nas bordas do bloco, que pode causar divergência de código no `warp` ao qual elas pertencem; *ii*) o acesso às vizinhanças do eixo $x$ (células azul claras

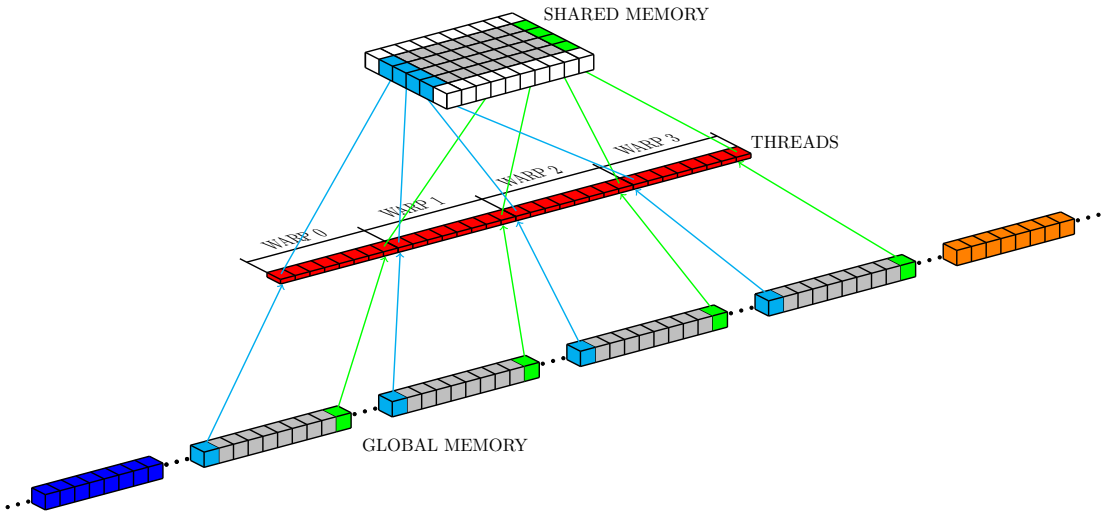


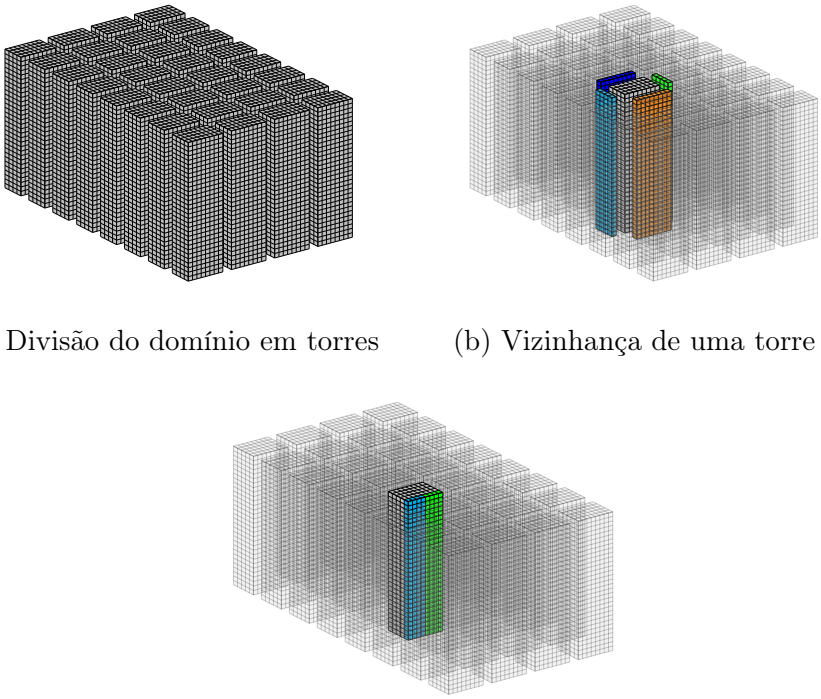

Figura 4.5: Carregamento de dados da vizinhança $X$ do bloco, da memória global para a compartilhada.

e verdes na Figura 4.5) pelas threads nas bordas do bloco, que não ocorre de forma coalescente.

### 4.3.1 A estrutura de dados

Para minimizar a divergência de código e os acessos não coalescentes à memória na JDC, propomos uma nova estrutura de dados, bem como uma estratégia de acesso aos dados para armazenar a variável dependente $U$ na memória global da GPU. A Figura 4.6 exibe uma representação da ideia geral por trás da nossa proposta. Na abordagem empregada na JDC, todo o domínio discreto é armazenado na forma de uma estrutura linear organizada segundo a ordenação linha majoritária (Figura 3.3). Nossa abordagem consiste em dividir o domínio discreto em subdomínios menores, de modo que seja possível armazenar os dados de forma independente para cada subdomínio. Além disso, incluímos um conjunto extra de células por subdomínio (*buffer* extra de memória). Estas células extras armazenaram os dados replicados das vizinhanças azul claro e verde (Figura 4.6c). O objetivo do *buffer* extra de memória é eliminar os acessos não coalescentes e a divergência de código quando um bloco carrega esses dados da memória global para a compartilhada.

A Figura 4.7 ilustra um exemplo 2D simplificado da estrutura de dados proposta. Esse exemplo mostra o efeito que a estrutura de dados tem sobre a organização linear dos dados armazenados na memória da GPU. Na Figura, as células cinzas representam os valores de $U$ que são acessados, atualizados e reescritos na memória global; as células azul claras e verdes representam as vizinhanças do bloco em $x$; as células azul escuras e



(a) Divisão do domínio em torres (b) Vizinhança de uma torre

(c) Estrutura de dados proposta

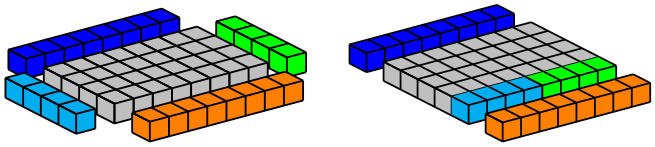
Figura 4.6: Representação da estrutura de dados 3D para um domínio discreto com $32 \times 32 \times 32$ células.

laranjas representam as vizinhanças em $y$. O uso da estrutura de dados proposta elimina os acessos desalinhados do tipo mostrado na Figura 3.4b, favorecendo acessos a posições de memória alinhadas como na Figura 3.4a.

### 4.3.2 A estratégia de acesso aos dados

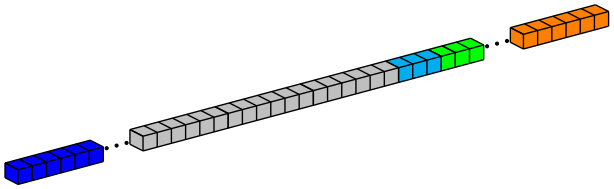
A estrutura de dados descrita na Seção 4.3.1, denominada de estrutura `towerDS` , descreve a forma como os dados são organizados na memória global de maneira a tornar mais eficiente o acesso à eles pelas threads de um mesmo WARP. Contudo, a estratégia completa de otimização depende de um algoritmo capaz de usar a `towerDS` para acelerar a computação de cada passo de tempo numérico. Podemos dividir essa estratégia de otimização do acesso aos dados em três etapas: $(i)$ mapear cada thread da grade nas posições da `towerDS` ; $(ii)$ ler os valores $U^n$ da memória global; e, $(iii)$ escrever os valores atualizados $U^{n+1}$ na memória global.

A Figura 4.8 mostra o fluxograma da janela deslizante modificado para a `towerDS` . Em vermelho vemos a única nova etapa a ser incluída no algoritmo de solução. Contudo, a carga dos dados de vizinhança na memória compartilhada também será modificada para tirar vantagem da organização dos dados definida pela `towerDS` .



(a) Distribuição geométrica de dados no domínio discreto.

(b) Distribuição geométrica de dados na estrutura de dados proposta.



(c) Distribuição de dados na memória global.

Figura 4.7: Uma representação simplificada da posição dos dados a serem acessados por um bloco com $8 \times 4$ threads.

O Algoritmo 1 descreve o processo desenvolvido para acessar os dados de vizinhança na memória global e copiá-los para a memória compartilhada. As variáveis `threadIdx` (posição da thread no bloco), `blockIdx` (posição do bloco na grade), `blockDim` (número de threads no bloco) e `gridDim` (número de blocos na grade) são intrínsecas do CUDA; seus valores são atribuídos pela API para cada thread quando o kernel é lançado para execução na GPU. A variável `coresize` recebe o número de células que serão realmente processadas em cada camada $Z$ de uma torre (células cinzas na Figura 4.6c). A variável `zlayersize` recebe o número total de posições existentes em cada camada $Z$ de uma torre, ou seja, o número de células reais mais o número de posições para hospedar a vizinhança $X$ da torre (*buffer* extra).

A estrutura de dados `towerDS` foi projetada para refletir as possíveis configurações multi-dimensionais da grade de threads do CUDA. Contudo, se mapeássemos uma thread para cada célula da estrutura iríamos designar threads para células cujo único objetivo é armazenar dados que não representam pontos reais do domínio. Designar threads para tais células seria atribuir a elas somente a tarefa de leitura e escrita dos dados. Essas threads ficariam inativas durante todo resto da execução do bloco, podendo gerar divergência de código e prejudicando a ocupação eficiente da GPU. Portanto, a grade de threads usada deve reproduzir a estrutura de células que representa o domínio real (Figura 4.6a) e não a estrutura de dados (Figure 4.6c). Como veremos a seguir, a reutilização de threads será empregada na leitura e escrita de dados nas células extras de cada torre (células azul claras e verdes).

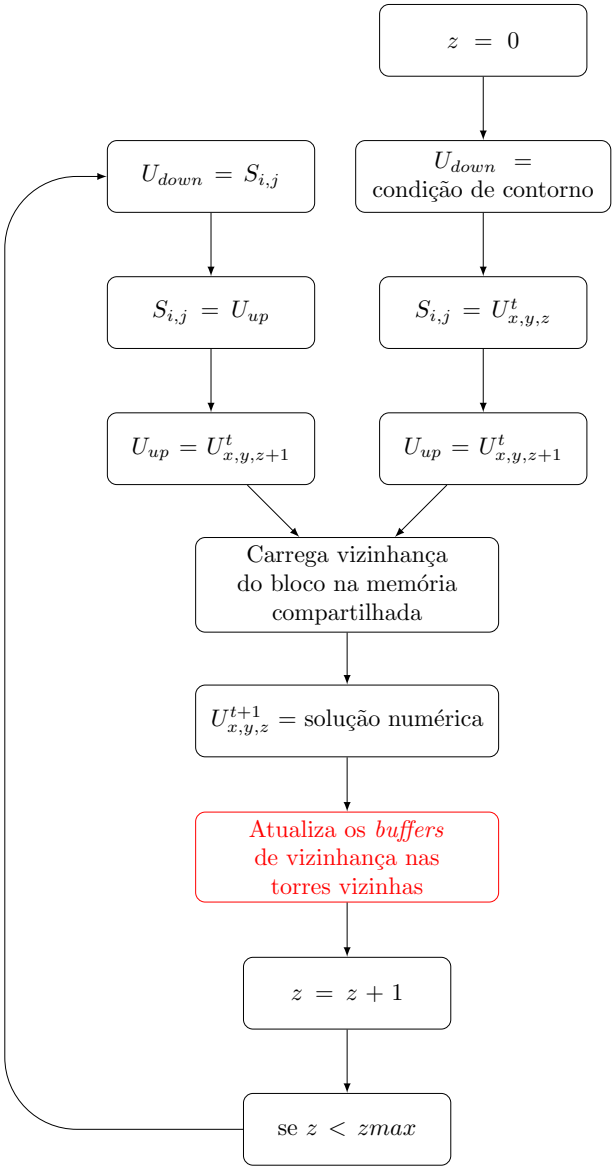



Figura 4.8: Tarefas executadas pelo kernel da abordagem de janela deslizante otimizada com a `towerDS` .

Nos Algoritmos 1 - 3 as variáveis da memória global contendo os campos de entrada (voltIN) e saída (voltOUT) para o potencial de membrana são distintos. O uso de variáveis distintas tem por objetivo manter a coerência durante a computação de múltiplos passos de tempo. Durante a execução de um `kernel` na GPU, não há qualquer tipo de comunicação entre blocos, portanto, para garantir que os dados acessados por um bloco não tenham sido modificados por outro, foram usadas variáveis distintas de entrada e saída no cálculo de cada passo de tempo. Esse cuidado é necessário devido a dependência das vizinhanças para o cálculo da solução numérica em cada célula do domínio.

No Algoritmo 1, as linhas 1 e 2 contém, respectivamente, o cálculo do índice da thread no bloco e do índice do bloco na grade, já as linhas de 3 à 6 contém a inicialização de

**Algoritmo 1** Processo de carga dos dados das vizinhanças $X$ e $Y$ na memória global usando a `towerDS` .

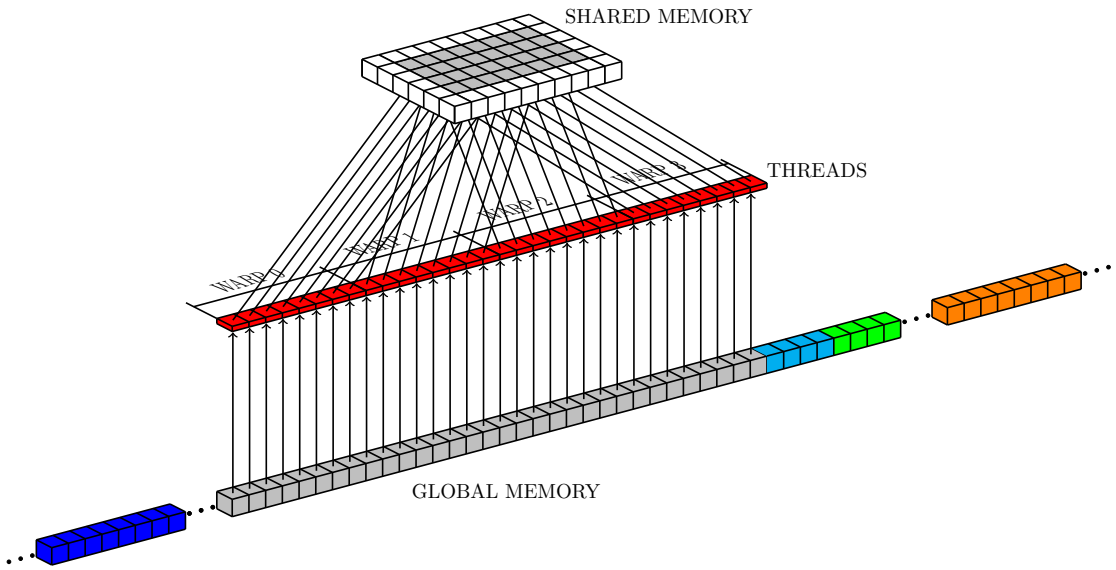
```
 1: thidx ⇐ threadIdx.y*blockDim.x + threadIdx.x                ▷ índice linear da thread
 2: bidx ⇐ blockIdx.y*gridDim.x + blockIdx.x                    ▷ índice linear do bloco
 3: msize_z ⇐ número de células do domíno discreto na direção Z
 4: coresize ⇐ blockDim.y*blockDim.x                            ▷ n^o de células por camada Z de uma torre
 5: zlayersize ⇐ blockDim.y*(blockDim.x + NHSIZE)               ▷ n^o de posições por camada Z de uma torre
 6: bnhsize ⇐ 2*( blockDim.x + blockDim.y )                     ▷ tamanho da vizinahnça XY do bloco
 7: l ⇐ thidx
 8: a0 ⇐ 2*blockDim.x
 9: a1 ⇐ 2*blockDim.x + blockDim.y
10: a3 ⇐ bidx*(zlayersize*msize_z) + coresize
11: a4 ⇐ (bidx - gridDim.x )*(zlayersize*msize_z) + coresize - blockDim.x + threadIdx.x
12: a5 ⇐ (bidx + gridDim.x )*(zlayersize*msize_z) + threadIdx.x
13: while l < bnhsize do
14:     if l < BLOCKDIM_X then
15:         if blockIdx.y > 0 then
16:             sidx ⇐ a4
17:             S[i][0] ⇐ voltIN[sidx]
18:         else
19:             Aplica-se a condição de contorno
20:     else if l < a0 then
21:         if blockIdx.y < (gridDim.y-1) then
22:             sidx ⇐ a5
23:             S[i][sy] ⇐ voltIN[sidx]
24:         else
25:             Aplica-se a condição de contorno
26:     else if l < a1 then
27:         sj ⇐ ( l - a0 )
28:         sidx ⇐ a3 + sj
29:         S[0][sj+1] ⇐ voltIN[sidx]
30:     else
31:         sj ⇐ ( l - a1 )
32:         sidx ⇐ a3 + blockDim.y + sj
33:         S[sx][sj+1] ⇐ voltIN[sidx]
34:     l ⇐ l + coresize
```

parâmetros dimensionais da estrutura `towerDS` . A variável `sidx` (linhas 16,22,28 e 32) guarda o índice da posição, na estrutura de dados, na qual está armazenado o valor do potencial de membrana que será operado pela `thread`. No algoritmo, o potencial de membrana está armazenado na variável `voltIN`. As linhas de 7 à 12 contém as equações de mapeamento usadas para definir as posições da `towerDS` que serão acessadas pela `thread`. A cópia dos dados de vizinhança, da memória global para a compartilhada, é executada nas linhas de 13 à 34. Cada bloco é inicializado com um número de `threads` igual ao valor armazenado na variável `coresize`. Como não são utilizadas `threads` exclusivas para leitura das vizinhanças, o processo iterativo, iniciado na linha 13, utiliza todas as threads disponíveis. Com isso, cada `warp` pode ser responsável por carregar mais de um conjunto de dados.

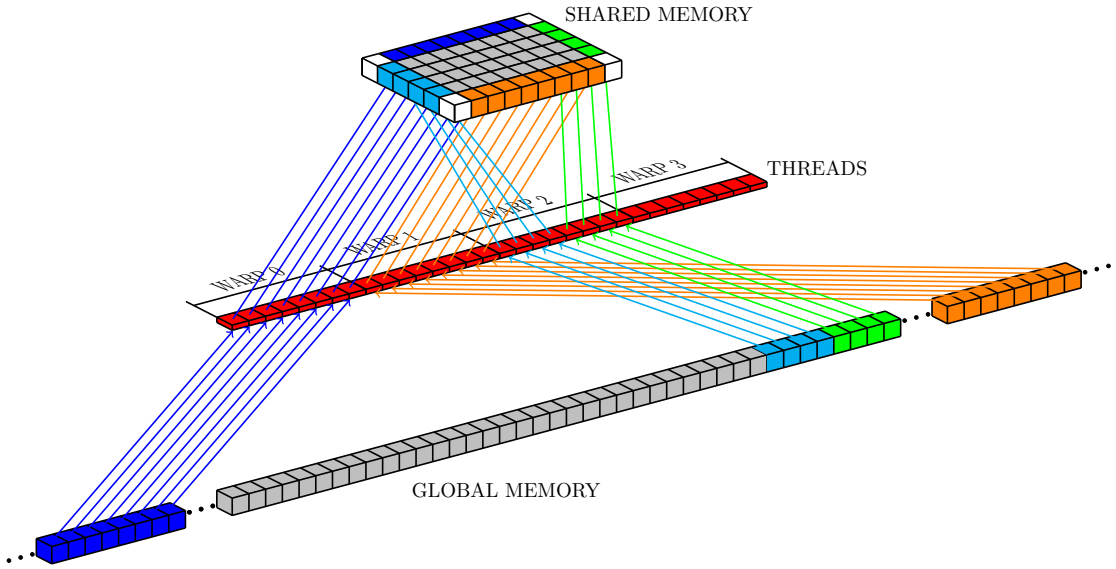
A Figura 4.9 ilustra o padrão de acesso aos dados para cada bloco. Neste exemplo, consideramos, por simplicidade, uma torre cujas camadas $z$ tenham dimensões $8 \times 4$, ou seja, oito células em $x$ e quatro em $y$. Os quadrados vermelhos representam as threads de um mesmo bloco e as setas representam os elementos acessados pelas threads. A Figura 4.9a mostra o padrão de acesso aos dados do núcleo (valores efetivamente operados pelas `threads`). O acesso aos dados do núcleo correspondem a etapa $S_{i,j} = U^t_{x,y,z}$ no fluxograma da Figura 4.8. A cópia das vizinhanças $X$ e $Y$ da memória global para compartilhada é representada na Figura 4.9b. As cores azul escuro e laranja representam a cópia das vizinhanças em $Y$ (linhas 15-26 do Algoritmo 1). Já as cores azul clara e verde representam a cópia dos dados das vizinhanças em $X$ (linhas 27-34 do Algoritmo 1), armazenados nos *buffers* extras de cada torre. O padrão de acesso para as vizinhanças em $Z$ é similar ao mostrado na 4.9a, mas os dados são carregados para os registradores ($U_{up}$ e $U_{down}$) e não para memória compartilhada. Imaginando que um `warp` seja composto por oito `threads` e que o conjunto de 128 bytes de dados coalescido pelo *hardware* da GPU seja representado por oito posições consecutivas de memória, os padrões de acesso exibidos na Figura 4.9 demonstram o alinhamento dos dados na `towerDS` proporciona ambiente favorável a acessos coalescentes para todos os dados necessários ao cálculo do passo de tempo numérico.

Ao usar a `towerDS` faz-se necessário introduzir uma nova etapa no processamento, a atualização dos *buffers* extras de vizinhança. Essa nova tarefa, exibida em vermelho no fluxograma da Figura 4.8, é descrita no Algoritmo 2. Nas linhas de 1 à 5 são definidas as dimensões da estrutura `towerDS` e os índices da `thread` e do bloco. As linhas de 7 à 10 contém as equações auxiliares de mapeamento na estrutura. Nas linhas de 11 à 27 encontra-se o processo iterativo no qual `thread` são designadas para atualizar os *buffers* de vizinhança. O processo de atualização é ilustrado na Figura 4.10; para atualizar os *buffers* de vizinhança nas torres à direita e à esquerda é utilizado um número de `threads` igual a `2*blockDim.y`.

O Algoritmo 3 descreve as tarefas desempenhadas pelo `kernel` que implementa a janela deslizante otimizada com a estrutura `towerDS` . As tarefas executadas pelo Algoritmo 3 foram organizadas de forma a reduzir a quantidade de computação no processo iterativo que percorre $Z$. Algumas barreias de sincronização são utilizadas para garantir a coerência duranta a computação. A barreira nas linhas 35, 54 e 72 garantem que: (i) todos os valores necessários estejam na memória compartilhada antes do início do cálculo da EDP; (ii) para $z > 0$, os *buffers* de vizinhança tenham sido atualizados nas torres adjacentes em $X$. Já as barreiras nas linhas 38, 57 e 75 garantem que todos os valores atualizados pelo




(a)




(b)

Figura 4.9: Padrão de leitura de dados da memória global.

cálculo do passo de tempo numérico estejam na memória compartilhada antes de serem atualizados na memória global. A variável auxiliar `Sout` atende a duas necessidades: (i) ela existe para prevenir que as `threads` de um `warp` não sobrescrevam na variável `S` valores que ainda não foram acessados por `threads` de outros `warps`; (ii) para permitir acesso por todas as `threads` do bloco aos valores de saída, possibilitando a execução do processo de atualização de vizinhanças com uso recursivo de `threads` (Algoritmo 2).

O objetivo principal da estratégia apresentada aqui é otimizar o desempenho da so-

**Algoritmo 2** Processo de atualização das vizinhanças $X$ e $Y$ na memória global usando a `towerDS` .

```
 1: msize_z ⇐ número de células do domíno discreto na direção Z
 2: coresize ⇐ blockDim.y*blockDim.x                ▷ nº de células por camada Z de uma torre
 3: zlayersize ⇐ blockDim.y*(blockDim.x + NHSIZE)   ▷ nº de posições por camada Z de uma torre
 4: thidx ⇐ threadIdx.y*blockDim.x + threadIdx.x                  ▷ índice linear da thread
 5: bidx ⇐ blockIdx.y*gridDim.x + blockIdx.x                      ▷ índice linear do bloco
 6: l ⇐ thidx
 7: a2 ⇐ 2*blockDim.y
 8: a3 ⇐ bidx*(zlayersize*msize_z) + coresize
 9: a6 ⇐ (bidx-1)*msize_z*zlayersize + coresize
10: a7 ⇐ (bidx+1)*msize_z*zlayersize + coresize
11: while l < a2 do
12:     if l < blockDim.y then
13:         if blockIdx.x > 0 then
14:             sidx ⇐ a6 + blockDim.y + l
15:             voltOUT[sidx] ⇐ S[1][l+1]
16:         else
17:             sidx ⇐ a3 + l
18:             voltOUT[sidx] ⇐ S[1][l+1]
19:     else
20:         sj ⇐ l - blockDim.y;
21:         if blockIdx.x < (gridDim.x-1) then
22:             sidx ⇐ a7 + sj
23:             voltOUT[sidx] ⇐ S[blockDim.x][sj+1]
24:         else
25:             sidx ⇐ a3 + blockDim.y + sj
26:             voltOUT[sidx] ⇐ S[blockDim.x][sj+1]
27:     l ⇐ l + coresize
```

lução numérica na GPU, através da redução da latência de acesso a memória global, por meio de acesos coalescentes, e maximizar a quantidade de threads ativas, pela redução da divergência de código e reuso de threads na escrita e leitura de dados da memória global. No Capítulo 5 é apresentada a avaliação dos benefícios da `towerDS` , aqui proposta, quando comparada a implementação direta JDC baseada na literatura [2, 30, 31, 45].

## 4.4 Estrutura de dados adaptativa

A estrutura de dados `towerDS` foi desenvolvida para acelerar o calculo da solução numérica do termo difusivo da Equação 2.1. Contudo, modelos complexos, tais como o OVVR [36], possuem um número expressivo de EDOs que não se beneficiam da estrutura[1]. Neste capítulo será apresentada uma estrutura de dados alternativa, que possibilita reduzir a quantidade de trabalho executada pela GPU para processar uma simulação numérica da eletrofisiologia do tecido cardíaco.

[1]Diferente da EDP o cálculo da solução numérica das EDOs não depende de vizinhança (vide Capítulo 2) e pro isso não pode ser acelerado com o uso da estrutura `towerDS` .

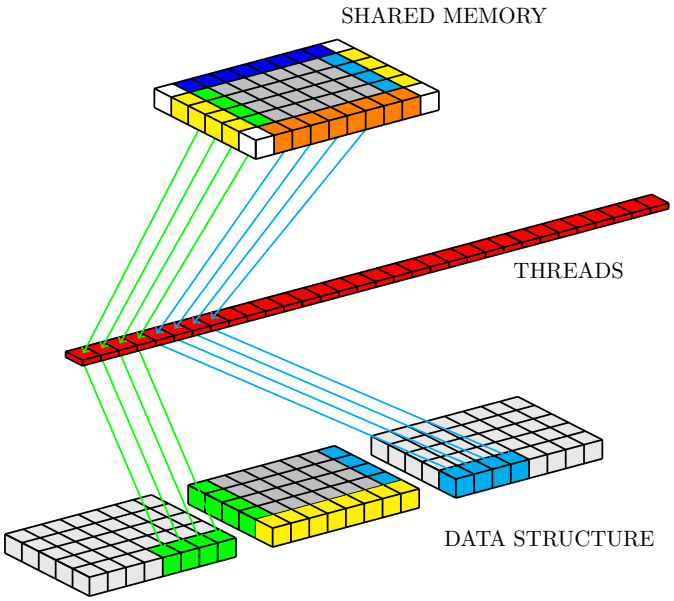



Figura 4.10: Padrão de escrita de dados nos blocos vizinhos.

### 4.4.1 Paralelização das solução das equações diferenciais ordinárias em GPU

A equação de reação difusão que modela a eletrofisiologia do tecido cardíaco é composta por uma EDP e um sistema de EDOs ( vide Seção 2 ). Quanto mais complexo o modelo maior o tamanho do sistema. Contudo, diferente da EDP, as EDOs apresentam apenas dependência temporal, ou seja, o cálculo da solução numérica é independente para cada célula do domínio espacial discreto. Podemos, portanto, designar uma única thread para resolver o sistema de EDOs em cada célula do espaço discreto. Desta forma, uma GPU é capaz de processar milhares de células concorrentemente e o cálculo da solução numérica para o sistema de EDOs pode ser massivamente paralelizado usando o CUDA.

Na solução da parte reativa da Equação 2.1, o acesso a memória global não mais configura um gargalo na solução paralela, uma vez que:

- a ordenação linha majoritária combinada com blocos cujo número de threads em $X$ seja múltiplo de meio `warp` permite acessos coalescentes a memória global;
- não há redundância na leitura dos dados;
- a quantidade de processamento por acesso é superior ao da EDP.

Nestes casos descobriu-se ser importante o gerenciamento correto dos recursos disponíveis *in-chip*, tal como os registradores e a memória compartilhada. Uma GPU atual é capaz de

**Algoritmo 3** Janela deslizante com a estrutura `towerDS` .

```
 1: msize_x ⇐ número de células do domíno discreto na direção X
 2: msize_y ⇐ número de células do domíno discreto na direção Y
 3: msize_z ⇐ número de células do domíno discreto na direção Z
 4: x ⇐ blockIdx.x*blockDim.x + threadIdx.x
 5: y ⇐ blockIdx.y*blockDim.y + threadIdx.y
 6: thidx ⇐ threadIdx.y*blockDim.x + threadIdx.x                ▷ índice linear da thread
 7: bidx ⇐ blockIdx.y*gridDim.x + blockIdx.x                    ▷ índice linear do bloco
 8: a0 ⇐ 2*blockDim.x
 9: a1 ⇐ 2*blockDim.x + blockDim.y
10: a2 ⇐ 2*blockDim.y
11: msize2d ⇐ msize_y*msize_x
12: inicializa array S[blockDim.x+NHSIZE][blockDim.y+NHSIZE]
13: inicializa array Sout[blockDim.x][blockDim.y]
14: if x < msize_x AND y < msize_y then
15:     i ⇐ threadIdx.x + 1
16:     j ⇐ threadIdx.y + 1
17:     sx ⇐ blockDim.x + 1
18:     sy ⇐ blockDim.y + 1
19:     zmax ⇐ msize_z - 1
20:     coresize ⇐ blockDim.y*blockDim.x
21:     zlayersize ⇐ blockDim.y*(blockDim.x + NHSIZE)
22:     idx ⇐ bidx*(zlayersize*msize_z) + thidx
23:     bnhsize ⇐ 2*( blockDim.x + blockDim.y )          ▷ tamanho da vizinahnça XY do bloco
24:     a3 ⇐ bidx*(zlayersize*msize_z) + coresize
25:     a4 ⇐ (bidx - gridDim.x )*(zlayersize*msize_z) + coresize - blockDim.x + threadIdx.x
26:     a5 ⇐ (bidx + gridDim.x )*(zlayersize*msize_z) + threadIdx.x
27:     a6 ⇐ (bidx-1)*msize_z*zlayersize + coresize
28:     a7 ⇐ (bidx+1)*msize_z*zlayersize + coresize
29:     S[i][j] ⇐ voltIN[idx]
30:     Stop ⇐ voltIN[(idx+zlayersize)]
31:     Sdown ⇐ Stop
32:     l ⇐ thidx
33:     ==> Cópia do dados para memória compartilhada (Algoritmo 1)
34:     ==> Cáculo da solução numérica da(s) EDO(s)
35:     ==> sincronização das threads
36:     Sout[(i-1)][(j-1)] ⇐ valor atualizado pelo cáluculo do passo de tempo numérico
37:     voltOUT[idx] ⇐ Sout[(i-1)][(j-1)]
38:     ==> sincronização das threads
39:     l ⇐ thidx
40:     ==> atualização dos buffers de vizinhança (Algoritmo 2)
41:     for 1 < z < zmax do
42:         idx ⇐ idx + zlayersize
43:         a3 ⇐ a3 + zlayersize
44:         a4 ⇐ a4 + zlayersize
45:         a5 ⇐ a5 + zlayersize
46:         a6 ⇐ a6 + zlayersize
47:         a7 ⇐ a7 + zlayersize
48:         Sdown ⇐ S[i][j]
49:         S[i][j] ⇐ Stop
50:         Stop ⇐ voltIN[(idx+zlayersize)]
51:         l ⇐ thidx
52:         ==> Cópia do dados para memória compartilhada (Algoritmo 1)
53:         ==> Cáculo da solução numérica da(s) EDO(s)
54:         ==> sincronização das threads
```

```
55:         Sout[(i-1)][(j-1)] ⇐ valor atualizado pelo cáluculo do passo de tempo numérico
56:         voltOUT[idx] ⇐ Sout[(i-1)][(j-1)]
57:         ==> sincronização das threads
58:         l ⇐ thidx
59:         ==> atualização dos buffers de vizinhança (Algoritmo 2)
60:     idx ⇐ idx + zlayersize
61:     a3 ⇐ a3 + zlayersize
62:     a4 ⇐ a4 + zlayersize
63:     a5 ⇐ a5 + zlayersize
64:     a6 ⇐ a6 + zlayersize
65:     a7 ⇐ a7 + zlayersize
66:     Sdown ⇐ S[i][j]
67:     S[i][j] ⇐ Stop
68:     Stop ⇐ Sdown
69:     l = thidx
70:     ==> Cópia do dados para memória compartilhada (Algoritmo 1)
71:     ==> Cáculo da solução numérica da(s) EDO(s)
72:     ==> sincronização das threads
73:     Sout[(i-1)][(j-1)] = valor atualizado pelo cáluculo do passo de tempo numérico
74:     voltOUT[idx] = Sout[(i-1)][(j-1)]
75:     ==> sincronização das threads
76:     l = thidx
77:     ==> atualização dos buffers de vizinhança (Algoritmo 2)
```

executar simultaneamente até 8 blocos por SM. Contudo, esse número está diretamente relacionado a quantidade de recursos *in-chip* disponíveis. Se, por exemplo, um bloco consome metade dos registradores disponíveis para o SM, não será possível alocar mais do que dois destes blocos no SM. Portanto, o uso dos recursos pode ser um limitante para a quantidade de blocos ativos em cada multiprocessador (SM) da GPU.

### 4.4.2 Mapeamento do tecido cardíaco

Simulações mais realistas do tecido cardíaco envolvem domínios irregulares que representam a geometria de um coração. Uma maneira de representar tal geometria é através do método de campo de fase (*phase-field method*) [17]. A aplicação desse método consiste no uso de um campo auxiliar $\phi$. Tal campo assume valores distintos dentro e fora do tecido cardíaco e varia suavemente na região que conecta essas duas regiões. A Figura 4.11 mostra um mapeamento do coração do porco usando um campo[2] $\phi$. Com o uso do campo de fase a Equação 2.1 assume a forma:

[2]O campo $\phi$ usado neste trabalho foi cedido pelo grupo do professor Flavio Fenton do *Georgia Institute of Technology*

$$\phi \frac{\partial U}{\partial t} = \nabla \cdot \mathbf{D} \phi \nabla U - \phi \frac{I_{ion}}{C_m} . \tag{4.1}$$

$$\tag{4.2}$$

O campo de fase assume valores 1 para a região do ventrículo decrescendo suavemente para 0 em células vazias (região sem tecido). Com $\mathbf{D}$ uniforme e valor unitário, a aproximação por diferenças centrada fica, no caso 3D:

$$\begin{aligned}
\nabla \cdot \phi \nabla U^n_{i,j,k} \simeq & \frac{(\phi_{i+1,j,k} + \phi_{i,j,k}) \left(U^n_{i+1,j,k} - U^n_{i,j,k}\right) - (\phi_{i,j,k} + \phi_{i-1,j,k}) \left(U^n_{i,j,k} - U^n_{i-1,j,k}\right)}{\Delta x} \\
& \frac{(\phi_{i,j+1,k} + \phi_{i,j,k}) \left(U^n_{i,j+1,k} - U^n_{i,j,k}\right) - (\phi_{i,j,k} + \phi_{i,j-1,k}) \left(U^n_{i,j,k} - U^n_{i,j-1,k}\right)}{\Delta y} \\
& \frac{(\phi_{i,j,k+1} + \phi_{i,j,k}) \left(U^n_{i,j,k+1} - U^n_{i,j,k}\right) - (\phi_{i,j,k} + \phi_{i,j,k-1}) \left(U^n_{i,j,k} - U^n_{i,j,k-1}\right)}{\Delta z} .
\end{aligned}$$

O campo de fase $\phi$ usado aqui é tri-dimensional e ocupa um paralelepípedo de dimensões $250 \times 326 \times 298$. A volume ativo, que corresponde a presença de tecido, ocupa aproximadamente 28% deste paralelepípedo. Portanto, ao processar todo o volume estamos desperdiçando um grande contingente de recursos processando pontos que correspondem a espaços vazios. Neste caso a estrutura de torre não é capaz de se ajustar a estrutura irregular, obrigando-nos a processar todo o domínio.

Pensando no domínio irregular que modela a geometria de um coração e como processá-lo de forma mais eficiente na GPU, propomos e desenvolvemos uma estrutura de dados adaptativa que se ajusta ao domínio reduzindo o desperdício de recursos com o processamento de células representando áreas sem tecido. Primeiramente, o domínio total é subdividido em paralelepípedos menores. No segundo passo todos os paralelepípedos que não possuírem em seu interior regiões de tecido são descartados. A Figura 4.12 mostra a região de tecido ajustada por cubos de tamanho $8 \times 8 \times 8$. A Figura 4.13 oferece uma vista superior da estrutura. Neste caso, a opacidade das células cinzas que representam a estrutura foi reduzida para ser possível vislumbrar como o volume definido pelo campo $\phi$ (pontos vermelhos) está contido na mesma. Com esta estrutura iremos, em nossa aplicação CUDA, designar um bloco de threads para processar cada um dos subdomínios.

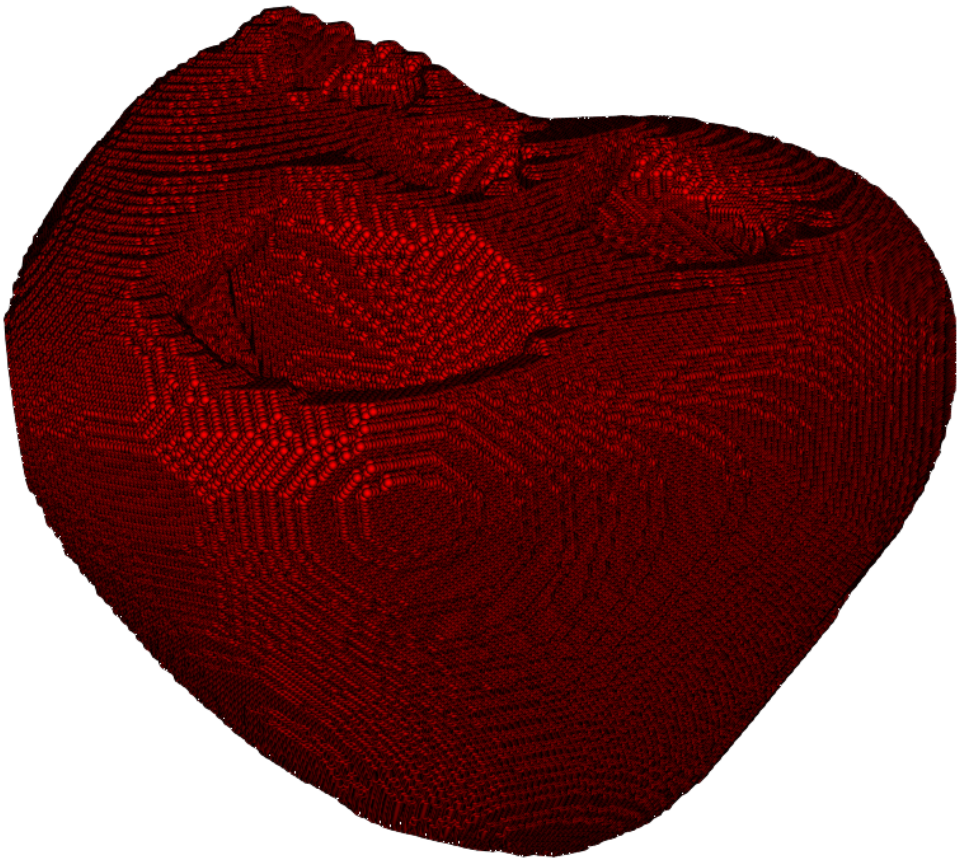


Figura 4.11: Visualização do campo $\Phi$ representando a geometria do coração do porco. As pequenas esferas vermelhas representam o tecido (células ativas).

### 4.4.3 A estratégia de armazenamento

O mapeamento do tecido com os subdomínios em forma de paralelepípedos é feita em forma de pre-processamento. Após identificar os paralelepípedos contendo tecido é preciso armazenar seus dados na memória global da GPU. Propomos aqui uma estratégia de armazenamento em quatro passos que possibilita, durante o cálculo do passo de tempo, tirar vantagem da coalescência de acesso a dados da GPU em um único acesso.

O primeiro passo consiste em atribuir a cada subdomínio um índice que determinará a ordem na qual os subdomínios serão armazenados. Para tal, percorre-se o conjunto completo de subdomínios seguindo a ordem linha majoritária, ou seja, percorremos toda uma linha e passamos para próxima, assim sucessivamente para cada camada Z. Cada subdomínio ativo encontrado recebe índice em ordem crescente. Daqui por diante, iremos nos referir aos subdomínios contendo regiões de tecido como ativos. O Algoritmo 4 apresenta este processo.

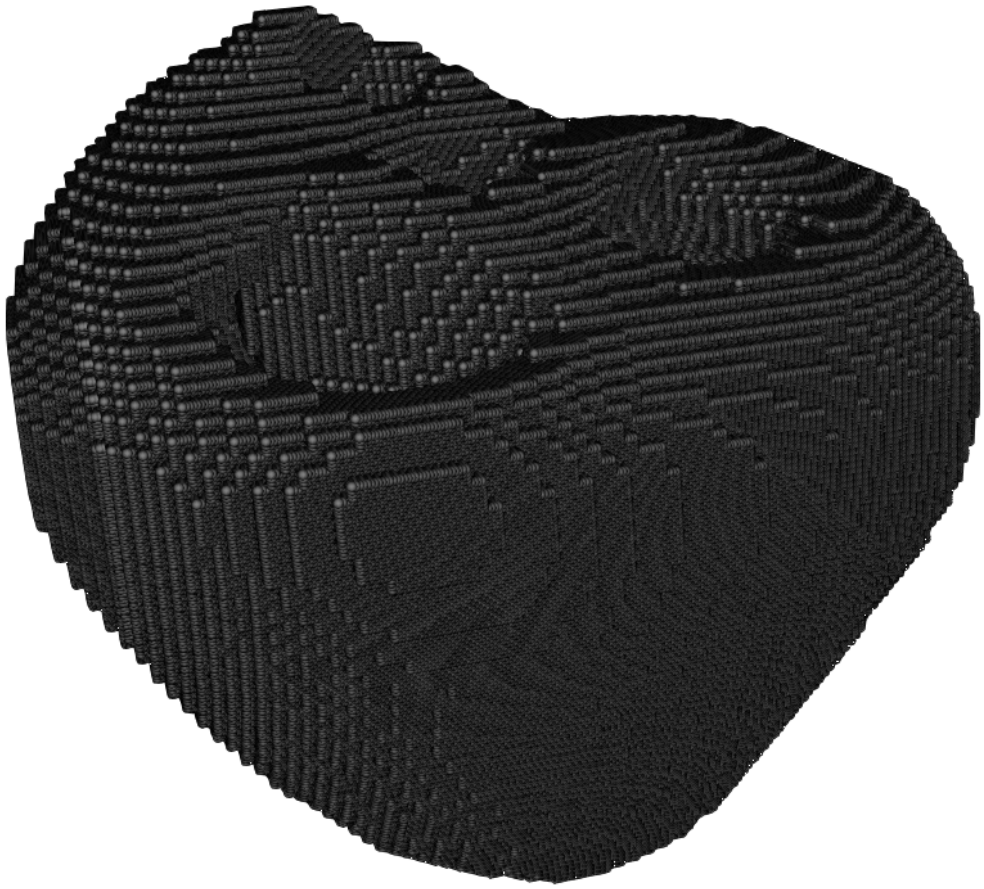


Figura 4.12: Estrutura usada para processar o tecido definido pelo campo de fase $\phi$. As esferas cinzas representam as células a serem armazenadas e processadas pelas threads. Elas estão organizadas em blocos de tamanho $8 \times 8 \times 8$ e contém tanto tecido como espaços vazios.

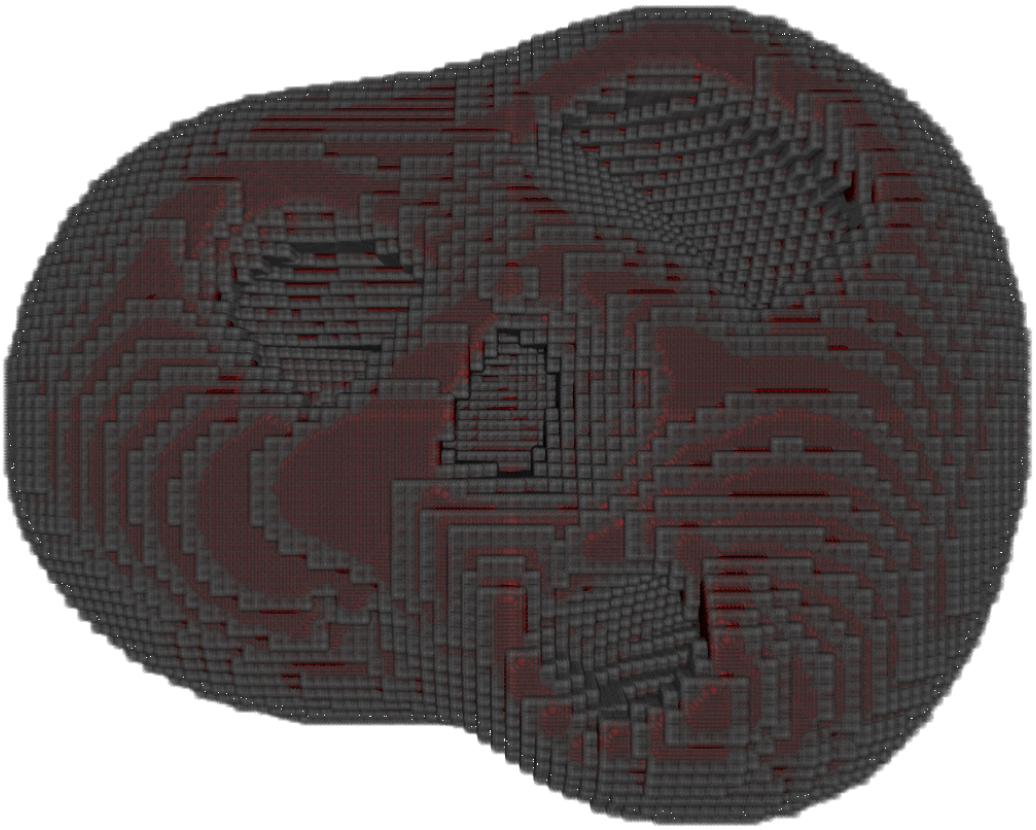


Figura 4.13: Vista superior da estrutura de dados em bloco. A opacidade das células cinzas que representam a estrutura foi reduzida para ser possível vislumbrar como o volume definido pelo campo $\phi$ (pontos vermelhos) está contido nela.

**Algoritmo 4** Atribuição de índices sequenciais aos subdomínios ativos

```
1: nsx ⇐ número de subdomínios em X
2: nsy ⇐ número de subdomínios em Y
3: nsz ⇐ número de subdomínios em Z
4: c ⇐ 0
5: for 0 ≤ k < nsz do
6:     for 0 ≤ j < nsy do
7:         for 0 ≤ i < nsx do
8:             idx ⇐ (k ∗ nsy + j) ∗ nsx + i
9:             if subdomínio[idx] contém tecido then
10:                idx_sub[c] ⇐ idx
11:                c ⇐ c + 1
```

O segundo passo consiste em armazenar cada subdomínio como uma *array* 1D que favoreça acessos coalescentes pelas threads do CUDA. Isso pode ser facilmente conseguido com o uso, novamente, da ordenação linha majoritária, conforme mostrado no Algoritmo 5. Contudo, essa solução mais simples fica inviabilizada pela dependência espacial imposta pela EDP que descreve o acoplamento das células (Equação 2.1).

**Algoritmo 5** Ordenação linear das células nos subdomínios ativos.

```
1: sddim_z ⇐ tamanho do subdomínio em Z
2: sddim_y ⇐ tamanho do subdomínio em Y
3: sddim_x ⇐ tamanho do subdomínio em X
4: sddim3D ⇐ sddim_z*sddim_y*sddim_x
5: for 0 ≤ c < (número de subdomínios ativos) do
6:     for 0 ≤ k < (sddim_z) do
7:         for 0 ≤ j < (sddim_y) do
8:             for 0 ≤ i < (sddim_x) do
9:                 idx ⇐ (k ∗ (sddim_y) + j) ∗ (sddim_x) + i + sddim3D ∗ idx_sub[c] ▷ posição de célula no domínio
10:                dsidx ⇐ (k ∗ (sddim_y) + j) ∗ (sddim_x) + i + sddim3D ∗ c ▷ posição de célula na estrutura de dados
11:                dscell[dsidx] ⇐ cell[idx]
```

A Figura 4.2 ilustra a vizinhança necessária ao processamento de um subdomínio. Precisamos, portanto, adequar o sistema de indexação proposto para contemplar tais vizinhanças. Usaremos aqui uma ideia similar a utilizada na estrutura em torres. *Buffers* extras de memória serão criados para armazenar a vizinhança de cada subdomínio ativo. Desta forma o processamento de cada subdomínio, durante o cálculo do passo de tempo, fica independente dos demais. O Algoritmo 6 mostra as modificações feitas no Algoritmo 5 necessárias para adequá-lo. A Figura 4.14 ilustra a distribuição linear dos dados de um subdomínio cubico e dos *buffers* que armazenam as vizinhanças.

O terceiro passo consiste em anotar o índice linear dos blocos (subdomínios) vizinhos

**Algoritmo 6** Ordenação linear das células nos subdomínios ativos com *buffers* extras de memória.

```
1: sddim_z ⇐ tamanho do subdomínio em Z
2: sddim_y ⇐ tamanho do subdomínio em Y
3: sddim_x ⇐ tamanho do subdomínio em X
4: sddim3D ⇐ sddim_z*sddim_y*sddim_x
5: vdim ⇐ (tamanho da vizinhança em Z) + (tamanho da vizinhança em Y) + (tamanho da
   vizinhança em X)
6: for 0 ≤ c < (número de subdomínios ativos) do
7:     for 0 ≤ k < (sddim_z) do
8:         for 0 ≤ j < (sddim_y) do
9:             for 0 ≤ i < (sddim_x) do
10:                idx ⇐ (k * (sddim_y) + j) * (sddim_x) + i + sddim3D * idx_sub[c]
11:                dsidx ⇐ (k * (sddim_y) + j) * (sddim_x) + i + (sddim3D + vdim) * c
12:                dscell[dsidx] ⇐ cell[idx]
```

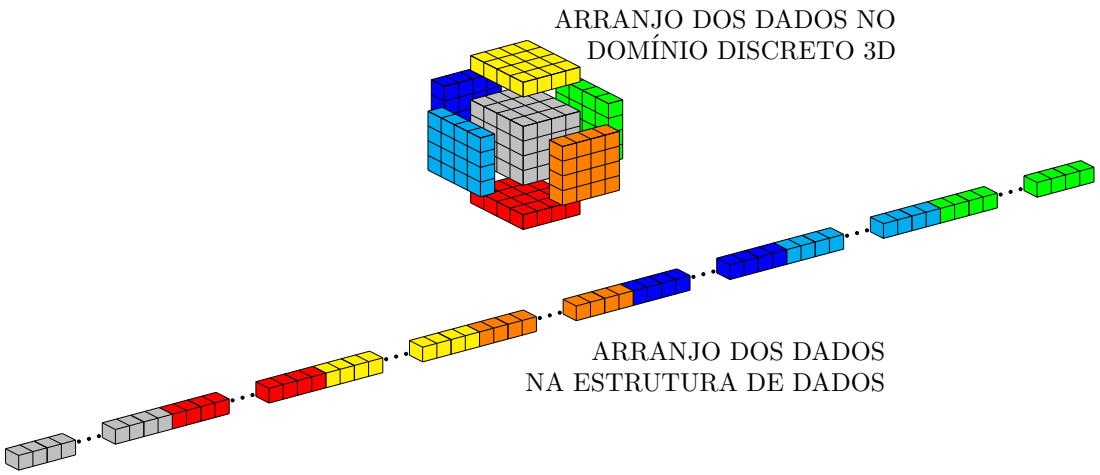



Figura 4.14: Distribuição dos dados de um bloco em um *array* 1D.

a cada bloco ativo. Como estamos lidando com vizinhanças ortogonais em um espaço 3D, para cada bloco precisaremos armazenar o índice de seis blocos vizinhos[3]. O fluxograma na Figura 4.15 apresenta de forma simplificada as tarefas executadas no mapeamento dos vizinhos. O Algoritmo 7 mostra uma versão modificada do Algoritmo 5 para incluir a identificação dos blocos vizinhos, incluindo toda a matemática usada na construção do mapa. O *array* `vizinhos` guarda, para cada bloco ativo, os índices dos blocos vizinhos. Contudo, existem duas classificações extras: (i) sempre que o bloco vizinho for inativo, o valor $-1$ será armazenado no lugar do índice; (ii) sempre que o bloco pertencer a borda do domínio, o valor $-5$ será armazenado no lugar do índice para o bloco vizinho referente a borda. Desta forma, poderemos atualizar os *buffers* extras com a informação correta sobre a vizinhança.

[3]Na prática, esse vetor de vizinhos é construído na CPU e enviado para GPU na etapa de pré-processamento. Uma vez construído, esse vetor não será novamente modificado durante a execução de uma simulação.

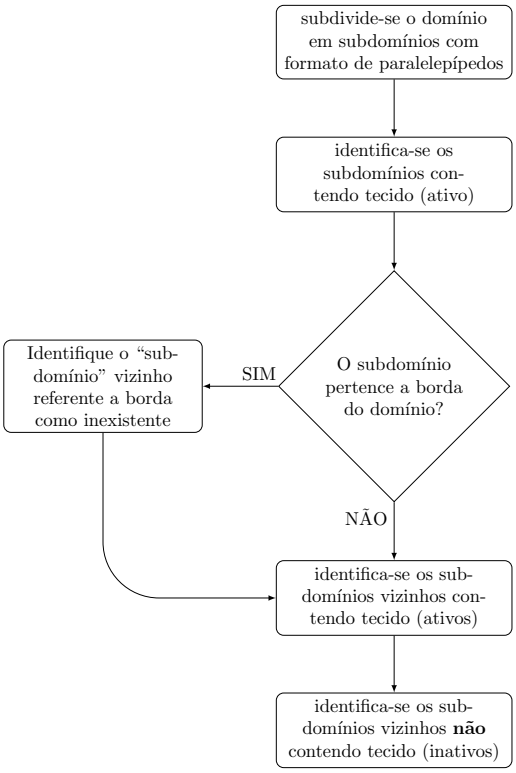



Figura 4.15: Tarefas executadas para mapear a posição dos vizinhos de cada subdomínio ativo na estrutura blockDS.

Os passos 1, 2 e 3 são executados somente uma vez. O quatro passo consiste em atualizar os *buffers* extras de cada cubo de dados com as informações corretas das vizinhanças. Esta etapa é executada por um `kernel` CUDA e, portanto, será abordado juntamente com a discussão das estratégias de acesso.

### 4.4.4 A estratégia de acesso

A ideia principal da estrutura de dados em bloco é designar um bloco de threads para processar um único subdomínio. Por conveniência, daqui por diante esta estrutura de dados será referenciada apenas como blockDS. Como os blocos foram organizados para armazenamento em uma estrutura linear podemos usar uma grade de blocos de threads também linear. Desta forma designamos uma thread para cada célula do subdomínio e podemos re-aproveitar threads para carregar as vizinhanças na memória compartilhada. Podemos também re-utilizar estes dados para atualizar os *buffers* de vizinhanças ao final da computação de cada passo de tempo.

O `kernel` responsável por calcular um passo de tempo pode ser dividido em quatro partes: (i) cópia dos dados do potencial de membrana da memória global para a compar-

**Algoritmo 7** Atribuição de índices sequenciais aos subdomínios ativos mais identificação de vizinhos.

```
1: nsx ⇐ número de subdomínios em X
2: nsy ⇐ número de subdomínios em Y
3: nsz ⇐ número de subdomínios em Z
4: ns2d ⇐ nsy * nsx
5: ns ⇐ nsz * ns2d
6: c ⇐ 0
7: for 0 ≤ idx < ns do
8:     if subdomínio[idx] contém tecido then
9:         active[idx] ⇐ 1                          ▷ identifica os subdomínios com tecido
10:        idx_sub[c] ⇐ idx                         ▷ armazena os índice dos subdomínios com tecido
11:        c ⇐ c + 1                                ▷ conta os subdomínios com tecido
12:    else
13:        active[idx] ⇐ 0
14: d ⇐ 0
15: for 0 ≤ s < c do
16:    idx ⇐ idx_sub[s]
17:    k ⇐ idx / ns2d                               ▷ Posição do subdomínio na direção Z
18:    j ⇐ (idx % ns2d) / nsx                       ▷ Posição do subdomínio na direção Y
19:    i ⇐ idx - (k*nsy + j)*nsx                    ▷ Posição do subdomínio na direção X
20:    if k = 0 then                                ▷ Caso que o subdomínio está na borda do domínio
21:        vizinho[d] ⇐ -5
22:    else if active[(idx - ns2d)] = 1 then        ▷ Caso que o subdomínio contém tecido
23:        vizinho[d] ⇐ (idx - ns2d)
24:    else                                         ▷ Caso que o subdomínio não contém tecido
25:        vizinho[d] ⇐ -1
26:    if (k+1) ≥ nsz then                          ▷ Caso que o subdomínio está na borda do domínio
27:        vizinho[d+1] ⇐ -5
28:    else if active[(idx + ns2d)] = 1 then        ▷ Caso que o subdomínio contém tecido
29:        vizinho[d+1] ⇐ (idx + ns2d)
30:    else                                         ▷ Caso que o subdomínio não contém tecido
31:        vizinho[d+1] ⇐ -1
32:    if j = 0 then                                ▷ Caso que o subdomínio está na borda do domínio
33:        vizinho[d+2] ⇐ -5
34:    else if active[(idx - nsx)] = 1 then         ▷ Caso que o subdomínio contém tecido
35:        vizinho[d+2] ⇐ (idx - nsx)
36:    else                                         ▷ Caso que o subdomínio não contém tecido
37:        vizinho[d+2] ⇐ -1
38:    if (j+1) ≥ nsy then                          ▷ Caso que o subdomínio está na borda do domínio
39:        vizinho[d+3] ⇐ -5
40:    else if active[(idx + nsx)] = 1 then         ▷ Caso que o subdomínio contém tecido
41:        vizinho[d+3] ⇐ (idx + nsx)
42:    else                                         ▷ Caso que o subdomínio não contém tecido
43:        vizinho[d+3] ⇐ -1
44:    if i = 0 then                                ▷ Caso que o subdomínio está na borda do domínio
45:        vizinho[d+4] ⇐ -5
46:                                                 ▷ Continua na próxima página.
```

```
47:   else if active[(idx - 1)] = 1 then          ▷ Caso que o subdomínio contém tecido
48:       vizinho[d+4] ⇐ (idx - 1)
49:   else                                        ▷ Caso que o subdomínio não contém tecido
50:       vizinho[d+4] ⇐ -1
51:   if (i+1) ≥ nsx then                         ▷ Caso que o subdomínio está na borda do domínio
52:       vizinho[d+5] ⇐ -5
53:   else if active[(idx + 1)] = 1 then          ▷ Caso que o subdomínio contém tecido
54:       vizinho[d+5] ⇐ (idx + 1)
55:   else                                        ▷ Caso que o subdomínio não contém tecido
56:       vizinho[d+5] ⇐ -1
57:   d ⇐ d + 6
```

tilhada, incluindo as vizinhanças; (ii) cálculo do passo de tempo numérico; (iii) escrita dos valores atualizados na memória global; (iv) atualização dos *buffers* de vizinhança. A parte (ii) é variável pois a quantidade de cálculos que precisam ser feitos depende do modelo celular utilizado. Os modelos complexos, que possuem um grande número de variáveis de estado, são muito exigentes com relação ao uso de registradores. Tais variáveis são lidas, operadas e reescritas na memória global sob demanda. Nesse caso, como cada thread acessa somente a posição de memória relativa a célula a qual ela foi designada, o fato de usarmos ordenação linha majoritária para indexar linearmente cada bloco possibilita acessos coalescentes a memória global. As partes (i), (iii) e (iv) envolvem a estratégia de acesso a estrutura de dados.

Na primeira parte do kernel, as threads irão carregar os dados da memória global para a compartilhada, como na janela deslisante. Uma diferença importante é que agora armazenamos um *array* 3D ao invés de um *array* 2D. As tarefa executadas por cada `thread` para copiar os dados da memória global para a compartilhada são mostradas na Figura 4.16, contudo, o processo de cópia é exibido mais detalhadamente no Algoritmo 8. A variável `lidx` (linha 16) guarda a posição na estrutura de dados linear correspondente a célula mapeada na thread. A variável `s` (linha 17) varia o índice linear da thread de forma que possamos re-utilizar threads para a tarefa de carregar os dados contidos nos buffers de vizinhança. As variáveis `sU` e `sP` são os *arrays* 3D na memória compartilhada, e tem dimensões igual a `bloco` + `vizinhança`. As variáveis `voltIN` e `phi` são os *arrays* lineares que armazenam, respectivamente, o potencial de membrana e o campo de fase $\phi$. As linhas de comando do Algoritmo 8 definem a execução das seguintes tarefas:

1. cópia os dados do subdomínio (linhas de 20 a 22);
2. cópia os dados do *buffer* correspondentes a vizinhança $Z$ (linhas de 23 a 28);
3. cópia os dados do *buffer* correspondentes a vizinhança $Y$ (linhas de 29 a 34);

4. cópia os dados do *buffer* correspondentes a vizinhança $X$ (linhas de 35 a 40);

As tarefas de cópia dos dados para memória compartilhada são executadas segundo um esquema de *round-robin*. A Figura 4.17 ilustra, para um bloco $2 \times 2 \times 2$, a execução sequencial da leitura dos dados por um bloco de threads.

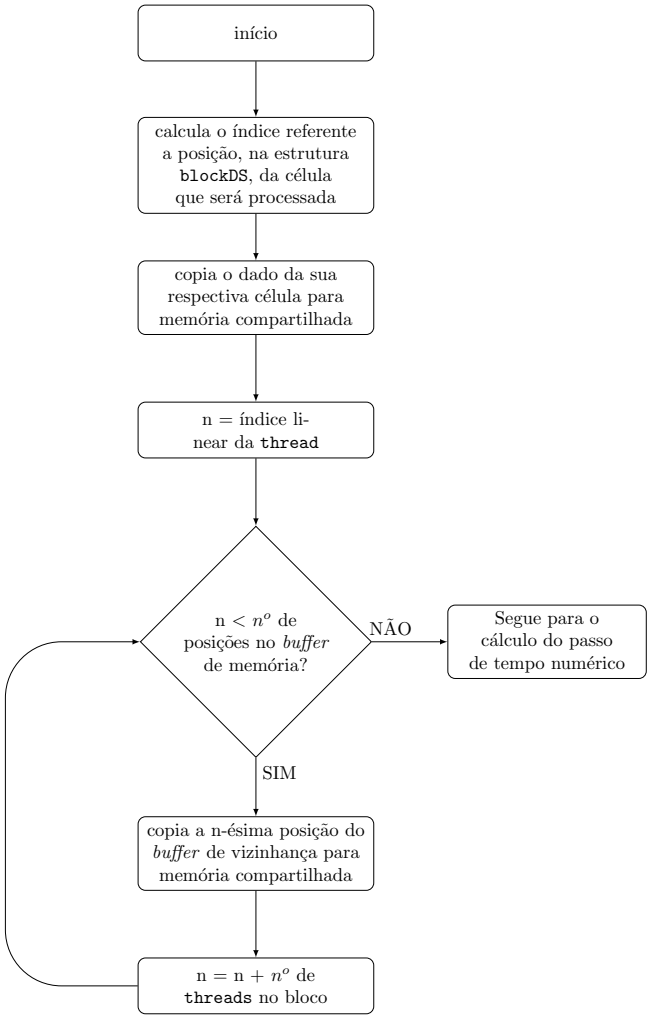



Figura 4.16: Procedimento de cópia dos dados da estrutura blockDS na memória global para a memória compartilhada.

Uma vez que todos os dados foram carregados na memória compartilhada, o kernel irá calcular a solução numérica para o próximo instante no domínio temporal discreto. Para o potencial de membrana, este valor é armazenado em um terceiro *array* 3D na memória compartilhada `Uout` – parte (ii). Na parte (iii) os valores do subdomínio processado pelo bloco serão atualizados. O processo é simples e direto pois cada thread já foi previamente mapeada em uma posição da estrutura de dados ( variável `lidx` na linha 16 do Algoritmo 8).

A parte (iv) cuida da atualização dos *buffers*, sendo apresentada esquematicamente na Figura 4.18 e descrita no Algoritmo 9. Através de um novo processo de *round-robin*, as threads de cada bloco irão copiar, para os *buffers* dos blocos vizinhos, os valores atualiza-

**Algoritmo 8** `kernel` parte (i) - carga dos dados do potencial de membrana na memória compartilhada.

```
1: memblockSize ⇐ n^o total de células no bloco núcleo + vizinhança
2: coreSize ⇐ n^o de threads no bloco ▷ mesmo que o n^o de células por subdomínio
3: nSize_z ⇐ n^o de células na vizinhança do bloco em Z ( z + 1 e z − 1 )
4: nSize_y ⇐ n^o de células na vizinhança do bloco em Z ( y + 1 e y − 1 )
5: nSize_x ⇐ n^o de células na vizinhança do bloco em X ( x + 1 e x − 1 )
6: bdimZ ⇐ n^o de threads por bloco na direção Z
7: bdimY ⇐ n^o de threads por bloco na direção Y
8: bdimX ⇐ n^o de threads por bloco na direção X
9: NBD ⇐ 1/2 do tamanho da vizinhança em cada direção
10: thZ ⇐ threadIdx.z ▷ índice Z da thread
11: thY ⇐ threadIdx.y ▷ índice Y da thread
12: thX ⇐ threadIdx.x ▷ índice X da thread
13: blkX ⇐ blockIdx.x ▷ índice X do bloco
14: tidx ⇐ (thZ*bdimY + thY)*bdimX + thX; ▷ índice linear da thread
15: lidx0 ⇐ blkX*memblockSize; ▷ posição da célula 0 do subdomínio/bloco
16: lidx ⇐ blkX*memblockSize + tidx; ▷ mapeamento da thread na célula
17: aux0 ⇐ bdimY*bdimX
18: aux1 ⇐ bdimZ*bdimX
19: s ⇐ tidx; ▷ s irá varia o índice da thread para carga da vizinhança
20: while s < memblockSize do
21:     memidx ⇐ lidx0 + s; ▷ posição atualmente sendo acessada
22:     if s < coreSize then
23:         sU[thX+NBD][thY+NBD][thZ+NBD] ⇐ gU[memidx]
24:         sP[thX+NBD][thY+NBD][thZ+NBD] ⇐ gP[memidx]
25:     else if s < (coreSize + nSize_z) then
26:         k ⇐ ((s - coreSize)/aux0) + (s ≥ (coreSize + NBD*aux0))*bdimZ
27:         j ⇐ (((s - coreSize)%aux0)/bdimX) + NBD
28:         i ⇐ (s - coreSize) - (k - (s ≥ (coreSize + NBD*aux0))*bdimZ)*aux0 - (j - NBD)*bdimX +
    NBD
29:         sU[i][j][k] ⇐ gU[memidx]
30:         sP[i][j][k] ⇐ gP[memidx]
31:     else if s < (coreSize+nSize_z+nSize_y) then
32:         j ⇐ ((s - coreSize - nSize_z)/aux1)+(s ≥ (coreSize + nSize_z + NBD*aux1))*bdimY
33:         k ⇐ (((s - coreSize-nSize_z)%aux1)/bdimX) + NBD;
34:         i ⇐ (s - coreSize - nSize_z) - ((j - (j ≥ (bdimY + NBD))*(bdimY))*aux1) - ((k - NBD)*bdimX)
    + NBD
35:         sU[i][j][k] ⇐ gU[memidx]
36:         sP[i][j][k] ⇐ gP[memidx]
37:     else if s < (coreSize+nSize_z+nSize_y+nSize_x) then
38:         i ⇐ ((s - coreSize - nSize_z - nSize_y)/(bdimZ*bdimY)) + (s ≥ (coreSize + nSize_z +
    nSize_y + NBD*bdimZ*bdimY))*bdimX
39:         k ⇐ (((s - coreSize - nSize_z)%aux1)/bdimX) + NBD
40:         j ⇐ (s - coreSize - nSize_z - nSize_y) - ((i - (i ≥ (bdimX + NBD))*(bdimX))*bdimZ*bdimY)
    - ((k - NBD)*bdimY) + NBD
41:         sU[i][j][k] ⇐ gU[memidx]
42:         sP[i][j][k] ⇐ gP[memidx]
43:     s ⇐ s + coreSize;
44: ==> sincronização das threads;
```

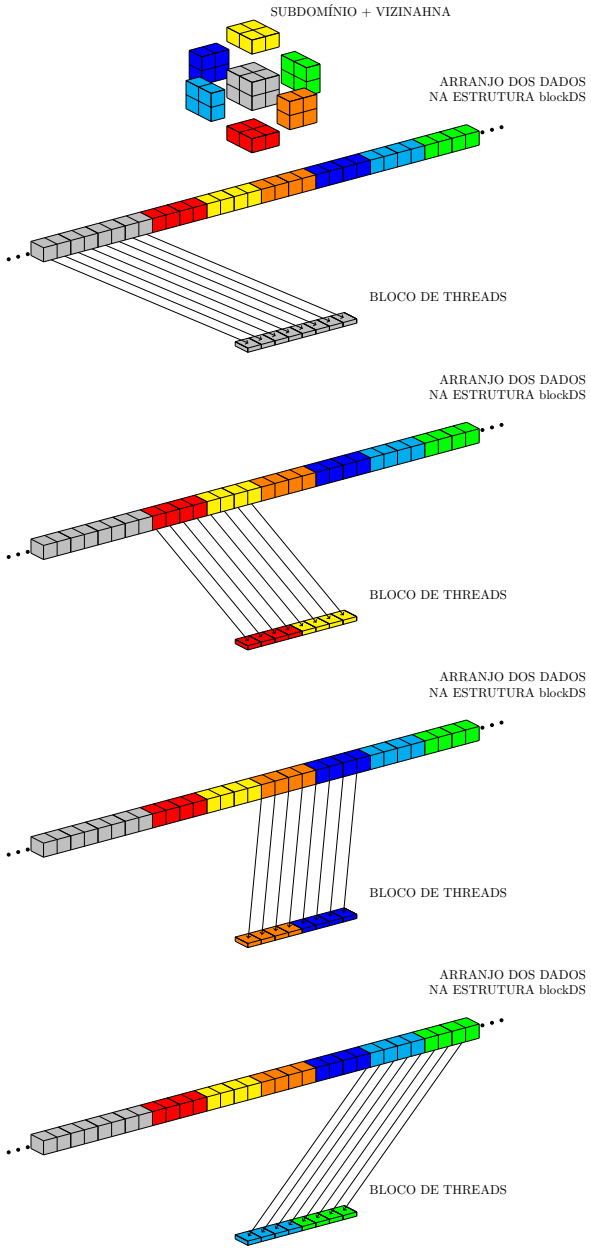



Figura 4.17: Execução sequencial da tarefa de cópia de dados da memória global para a memória compartilhada.

dos do potencial de membrana computados pelo bloco ao qual pertencem. Primeiro são atualizados os *buffers* nos vizinhos em $Z$ (linhas de 3 a 14); em seguida são atualizados os *buffers* nos vizinhos em $Y$ (linhas de 15 a 26); por fim são atualizados os *buffers* nos vizinhos em $X$ (linhas de 27 a 38). Nesta parte são aplicadas as condições de contorno do modelo. Neste caso, diferenciamos vizinhos vazios (não contendo tecido, mas dentro do domínio) dos vizinhos externos (blocos vizinhos que estariam foram do domínio). A variável `nblock` contém o vetor de vizinhos construído com o Algoritmo 7. Vizinhos marcados com -1 significam blocos vazios, enquanto vizinhos marcados com -5 significam blocos externos. Nesse último caso as condições de contorno são aplicadas. A parte (iv)

do `kernel` se confunde com o passo 4 da estratégia de armazenamento. A execução do Algoritmo 9 é necessária na etapa de pré-processamento, após a inicialização dos campos segundo a condição inicial.

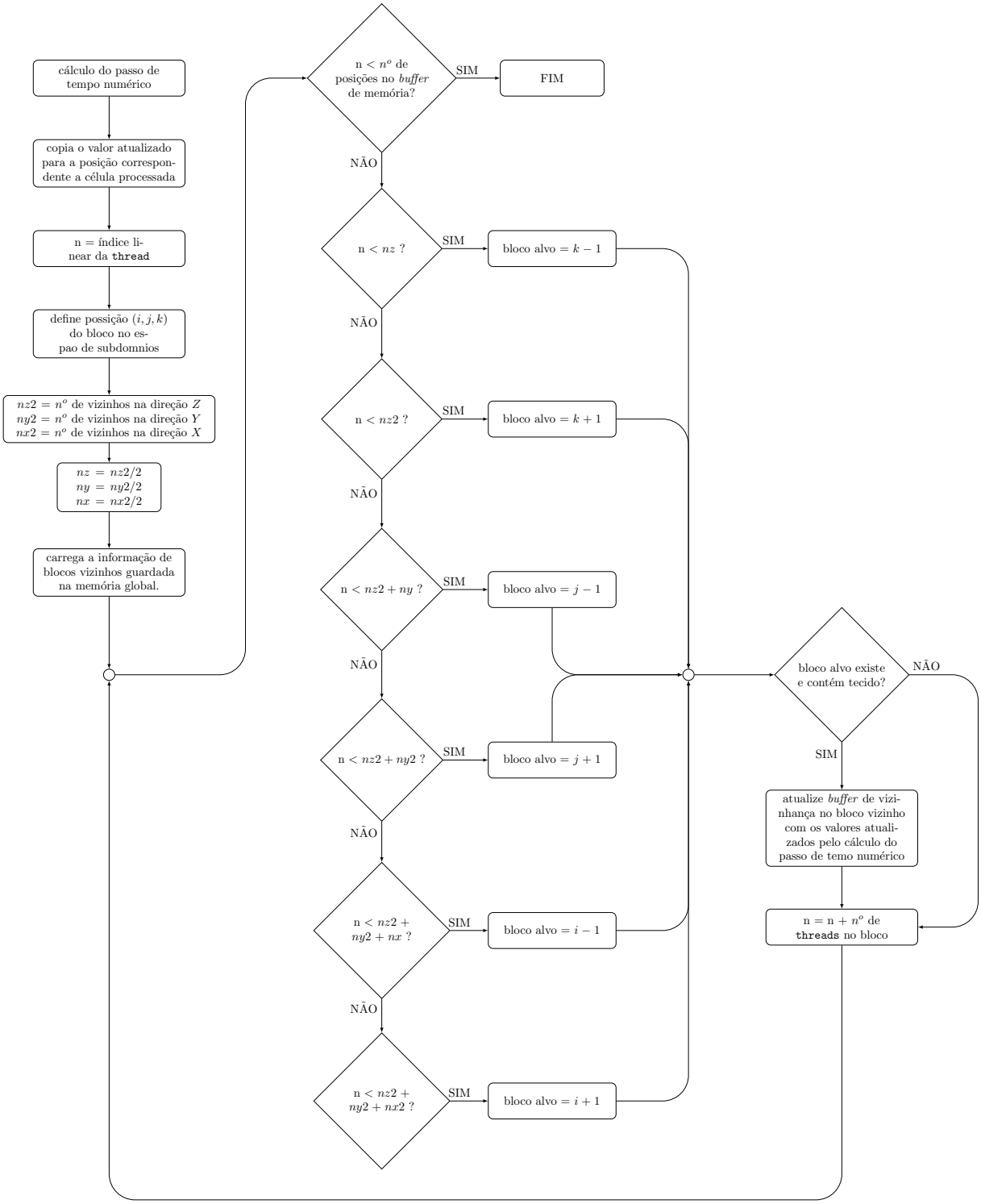



Figura 4.18: Procedimento de atualização dos *buffers* de vizinhança da estrutura blockDS na memória global.

Tal como na estrutura de dados em torres, também utilizamos dois `arrays` para

armazenar o potencial de membrana. Desta forma, garantimos que nenhum valor tenha sido indevidamente alterado durante o cálculo do passo de tempo numérico. Contudo, no caso da estrutura em blocos, cada bloco é completamente independente dos demais durante o cálculo do passo de tempo. Portanto, caso o espaço na memória global seja um problema, a parte (iv) pode ser colocada em um `kernel` separado. Isso permite o uso de um único `array`, pois os *buffers* de vizinhança serão atualizados após o término do cálculo de cada passo de tempo, com a chamada do novo `kernel`, garantindo assim que seus valores não sejam alterados indevidamente durante o cálculo do passo de tempo.

A estrutura de dados blockDS permite reduzir o tamanho do domínio enquanto usa um arranjo linear para facilitar acessos coalescentes para a memória global. Tal redução depende do tamanho do bloco utilizado, mas pode chegar a 69% do volume total, como veremos no Capítulo 5 (Seção 5.4.2). O blockDS também torna possível ajustar uma grade regular de threads do CUDA ao formado irregular do coração do porco, definido pela função de fase $\phi$. A redução do domínio influencia diretamente na quantidade de trabalho e no tamanho da grade de threads, tendo impacto direto no tempo de processamento da solução numérica.

## 4.5 Multi-GPUs

Durante os experimentos observamos que o espaço de memória global da GPU pode ser um problema. Mesmo para um modelo simples, como o Karma, e uma grade com $1024 \times 1024 \times 1024$ células, por exemplo, são necessários cerca de $12,5GB$ de espaço na memória. Para resolver essa restrição, estendemos nossa solução em torres para um ambiente com múltiplas GPUs.

Os *clusters* de GPUs apresentam uma memória distribuída, ou seja, cada GPU tem sua própria memória. Assim, para executar a simulação (as simulações de dinâmica elétrica cardíaca) em um cluster multi-GPU, é necessário dividir o domínio em subdomínios, para que cada GPU possa calcular parte do problema. Como cada GPU é responsável por computar um subdomínio e a solução numérica por diferenças finitas requer informações de células vizinhas, cada GPU deve se comunicar com outras GPUs para enviar e receber informações de células processadas por outras GPUs. Optamos por dividir o domínio na direção $Z$ devido à simetria da estrutura de dados em torre nesse eixo. Cada subdomínio tem duas áreas extras de memória (*buffers* de comunicação), nos quais são armazenados os dados de $U$ referentes aos limites superior e inferior em $Z$. A Figura 4.19 ilustra os

**Algoritmo 9** `kernel` parte (iv) - atualização dos dados nos *buffers* de vizinhança.

```
1: s ⇐ tidx;                                   ▷ s irá varia o índice da thread para carga da vizinhança
2: while s < memblockSize-coreSize do
3:     if s < nSize_z then
4:         aux ⇐ ( s < NBD*bdimY*bdimZ )
5:         k ⇐ (bdimZ-NBD)*(s<(NBD*bdimX*bdimY)) + ((s-((s ≥ (NBD*bdimX*bdimY)) *
   (NBD*bdimY*bdimX)))/(bdimX*bdimY));
6:         j ⇐ ((s%(bdimY*bdimX))/bdimX);
7:         i ⇐ s - (((s/(bdimX*bdimY)))*(bdimX*bdimY)) - j*bdimX;
8:         if nblock[(6*blkX+aux)] > -1 then
9:             memidx ⇐ (nblock[(6*blkX+aux)]*memblockSize) + coreSize + s
10:            voltOUT[memidx] ⇐ Uout[i][j][k];
11:        else if nblock[(6*blkX+aux)] = -5 then
12:            memidx ⇐ lidx0 + coreSize + s + (NBD*bdimY*bdimX)*(aux-(!aux))
13:            k ⇐ (NBD>k)*(NBD-k) + (NBD<k)*(k-NBD)
14:            voltOUT[memidx] ⇐ Uout[i][j][k]
15:    else if s < (nSize_z+nSize_y) then
16:        aux ⇐ ( s < (nSize_z + NBD*bdimZ*bdimX) )
17:        j ⇐ ((bdimY-NBD)*(s < (nSize_z+(NBD*bdimZ*bdimX))))+((s-nSize_z-
   (NBD*bdimZ*bdimX)*(s ≥ (nSize_z+(NBD*bdimZ*bdimX))))/(bdimZ*bdimX))
18:        k ⇐ (((s-nSize_z)%(bdimZ*bdimX))/bdimX)
19:        i ⇐ (s-nSize_z) - (((s-nSize_z)/(bdimZ*bdimX))*bdimZ*bdimX) - (k*bdimX)
20:        if nblock[(6*blkX+2+aux)] > -1 then
21:            memidx ⇐ (nblock[(6*blkX+2+aux)]*memblockSize) + coreSize + s
22:            voltOUT[memidx] ⇐ Uout[i][j][k]
23:        else if nblock[(6*blkX+2+aux)] == -5 then
24:            memidx ⇐ lidx0 + coreSize + s + (NBD*bdimZ*bdimX)*(aux-(!aux))
25:            j ⇐ (NBD>j)*(NBD-j) + (NBD<j)*(j-NBD)
26:            voltOUT[memidx] ⇐ Uout[i][j][k]
27:    else if s < (nSize_z+nSize_y+nSize_x) then
28:        aux ⇐ ( s < (nSize_z+nSize_y+(NBD*bdimZ*bdimY)) )
29:        i ⇐ ((bdimX-NBD)*(s<(nSize_z+nSize_y+(NBD*bdimZ*bdimY))))+((s-nSize_z-nSize_y-
   (NBD*bdimZ*bdimY)*(s ≥ (nSize_z+nSize_y+(NBD*bdimZ*bdimY))))/(bdimZ*bdimY))
30:        k ⇐ (((s-nSize_z-nSize_y)%(bdimZ*bdimY))/bdimY)
31:        j ⇐ s-nSize_z-nSize_y - ((s-nSize_z-nSize_y)/(bdimZ*bdimY))*(bdimZ*bdimY) - k*bdimY
32:        if nblock[(6*blkX+4+aux)] > -1 then
33:            memidx ⇐ (nblock[(6*blkX+4+aux)]*memblockSize) + coreSize + s
34:            voltOUT[memidx] ⇐ Uout[i][j][k]
35:        else if nblock[(6*blkX+4+aux)] ⇐ -5 then
36:            memidx ⇐ lidx0 + coreSize + s + (NBD*bdimZ*bdimY)*(aux-(!aux))
37:            i ⇐ (NBD>i)*(NBD-i) + (NBD<i)*(i-NBD)
38:            voltOUT[memidx] ⇐ Uout[i][j][k]
39:    s ⇐ s + coreSize
```

dados armazenados em cada área de memória extra.

A cada passo de tempo numérico calculado, cada GPU copia suas bordas em $Z$ para a memória RAM da CPU. Em seguida, cada CPU copia os dados para os respectivos subdomínios vizinhos em outras GPUs. Considere, por exemplo, o $k$-ésimo subdomínio, atribuído a GPU de número $k$. Ao final do cálculo do $n$-ésimo passo de tempo a GPU de número $k$ cópia as bordas superior e inferior do $k$-ésimo subdomínio para a memória RAM da CPU. Antes do início da computação do passo de tempo $n + 1$ começar, a CPU copia

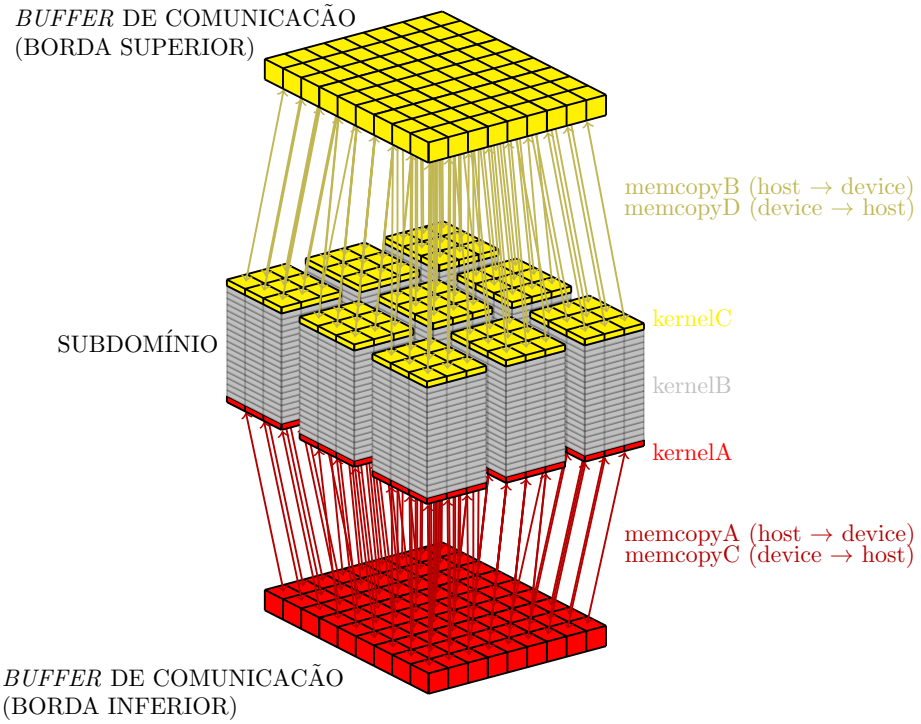



Figura 4.19: *Buffers* de memória usados para comunicar as bordas $Z$ dos subdomínios entre diferentes GPUs. No lado direito da figura encontra-se a esquema usado ne estratégia multi-gpus/multi-streams.

a borda superior do $k$-ésimo subdomínio para o *buffer* de comunicação correspondente na GPU $k+1$, e a borda inferior do $k$-ésimo subdomínio para o *buffer* de comunicação correspondente na GPU $k-1$. A Figura 4.20 ilustra o padrão de comunicação da solução multi-GPUs proposta. Os retângulos cinzas representam os subdomínios; as barras coloridas dentro de subdomínios representam suas bordas em $Z$; as barras coloridas fora dos subdomínios representam suas vizinhanças em $Z$; as setas coloridas representam as cópias de memória.

Em aplicações multi-GPUs, a cópia de memória CPU↔GPU pode ser tornar um gargalo. Contudo, o uso de `streams` pode ser usado para criar concorrência entre processamento e cópia de memória, reduzindo a latência da cópia de memória [44]. Na abordagem proposta, a cada cálculo de um passo de tempo, copiamos dados da memória da CPU para GPU (*buffers* de comunicação), executamos um único kernel para calcular a aproximação numérica para o passo de tempo e depois copiamos os dados da memória de GPU para CPU. Nesse caso, a computação e a comunicação são executadas sequencialmente e o desempenho fica limitado pela cópia de memória entre CPU e GPU, que representa a etapa mais lenta.

Com isso em mente, propomos o uso de três `streams`: um para calcular a parte

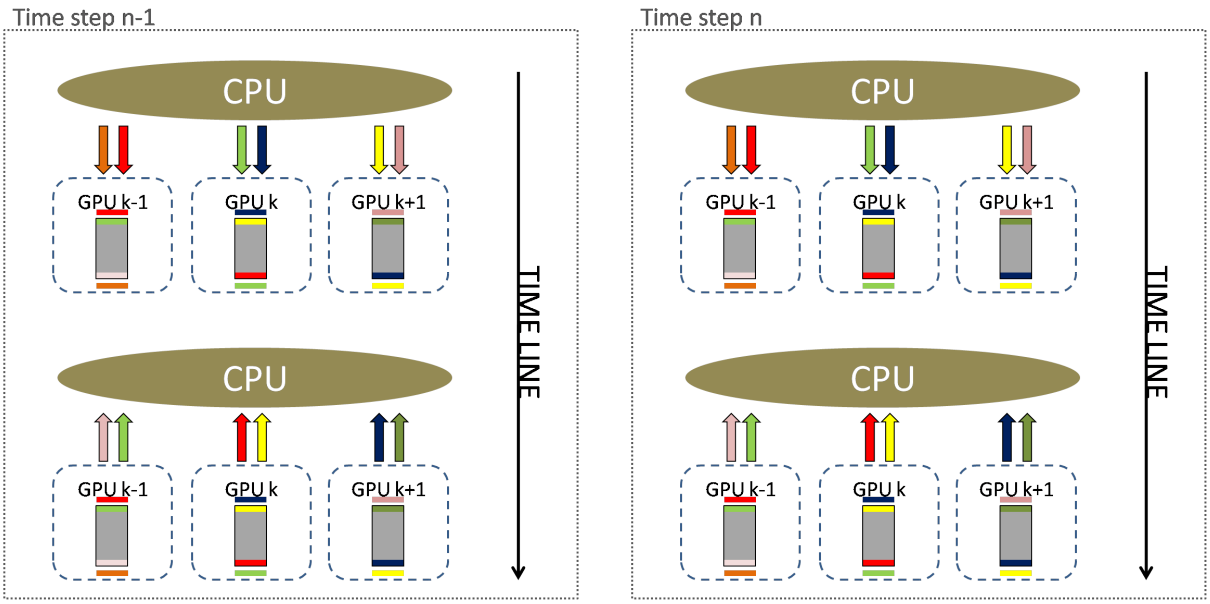



Figura 4.20: Estratégia de comunicação para três GPUs com índices sequenciais. A figura ilustra o processo para o cálculo de dois passos também sequenciais.

superior do subdomínio (regiões em amarelo na Figura 4.19) e comunicá-la, um para calcular a parte inferior do subdomínio (regiões em vermelho na Figura 4.19) e comunicá-la e o terceiro para calcular o restante do subdomínio (regiões em cinza na Figura 4.19). Essa abordagem tenta sobrepor as etapas e minimizar o tempo necessário para calcular um certo número de passos de tempo. Esta nova abordagem consiste em dividir o processamento de um passo de tempo em três kernels: um para o limite inferior do subdomínio (`kernelA`); o segundo para o limite superior do subdomínio (`kernelC`); o terceiro para todas as demais camadas $Z$ (`kernelB`). A Figura 4.19 ilustra a região de atuação de cada `kernel`. Já a etapa de cópia de memória é composta por quatro tarefas: as tarefas `memcopyA` e `memcopyC` copiam o *buffer* de comunicação inferior da CPU para a GPU (`memcopyA`) e da GPU para a CPU (`memcopyC`); as tarefas `memcopyB` e `memcopyD` copiam o *buffer* de comunicação superior da CPU para a GPU (`memcopyB`) e da GPU para a CPU (`memcopyD`).

A estratégia de torres para ambientes mutli-GPU permite simulações da eltrofisiologia cardíaca em grades espaciais mais refinadas. Contudo, a comunicação de vizinhanças pode ser um limitante. As estratégias propostas nesta seção podem auxiliar a reduzir a latência de comunicação acelerando as simulações numéricas.

# Capítulo 5

# Experimentos e Resultados

Neste capítulo serão apresentados experimentos com as diferentes estratégias de paralelismo e estruturas discutidas no Capítulo 4. O desempenho das abordagens baseadas em estruturas de dados serão comparados ao da abordagem clássica de janela deslizante. Por simplicidade, daqui por diante, iremos nos referir as quatro diferentes abordagens como:

- JDC (janela deslizante clássica, Seção 4.2);
- towerDS (estrutura de dados em torre, Seção 4.3);
- multi-tower (estrutura de dados em torre para ambientes multi-GPU, Seção 4.5);
- blockDS (estrutura de dados adaptativa, Seção 4.4).

Em todas as simulações e experimentos aqui descritos foi utilizado um domínio espacial discreto com passo de tempo $dt$ igual a $0.05ms$. O domínio espacial discreto varia com o experimento, contudo, a resolução é a mesma em todas as direções com $dx = dy = dz$.

## 5.1 Ambiente computacional

Os experimentos conduzidos para avaliar as diferentes estratégias paralelas abordadas nesta pesquisa consistem no cálculo de um conjunto de passos de tempo do MDF. Dadas as características do *hardware* da GPU[1], não é possível calcular todos os passos de tempo numéricos como uma única execução de um `kernel`. Portanto, nas implementações avaliadas aqui, cada passo de tempo é calculado, iterativamente, com a execução de um `kernel`.

[1]Não há comunicação entre blocos durante a execução de um `kernel`

Os experimentos computacionais foram conduzidos em dois ambientes com uma única GPU e um ambiente com oito GPUs. As GPUs utilizadas foram: uma GTX Titan X (arquitetura Maxwell), uma GTX 1080 Ti (arquitetura Pascal) e uma Tesla P100 (arquitetura Pascal)[2]. As configurações principais do *hardware* das máquinas que hospedam cada GPU podem ser encontradas na Tabela 5.1. Os experimentos envolvendo múltiplas GPUs foram condizidos em um cluster DGX-1 da Nvidia.

Tabela 5.1: CPUs and GPUs characteristics.

| CPU | CPU cores | host RAM memory | GPU | GPU cores | GPU RAM memory |
|---|---|---|---|---|---|
| Intel Core i7-4790 (3.60GHz) | 4 | 32 GB | Geforce GTX Titan X | 3072 | 12 GB |
| Intel Xeon E5-2620 (2.1GHz) | 8 | 32 GB | Geforce GTX 1080 Ti | 3584 | 12 GB |
| 2x Intel Xeon E5-2698 (2.2GHz) | $2 \times 20$ | 512 GB | $8 \times$ Tesla P100 | $8 \times 3584$ | $8 \times 16$ GB |

A implementações para os experimentos com uma única GPU foram compiladas com o CUDA na versão 10.0, enquanto os experimentos na DGX foram compilados com o CUDA versão 9.0.176. Os resultados preliminares não apontaram variação de desempenho e resultados com a variação da versão de CUDA. O `nvprof`, foi usado como ferramenta de *profile* do código e para coletar os tempos de execução dos `kernels` de cada implementação.

## 5.2 Avaliação de desempenho da towerDS usando o estêncil de diferenças centradas

A primeira avaliação de desempenho da estrutura towerDS foi baseada na computação do estêncil de diferenças centradas (Equação 3.1). As Equações 5.1 e 5.2 representam aproximações, respectivamente, de $2^a$ e $4^a$ ordens, e seus coeficientes foram adaptados do modelo Karma [26, 15].

[2]Todas as linhas de GPUs da Nvidia fornecem suporte a computação de propósito geral. Contudo, a linha de GPUs GeForce GTX está mais voltada para tarefas de computação gráfica como a execução de jogos. Já a linha Tesla está direcionada para tarefas de processamento de alto desempenho. Segundo dados fornecidos pela Nvidia , para valores com precisão simples uma GTX 1080 Ti atinge um valor de pico inferior a 0.18 Tflops, enquanto uma Tesla P100 pode chegar a 21.2 Tflops.

$$\begin{aligned} U^{t+1}_{i,j,k} &= U^{t}_{i,j,k} + \big[0.02\left(U^{t}_{i,j,k+1} + U^{t}_{i,j,k-1} + U^{t}_{i,j+1,k} + \right. \\ &\quad \left. U^{t}_{i,j-1,k} + U^{t}_{i+1,j,k} + U^{t}_{i-1,j,k} - 6U^{t}_{i,j,k}\right)\big] \end{aligned} \tag{5.1}$$

$$\begin{aligned} U^{t+1}_{i,j,k} &= U^{t}_{i,j,k} + \big\{0.002\left[16\left(U^{t}_{i,j,k+1} + U^{t}_{i,j,k-1} + U^{t}_{i,j+1,k} + \right.\right. \\ &\quad \left. U^{t}_{i,j-1,k} + U^{t}_{i+1,j,k} + U^{t}_{i-1,j,k}\right) - \left(U^{t}_{i,j,k+2} + U^{t}_{i,j,k-2} + U^{t}_{i,j+2,k}\right. \\ &\quad \left.\left. + U^{t}_{i,j-2,k} + U^{t}_{i+2,j,k} + U^{t}_{i-2,j,k} + 90U^{t}_{i,j,k}\right)\big]\big\} \end{aligned} \tag{5.2}$$

As Figuras 5.1 e 5.2 mostram como o tempo médio de execução do `kernel` varia com o tamanho dos blocos utilizados para representar um domínio espacial 3D com $512^3$ pontos, ou células. A parte superior das figuras mostram o tempo em milisegundos, enquanto a parte inferior mostra todos os tempos em relação ao menor:

$$\text{tempo relativo} = \frac{\text{tempo médio de execução do } \texttt{kernel}}{\textbf{menor} \text{ tempo médio de execução do } \texttt{kernel}}. \tag{5.3}$$

Observa-se que o tempo de execução do `kernel` não está ligado somente ao número de threads, mas a distribuição dessas threads nas direções $X$ e $Y$. No caso da JDC, vemos que causa-se uma forte degradação no desempenho ao aumentarmos o número de threads na direção $Y$, mantendo baixo o número de threads na direção $X$. Isso ocorre porque ao aumentarmos o número de threads no bloco, aumentamos também a demanda de cada bloco por memória compartilhada, limitando assim o número de blocos executados de forma concorrente na GPU. Contudo, ao aumentarmos o número de threads na direção $X$ o tempo de execução do kernel cai. Essa queda no tempo de execução é causada pela redução no número de halos de vizinhanças entre blocos na direção $X$ (mais threads por bloco em $X$ menos blocos são necessários para representar o domínio). Isso mostra que a degradação de desempenho causada pelos acessos aos halos em $X$ é maior que a causada pela redução no número de blocos executados de forma concorrente. A Tabela 5.2 mostra os melhores resultados da cada uma das abordagens para os dois estênceis testados. Os resultados demonstram uma redução considerável no tempo de execução ao usar o towerDS, sendo esta abordagem $1,23$ mais rápida que o JDC no caso do estêncil de $2^a$ ordem e $2,92$ no caso do estêncil de $4^a$ ordem.

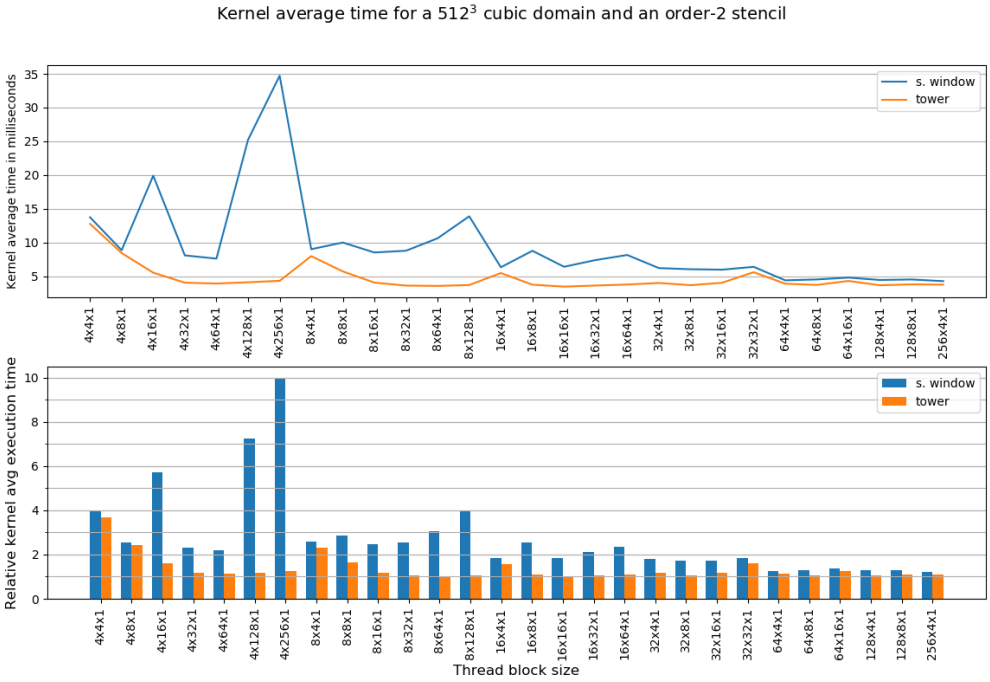



Figura 5.1: Tempo médio de execução para calcular um passo de tempo para um estêncil de ordem 2.

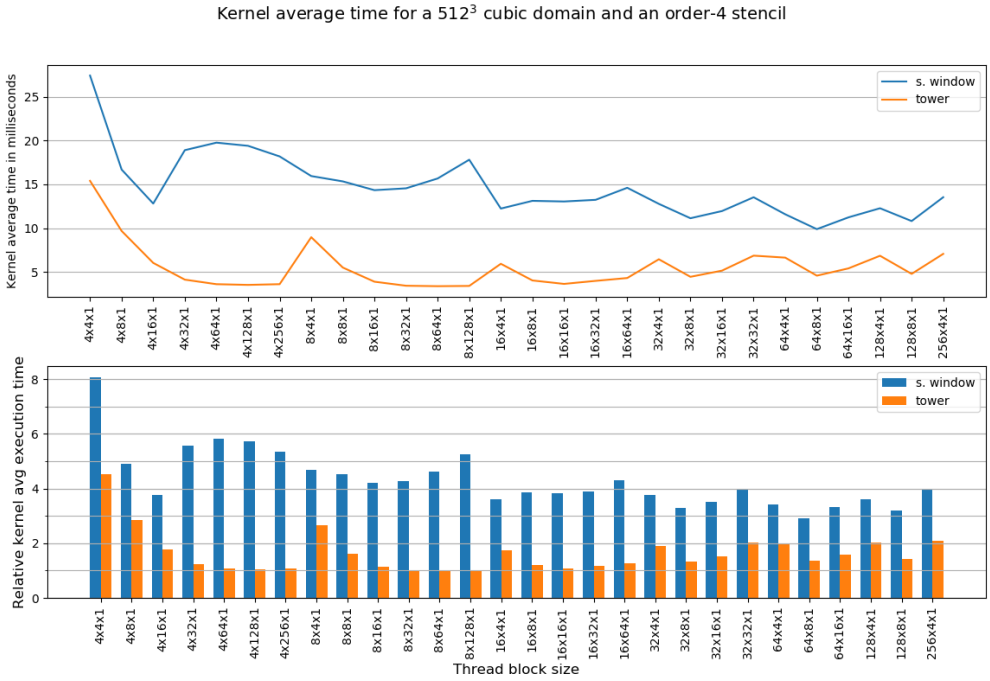



Figura 5.2: Tempo médio de execução para calcular um passo de tempo para um estêncil de ordem 4.

## 5.3 Simulações com o modelo Karma de 2 variáveis

Na segunda etapa de experimentos, as abordagens JDC e towerDS foram avaliadas quanto ao seu desempenho em simulações da eletrofisiologia cardíaca usando o modelo Karma

| estêncil | abordagem | tamanho do bloco | tempo (ms) |
|---|---|---|---|
| $2^a$ ordem | JDC | $256 \times 4 \times 1$ | 4.288 |
| | towerDS | $16 \times 16 \times 1$ | 3.474 |
| $4^a$ ordem | JDC | $64 \times 8 \times 1$ | 9.897 |
| | towerDS | $8 \times 64 \times 1$ | 3.394 |

Tabela 5.2: Configurações de bloco que resultaram nos menores tempos de execução para o `kernel`.

(Seção 2.2). Neste conjunto de experimentos foram avaliados dois tamanhos de domínios cúbicos, $256^3$ e $512^3$, e as três GPUs disponíveis no ambiente computacional dos testes.

## 5.3.1 Condições iniciais e de contorno

### 5.3.1.1 Condição inicial

A condição inicial nas simulações com o Karma são dadas pelas Equações 5.4 e 5.5. O potencial de repouso das células é dados por $U^0_{ijk} = 0,0$ enquanto o estímulo é provocado pela elevação do potencial para $U^0_{ijk} = 3,0$ em 20% das camadas $Z$.

$$v^0_{ijk} = 0,5 \quad \forall z_k \tag{5.4}$$

$$U^0_{ijk} = \begin{cases} 3,0 & \text{para } z_k \leq 0.05 \text{ (tamanho do domínio em z)} \\ 0,0 & \text{caso contrário} \end{cases} \tag{5.5}$$

### 5.3.1.2 Condições de contorno

As condições de contorno aplicadas as simulação com o modelo Karma são conhecidas como condições de Neumann de fluxo nulo. A condição de contorno é definida pela criação de "células fantasmas" cujo objetivo é criar um "refelxo" da onda. A Figura 5.3 ilustra essa "reflexão" para um pequeno domínio 2D, par o qual a vizinhança interna de cada borda é usada como a vizinhança exterior ao domínio. O objetivo das células fantasmas é exinguir a propagação do pulso nas bordas do domínio. A Figura 5.4 ilustra a interação entre dois pulsos e a extinção dos mesmos na sua interação. A figura também mostra o comportamento do estímulo quando atinge as bordas. As condições de contorno reproduzem o efeito da interação entre a propagação de dois estímulos.

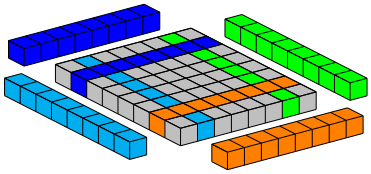


Figura 5.3: Esquema de células refletidas aplicada como condições de contorno a um pequeno domínio 2D.

### 5.3.2 Avaliação de desempenho com uma única GPU

O desempenho das abordagens JDC e towerDS foi avaliado para dois domínios espaciais 3D com resoluções diferentes, um com $256^3$ células e outro com $512^3$. A Tabela 5.3 mostra o número de células armazenadas por variável para cada um dos domínios, bem como a quantidade de memória ocupada na memória global da GPU. Ambas as variáveis do

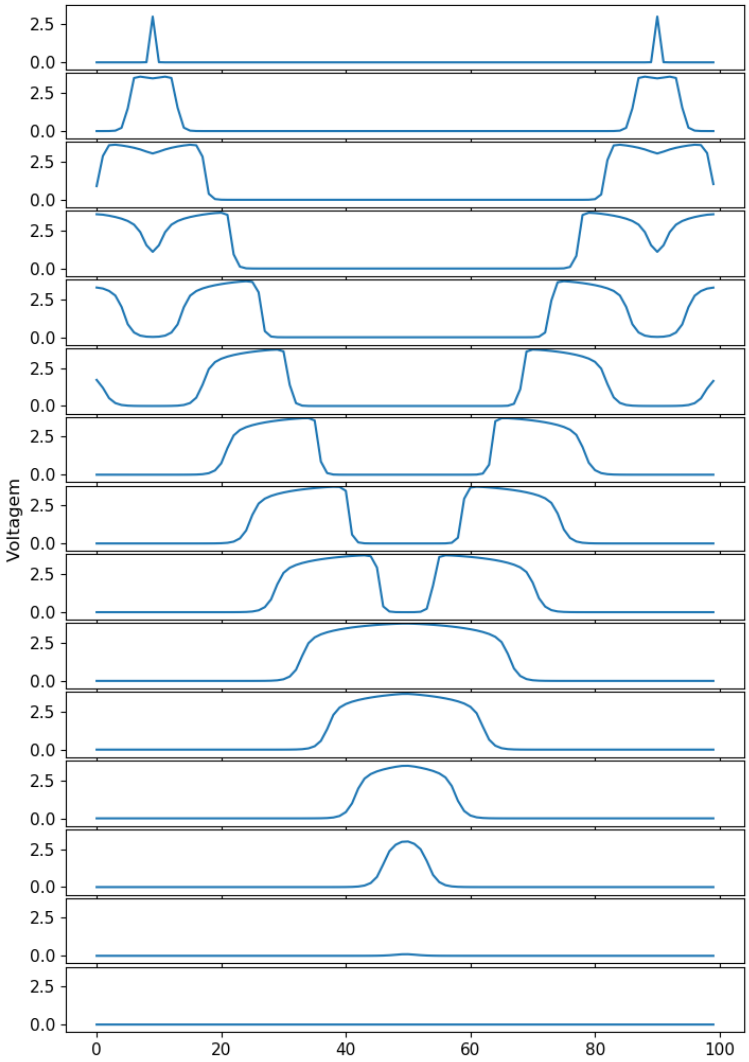



Figura 5.4: Simulação 1D da propagação de dois estímulos.

modelo Karma são inicializadas diretamente na memória global da GPU com valores de repouso pré-definidos. O espaço de memória (em bytes) ocupado pela estrutura de dados towerDS pode ser calculado, para uma simulação com precisão simples, pela equação:

$$mem = 4 * (2 * ds\_size + mesh\_size) \quad . \tag{5.6}$$

onde $ds\_size$ é o número de células na estrutura e $mesh\_size$ o número de células no domínio espacial discreto. $ds\_size$ é dependente do tamanho das torres usadas na estrutura para representar o domínio espacial. Por exemplo, para ajustar um domínio 3D com $64^3$ células, podemos utilizar 2×8×1 torres com 32×8×64 células cada uma, resultando em $ds\_size = 278528$ células. Por outro lado, no caso do JDC, o número de células armazenadas por variável será sempre igual ao número de células no domínio.

| domínio | número de células | | espaço na memória | |
|---|---|---|---|---|
| | JDC | towerDS | JDC | towerDS |
| $256^3$ | $256^3$ | 17.825.792 | 192 | 200 |
| $256^3$ | $512^3$ | 142.606.336 | 1536 | 1600 |

Tabela 5.3: Espaço de memória necessário para armazenar as variáveis do modelo Karma. O espaço de memória é apresentado em Mb.

As Figuras 5.5, 5.6 e 5.7 mostram como o tempo total de processamento na GPU se comporta com relação ao tamanho do bloco de threads usado com o `kernel`. Cada figura mostra tal comportamento para cada uma das GPUs em nosso ambiente computacional (veja Seção 5.1). Os resultados mostraram que aumentar o tamanho do bloco em $X$ causa uma redução significativa no tempo de processamento geral da GPU. Este comportamento está relacionado a redução no número de halos de vizinhança entre blocos na direção $X$ e, consequentemente, em nossa principal causa de divergência de código e acessos não coalescentes à memória global (Seção 4.2). Por outro lado, aumentar o número de threads em $Y$ causa aumento no tempo de execução devido a redução no número de blocos ativos por SM, como discutido na Seção 5.2.

A Tabela 5.4 apresenta os arranjos de blocos e threads com os quais foi obtido o melhor desempenho em ambas as abordagens, towerDS e JDC. Em geral, os menores tempos de execução foram obtidos com as threads organizadas em blocos de tamanho $64 \times 4 \times 1$. No caso da P100, o arranjo de blocos que apresentou o melhor desempenho depende do tamanho do domínio e da estratégia de implementação utilizada. Nenhuma das métrica avaliadas com o profile da Nvidia  apontou uma possível razão para o resultado diferente

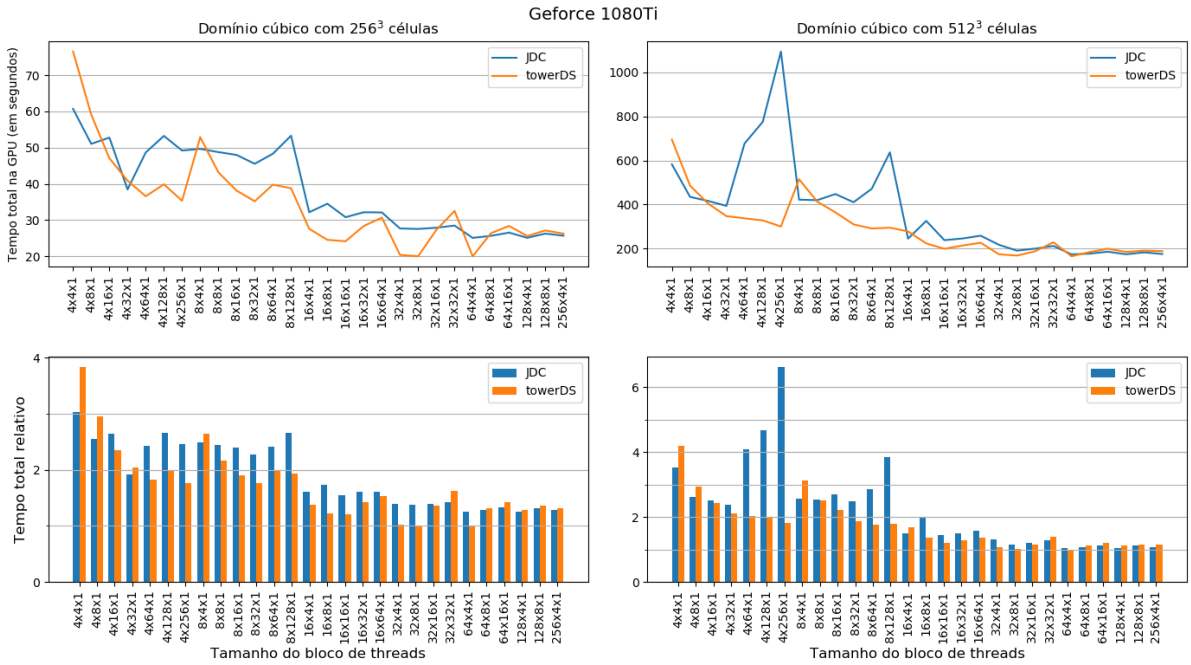



Figura 5.5: Tempo de processamento em função do tamanho do bloco. Neste experimento foram calculados 20.000 passos de tempo na GPU GTX 1080 Ti.

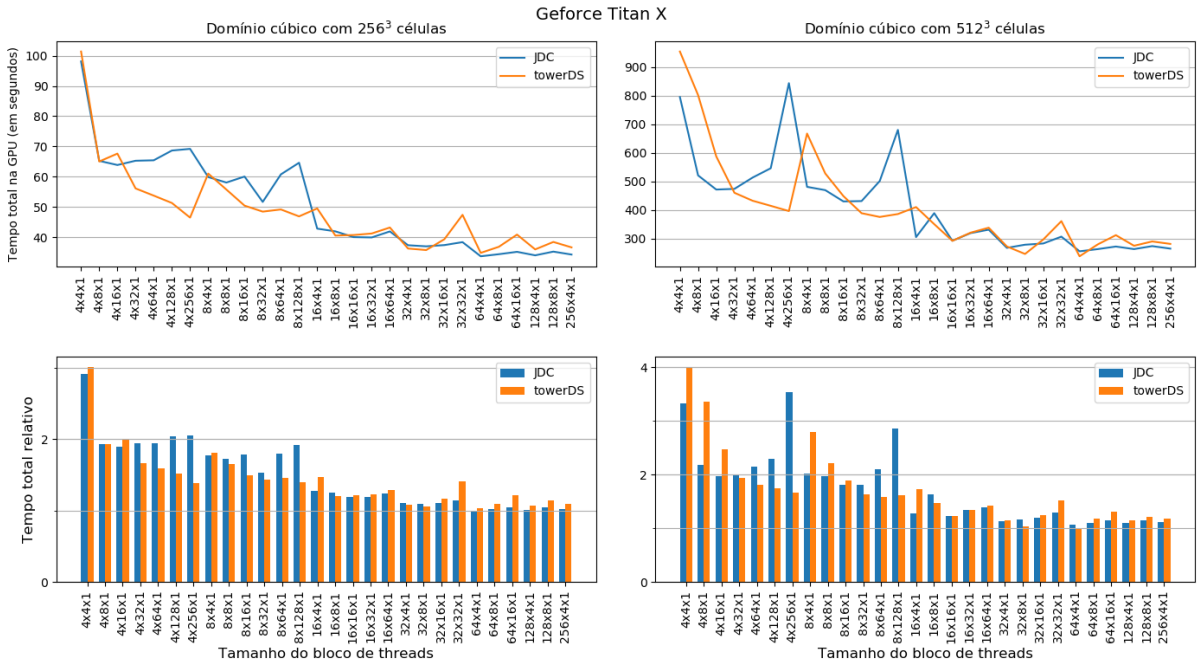



Figura 5.6: Tempo de processamento em função do tamanho do bloco. Neste experimento foram calculados 20.000 passos de tempo na GPU GTX Titan X.

na P100.

Quando os tempos de execução da JDC com a towerDS são comparados, towerDS consegue executar o mesmo processamento em tempos 40.71% menores para um domínio

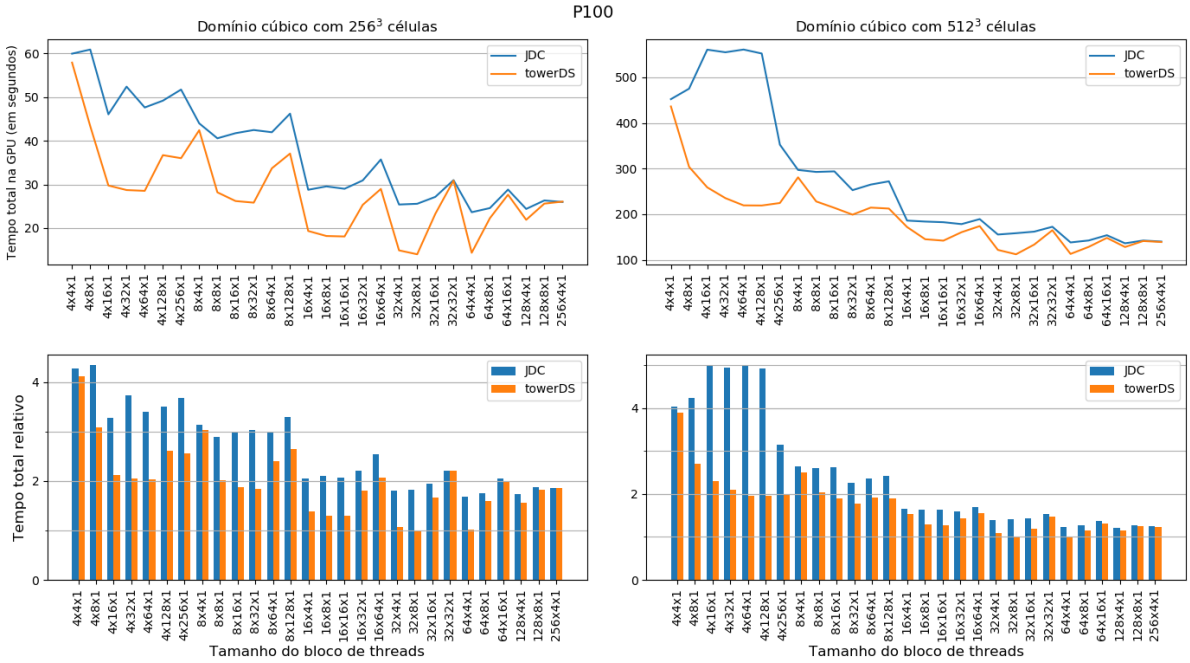



Figura 5.7: Tempo de processamento em função do tamanho do bloco. Neste experimento foram calculados 20.000 passos de tempo na GPU Tesla P100.

| domínio | GPU | JDC | | towerDS | |
|---|---|---|---|---|---|
| | | tempo (s) | bloco | tempo (s) | bloco |
| $256^3$ | 1080Ti | 25.0849 | $64 \times 4 \times 1$ | 20.0084 | $64 \times 4 \times 1$ |
| | Titan X | 33.7281 | $64 \times 4 \times 1$ | 34.8525 | $64 \times 4 \times 1$ |
| | P100 | 23.6460 | $64 \times 4 \times 1$ | 14.0196 | $32 \times 8 \times 1$ |
| $512^3$ | 1080Ti | 174.3350 | $64 \times 4 \times 1$ | 165.5145 | $64 \times 4 \times 1$ |
| | Titan X | 255.8197 | $64 \times 4 \times 1$ | 238.4309 | $64 \times 4 \times 1$ |
| | P100 | 136.3548 | $128 \times 4 \times 1$ | 112.2918 | $32 \times 8 \times 1$ |

Tabela 5.4: Configurações de blocos com menores tempos de processamento ao calcular $20,000$ passos de tempo. Os tempos de processamento são dados em segundos.

com $256^3$ células processado com uma Tesla P100. Para um domínio de tamanho $512^3$ a redução no tempo de processamento chega a $17,65\%$ também usando uma Tesla P100.

### 5.3.3 Experimentos com múltiplas GPUs

Os experimentos multi-GPU foram realizados em uma NVIDIA DGX-1. Este cluster de GPU é composto por oito GPUs NVIDIA Tesla P100, cada uma com núcleos 3584 CUDA. Os resultados obtidos com uma implementação direta da abordagem multi-GPU, proposta na Seção 4.5, mostram que quando aumentamos o número de GPUs, o tempo total também aumenta (linhas azuis e vermelhas na Figura 5.8). Em um ambiente com múltiplas GPUs precisamos dividir o problema e distribuí-lo para ser resolvido entre as GPUs disponíveis.

No caso das simulações cardíacas tratadas aqui, a dependência da vizinhança no cálculo da solução numérica impõe a necessidade de comunicação entre as GPUs ao final do cálculo de cada passo de tempo (vide Seção 4.5). Portanto, o incremento no número de GPUs usadas implica em um maior número de divisões do domínio e, consequentemente, em um maior tráfego de informações ao final do cálculo de cada passo de tempo, impactando o tempo de execução da simulação.

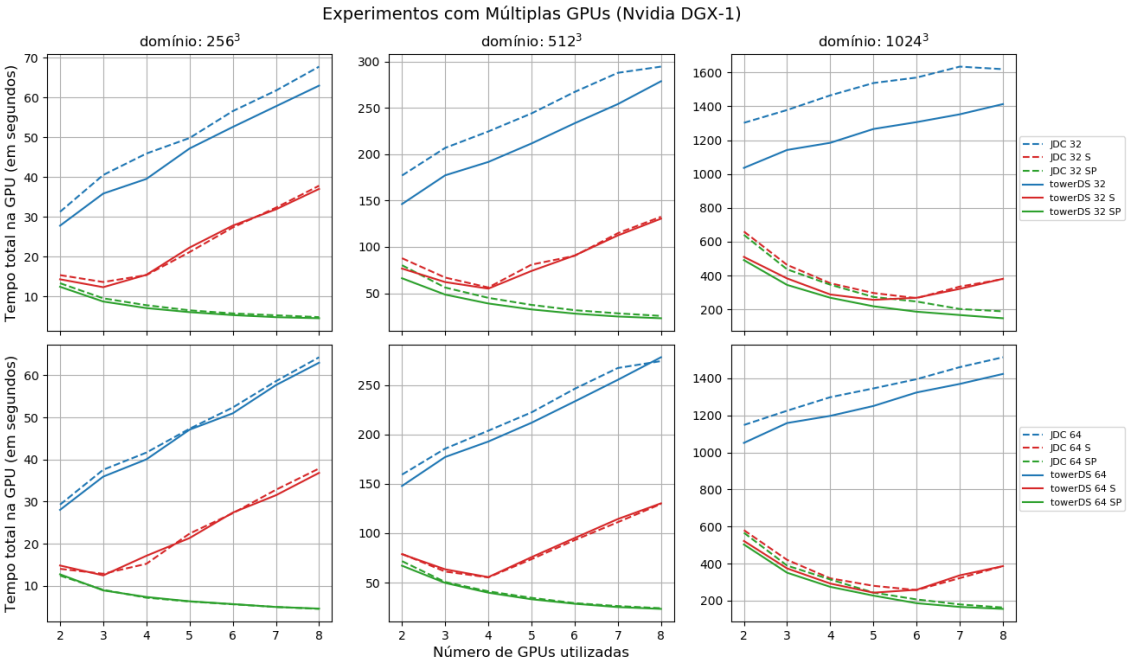



Figura 5.8: Desempenho das diferentes estratégias multi-GPUs. A letra S nas legendas identificam as estratégias com CUDA `streams`. As letras SP identificam o uso de `streams` + memória paginada (*page-lock memory*).

Para melhorar o desempenho da implementação com múltiplas GPUs foram usados dois recursos do CUDA, os `streams` e a memória paginada. Com o uso dos `streams` pode-se criar diferentes fluxos de execução na GPU, possibilitando sobrepor computação e cópia de memória entre a CPU e a GPU (vide Seção 4.5). Também foi avaliado o uso combinado dos *streams* com o uso de memória paginada alocada no `HOST`. O uso da memória paginada possibilita cópias de memória mais velozes entre a CPU e a GPU[44], porque garante que os dados não serão paginados para o disco, e estarão disponíveis na memória física a qualquer momento [44].

A Figura 5.8 apresenta os resultados obtidos com a abordagem que utiliza `streams`, e também a escalabilidade com múltiplas GPUs. Em nossos experimentos foram calculados 20.000 passos de tempo para três domínios com tamanhos diferentes: $256^3$, $512^3$ e $1024^3$. Avaliamos o comportamento de três abordagens usando nossa estratégia de cópia

de memória: uma padrão, com apenas um único `stream`; uma usando a estratégia com três `streams` para sobrepor a computação e a cópia da memória (identificada pela letra S na legenda da Figura 5.8); uma usando a estratégia com `streams` e o uso de memória paginada (*page-lock memory* ou *pinned memory*, identificada por letras SP na Figura 5.8). A figura mostra resultados com duas configurações de bloco: na parte de cima da figura (primeira linha de gráficos) são exibidos os resultados dos experimentos com blocos de tamanho $32 \times 8 \times 1$, com os quais obteve-se o tempo de execução mais baixo para o towerDS; na parte inferior (segunda linha) são exibidos os resultados dos experimentos com blocos de tamanho $64 \times 4 \times 1$, com os quais obteve-se o tempo de execução mais baixo para a JDC.

A escalabilidade da abordagem multi-GPU proposta é sub-linear quando combinada ao uso de memória paginada (Figura 5.8). De fato, oito GPUs obtêm um ganho de desempenho ligeiramente superior a sete, por exemplo. Por outro lado, quatro GPUs apresentaram um desempenho quase quatro vezes mais rápido que uma única GPU. Assim sendo, quatro GPUs obtiveram o melhor desempenho considerando a escalabilidade. No entanto, quanto mais GPUs maior a quantidade de memória distribuída, fator importante quando aumentamos a resolução do domínio. A escalabilidade da abordagem multi-GPU está relacionada à quantidade de dados processados por cada núcleo versos o volume de comunicação de dados. O comportamento do tempo de processamento com o número de GPUs apresentado na Figura 5.8 indica que com o aumento do número de GPUs o tempo de comunicação torna-se dominante, prejudicando o desempenho. Contudo, o uso de alocação de memória paginada, reduz consideravelmente este efeito. A Tabela 5.5 mostra os tempos obtidos com a abordagem multi-GPU mais a memória paginada e usando oito GPUs. Para um domínio com $256^3$, uma simulação de $1s$ do tempo real (20.000 passos de tempo) seria apenas quatro vezes mais lento que o tempo real. Nesse caso a JDS e a towerDS apresentam desempenhos similares, sendo o tempo médio de execução da towerDS ligeiramente superior. A análise dos resultados experimentais indicam que o tempo gasto para comunicar os *buffers* de vizinhança $Z$ entre a CPU e as GPUs é maior no caso da towerDS. Os dados das vizinhança $Z$ a serem comunicadas através dos *buffers* estão alinhados em posições adjacentes de memória no caso da JDC, mas no caso da towerDS esses dados estão dispersos devido a subdivisão do domínio em torres.

Por fim, a Figura 5.9 mostra o número de células processadas por segundo para todas as diferentes GPUs que testamos. Nós escolhemos a configuração de bloco mais rápida para cada caso. Conseguimos a mesma quantidade de células processadas por segundo para os domínio de tamanho $256^3$ e $512^3$. Como esse resultado foi alcançado com os

| domínio | JDC | | towerDS | |
|---|---|---|---|---|
| | time | block size | time | block size |
| $256^3$ | 4.4975 | $64 \times 4 \times 1$ | 4.5418 | $32 \times 8 \times 1$ |
| $512^3$ | 24.0151 | $64 \times 4 \times 1$ | 23.0818 | $32 \times 8 \times 1$ |
| $1024^3$ | 162.9126 | $64 \times 4 \times 1$ | 147.9537 | $32 \times 8 \times 1$ |

Tabela 5.5: Tempo de processamento total para calcular $20,000$ passos de tempo.

mesmos tamanhos de bloco, conclui-se que a ocupação total da GPU foi atingida em ambos os casos. Por outro lado, nossa abordagem multi-GPU funcionou melhor para malhas maiores, sendo mais eficiente para a malha de $1024^3$. A Tabela 5.6 mostra o número de passos de tempo calculados por segundo em nosso cenários mais rápidos, com uma ou mais GPUs.

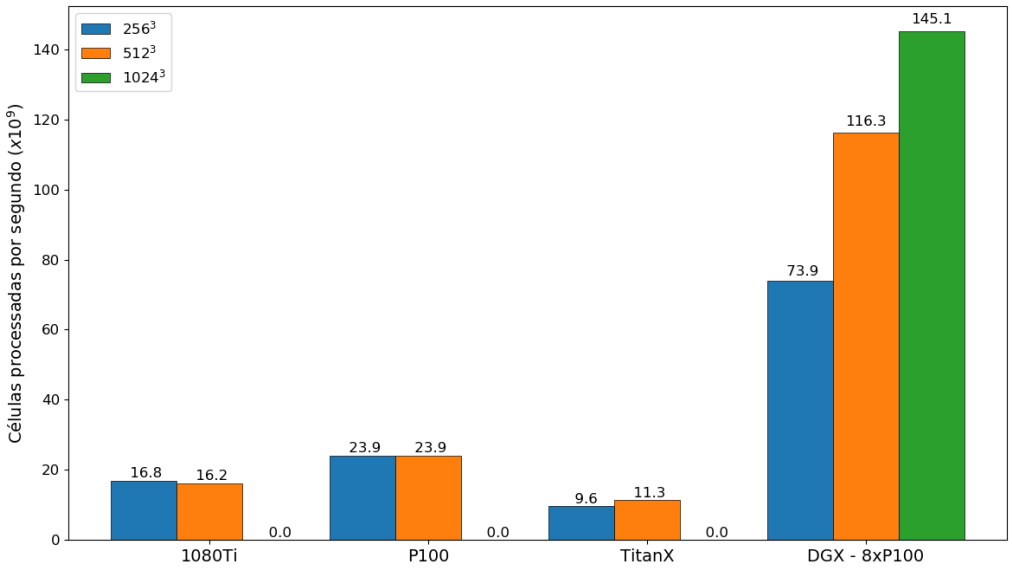



Figura 5.9: Número médio de células processadas por segundo com nossas configurações de bloco mais velozes.

| GPU | $256^3$ | $512^3$ | $1024^3$ |
|---|---|---|---|
| P100 | 1426 | 178 | — |
| DGX - 8xP100 | 4405 | 866 | 135 |

Tabela 5.6: Número de passos de tempo calculados por segundo para os cenários mais rápidos em nossos experimentos com uma e múltiplas GPUs.

### 5.3.4 Coerência entre a simulações

As abordagens JDC e towerDS foram usadas para executar simulações com o modelo Karma. Desta forma, pode-se avaliar a coerência entre os resultados e verificar a introdução de variabilidade nos resultados pelo uso da estrutura de dados towerDS.

A Figura 5.10 mostra o valor do potencial de membrana para uma simulação com as condições iniciais e de contorno descritas na Seção 5.3.1. Os valores apresentados foram coletados para pontos com diferentes valores de $z$, mas com o mesmo par ordenado $(x, y)$. Foram armazenadas 2.000 amostras coletadas a cada 10 passos de tempo durante a simulação de $1s$ do tempo real (20.000 passos de tempo). Verificamos que os dados são consistentes e não há diferença entre eles.

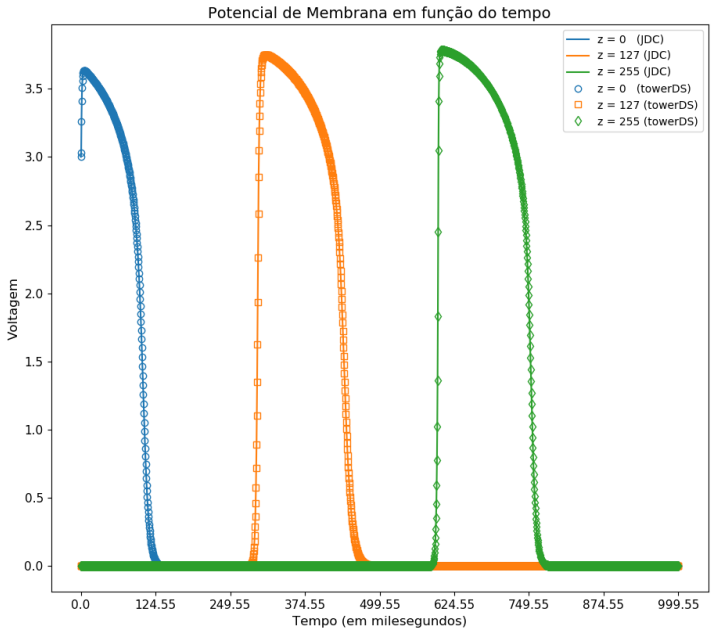



Figura 5.10: Propagação do estímulo elétrico na direção $Z$.

A Figura 5.10 mostra uma simulação feita com a implementação towerDS do modelo Karma para um domínio cúbico com $256^3$ células. Os frames são espaçados por 25 milisegundos. Para avaliar a capacidade da implementação reproduzir uma onda espiral, foi introduzido um estímulo da forma:

$$\mathrm{U}^{8,500}_{x,y,z} = \begin{cases} 3.0f & \text{se} \quad 60 < z < 100 \quad \text{e} \quad 117 \leq x \leq 137 \\ \mathrm{U}^{8,500}_{x,y,z} & \text{caso contrário} \end{cases} \tag{5.7}$$

A implementação do Karma, usando a abordagem towerDS, é consistente e capaz de reproduzir características relevantes do tecido cardíaco como as ondas espirais, comumente associadas a problemas de arritmias cardíacas. Ela também possibilita rodar simulações em tempos menores que a abordagem clássica JDC, principalmente para domínios maiores

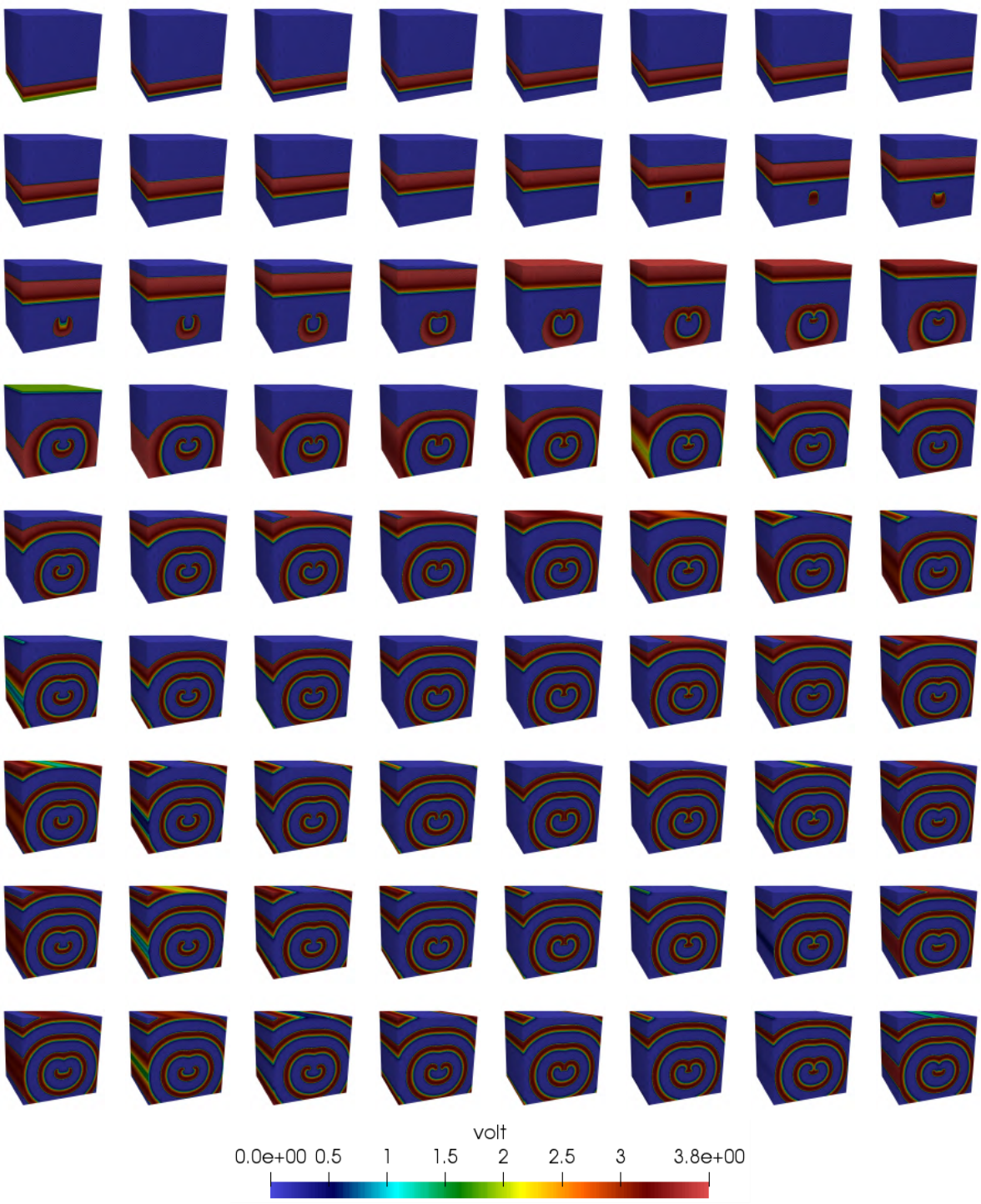



Figura 5.11: Onda espiral simulada com a implementação towerDS. O intervalo de tempo entre as imagens é de $0,025$ segundos.

que $256^3$. Além disso, para domínios com $256^3$ células, as implementações multi-GPUs do Karma permitem simular 1 segundo de atividade elétrica do coração com apenas 4 segundos de simulação (4× mais lento).

## 5.4 Simulações com o modelo OVVR de 41 variáveis

Nesta seção iremos avaliar o desempenho de nossas implementações paralelas para o modelo OVVR. Como esse modelo demanda uma quantidade de processamento consideravelmente superior ao Karma, o foco dos experimentos é avaliar o comportamento da implementação paralela usando a estrutura de dados adaptativa blockDS.

### 5.4.1 Condições iniciais e de contorno

#### 5.4.1.1 Condição inicial

A condição inicial nas simulações com o OVVR foi adaptada de [36]. O potencial de repouso das células é dados por $U^0_{ijk} = -87,4$ enquanto o estímulo é provocado pela elevação do potencial para $U^0_{ijk} = 0,0$ em 20% das camadas $Z$. As demais variáveis são inicializadas com os valores do artigo original.

$$U^0_{ijk} = \begin{cases} 87,4 & \text{para } z_k \leq 0.05 \text{ e } \phi_{i,j,k} > 0.0005 \\ 00,0 & \text{caso contrário} \end{cases} \tag{5.8}$$

onde $z_k$ representa o número de células do domínio na direção $Z$.

#### 5.4.1.2 Condições de contorno

As condições de contorno adotadas nas bordas do domínio foram as mesmas usadas com o modelo Karma (Seção 5.3.1). Contudo, o uso do campo de fase $\phi$ que define a estrutura do coração do porco impõe novas condições de contorno a serem aplicadas às simulações. Neste caso, a reflexão de pontos, aplicada as bordas do domínio, também foi apicada nas fronteiras definidas pelo campo de fase. Tais condições de contorno são descritas no Algoritmo 10, e são aplicadas durante o processo de computação da solução numérica do `kernel`. Desta forma, o valor do potencial de membrana nas células fora da estrutura do coração é mantido no valor de repouso, representando a condição de fluxo nulo.

**Algoritmo 10** Condições de contorno nas fronteiras do campo de fase $\phi$.

1: **if** $\phi_{i,j,k} \leq 0.0005$ e $\phi_{i-1,j,k} > 0.0005$ **then**
2: $\quad U^t_{i,j,k} = U^t_{i-2,j,k}$
3: **if** $\phi_{i,j,k} \leq 0.0005$ e $\phi_{i+1,j,k} > 0.0005$ **then**
4: $\quad U^t_{i,j,k} = U^t_{i+2,j,k}$
5: **if** $\phi_{i,j,k} \leq 0.0005$ e $\phi_{i,j-1,k} > 0.0005$ **then**
6: $\quad U^t_{i,j,k} = U^t_{i,j-2,k}$
7: **if** $\phi_{i,j,k} \leq 0.0005$ e $\phi_{i,j+1,k} > 0.0005$ **then**
8: $\quad U^t_{i,j,k} = U^t_{i,j+2,k}$
9: **if** $\phi_{i,j,k} \leq 0.0005$ e $\phi_{i,j,k-1} > 0.0005$ **then**
10: $\quad U^t_{i,j,k} = U^t_{i,j,k-2}$
11: **if** $\phi_{i,j,k} \leq 0.0005$ e $\phi_{i,j,k+1} > 0.0005$ **then**
12: $\quad U^t_{i,j,k} = U^t_{i,j,k+2}$

### 5.4.2 Tamanho do domínio

Diferente do estudo com o modelo Karma, aqui o tamanho do domínio está condicionado as dimensões do campo de fase. Neste caso o domínio espacial tem dimensões $250 \times 326 \times 298$. Enquanto a JDC e a towerDS processam todas células, a blockDS processa somente uma parte do domínio (veja Seção 4.4). A Figura 5.12 mostra, por tamanho de bloco usado, a fração de células processadas ao empregar a blockDS. A fração de células ativas definidas pelo campo de fase $\phi$, ou seja, onde efetivamente há tecido, é de aproximadamente 27% do domínio.

### 5.4.3 Avaliação de desempenho com uma única GPU

O estudo comparativo das implementações paralelas do modelo OVVR foram conduzidas nas duas GPUs com mesma architetura, a GTX 1080Ti e a Tesla P100. As Figuras 5.13 e 5.14 mostram o tempo total de processamento na GPU demandado por cada uma das implementações para o cálculo de 20.000 passos de tempo. Em ambas as GPUs a implementação usando a blockDS é mais de 6× mais rápida que as as outras versões na 1080Ti e mais de 10× mais rápida na P100. Nas partes inferiores de ambas as figuras pode-se observar, com mais clareza, o comportamento do tempo total de processamento na GPU em função do tamanho do bloco. A variação do tempo é inferior a $3s$ em ambas as GPUs. As Tabelas 5.7 e 5.8 mostram, respectivamente, o melhor resultado (menor tempo de processamento) para cada abordagem paralela para cada GPU. A Figura 5.15 mostra, para os cenários mais rápidos, o número médio de células processadas por segundo. No caso do OVVR há uma elevada demanda por registradores, limitando o número de blocos

ativos por SM e, consequentemente, degradando o desempenho da JDC e da towerDS. O efeito é ainda mais notável na towerDS (pior desempenho) devido a presença das variáveis usadas no mapeamento das threads nas posições de memória, tais variáveis não estão presentes na JDC.

| GPU | blockDS | JDC | towerDS |
|---|---|---|---|
| 1080Ti | $344,5$ | $1851,5$ | $3049,4$ |
| P100 | $264,8$ | $2681,4$ | $2821,4$ |

Tabela 5.7: Tempo de GPU necessário para processar 20.000 passos de tempo na implementação do modelo OVVR com a estrutura de dados adaptativa blockDS.

| GPU | blockrDS | JDC | towerDS |
|---|---|---|---|
| 1080Ti | 58 | 10 | 6 |
| P100 | 75 | 7 | 7 |

Tabela 5.8: Número médio de passos de tempo calculados por segundo na implementação do modelo OVVR com a estrutura de dados adaptativa blockDS.

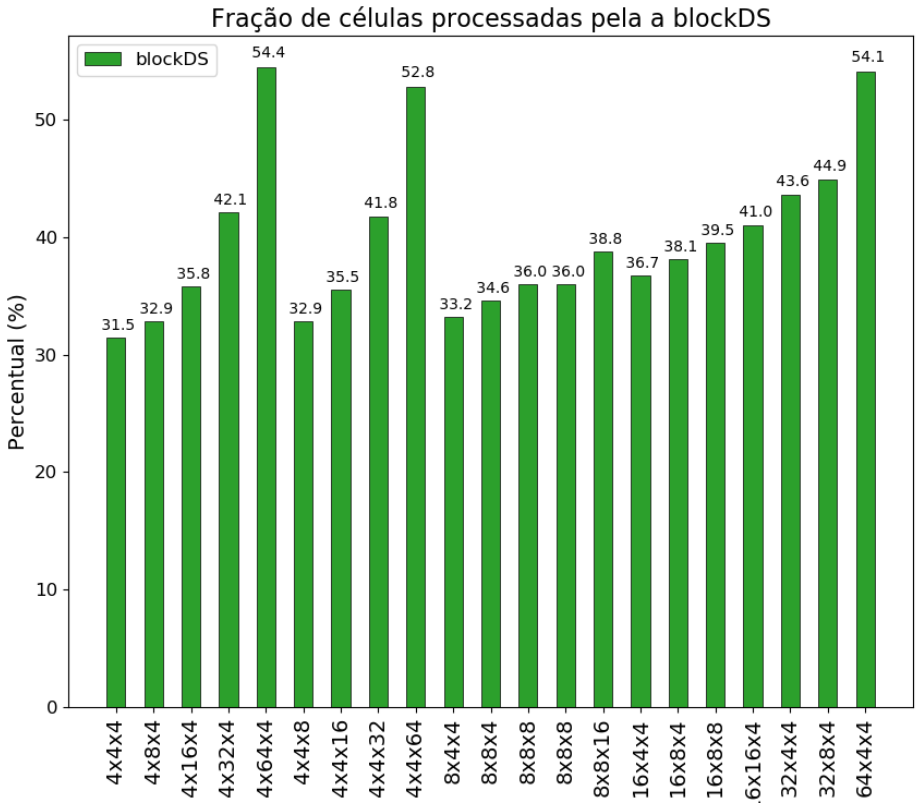



Figura 5.12: Fração de células processadas com a blockDS.

### 5.4.4 Simulação 2D

Para avaliar o comportamento da simulação numérica com a blockDS, executamos um experimento 2D usando uma camada do campo de fase. A Figura 5.16 mostra oito *frames* de uma simulação de $600ms$ de atividade elétrica. Pode-se observar a propagação coerente do estímulo através da área do tecido, respeitando os espaços vazios. A título de comparação, são exibidos, lado a lado, os resultados da simulação com a blockDS e com a JDC. A terceira coluna da figura mostra a diferença nos valores do potencial de membrana entre as duas simulações.

A abordagem paralela com a estrutura de dados adaptativa blockDS é consideravel-

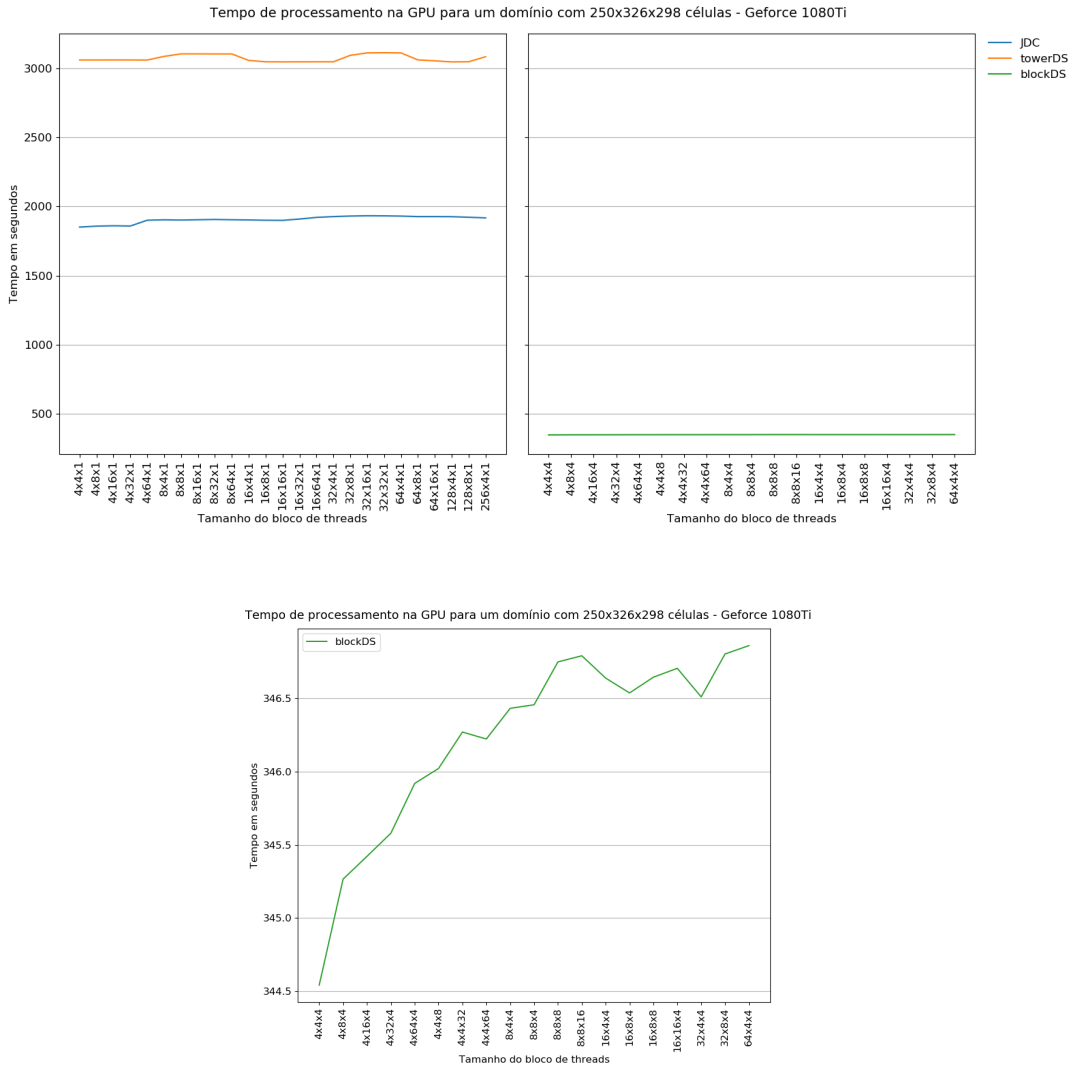



Figura 5.13: Tempo de processamento em função do tamanho do bloco. Neste experimento foram calculados 20.000 passos de tempo na GPU GTX 1080 Ti. O gráfico de baixo foca na blockDS.


Figura 5.14: Tempo de processamento em função do tamanho do bloco. Neste experimento foram calculados 20.000 passos de tempo na GPU Tesla P100. O gráfico de baixo foca na blockDS.

mente mais rápida que as demais abordagens avaliadas. Além disso, o experimento 2D mostrou que a redução do domínio feita pela blockDS não tem efeitos significativos nos resultados numéricos, permanecendo a simulação coerente com uma simulação com o domínio completo (caso da JDC e da towerDs). Existem diferenças entre as frentes de onda que podem ser observadas nas segunda, terceira e quarta linhas da Figura 5.16. Essa diferença é da ordem de $2,0 \times 10^{-3}$ nas segunda e quarta linhas, e $5.0 \times 10^{-4}$ na terceira linha. Essas pequenas diferenças podem ter sido geradas na exportação dos dados.

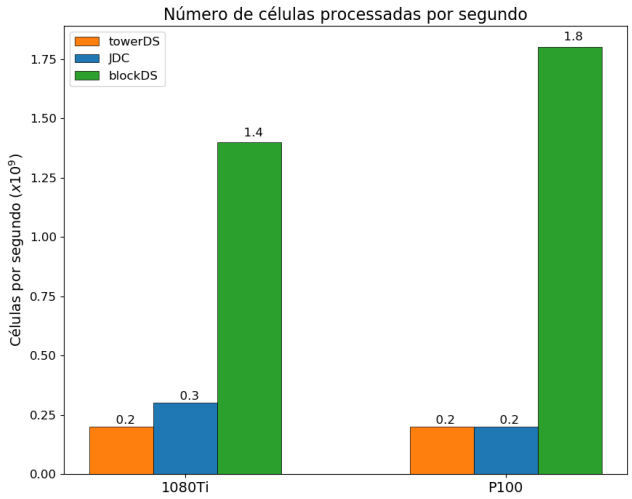



Figura 5.15: Número de células processadas por segundo.

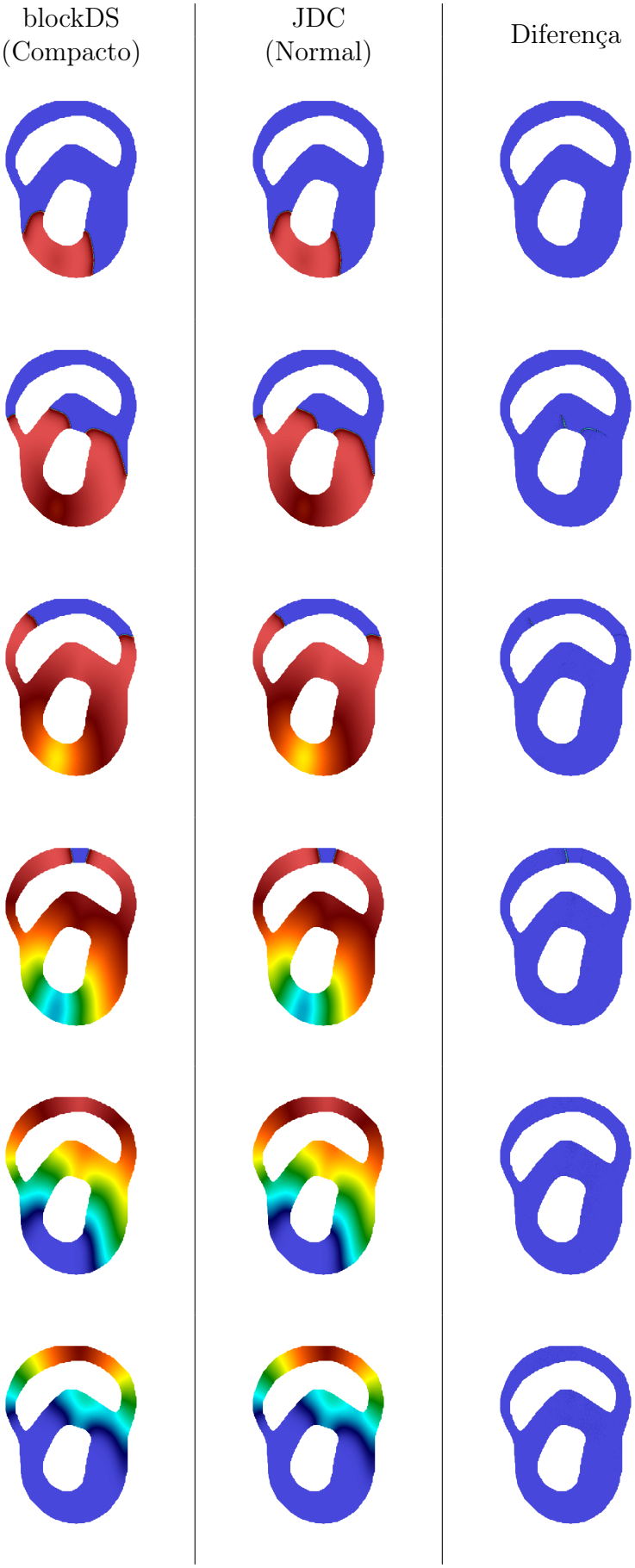



Figura 5.16: Simulação 2D de um pulso em uma camada z da estrutura do coração do porco.

# Capítulo 6

# Conclusão

Simulações da eletrofisiologia do tecido cardíaco tem grande importância nas pesquisas médicas para tratar arritmias e outros problemas do coração. Nesta pesquisa de tese foram desenvolvidas e adaptadas metodologias para executar simulações numéricas da eletrofisiologia do tecido cardíaco usando modelos matemáticos e computação paralela. As metodologias de implementação propostas foram projetadas para simulações utilizando ambientes híbridos compostos por CPUs e GPUs. O foco dos esforços empregados estava em extrair o máximo de capacidade de processamento paralelo de uma GPU, buscando simulações próximas ao tempo real.

Na primeira etapa de pesquisa buscamos na literatura quais métodos vem sendo empregados para a aceleração das simulações da eletrofisiologia cardíaca em GPUs. A despeito do método numérico usado, em sua maioria, os autores empregam técnicas baseadas em estruturas de dados lineares nas quais um domínio bi ou tri-dimensional é linearizado usando a ordenação linha majoritária. Neste trabalho o método numérico usado para calcular a solução da equação de reação-difusão (Equação 2.1) foi o método de diferenças finitas. Este é um método simples, mas eficiente para simulações de fenômenos de arritmia com a ocorrência de ondas espirais no tecído cardíaco. Em 2009, Micikevicius [31] propôs uma técnica de paralelização do método de diferenças finitas em GPU. Tal técnica foi utilizada por diversos autores (vide Capítulo 3).

A técnica proposta por Micikevicius, chamada janela deslisante (JDC), consiste em usar blocos bi-dimensionais para processar domínios tri-dimensionais com uma iteração na direção $Z$. A cada iteração, a JDC usa a memória compartilhada e os registradores da GPU para armazenar os dados a serem processados por um bloco. Contudo, essa técnica é, ainda hoje, usada em combinação com dados armazenados linearmente usando ordenação linha majoritária, mas tal combinação pode acarretar problemas de acesso

aos dados nas vizinhanças dos blocos (vide Capítulos 3 e 4). Tendo isto em vista, na segunda etapa da pesquisa, foi desenvolvida uma estrutura de dados que, se comparada a JDC com linha majoritária, reduz o tempo de computação do passo de tempo numérico. Os resultados experimentais mostram que dependendo do tamanho do domínio e da GPU utilizada, a combinação da técnica de janela deslizante com a estrutura de dados em torre, aqui proposta, pode reduzir o tempo de uma simulação em até 41%, quando comparado a JDC com ordenação linha majoritária. Com uma Tesla P100, nossa implementação do modelo Karma de duas variáveis simula um segundo do tempo real (20.000 passos de tempo numérico) em $14s$, para um domínio com $256^3$ células, e em $112s$, para um domínio com $512^3$ células. Nossa implementação do Karma com a estrutura em torre processa 23.9 bilhões de células por segundo em uma GPU Tesla P100, tendo desempenho melhor que trabalhos anteriores usando a JDC.

Os experimentos realizados mostram que o número de células pode ser um limitante para as simulações em GPU, pois poderia não haver espaço suficiente na memória global para armazenar as variáveis. Portanto, na terceira etapa foi desenvolvida uma estratégia de comunicação para possibilitar executar uma simulação usando múltiplas GPUs. Usando uma DGX-1 da Nvidia, com 8 GPUs P100, nos aproximamos do tempo real simulando 1 segundo de atividade elétrica do coração em $\sim$ 4.5 segundos. Neste caso o domínio era composto por $256^3$ células. Para um domínio com 1024 células, a implementação multi-GPU desenvolvida nesta pesquisa foi capaz de processar, usando 8 GPUs P100, 145 bilhões de células por segundo. O modelo matemático usado nesta implementação também foi o Karma de duas variáveis. Os resultados obtidos mostram um impressionante ganho de performance com o uso da memória paginada. Recentemente Kaboudian et al. (2019) [25] relatou a capacidade de processar $9,96 \times 10^10$ células por segundo utilizando a API WebGL e uma GPU Titan X com arquitetura Pascal. Tais resultados foram obtidos com um modelo de três variáveis muito similar ao Karma, contudo o domínio do experimento era um domínio 2D com apenas $512^3$ células.

A quarta etapa da pesquisa focou em um modelo eletrofisiológico mais realista, e consequentemente mais complexo. Foi usado o modelo OVVR [36] de 41 variáveis, e um campo de fase $\phi$ que representa a geometria do coração (vide Seção 5.4.1). Desenvolveu-se uma nova estrutura de dados capaz de se adaptar a geometria do coração, mas mantendo uma distribuição de dados capaz de atender as condições de coalescência nas transações de memória. Foi desenvolvido também um algoritmo para mapear as `threads` de cada bloco nas diversas posições de memória que serão acessadas (vide Seção 4.4). Simulações em duas dimensões não mostraram diferenças entre os resultados numéricos obtidos com a

estrutura adaptativa, que reduz o tamanho do domínio, e a solução com a JDC. O campo $\phi$ utilizado em nossos experimentos é composto por $250 \times 326 \times 298$ células. Nessas condições, a implementação do OVVR, usando a estrutura de dados adaptativa, chega a ser dez vezes mais rápida que ambas a JDC e a implementação com a estrutura em torre.

Os experimentos realizados nesta pesquisa envolveram não somente o desenvolvimento e implementação de técnicas computacionais, mais também a implementação dos modelos Karma e OVVR. Em particular a implementação do simulador em GPU com o modelo OVVR foi um desafio por tratar-se de um modelo complexo composto por dezenas de equações [36].

Nesta pesquisa de tese buscamos implementações em GPU otimizadas para executar simulações da eletrofisiologia cardíaca em tempos próximos ao tempo simulado. Conseguimos aprimorar o método clássico de paralelização em GPU do método de diferenças centradas, proposto por Micikevicius em 2009 e muito utilizado ao longo dos anos. Também propusemos uma método novo capaz de reduzir a carga de trabalho adaptando-se a geometria do coração e a presença ou não de tecido nas células do domínio discreto. Nossa implementação com múltiplas GPUs chega próximo ao tempo real para um modelo mais simples. Já com a implementação usando a estrutura adaptativa conseguimos executar uma simulação, usando um modelo complexo com 41 variáveis, num domínio com $\sim 2,41 \times 10^7$ células, em apenas $265s$. Até onde nosso levantamento da literatura mostrou, não há trabalhos implementando o modelo OVVR 3D em GPUs usando CUDA e uma abordagem como a blockDS (vide Capítulo 3).

## 6.1 Trabalhos futuros

Esta pesquisa de tese concretizou seus objetivos, mas abriu portas para questões que podem ser investigadas em trabalhos futuros.

Os resultados encontrados no experimentos com o estêncil de diferenças finitas 5.2 sugerem investigar a relação entre o número de passos de tempo processados por segundo para estênceis de ordem superior a 4.

Holewinski (2012) [22] e Zumbusch (2012) [52] propõe uma estratégia paralela para diferenças finitas em GPUs chamada de agrupamento espaço-temporal (*space-time blocking*). Essa estratégia consistem em usar a memória compartilhada de forma que um mesmo kernel possa calcular mais de um passo de tempo numérico. Combinar esta técnica com as estruturas de dados propostas tem grandes chances de resultar em uma aceleração

ainda maior das simulações numéricas da dinâmica da eletrofisiologia do coração.

Outra questão relevante reside na real eficiência do método clássico da janela deslisante. Alguns resultados preliminares indicaram que esta pode não ser a abordagem mais eficiente nas GPUs modernas. Estudos mais profundos sobre uso de memória compartilhada e "cacheamento de dados" podem ser realizados para averiguar sob quais condições o uso da memória compartilhada é a melhor estratégia.

# Referências


[1] Abdelkhalek, R.; Calandra, H.; Coulaud, O.; Latu, G.; Roman, J. Fast seismic modeling and reverse time migration on a graphics processing unit cluster. *Concurrency and Computation: Practice and Experience 24*, 7 (2012), 739–750.

[2] Bartocci, E.; Cherry, E. M.; Glimm, J.; Grosu, R.; Smolka, S. A.; Fenton, F. H. Toward real-time simulation of cardiac dynamics. In *Proceedings of the 9th International Conference on Computational Methods in Systems Biology* (New York, NY, USA, 2011), CMSB '11, ACM, pp. 103–112.

[3] Beeler, G. W.; Reuter, H. Reconstruction of the action potential of ventricular myocardial fibres. *the Journal of Physiology 268*, 1 (1977), 177–210.

[4] Bernaschi, M.; Fatica, M.; Melchionna, S.; Succi, S.; Kaxiras, E. A flexible high-performance lattice boltzmann gpu code for the simulations of fluid flows in complex geometries. *Concurrency and Computation: Practice and Experience 22*, 1 (2010), 1–14.

[5] Bondarenko, V. E.; Szigeti, G. P.; Bett, G. C.; Kim, S.-J.; Rasmusson, R. L. Computer model of action potential of mouse ventricular myocytes. *American Journal of Physiology-Heart and Circulatory Physiology 287*, 3 (2004), H1378–H1403.

[6] Bueno-Orovio, A.; Cherry, E. M.; Fenton, F. H. Minimal model for human ventricular action potentials in tissue. *Journal of Theoretical Biology 253*, 3 (2008), 544–560.

[7] Butters, T. D.; Aslanidi, O. V.; Zhao, J.; Smaill, B.; Zhang, H. A novel computational sheep atria model for the study of atrial fibrillation. *Interface Focus 3*, 2 (2013).

[8] Campos, J. O.; Oliveira, R. S.; dos Santos, R. W.; Rocha, B. M. Lattice boltzmann method for parallel simulations of cardiac electrophysiology using {GPUs}. *Journal of Computational and Applied Mathematics 295* (2016), 70–82. {VIII} Pan-American Workshop in Applied and Computational Mathematics.

[9] Chen, S.; Doolen, G. D. Lattice boltzmann method for fluid flows. *Annual Review of Fluid Mechanics 30*, 1 (1998), 329–364.

[10] Cherry, E. M.; Fenton, F. H. Suppression of alternans and conduction blocks despite steep apd restitution: electrotonic, memory, and conduction velocity restitution effects. *American Journal of Physiology-Heart and Circulatory Physiology 286*, 6 (2004), H2332–H2341.

[11] Cherry, E. M.; Fenton, F. H. Visualization of spiral and scroll waves in simulated and experimental cardiac tissue. *New Journal of Physics 10*, 12 (2008), 125016.

[12] Clayton, R.; Panfilov, A. A guide to modelling cardiac electrical activity in anatomically detailed ventricles. *Progress in Biophysics and Molecular Biology 96*, 1-3 (2008), 19–43. Cardiovascular Physiome.

[13] Clua, E.; Panaro Zamith, M. Programming in cuda for kepler and maxwell architecture. *Revista de Informática Teórica e Aplicada 22*, 2 (11 2015), 233–257.

[14] Dematté, L.; Prandi, D. Gpu computing for systems biology. *Briefings in Bioinformatics 11*, 3 (2010), 323–333.

[15] Fenton, F.; Karma, A. Vortex dynamics in three-dimensional continuous myocardium with fiber rotation: Filament instability and fibrillation. *Chaos 8*, 1 (1998), 20–47.

[16] Fenton, F. H.; Cherry, E. M. Models of cardiac cell. *Scholarpedia 3*, 8 (2008), 1868.

[17] Fenton, F. H.; Cherry, E. M.; Karma, A.; Rappel, W.-J. Modeling wave propagation in realistic heart geometries using the phase-field method. *Chaos: An Interdisciplinary Journal of Nonlinear Science 15*, 1 (2005), 013502.

[18] Giles, M.; László, E.; Reguly, I.; Appleyard, J.; Demouth, J. Gpu implementation of finite difference solvers. In *Proceedings of the 7th Workshop on High Performance Computational Finance* (Piscataway, NJ, USA, 2014), WHPCF '14, IEEE Press, pp. 1–8.

[19] Harris, M. J.; Coombe, G.; Scheuermann, T.; Lastra, A. Physically-based visual simulation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Aire-la-Ville, Switzerland, Switzerland, 2002), HWWS '02, Eurographics Association, pp. 109–118.

[20] Hodgkin, A. L.; Huxley, A. F. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology 117*, 4 (1952), 500–544.

[21] Hoffman, J. D.; Frankel, S. *Numerical methods for engineers and scientists*. CRC press, 2001.

[22] Holewinski, J.; Pouchet, L.-N.; Sadayappan, P. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing* (New York, NY, USA, 2012), ICS '12, ACM, pp. 311–320.

[23] Hosoi, A.; Washio, T.; Okada, J.; Kadooka, Y.; Nakajima, K.; Hisada, T. A multi-scale heart simulation on massively parallel computers. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–11.

[24] Iyer, V.; Mazhari, R.; Winslow, R. L. A computational model of the human left-ventricular epicardial myocyte. *Biophysical Journal 87*, 3 (2004), 1507–1525.

[25] Kaboudian, A.; Cherry, E. M.; Fenton, F. H. Real-time interactive simulations of large-scale systems on personal computers and cell phones: Toward patient-specific heart modeling and other applications. *Science Advances 5*, 3 (2019).

[26] Karma, A. Spiral breakup in model equations of action potential propagation in cardiac tissue. *Phys. Rev. Lett. 71* (Aug 1993), 1103–1106.

[27] Kirk, D. B.; Wen-Mei, W. H. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[28] Mahajan, A.; Shiferaw, Y.; Sato, D.; Baher, A.; Olcese, R.; Xie, L.-H.; Yang, M.-J.; Chen, P.-S.; Restrepo, J. G.; Karma, A.; Garfinkel, A.; Qu, Z.; Weiss, J. N. A rabbit ventricular action potential model replicating cardiac dynamics at rapid heart rates. *Biophysical Journal 94*, 2 (2008), 392 – 410.

[29] Mena, A.; Ferrero, J. M.; Matas, J. F. R. Gpu accelerated solver for nonlinear reaction–diffusion systems. application to the electrophysiology problem. *Computer Physics Communications 196* (2015), 280 – 289.

[30] Michéa, D.; Komatitsch, D. Accelerating a three-dimensional finite-difference wave propagation code using gpu graphics cards. *Geophysical Journal International 182*, 1 (2010), 389–402.

[31] Micikevicius, P. 3d finite difference computation on gpus using cuda. In *Proceedings of 2nd workshop on general purpose processing on graphics processing units* (2009), ACM, pp. 79–84.

[32] Moreno, J. D.; Zhu, Z. I.; Yang, P.-C.; Bankston, J. R.; Jeng, M.-T.; Kang, C.; Wang, L.; Bayer, J. D.; Christini, D. J.; Trayanova, N. A., et al. A computational model to predict the effects of class i anti-arrhythmic drugs on ventricular rhythms. *Science translational medicine 3*, 98 (2011), 98ra83–98ra83.

[33] Niederer, S.; Mitchell, L.; Smith, N.; Plank, G. Simulating human cardiac electrophysiology on clinical time-scales. *Frontiers in Physiology 2*, 14 (2011).

[34] Nimmagadda, V. K.; Akoglu, A.; Hariri, S.; Moukabary, T. Cardiac simulation on multi-gpu platform. *The Journal of Supercomputing 59*, 3 (2012), 1360–1378.

[35] NVIDIA. CUDA C programming guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2016. [Online; accessed 25-September-2016].

[36] O'Hara, T.; Virág, L.; Varró, A.; Rudy, Y. Simulation of the undiseased human cardiac ventricular action potential: Model formulation and experimental validation. *PLOS Computational Biology 7*, 5 (05 2011), 1–29.

[37] Oliveira, R. S.; Rocha, B. M.; Burgarelli, D.; Meira, W.; dos Santos, R. W. *An Adaptive Mesh Algorithm for the Numerical Solution of Electrical Models of the Heart*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 649–664.

[38] Porter-Sobieraj, J.; Cygert, S.; Kikoła, D.; Sikorski, J.; Słodkowski, M. Optimizing the computation of a parallel 3d finite difference algorithm for graphics processing units. *Concurrency and Computation: Practice and Experience 27*, 6 (2015), 1591–1602.

[39] Rocha, B. M.; Campos, F. O.; Amorim, R. M.; Plank, G.; dos Santos, R. W.; Liebmann, M.; Haase, G. Accelerating cardiac excitation spread simulations using graphics processing units. *Concurrency and Computation: Practice and Experience 23*, 7 (2011), 708–720.

[40] Rush, S.; Larsen, H. A practical algorithm for solving dynamic membrane equations. *IEEE Transactions on Biomedical Engineering BME-25*, 4 (July 1978), 389–392.

[41] Saad, Y. *Iterative methods for sparse linear systems*. Siam, 2003.

[42] Sachetto Oliveira, R.; Martins Rocha, B.; Burgarelli, D.; Meira, W.; Constantinides, C.; Weber dos Santos, R. Performance evaluation of gpu parallelization, space-time adaptive algorithms, and their combination for simulating cardiac electrophysiology. *International Journal for Numerical Methods in Biomedical Engineering 34*, 2 (2018), e2913–n/a. e2913 cnm.2913.

[43] Sachse, F. B. *Computational Cardiology: Modeling of Anatomy, Electrophysiology and Mechanics*, vol. 2966. Springer, 2004.

[44] Sanders, J.; Kandrot, E. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

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

[46] ten Tusscher, K. H. W. J.; Panfilov, A. V. Alternans and spiral breakup in a human ventricular tissue model. *American Journal of Physiology - Heart and Circulatory Physiology 291*, 3 (2006), H1088–H1100.

[47] Trayanova, N. Defibrillation of the heart: insights into mechanisms from modelling studies. *Experimental physiology 91*, 2 (2006), 323–337.

[48] Trayanova, N. A. Whole-heart modeling applications to cardiac electrophysiology and electromechanics. *Circulation research 108*, 1 (2011), 113–128.

[49] Victorri, B.; Vinet, A.; Roberge, F. A.; Drouhard, J.-P. Numerical integration in the reconstruction of cardiac action potentials using hodgkin-huxley-type models. *Computers and Biomedical Research 18*, 1 (1985), 10–23.

[50] Xia, Y.; Wang, K.; Zhang, H. Parallel optimization of 3d cardiac electrophysiological model using gpu. *Computational and Mathematical Methods in Medicine 2015* (2015), 1–10.

[51] Zahid, S.; Whyte, K. N.; Schwarz, E. L.; Blake III, R. C.; Boyle, P. M.; Chrispin, J.; Prakosa, A.; Ipek, E. G.; Pashakhanloo, F.; Halperin, H. R., et al. Feasibility of using patient-specific models and the "minimum cut" algorithm

to predict optimal ablation targets for left atrial flutter. *Heart rhythm 13*, 8 (2016), 1687–1698.

[52] ZUMBUSCH, G. Vectorized higher order finite difference kernels. In *International Workshop on Applied Parallel Computing* (2012), Springer, pp. 343–357.