Raphael Pereira Cordeiro

Agrupando dados e *kernels* de um simulador cardíaco em um ambiente multi-GPU

Dissertação apresentada ao Programa de Pós-graduação em Modelagem Computacional, da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do grau de Mestre em Modelagem Computacional.

Orientador: Prof. D. Sc. Marcelo Lobosco

Coorientador: Prof. D. Sc. Rodrigo Weber dos Santos

Coorientador: Prof. D. Sc. Rafael Sachetto Oliveira

Juiz de Fora

2017

Raphael Pereira Cordeiro,

Agrupando dados e *kernels* de um simulador cardíaco em um ambiente multi-GPU/ Raphael Pereira Cordeiro. – Juiz de Fora: UFJF/MMC, 2017.

XIII, 66 p.: il.; 29, 7cm.

Orientador: Marcelo Lobosco

Coorientador: Rodrigo Weber dos Santos

Coorientador: Rafael Sachetto Oliveira

Dissertação (mestrado) – UFJF/MMC/Programa de Modelagem Computacional, 2017.

Referências Bibliográficas: p. 63 – 66.

1. Modelagem computacional. 2. Computação paralela. 3. GPU. 4. Eletrofisiologia cardíaca. 5. Computação de alto desempenho. I. Lobosco, Marcelo *et al.*. II. Universidade Federal de Juiz de Fora, MMC, Programa de Modelagem Computacional.

Raphael Pereira Cordeiro

Agrupando dados e *kernels* de um simulador cardíaco em um ambiente multi-GPU

Dissertação apresentada ao Programa de Pós-graduação em Modelagem Computacional, da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do grau de Mestre em Modelagem Computacional.

Aprovada em 10 de Março de 2017.

BANCA EXAMINADORA

Prof. D. Sc. Marcelo Lobosco - Orientador
Universidade Federal de Juiz de Fora

Prof. D. Sc. Rodrigo Weber dos Santos - Coorientador
Universidade Federal de Juiz de Fora

Prof. D. Sc. Rafael Sachetto Oliveira - Coorientador
Universidade Federal de São João del-Rei

Prof. D. Sc. Bernardo Martins Rocha
Universidade Federal de Juiz de Fora

Prof. D. Sc. Carolina Ribeiro Xavier
Universidade Federal de São João del-Rei

Prof. D. Sc. Cristiana Barbosa Bentes
Universidade do Estado do Rio de Janeiro

Dedico este trabalho a meus amigos e parentes que tiraram os obstáculos à minha frente, permitindo que eu pudesse sempre seguir em frente.

AGRADECIMENTOS

Agradeço, à minha mãe (*in memorian*), por me ensinar que, de todo o legado que me deixou, o mais importante foi a educação. Nunca teria chegado até aqui se ela não tivesse acreditado em mim.

Agradeço às minhas tias Heloísa e Silvana, além de meus demais tios e primos, por me ajudarem a seguir em frente.

Agradeço aos amigos e colegas pela companhia e companheirismo, em especial aos da PGMC pelas horas e horas de estudo, risadas e pelas "gordices".

Agradeço aos meus orientadores Marcelo Lobosco, Rodrigo Weber e Rafael Sachetto pela paciência e dedicação. Nunca me esquecerei.

Agradeço à minha esposa que, durante esses dois anos deixou de ser minha noiva e não saiu do meu lado, com paciência e amor. Nunca vou deixar de te amar.

Por fim, porém mais importante, agradeço a Deus.

"Porquanto é o SENHOR quem concede sabedoria, e da sua boca procedem a inteligência e o discernimento.

Provérbio de Salomão

RESUMO

A modelagem computacional é uma ferramenta útil no estudo de diversos fenômenos complexos, como o comportamento eletro-mecânico do coração em condições normais e patológicas, sendo importante para o desenvolvimento de novos medicamentos e métodos de combate às doenças cardíacas. A alta complexidade de processos biofísicos se traduz em complexos modelos matemáticos e computacionais, o que faz com que simulações cardíacas necessitem de um grande poder computacional para serem executadas. Logo, o estado da arte em simuladores cardíacos é implementado para ser executado em arquiteturas paralelas. Este trabalho apresenta a implementação e avaliação de um método com dados e *kernel* agregados, método este utilizado para reduzir o tempo de computação de códigos que executam em ambientes computacionais compostos de múltiplas unidades de processamento gráfico (*Graphics Processing Unit* ou simplesmente GPUs). Este método foi testado na computação de uma importante parte da simulação da eletrofisiologia do coração, a resolução das equações diferenciais ordinárias (EDOs), resultando em uma redução pela metade do tempo necessário para a sua resolução, quando comparado com o esquema onde este método não foi implementado. Com o uso da técnica proposta neste trabalho, o tempo total de execução das simulações cardíacas foi reduzido em até 25%.

Palavras-chave: Modelagem computacional. Computação paralela. GPU. Eletrofisiologia cardíaca. Computação de alto desempenho.

ABSTRACT

Computational modeling is a useful tool to study many distinct and complex phenomena, such as to describe the electrical and mechanical behavior of the heart, under normal and pathological conditions. The high complexity of the associated biophysical processes translates into complex mathematical and computational models. This, in turn, translates to cardiac simulators that demand a lot of computational power to be executed. Therefore, most of the state-of-the-art cardiac simulators are implemented to run in parallel architectures. In this work a new coalesced data and kernel scheme is evaluated. Its objective is to reduce the execution costs of cardiac simulations that run on multi-GPU environments. The new scheme was tested for an important part of the simulator, the solution of the systems of Ordinary Differential Equations (ODEs). The results have shown that the proposed scheme is very effective. The execution time to solve the systems of ODEs on the multi-GPU environment was reduced by half, when compared to a scheme that does not implemented the proposed data and kernel coalescing. As a result, the total execution time of cardiac simulations was 25% faster.

Keywords: Computational modeling. Parallel computing. GPU. Cardiac electrophysiology. High performance computing.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

1 INTRODUÇÃO

1.1 Motivação

Doenças cardíacas estão entre as principais causas de morte no mundo, sendo a causa da morte de cerca de 17,5 milhões de pessoas em 2012, o que corresponde a 31% das mortes ocorridas nesse ano [6], número que continua aumentando a cada ano. Para o desenvolvimento de novos medicamentos e novos tratamentos é essencial a plena compreensão de alguns aspectos, como a deformação mecânica do coração, sua fisiologia e fisiopatologia [7]. Neste cenário, a modelagem matemática e computacional tem sido uma importante ferramenta, que tornou possível, por exemplo, a visualização de fenômenos que até então existiam apenas em teoria, como ondas espirais, fenômeno associado à arritmia cardíaca [3].

Não surpreendentemente, a alta complexidade dos processos biofísicos se traduz igualmente em complexos modelos matemáticos e computacionais. Modelos cardíacos modernos são descritos por equações diferenciais parciais (EDP) não lineares e equações diferenciais ordinárias (EDOs) com milhões de variáveis e centenas de parâmetros, no caso de modelos discretos.

O modelo bidomínio [8], considerado a mais completa descrição da atividade elétrica do tecido cardíaco, sob adequados pressupostos pode ser reduzido a um modelo mais simples e não tão caro computacionalmente, chamado modelo monodomínio. Ainda assim, simulações em larga escala, como a derivada da discretização de um coração inteiro, continuam sendo um desafio. As dificuldades encontradas e a complexidade associada à implementação e uso destes modelos são justificadas pelos benefícios alcançados por eles.

Entretanto, a demanda pela discretização fina para a solução das EDPs e a heterogênea e não-uniforme condutividade elétrica do tecido cardíaco impediram, no passado, o estudo desse fenômeno em tecidos cardíacos que incluem estruturas microscópicas. Uma solução para este problema foi apresentada por Barros em [3] usando uma plataforma que alia o uso de unidades de processamento computacional (CPU) a unidades de processamento gráfico (GPU), composta por um *cluster* de computadores equipados com GPUs. A solução foi baseada na integração de duas ferramentas para computação de alto desempenho

já utilizadas em simulações de tecidos cardíacos, sendo elas a interface de trocas de mensagens (MPI) e unidades de processamento gráfico de uso geral (GPGPU).

Porém, foi observado em sistemas multi-GPU que possuem mais processos em CPUs do que em GPUs, que os múltiplos acessos simultâneos às placas gráficas podem causar uma sobrecarga nas mesmas. Arquiteturas mais atuais já possuem tecnologia capaz de evitar tal sobrecarga, chamada *Hyper-Q*, porém muitos pesquisadores e instituições não possuem acesso à tais arquiteturas.

1.2 Objetivos

O principal objetivo desta dissertação foi o de melhorar o desempenho de um simulador cardíaco que executa em ambientes compostos de múltiplas GPUs, usando para isso o agrupamento de dados e de *kernels*.

A versão multi-GPU do simulador cardíaco foi projetada para executar em um ambiente de memória distribuída composto de vários nós, onde por sua vez cada nó é composto por múltiplos núcleos CPUs e múltiplas GPUs. Tipicamente o número de núcleos é superior ao número de GPUs. Os núcleos CPUs são utilizados na computação das EDPs, enquanto as EDOs são computadas nas GPUs. A divisão dos dados a serem processados se dá pela quantidade de processos que executam nas CPUs. Desta forma, uma única GPU pode ser responsável pelo processamento dos dados de várias CPUs, tendo em vista que cada CPU invoca um *kernel* CUDA para ser executado na GPU.

O agrupamento de dados e de *kernels* tem por objetivo reduzir o número de *kernels* a serem invocados, de modo que estes sejam iguais ao número de GPUs disponíveis, reduzindo o seu impacto no barramento e Sistema Operacional. A hipótese é que este agrupamento possa reduzir os custos associados à resolução das EDOs nas GPUs, principalmente decorrentes da serialização na execução dos múltiplos *kernels* pela GPU.

Arquiteturas de GPUs mais atuais possuem a tecnologia *Hyper-Q*, que permite que vários núcleos CPUs invoquem simultaneamente *kernels* em uma única GPU, aumentando assim sua utilização. Segundo a NVIDIA [9], a tecnologia *Hyper-Q* aumenta o número total de conexões (filas de trabalho) entre o processador e a GPU, permitindo que até 32 conexões simultâneas sejam gerenciadas por hardware, reduzindo ou mesmo eliminando a serialização entre *kernels* CUDA. Deste modo, a pergunta que esta dissertação tenta

responder é: pode-se obter resultados equivalentes, em GPUs mais antigas, usando-se uma abordagem por software? A resposta a esta pergunta é importante, pois nem sempre é possível trocar todas as GPUs de um ambiente computacional, principalmente quando se trabalha com um grande número destas, como pode ocorrer em um *cluster* de computadores. Este trabalho demonstrará que os agrupamentos propostos são uma alternativa de software para computadores com GPUs que não possuem a tecnologia *Hyper-Q*. Por outro lado, esperamos que os custos adicionais (*overheads*) decorrentes de seu uso não reduzam, de forma significativa, o desempenho das aplicações quando executadas em GPUs que possuam a tecnologia *Hyper-Q*.

1.3 Método

Para verificar se a hipótese deste trabalho é válida, utilizou-se o seguinte método. Primeiro, executou-se o simulador em sua versão original, e coletou-se seus tempos de execução. Na sequência, a versão original do simulador foi modificada, de modo a implementar o agrupamento de dados e *kernels*. Chamaremos a versão original de KNA (*kernel* não agregado) e a versão modificada de KA (*kernel* agregado). A versão KA foi então executada no mesmo ambiente e com os mesmos parâmetros que a versão KNA, e seus tempos de execução são coletados. Procedeu-se então a comparação dos tempos. Caso ocorra um ganho de desempenho, ou seja, o tempo da versão KA seja menor que o tempo da versão KNA, assume-se que a hipótese é verdadeira.

Deve-se, contudo, verificar ainda se a solução proposta impõe baixo custo adicional para sua implementação. Para verificar o impacto, em termos de custo, da implementação do método KA, o teste anterior é repetido em uma GPU que possua o suporte a tecnologia *Hyper-Q*. Assume-se que a diferença nos tempos de execução das versões KA e KNA seja decorrente do custo adicional imposto pelo software que implementa o novo método.

Em ambas as versões foi utilizado um modelo microscópico para a descrição do tecido cardíaco conforme descrito por Barros em [2]. A discretização feita divide a reação e difusão do modelo monodomínio em um sistema não linear de EDOs e um sistema linear de EDPs. O cálculo do potencial de ação é feito utilizando-se o modelo Bondarenko [10], modelo este que contém 41 equações para descrever o comportamento de células cardíacas de camundongos. Como a resolução do sistema de EDPs é feito nas CPUs, através das

bibliotecas MPI e PETSc, não se espera que estas sejam muito impactadas pelo método KA. A solução dos sistemas de EDOs, feita nas unidades gráficas, é que deve sofrer o maior impacto.

1.4 Organização da dissertação

Esta dissertação foi dividida em seis partes, sendo a primeira a introdução do trabalho.

A segunda e a terceira partes formam o referencial teórico do trabalho. O segundo capítulo apresenta os modelos matemáticos e computacionais utilizados, assim como parte da teoria em eletrofisiologia que serviu como base para o desenvolvimento dos modelos. O terceiro capítulo foca na apresentação das técnicas de computação paralela que foram utilizadas no desenvolvimento do código.

O quarto capítulo apresenta os métodos. São discutidas as técnicas empregadas na paralelização dos modelos computacionais usados na simulação do tecido cardíaco, incluindo o método multi-GPU anterior e o método proposto no escopo deste trabalho. O quinto capítulo apresenta os resultados de simulações que demonstram a efetividade da solução proposta neste trabalho.

Por fim são apresentadas as conclusões, com as considerações finais e planos para trabalhos futuros.

2 MODELAGEM CARDÍACA

O coração é uma bomba eletromecânica cuja função é bombear o sangue para o restante do corpo, função que depende da contração coletiva e sincronizada de milhões de células musculares, chamadas miócitos. Os miócitos se ligam uns aos outros em série, organizando-se na forma de fibras musculares estriadas. Neste capítulo iremos realizar uma breve introdução sobre como este comportamento pode ser modelado matematicamente a fim de podermos simular computacionalmente não somente uma célula, mas todo um tecido cardíaco.

2.1 O potencial de ação

Na perspectiva da atividade elétrica, a propriedade mais importante dos miócitos é o fato destas células poderem responder ativamente a estímulos elétricos, podendo ser chamadas de células excitáveis.

Em condições de repouso, as células mantém uma concentração iônica interna diferente daquelas ao seu redor. A carga elétrica dos íons faz com que haja uma diferença de potencial ao longo da membrana, o que é chamado de potencial transmembrânico.

Se um estímulo elétrico suficientemente forte é aplicado à uma célula excitável, o potencial transmembrânico muda, fazendo com que as propriedades condutoras da membrana celular se alterem, resultando em um rápido fluxo de íons positivos para dentro da célula, processo chamado de despolarização. Após a despolarização o potencial retorna para o estado de repouso, o que é chamado de repolarização. O ciclo completo de despolarização e repolarização recebe o nome de potencial de ação [11], que é ilustrado pela Figura 2.1.

Uma característica importante das células excitáveis é o fato de serem refratárias à estimulação durante o potencial de ação, ou seja, um período de tempo em que células já excitadas não podem ser reexcitadas, o que faz com que a frequência do potencial de ação seja limitada. O tempo para que essas células possam ser estimuladas novamente é chamado de período refratário. Existe ainda um período refratário relativo, período em que alguns dos canais iônicos já estão ativos e, por isso, é possível a estimulação do

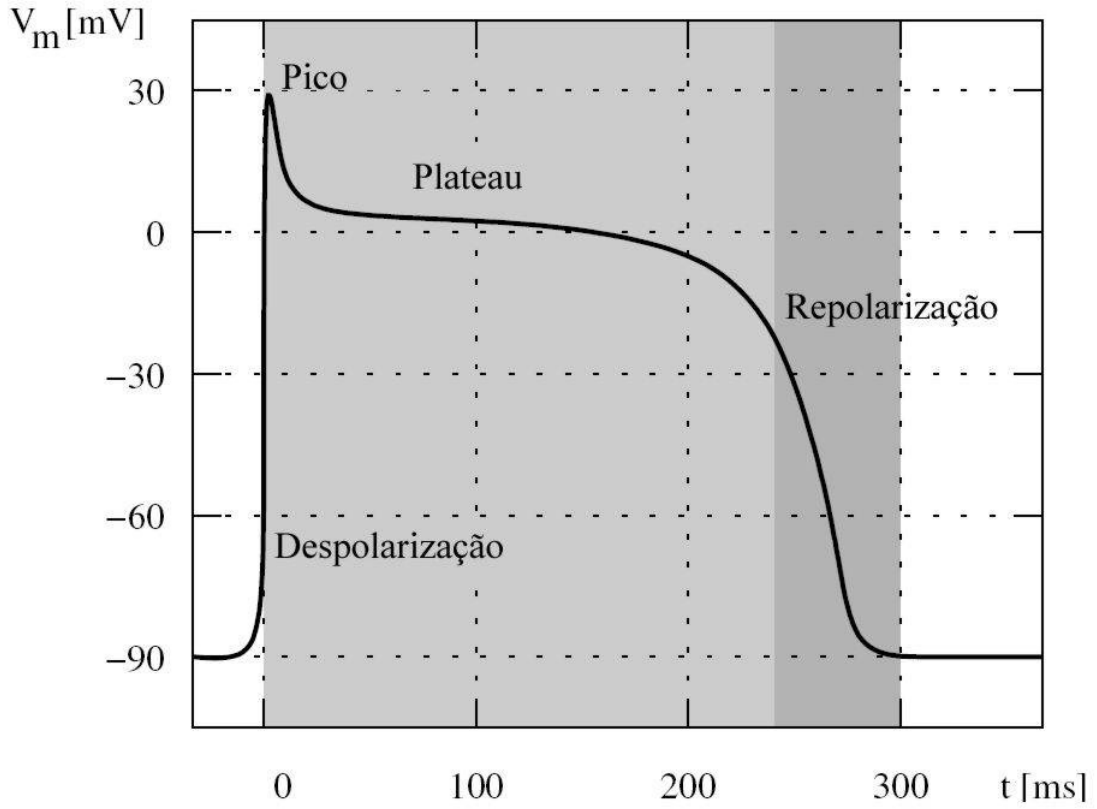

Figura 2.1: Etapas do potencial de ação medido de um miócito. Adaptado de [1].

potencial de ação. Porém, como nem todos os canais estão ativos, a corrente necessária para ativação deverá ser maior [12], conforme a Figura 2.2 ilustra.

2.2 Modelo Celular de Hodgkin e Huxley

Usando como base de estudo o axônio gigante de uma lula, Alan Hodgkin e Andrew Huxley desenvolveram o primeiro modelo quantitativo da propagação de um potencial de ação [13]. O modelo Hodgkin-Huxley foi proposto, inicialmente, para explicar a propagação de um sinal elétrico nos axônios de lulas, porém a ideia de seu estudo foi estendida e aplicada para uma grande variedade de células excitáveis. O simples modelo celular fez com que um novo campo de estudo surgisse na matemática aplicada: o estudo de sistemas excitáveis [14].

A corrente iônica estudada neste modelo é a soma das seguintes correntes: a corrente iônica de sódio, I_{Na}, a corrente iônica de potássio, I_K, e uma corrente de fuga, I_L, que seria um somatório de todas as demais correntes não descritas no modelo. Logo:

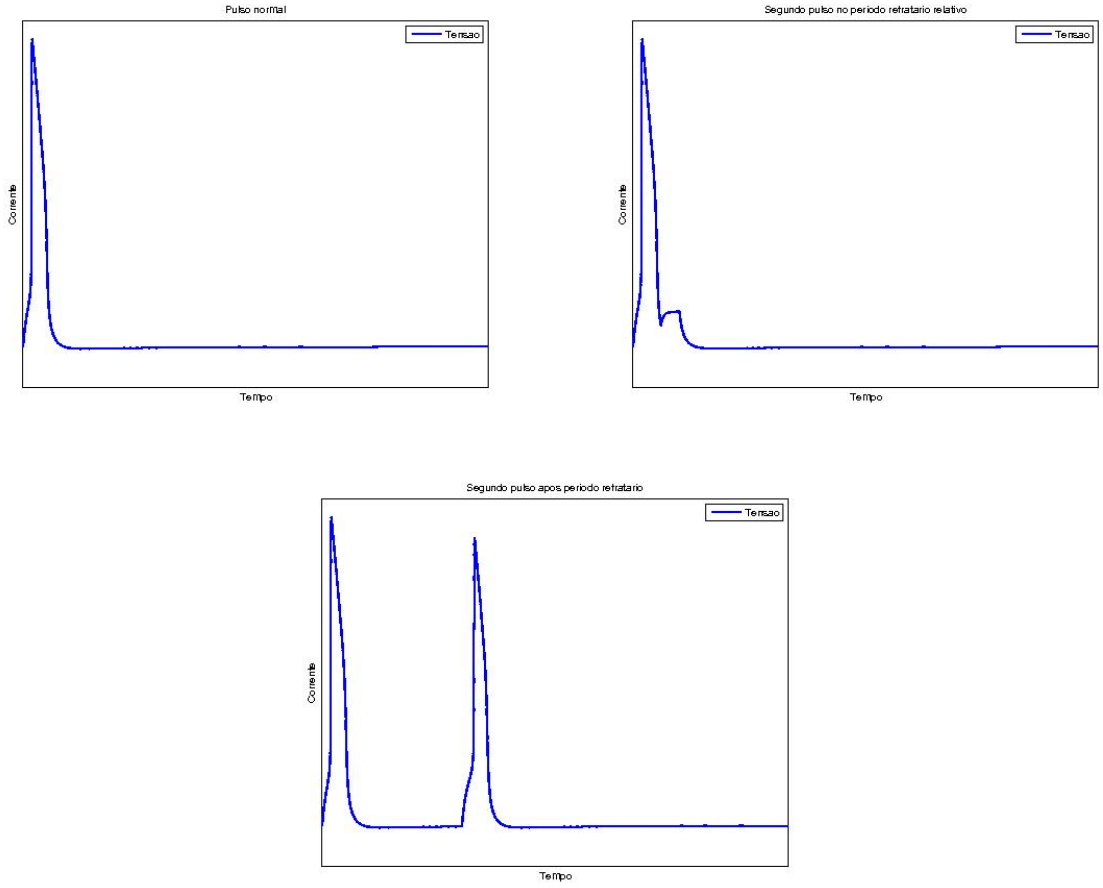

Figura 2.2: Contração do coração mostrando o período refratário e o refratário relativo.

$$I_{ion} = I_{Na} + I_K + I_L. \tag{2.1}$$

As equações de cada uma das correntes são obtidas a partir do modelo linear. g_{Na}, g_K e g_L são as condutâncias máximas para cada um dos canais. V_{Na}, V_K e V_L são os potenciais de equilíbrio de Nernst dos canais e m, n e h as probabilidades das subunidades dos mesmos. Levando-se em consideração o fato do canal de sódio possuir três subunidades m e uma h, o canal de potássio quatro subunidades n e a corrente de fuga não possuir subunidades, as equações das correntes serão:

$$I_{Na} = g_{Na} m^3 h (V - V_{Na}), \tag{2.2}$$

$$I_K = g_K n^4 (V - V_K), \tag{2.3}$$

$$I_L = g_L (V - V_L). \tag{2.4}$$

Para mais detalhes sobre os valores de cada subunidade, sugere-se a consulta a Sachetto [1] ou Keener [14].

É importante saber que a corrente total que atravessa a membrana celular, I_m, é dada pela soma da corrente capacitiva com a corrente iônica [15] e, portanto, é dada por:

$$I_m = I_{ion} + C_m \frac{dV}{dt} \tag{2.5}$$

onde C_m é a capacitância da membrana celular.

Por fim, substituindo as correntes (2.3), (2.4) e (2.4) na equação (2.5), chega-se a:

$$-C_m \frac{dV}{dt} = g_{Na} m^3 h (V - V_{Na}) + g_K n^4 (V - V_K) + g_L (V - V_L). \tag{2.6}$$

Em simulações computacionais adiciona-se uma corrente externa I_{ext} que deve ser aplicada como estímulo elétrico.

2.3 Modelo Celular de Bondarenko

Em 2003 Bondarenko e colaboradores (BDK) publicaram o primeiro modelo para simular miócitos ventriculares de camundongos [10]. O modelo possui 41 equações diferenciais ordinárias (EDOs) para o cálculo de 15 correntes transmembrânicas que, somadas, resultarão na corrente iônica.

Neste modelo os canais iônicos são representados por cadeias de Markov. Um processo de Markov é um sistema finito e discreto formado por estados, onde existe a probabilidade de transição entre os estados. Uma cadeia de Markov é a repetida execução de um processo de Markov onde o próximo estado será escolhido aleatoriamente, tendo como dependência apenas o estado atual [2].

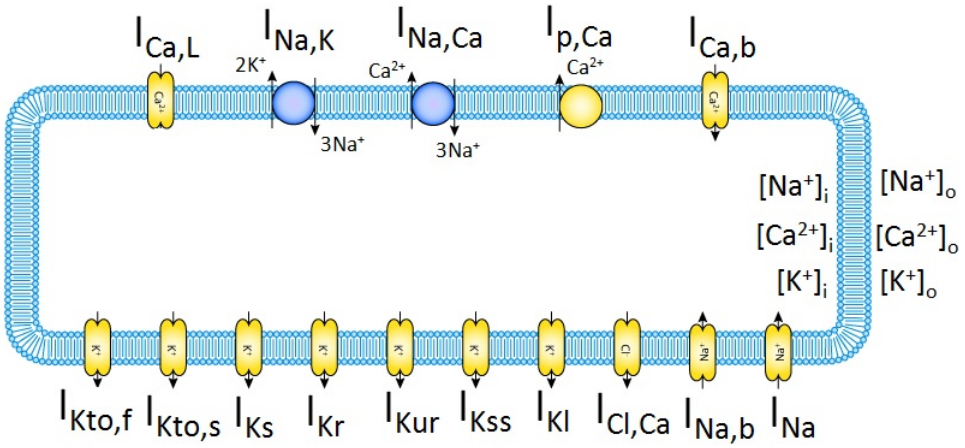

Figura 2.3: Circuito do medelo BDK. Adaptado de [2].

2.4 Modelos para o Tecido

As células cardíacas são conectadas por *gap junctions*, junções que funcionam como conexões elétricas entre elas, conectando-as no tecido cardíaco. Células estimuladas transmitem sinal elétrico para as adjacentes através destas junções, o que faz com que um estimulo elétrico aplicado em uma parte do coração possa se propagar por todo o órgão.

Como uma célula é capaz de estimular as adjacentes, ela não deve ser tratada individualmente e, por isso, o modelo para estudá-las deve ser um modelo para o tecido cardíaco. Apresentamos o Monodomínio, que é uma simplificação do modelo Bidomínio.

2.4.1 Modelo do Monodomínio

Dentre os modelos que descrevem a atividade elétrica do tecido cardíaco, o modelo bidomínio é considerado o mais completo.

No modelo bidomínio cada ponto no espaço representa uma fração do espaço intracelular e uma do espaço extracelular, de forma que cada ponto tenha dois potenciais elétricos, V_i e V_e, e duas correntes, i_i e i_e, com o índice i representando o espaço intracelular e o índice e representando o espaço extracelular.

Em 1978 Tung e Geselowitz [16] proporam as equações de formulação padrão do modelo

Bidomínio, dadas por:

$$\nabla \cdot (D_i \nabla V_m) + \nabla \cdot (D_i \nabla V_e) = \beta \left(C_m \frac{\partial V_m}{\partial t} + I_{ion} \right), \tag{2.7}$$

$$\nabla \cdot (D_i \nabla V_m) + \nabla \cdot ((D_i + D_e) \nabla V_e) = 0. \tag{2.8}$$

onde D_i e D_e são os tensores de condutividade e β é a razão superfície-para-volume da membrana, necessária para converter a corrente transmembrânica da unidade de área para a unidade de volume.

O bidomínio é um modelo de equações diferenciais parciais de alto custo computacional [17]. É possível simplificá-lo e obter o modelo Monodomínio que descreve apenas o comportamento do potencial transmembrânico V ao considerar que os meios extracelular e intracelular possuem uma taxa igual de anisotropia, ou seja, $D_e = \lambda D_i$, onde λ é uma constante.

Logo, eliminando D_e das equações (2.7) e (2.8), obtemos:

$$\nabla \cdot (D_i \nabla V) + \nabla \cdot (D_i \nabla V_e) = \beta \left(C_m \frac{\partial V}{\partial t} + I_{ion} \right), \tag{2.9}$$

$$\nabla \cdot (D_i \nabla V) + (1 + \lambda) \nabla \cdot (D_i \nabla V_e) = 0. \tag{2.10}$$

Isolando $\nabla \cdot (D_i \nabla V_e)$ em (2.10), substituindo em (2.9) e rearranjando os termos, ficamos com:

$$\frac{\lambda}{1 + \lambda} \nabla \cdot (D_i \nabla V) = \beta \left(C_m \frac{\partial V}{\partial t} + I_{ion} \right) \tag{2.11}$$

o que será, caso o tecido seja isotrópico, igual a

$$D \nabla^2 V = \beta \left(C_m \frac{\partial V}{\partial t} + I_{ion} \right), \tag{2.12}$$

onde D é um escalar.

Como dito anteriormente, o modelo Monodomínio é uma simplificação do modelo Bidomínio, o que implica no fato de que ele deve ser usado em casos mais limitados. Ele é gerado usando-se como hipótese que as taxas de anisotropia são iguais para os meios intracelular e extracelular, o que já foi provado por experimentos não ser verdade. Além disso, o parâmetro λ é difícil de se especificar de forma a se obter uma boa aproximação

das condutividades. Com isso, o modelo Bidomínio se mostra mais útil para simulações mais realísticas, enquanto que o Monodomínio deve ser usado apenas para estudo mais simples [17].

2.5 O Modelo Microscópico

Barros et. al. [3] desenvolveram um modelo bi-dimensional para o tecido cardíaco. O modelo leva em conta a microestrutura do tecido cardíaco, a distribuição heterogênea e uma discretização espacial de $8\mu m \times 8\mu m$. A Figura 2.4 apresenta um modelo padrão para as conexões dos miócitos. Nesta unidade básica são conectados entre si 32 miócitos cardíacos de diferentes formatos e número de células vizinhas. Os valores da média e desvio padrão do comprimento e largura das células são, respectivamente, $120,9 \pm 27,8\mu m$ e $18,3 \pm 3,5\mu m$. Em média cada célula se conecta a 6 miócitos vizinhos. O modelo ainda considera uma profundidade homogênea $d = 10\mu m$.

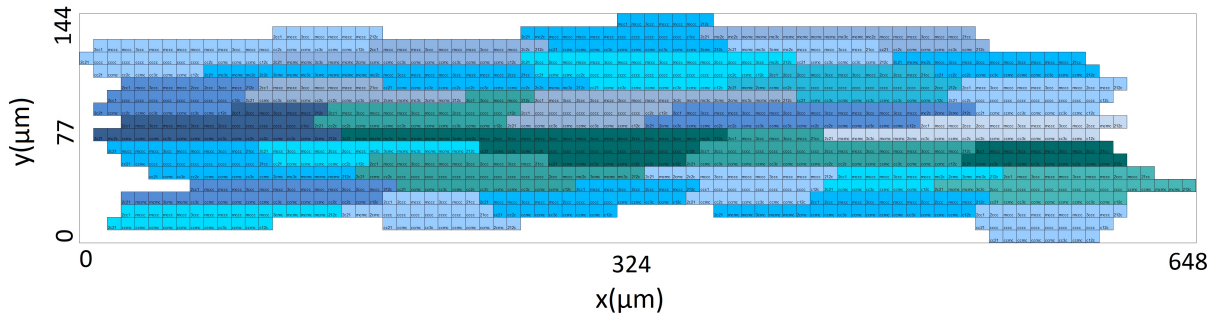

Figura 2.4: Unidade básica da distribuição do miócito cardíaco com um total de 32 células. As células estão dispostas em diferentes cores ao longo do eixo x. A unidade básica se estende por $648\mu m$ na direção longitudinal por $144\mu m$ na direção transversal. (Adaptado de [3].)

Esta unidade básica foi criada de forma a permitir que tecidos maiores fossem gerados à partir da conexão de múltiplas unidades, conforme pode ser verificado na Figura 2.5.

O código para simulação computacional desse modelo foi desenvolvido de forma que o usuário possa definir, para cada volume discretizado $Vol_{i,j}$ de área $h \times h$, os valores da condutividade ou condutância para as faces do norte ($\sigma_{x_{i,j+1/2}}$), do sul ($\sigma_{x_{i,j-1/2}}$), do leste ($\sigma_{x_{i+1/2,j}}$) ou do oeste ($\sigma_{x_{i-1/2,j}}$) podendo estes ser qualquer valor não negativo. Além disso

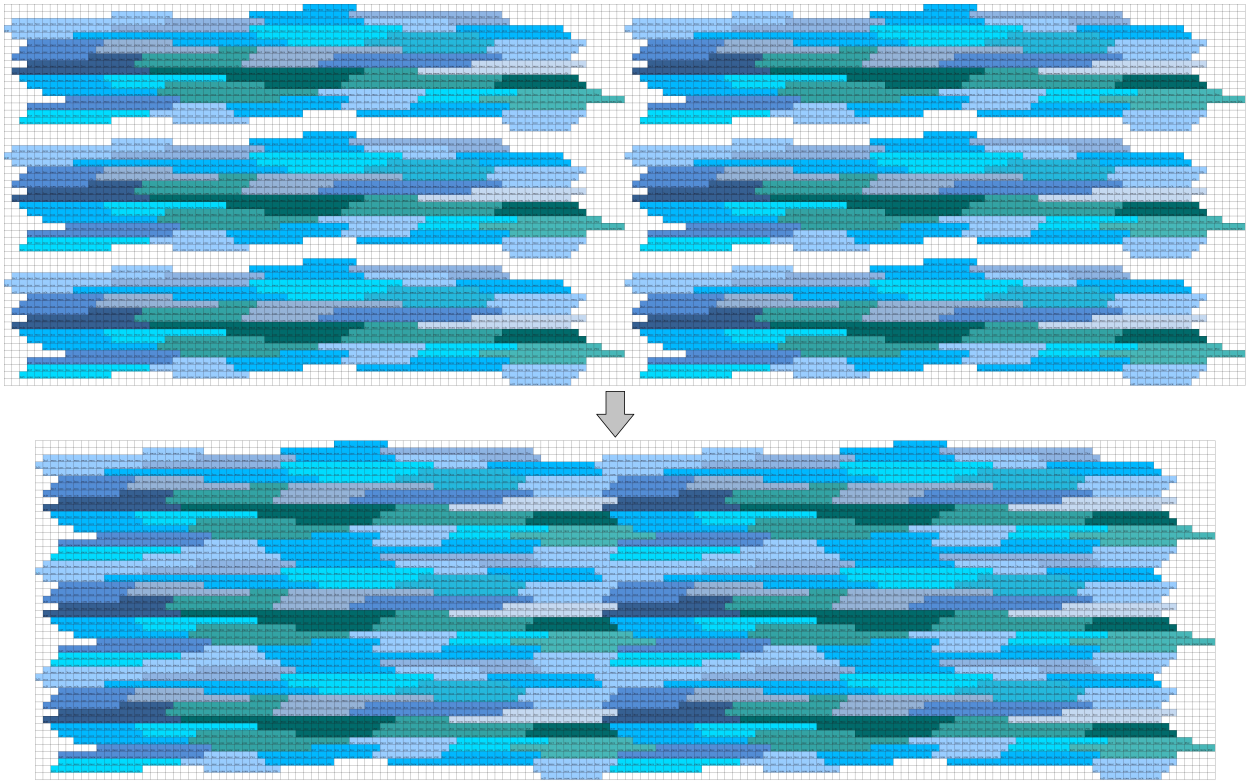

Figura 2.5: Seis unidades básicas sendo combinadas para gerar um tecido. (Adaptado de [3].)

foram escolhidos apenas 5 tipos de conexão entre vizinhos: membrana com $\sigma_m = 0,0$; citoplasma com $\sigma_c = 0,4\mu S/\mu m$; *plicate gap junctions* com $G_p = 0,5\mu S$; *interplicate* com $G_i = 0,33\mu S$ e *combined plicate* com $G_c = 0,062\mu S$. Nas conexões anteriores G é usado para condutância enquanto que σ para condutividade. A distribuição dos diferentes *gap junctions* não foi gerada aleatoriamente, mas escolhida de forma manual. Com estas definições e valores de condutividade, a velocidade de condução ao longo da fibra encontrada foi perto de $410\mu m/ms$ (LP) e de $130\mu m/ms$ a velocidade de condução na direção transversal da fibra (TP), o que resulta numa razão LP/TP de $0,32$ razão que está próxima da proporção encontrada em [18].

2.6 Discretização do Tempo e Espaço

As partes de reação e difusão das equações de monodomínio foram separadas utilizando o separador de Godunov [11]. Com isso, cada passo de tempo envolverá a solução de dois

problemas diferentes: um sistema não linear de EDOs

$$\begin{aligned} \frac{\partial V}{\partial t} &= \frac{1}{C_m}[-I_{ion}(V,\eta) + I_{stim}], \\ \frac{\partial \eta}{\partial t} &= f(V,\eta), \end{aligned} \tag{2.13}$$

e uma EDP parabólica

$$\beta \left(C_m \frac{\partial V}{\partial t} \right) = \nabla \cdot (\sigma \nabla V). \tag{2.14}$$

A derivada no tempo presente na equação (2.14) que opera em V pode ser aproximada por um esquema implícito de Euler de primeira ordem resultando em

$$\frac{\partial V}{\partial t} = \frac{V^{n+1} - V^n}{\Delta t_p}, \tag{2.15}$$

onde V^n representa o potencial transmembrânico no tempo t_n e Δt_p é o passo de tempo usado para avançar na EDP.

O sistema de EDOs é do tipo *stiff* [19], o que demanda um passo de tempo muito pequeno para métodos explícitos. Testes foram feitos com os métodos de Euler e de Rush-Larsen [20] e o passo de tempo demandado para ambos os métodos foi de $\Delta t_o = 0.0001ms$ para se obter estabilidade [3]. Como o método de Rush-Larsen é mais caro computacionalmente para cada passo de tempo, o método de Euler foi escolhido como melhor opção.

A solução das EDOs e das EDPs podem ser tratadas como problemas desacoplados, além disso o uso de um método incondicionalmente estável para as EDPs permite que o passo de tempo na solução dessas possa ser maior. O valor escolhido foi $\Delta t_p = 0.01ms$, o que acarretou num erro de apenas 0.01%, conforme calculado em [3].

O método dos volumes finitos é um método matemático utilizado para se obter uma versão discreta de equações diferenciais parciais [21, 22]. Este método foi utilizado para discretizar a parte espacial do termo de difusão na equação (2.14) e, para isto, foi considerada uma malha uniforme bidimensional constituída de quadriláteros regulares chamados de volumes. $Vol_{i,j}$ é o volume na posição i, j da malha.

Os cálculos são apresentados em [2, 3] porém o resultado são as equações para cada volume finito $Vol_{i,j}$, dadas por

$$\begin{gathered}(\sigma_{i+1/2,j} + \sigma_{i-1/2,j} + \sigma_{i,j+1/2} + \sigma_{i,j-1/2} + \alpha)V^*_{i,j} - \\ \sigma_{i,j-1/2}V^*_{i,j-1} - \sigma_{i+1/2,j}V^*_{i+1,j} - \sigma_{i,j+1/2}V^*_{i,j+1} - \sigma_{i-1/2,j}V^*_{i-1,j} = \alpha V^n_{i,j}\end{gathered} \tag{2.16}$$

$$\begin{aligned}C_m\frac{V^{n+1}_{i,j} - V^*_{i,j}}{\Delta t_o} &= -I_{ion}(V^*_{i,j}, \eta^n) \\ \frac{\eta^{n+1} - \eta^n}{\Delta t_o} &= f(\eta^n, V^*, t)\end{aligned} \tag{2.17}$$

onde η é um vetor de variáveis de estado, $\alpha = (\beta C_m h^2)/\Delta t_p$, n é o passo de tempo atual, $*$ é um passo de tempo intermediário, $n+1$ é o próximo passo de tempo, σ pode ser qualquer uma das condutâncias de *gap junction* (G_p, G_i, G_c) dividida pela profundidade d ou valor de condutância (σ_c ou σ_m) conforme definidos na seção 2.5.

Primeiramente deve ser resolvido o sistema linear (2.16) avançando no passo de tempo Δt_p e, depois, resolver o sistema não linear de EDOs (2.17) N_o vezes até que $N_o\Delta t_o = \Delta t_p$.

3 COMPUTAÇÃO PARALELA

3.1 Introdução

Normalmente uma aplicação é escrita de forma sequencial e cada uma de suas linhas de código é executada em sequência. Assim, cada instrução é executada uma após a outra. A soma de duas matrizes 100×100, por exemplo, leva o tempo da execução de dez mil somas para finalizar. Com o poder de processamento atual, dez mil somas são executadas instantaneamente, porém se levarmos em consideração matrizes com mais de um milhão de linha e colunas, cuja soma deve ser calculada milhões de vezes, esse problema passa a ser computacionalmente custoso, ou seja, muito demorado.

Percebe-se que a soma de matrizes é uma aplicação onde cada soma pode ser feita de forma independente das demais. Se os dados de cada matriz forem divididos igualmente entre dois computadores (ou processadores), cada um desses demorará metade do tempo original para a executar sua parte das somas. O processo de divisão dos dados (ou de instruções, ou de tarefas) entre dois ou mais computadores (ou processadores ou núcleos), com o objetivo de reduzir o tempo de computação ou lidar com uma quantidade maior de dados é o que é chamado de paralelização.

Neste capítulo serão brevemente apresentadas duas arquiteturas utilizadas na execução de aplicações paralelas, memória compartilhada e distribuída. Adicionalmente, será apresentada a arquitetura de placas gráficas que vêm, na última década, sendo utilizadas para acelerar a execução de aplicações paralelas. As ferramentas de programação utilizadas nessas arquiteturas são então apresentadas, com foco naquelas utilizadas no desenvolvimento do simulador do coração. Assumiremos ao longo de todo o trabalho o paralelismo baseado na divisão de dados entre as unidades de processamento (paralelismo de dados).

3.2 Memória Compartilhada

Computadores uniprocessados, como o nome indica, são compostos por um único processador que executa programas armazenados em sua memória. Uma maneira natural

de se estender essa arquitetura é ter vários processadores conectados à mesma memória, de forma que cada processador possa acessar qualquer endereço de memória, numa configuração chamada *memória compartilhada*. Em ambos os casos, a conexão entre o(s) processador(es) e a memória acontece através de um barramento [23], e uma única cópia do Sistema Operacional gerencia a máquina.

Em um sistema de memória compartilhada com múltiplos processadores o barramento pode conectar todos os processadores diretamente à memória principal, sistema chamado de UMA (*uniform memory access*), onde os tempos de acesso de cada processador serão os mesmos para todas as localizações de memória. Outra alternativa seria cada processador ter uma conexão direta a uma porção da memória principal, mas mantendo o acesso as porções de memória associadas aos demais. O acesso aos endereços localizados na porção de memória local de um processador ocorre de modo mais rápido do que os acessos realizados aos demais blocos, sistema este chamado de NUMA (*nonuniform memory access*).

Nestas arquiteturas, a implementação do paralelismo de dados é simples. Cada processador é responsável pelo processamento de uma porção dos dados da aplicação, que podem ser acessados diretamente pela memória compartilhada. A colaboração entre processadores se dá diretamente por leituras e escritas em posições de memória compartilhada. Podem ser necessárias operações de sincronização para evitar condições de corrida durante esta colaboração, ou seja, durante o processamento de dados e geração dos resultados.

3.3 Memória Distribuída

Em uma arquitetura de memória distribuída, os computadores ou processadores funcionam de modo independente, sendo chamados de nós, possuindo cada nó sua própria memória, disco e cópia do Sistema Operacional. Os computadores são interligados através de uma rede de comunicação.

A implementação do paralelismo de dados nessa arquitetura é mais complexo. Os dados precisam ser enviados para cada processador, o que é feito pela aplicação através de troca de mensagens. Caso os processos precisem se comunicar ou sincronizar durante o processamento, outras mensagens deverão ser trocadas. Observa-se então que a rede de

comunicação é de extrema importância para que seja possível obter um bom desempenho. Uma comunicação lenta pode comprometer todo o desempenho, por isso redes de baixa latência normalmente são utilizadas junto com protocolos especiais de comunicação.

Dentre os sistemas de memória distribuída, um dos mais utilizados é chamado de *cluster*. Um *cluster* é composto por uma série de computadores independentes interconectados por uma rede.

Nos últimos anos os *clusters* tornaram-se sistemas híbridos, visto que cada nó é, geralmente, um sistema de memória compartilhada composto por processadores com um ou mais núcleos de processamento [24]. Também observa-se uma tendência de agregar, a cada nó, unidades especializadas no processamento de grandes quantidades de dados, os chamados aceleradores. GPUs (*General Processing Units*) e FPGAs (*Field-programmable Gate Arrays*) são exemplos de aceleradores.

3.3.1 MPI

Dentre as formas de programação em ambientes de memória distribuída, a mais usada é a troca de mensagens (*message-passing*). Um padrão de troca de mensagens contém, no mínimo, uma função de envio e uma de recebimento. Quando um processo necessita enviar uma mensagem, ele chama uma função de envio (*send*) e o processo que vai receber a mensagem chama a função de recebimento (*receive*).

Um grupo formado por acadêmicos e industriais desenvolveu o padrão MPI (*message-passing interface*) para a troca de mensagens. O MPI não é uma nova linguagem de programação, mas sim um conjunto de rotinas encapsuladas em uma biblioteca que permitem o uso mais portátil e prático de trocas de mensagens [23]. Cabe aos fabricantes implementar e otimizar o padrão MPI para uma dada linguagem de programação, arquitetura e topologia de rede. Hoje é possível a utilização de MPI em diversas linguagens, como C, C++ e FORTRAN [2].

Ao executar um programa usando MPI, o usuário deverá especificar o número de processos a serem criados. Estes processos são univocamente identificados por inteiros não-negativos consecutivos denominados *ranks*, começando no primeiro processo, numerado com o *rank* 0, até o p-ézimo processo, numerado com o *rank* $p - 1$ [24].

Mesmo havendo um amplo e crescente número de rotinas, um programa simples em MPI precisa de apenas algumas para executar corretamente. De modo geral, as rotinas

indispensáveis são as relacionadas à inicialização do ambiente; identificação dos processos; troca de mensagens; e finalização do ambiente e liberação de recursos alocados. Um modelo geral de organização de código, baseado no número do identificador do processo, é ilustrado no Algoritmo 1.

Algoritmo 1: Pseudo-código de um programa simples em MPI

```
2:  main
4:    ... inicializa MPI ...
6:    ... processos recebem o número de seu rank e total de processos ...
8:      if rank == remetente do
10:       ... envia mensagem ...
12:     else do
14:       ... recebe mensagens de todos os demais processos ...
16:     end-else
18:     end-if
20:   ... finaliza MPI ...
    end-main
```

3.3.2 PETSc

A biblioteca PETSc (*Portable, Extensible Toolkit for Scientific Computation*) consiste de uma variedade de estruturas de dados e rotinas para a solução escalável de aplicações científicas modeladas a partir de equações diferenciais parciais. O padrão MPI é utilizado em todas as comunicações envolvendo trocas de mensagens.

PETSc inclui um conjunto de solucionadores de equações lineares e não lineares em paralelo, além de integradores que podem ser utilizados em códigos escritos nas linguagens Fortran, C, C++, Python ou MATLAB, neste último apenas para códigos sequenciais [25].

O PETSc deve ser iniciado com *PetscInitialize* e finalizado com *PetscFinalize*. O uso dessas funções dispensa a iniciação e finalização do MPI. O MPI possui um comunicador global, chamado de *MPI_COMM_WORLD*, trocado no PETSc por *PETSC_COMM_WORLD*. O padrão também possui seus próprios tipos de dados, como *PetscInt*, *PetscReal*, *PetscScalar* ou *PetscBool*. Também há suporte para criação de tipos de dados homogêneos e heterogêneos. Por exemplo, para utilizar vetores e matrizes, as variáveis devem ser definidas como *Vec* e *Mat*, respectivamente. O usuário pode facilmente criar vetores e matrizes paralelas de forma simples, sem a necessidade da utilização direta de MPI com esses tipos de dados [25, 26].

3.4 GPGPU

GPGPUs (*general-purpose graphics processing unit*), ou simplesmente GPUs, são placas de vídeo que podem ser programadas para realizar processamentos que tipicamente seriam executados por CPUs. Elas estão presentes em grande parte dos computadores modernos e são capazes de realizar uma mesma operação em um grande conjunto de dados [27, 28].

Seu grande poder de processamento se dá graças à presença de uma grande quantidade de unidades de processamento, as chamadas unidades lógico-aritméticas (do inglês ALUs - *arithmetic logic units*). Enquanto as CPUs possuem apenas algumas unidades, as GPUs geralmente possuem centenas, o que faz com que uma quantidade muito maior de dados possa ser processada simultaneamente, como pode ser visto na Figura 3.1.

As unidades de processamento das GPUs executam *threads*, que organizam-se de forma hierárquica. Uma *thread* de execução é a menor sequência de instruções programáveis que pode ser executada de maneira independente. Um conjunto de *threads* cooperativas entre si pode ser agrupado num bloco de *threads*. Dentro de um bloco, cada *thread* tem seu próprio número de identificação, ou *thread ID*. Um grupo de blocos gera um *grid*.

Mesmo com grande número de unidades de processamento disponíveis, o uso das primeiras GPUs tinha algumas desvantagens. Primeiramente, o programador que quisesse utilizar a placa gráfica teria que ter grande conhecimento da API gráfica OpenGL e da arquitetura da GPU, visto que deveria transformar seu código em uma sequência de operações gráficas equivalentes, eventualmente convertendo parâmetros e resultados das operações. Acessos aleatórios à memória não eram permitidos, dificultando tanto a realização de operações de escrita quanto de leitura. Além disso, não havia suporte à precisão dupla. Para resolver esses problemas, foram criadas pela NVIDIA a interface CUDA e a arquitetura G80.

3.4.1 NVIDIA CUDA

Em 2007 a NVIDIA lançou uma interface de programação de fácil utilização para as GPUs, chamada CUDA (*Compute Unified Device Architecture*), o que permitiu a programação de GPUs sem a necessidade de aprender as complicadas linguagens anteriores ou ter que pensar os programas em termos de operações gráficas primitivas. Porém, por ser uma exclusividade da NVIDIA, é necessário ter uma placa gráfica da empresa com suporte à

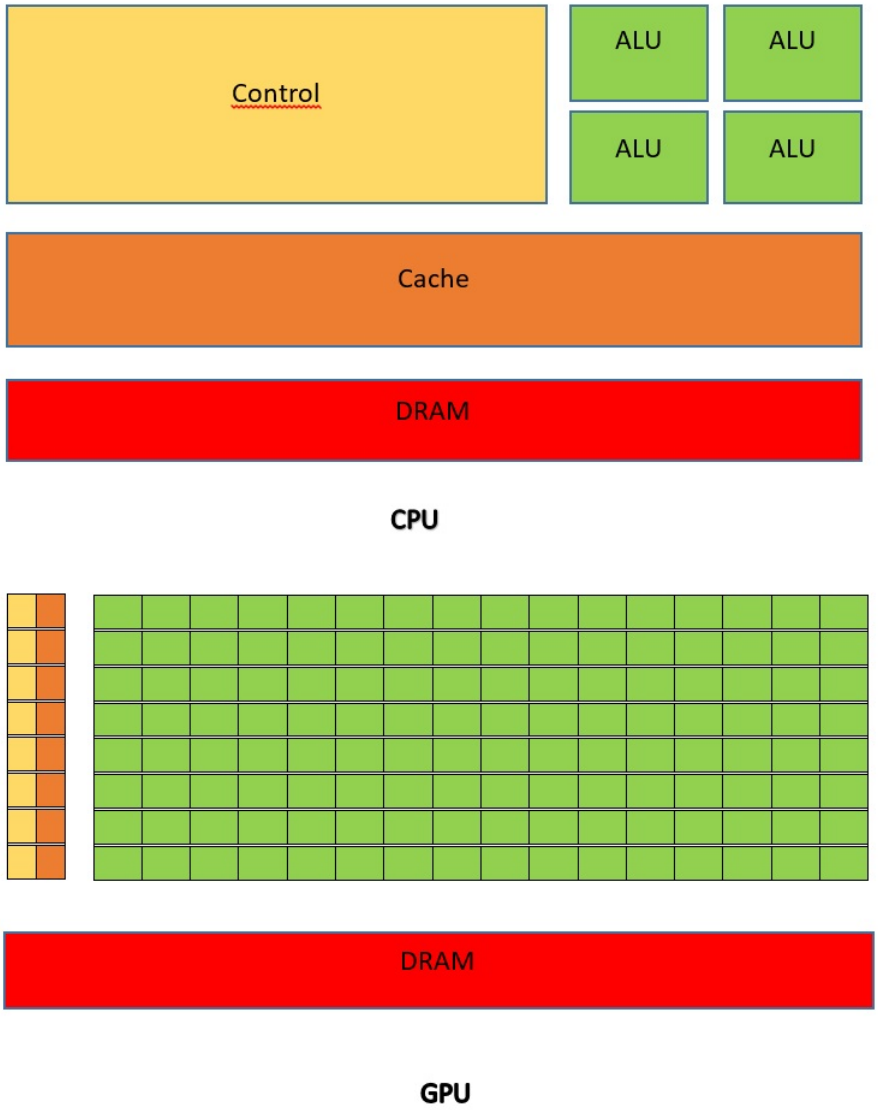

Figura 3.1: Comparação entre a arquitetura de uma CPU e de uma GPU. Retirado de [4].

interface.

CUDA é uma extensão da linguagem C, que permite ao programador definir que código será executado na CPU (chamado em CUDA de *host*), ou na GPU (chamado de *device*). Em CUDA geralmente o *host* dispara tarefas (chamadas de *kernel*) que utilizarão várias unidades de processamento da GPU em paralelo. Cada unidade de processamento da GPU executará instruções de um código para um determinado conjunto de dados. Para executar um *kernel*, o programador deve definir, além de seus parâmetros, quantas *threads* CUDA serão criadas para sua execução[27]. Por exemplo, a instrução **kernel_function<<<nBlocks,nThreads>>>(args);** chama a função **kernel_function**, criando um grid de **nBlocks** blocos, onde cada bloco possui

nThreads *threads*. Assim a função será executada em paralelo por um total de $nBlocks \times nThreads$ threads. **args** se refere aos argumentos ordinários da função **kernel_function**.

Por possuírem regiões de memória distintas, os dados que serão passados como parâmetros para a GPU, bem como os resultados da função, deverão ser copiados entre a memória principal e a memória da GPU. Essas operações de cópia podem ser custosas, dependendo das características do hardware e quantidade de dados transmitidos.

3.4.2 Streams CUDA

Streams podem ser definidas como filas de trabalho virtual na GPU usadas para operações assíncronas, ou seja, operações que não bloqueiam a CPU [27]. Elas permitem que N *kernels* independentes sejam executadas em *pipeline*.

Normalmente, as chamadas de *kernels* e as cópias entre GPUs e CPUs acontecem em uma *stream*, chamada de *stream* padrão, sendo executadas na ordem exata em que foram enviadas. No entanto, o uso de múltiplas *streams* permite que certos comandos, como os de cópias de dados, possam ser intercalados e executados simultaneamente.

3.5 Arquiteturas de GPUs

3.5.1 Tesla

Em 2006 a NVIDIA lançou a GeForce 8800, também chamada de Tesla, arquitetura baseada num vetor escalável de processadores [4].

Dentre as novidades desta arquitetura estão a troca de *pipelines* de vértices e *pixels* por processadores únicos, que executam operações com vértices, geometrias e *pixels*. Foi a primeira GPU a utilizar o processamento escalar com *threads*, eliminando, com isso, a necessidade do programador de gerenciar manualmente registradores vetoriais. Além disso foram introduzidas a memória compartilhada, a sincronização com barreira para comunicação entre *threads*, e o modelo de execução SIMT (*single-instruction, multiple-thread*).

Do ponto de vista da arquitetura, as SPAs (*streaming processor arrays*) realizam todos os cálculos programáveis da GPU, além de prover controle e gerenciamento das

threads. Nas SPAs estão os TPCs (*Texture/processor cluster*). Seu número determina o desempenho de processamento da GPU. As TPCs possuem um *geometry controler*, um SMC (*streaming multiprocessor controler*), dois SMs (*streaming multiprocessor*) e uma unidade de textura. Sua arquitetura é ilustrada na Figura 3.2.

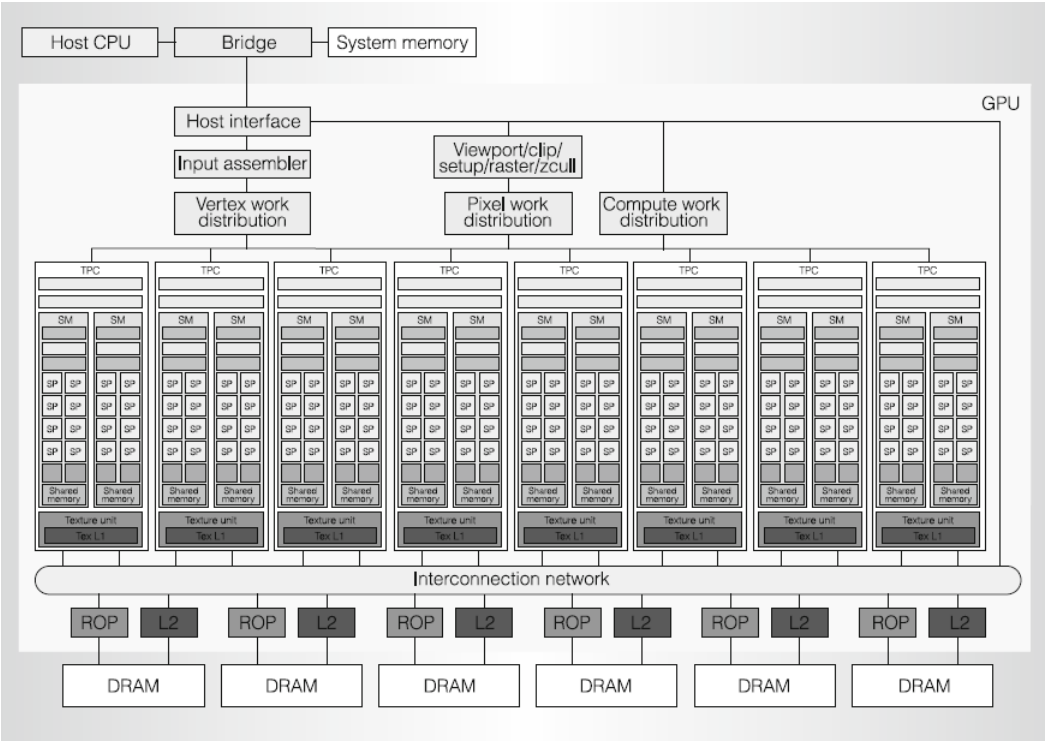

Figura 3.2: Arquitetura Tesla. Retirado de [4].

A fim de gerenciar e executar eficientemente centenas de *threads* executando diversos programas diferentes, as SMs da Tesla usam a arquitetura SIMT, que cria, gerencia, escalona e executa *threads* em grupos compostos por 32 elementos, chamados de *warp*. Cada SM pode gerenciar até 24 *warps*, totalizando 768 *threads*. Cada uma dessas *threads* executa inicialmente a mesma instrução, mas são livres para executar de forma independente as demais instruções [4].

Três tipos de memória são acessíveis para leitura e escrita na arquitetura: a memória local, pertence individualmente a cada *thread*, não possuindo compartilhamento com as demais; a memória compartilhada é uma memória de acesso de baixa latência quando *threads* cooperam entre si no mesmo SM, mais utilizada dentro dos blocos; e a memória global, que compartilha dados entre todas as *threads* da aplicação, sendo utilizada para comunicação entre *grids*.

Para aumentar a largura de banda de memória e reduzir os custos de acesso à memória,

múltiplas instruções de acesso às memórias local e global, executadas por distintas *threads* de um mesmo *warp*, podem ser agrupadas em um único acesso à memória, desde que alguns critérios sejam atendidos [4]: basicamente os acessos devem ser feitos à endereços consecutivos de memória e o endereço acessado pela *thread* 0 deve ser múltiplo de 128.

3.5.2 Fermi

A arquitetura Fermi (GF100) foi lançada em 2011 trazendo agora 3 bilhões de transistores em uma GPU. Dentre as suas principais melhorias, podem ser citados um aumento no tamanho da memória compartilhada, troca mais rápida de contexto entre *threads*, operações de acesso à memória mais rápidas, e melhora no desempenho das operações de precisão dupla. Os três espaços de endereçamento, antes separados, agora foram unificados para leitura e escrita, o que possibilitou a utilização da linguagem C++. Na linguagem C++ todas as variáveis e funções residem em objetos que são passados via ponteiros. Fermi possibilita o uso de ponteiros unificados para passar objetos em qualquer espaço de memória uma vez que sua unidade de tradução de endereços no *hardware* unifica automaticamente mapas de referência de ponteiros para os espaços de memória corretos. Algumas funcionalidades das linguagens orientadas à objetos, como herança e polimorfismo, necessitam de resolução de endereços em tempo de execução, via ponteiros, para funcionar.

A GF100 possui quatro GPCs (*graphics processor clusters*). Cada GPC possui quatro SMs que, por sua vez, possuem 32 CUDA *cores*, conforme ilustrado pela Figura 3.3.

3.5.3 Kepler

A arquitetura Kepler trazia consigo 7,1 bilhões de transistores, além da simplificação da criação de programas paralelos através de novos métodos de otimização da paralelização.

A nova SMX (*streaming multiprocessor*) possui 192 CUDA *cores* de precisão simples, 64 de precisão dupla, 32 unidades de função especial (SFU) e 32 unidades de leitura e escrita (LD/ST). O número de registradores por *threads* aumentou de 63 na arquitetura Fermi para 255. A arquitetura é ilustrada na Figura 3.4.

Um dos novos recursos presente nesta arquitetura é a *Shuffle instruction*. Nas arquiteturas anteriores, a comunicação entre *threads* na mesma *warp* requeriam um espaço de memória separado, além de operações para carregar os dados através da memória

Figura 3.3: Arquitetura Fermi.

compartilhada. Com este novo recurso *threads* podem compartilhar memória, desde que estejam num mesmo *warp* [9].

O recurso do **paralelismo dinâmico** (*Dynamic Parallelism*) permite que, num sistema integrado GPU-CPU, a GPU tenha independência para poder chamar novos *kernels*, sincronizar os resultados além de escalar esse trabalho sem envolver a CPU.

Um novo recurso, **Hyper-Q** (hiper fila), permite que múltiplas CPUs enviem trabalhos para uma única GPU ao mesmo tempo. O recurso aumenta o número total de conexões entre o *host* e a lógica do distribuidor de trabalho da GPU, permitindo 32 conexões simultâneas gerenciadas por hardware.

3.5.4 Maxwell

Em 2012 surgiu a arquitetura Maxwell (GM206), desenvolvida pela NVIDIA para reduzir em até 4 vezes o consumo de energia em relação à geração anterior [5].

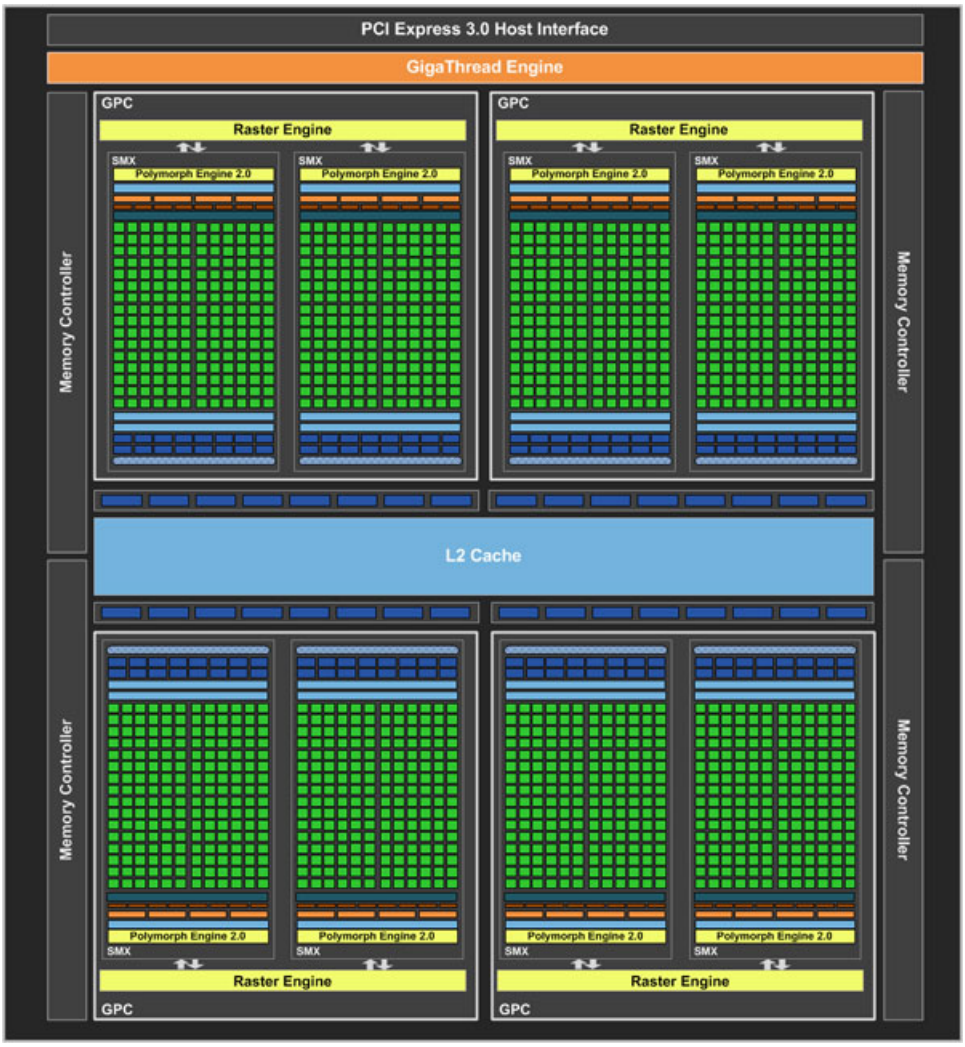

Figura 3.4: Arquitetura Kepler.

GM206 é composto por dois GPCs com oito Maxwell SMs (SMM) em cada e 2 controladores de memória. Cada SMM possui 128 CUDA *cores* e 8 unidades de textura, totalizando 1.024 CUDA *cores* e 64 unidades de textura, conforme apresentado na Figura 3.5.

Comparado com as GPUs baseadas na arquitetura Kepler, o SM do Maxwell foi reconfigurado para aumentar a eficiência. Cada SMM possui quatro escaladores de *warp*, que são capazes de despachar duas instruções a cada *clock*. Com isso foi possível reduzir computações redundantes, aumentando a eficiência, tanto do ponto de vista do desempenho, quanto do ponto de vista energético [5].

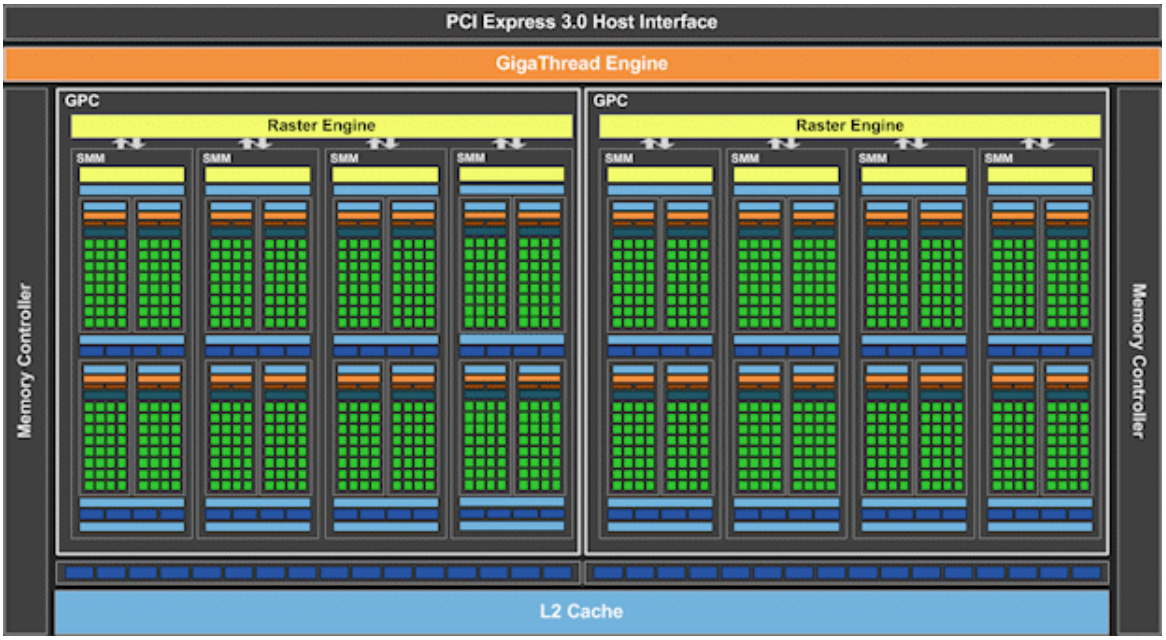

Figura 3.5: Arquitetura Maxwell. Retirado de [5].

4 MÉTODOS

Este capítulo tem como foco apresentar o método usado para verificar se o desempenho de um simulador cardíaco que executa em um ambiente composto de múltiplas GPUs (que será referenciado como multi-GPU) pode ser melhorado usando um esquema baseado em agrupamento de dados e *kernels*.

Como a proposta de agrupamento apresentada neste trabalho é uma extensão do trabalho de Barros [3], o método consiste em inicialmente avaliar o desempenho dessa versão anterior em diversos ambientes computacionais. Na sequência, uma versão do código que implementa o esquema de agrupamento de dados e *kernels* é avaliada, usando os mesmos parâmetros e nos mesmos ambientes computacionais. O ganho de desempenho é então medido para determinar se o esquema de agrupamento foi ou não efetivo em seu propósito.

Esta seção apresenta a implementação do simulador cardíaco usada como base para o desenvolvimento do trabalho. Na sequência é apresentado o esquema proposto como forma de melhorar o desempenho do simulador, bem como sua implementação. No próximo capítulo, serão apresentados os detalhes dos experimentos conduzidos para a avaliação da proposta.

4.1 Simulador Cardíaco

Simulações de larga escala resultantes de uma discretização fina de um tecido cardíaco são computacionalmente caras. Se usarmos, por exemplo, uma discretização de $8\mu m$ do Modelo Microscópico (descrito na seção 2.5) em um tecido de $1cm \times 1cm$ usando o modelo Bondarenko (BDK), que possui 41 variáveis, como modelo das células cardíacas, serão computadas um total de $1250 \times 1250 \times 41 = 64.062.500$ variáveis a cada passo de tempo. Para simular apenas $100ms$ de atividade cardíaca, quando $\Delta t_0 = 0,0001ms$, os 64 milhões de valores calculados no sistema de EDOs devem ser computados um milhão de vezes, enquanto que as EDPs, com um milhão e meio de valores, devem ser calculadas 10 mil vezes (o número distinto de passos para a resolução da EDO e EDP devem-se aos valores diferentes usados para os passos de tempo). A maior motivação de se usar

computação paralela em simulações cardíacas vem do fato de que a simulação sequencial de um tecido, nas condições citadas anteriormente, pode demorar 546.507 segundos[2]. São mais de 6 dias de cálculos para simular apenas $10ms$ em um tecido de $1cm^2$, o que tornaria a simulação de um coração inteiro inviável, caso utilizado um código sequencial. Na versão utilizada como base para este trabalho, duas ferramentas de paralelização foram empregadas para melhorar o desempenho: a interface MPI e CUDA, usadas simultaneamente para a execução do código do simulador em um *cluster* de computadores equipado com múltiplas GPUs.

De fato, MPI foi utilizado em vários trabalhos anteriores para a paralelização de modelos fisiológicos [8, 29, 30]. Utilizando 64 processos, inclusive, já foi reportado por um dos trabalhos a obtenção de *speedups* de 60,4 para um tecido de $0,5cm \times 0,5cm$ e de 61,2 para um tecido de $1cm^2$[3].

O uso em conjunto das duas ferramentas citadas anteriormente se provou muito eficiente. Utilizando GPUs na resolução das EDOs e MPI na resolução das EDPs o tempo total de resolução com um tecido de $0,5cm \times 0,5cm$ foi de $401,8$ segundos em média, obtendo um *speedup* de 343, enquanto que com um tecido de $1cm^2$ o tempo total foi de 1.302 segundos e o *speedup* foi de 420 [2].

A implementação dessa versão que utiliza simultaneamente CPUs e GPUs se baseia em MPI e CUDA. Como o número de núcleos CPUs disponíveis em cada máquina, em geral, é muito superior ao número de GPUs, mais de um processo MPI tenta executar *kernels* CUDA em uma mesma GPU.

É possível que esta configuração, em que múltiplas CPUs invocam vários *kernels* para serem executados em uma mesma GPU, possa levar a um atraso na execução do código sob responsabilidade das placas gráficas. Estes processos MPI tentarão acessar as GPUs aproximadamente ao mesmo tempo, possivelmente causando diversas sobrecargas. Primeiramente, os diversos processos MPI realizarão chamadas de sistema simultaneamente, o que pode sobrecarregar o Sistema Operacional. Os simultâneos acessos às GPUs também poderão causar uma sobrecarga no barramento PCI, pois diversas transferências de dados ocorrerão aproximadamente ao mesmo tempo, e deverão ser finalizadas antes que os *kernels* possam ser executados. Além disso o próprio *hardware* das GPUs poderá ser sobrecarregado, na medida que o seu escalonador/*dispatcher* terá de lidar com várias invocações simultâneas de *kernels*.

Ao longo dos estudos realizados no escopo deste trabalho, duas alternativas foram discutidas como forma de resolver, ou mesmo minimizar estes problemas: o uso de *streams* CUDA e o uso do agrupamento de dados e *kernels*. O uso de *streams* CUDA para reduzir o tempo de execução do simulador se baseia na observação de que transferências de dados poderiam ser feitas em paralelo com a execução dos *kernels*. O agrupamento dos dados, por sua vez, se baseia na hipótese de que uma única chamada ao SO, acesso ao barramento e invocação de *kernel* CUDA é possivelmente menos custosa do que múltiplas operações, e de implementação simples. A ideia da implementação é que um dos vários processos MPI que acessem uma mesma GPU seja escolhido para centralizar as invocações de *kernels* CUDA. Assim esse processo será o único a se comunicar com esta GPU, de modo que uma única cópia de dados ocorrerá para a GPU, bem como a invocação de um único *kernel* CUDA e uma única cópia de resultados de volta para a CPU. Naturalmente que a implementação implica em custos adicionais, por exemplo, imporia um tempo extra para copiar os dados de todos os processos MPI para um único processo antes da invocação do *kernel*, bem como para a distribuição dos resultados colhidos por este único processo MPI para todos os demais processos.

Esta segunda alternativa será chamada de método dos *Kernels* Agregados, ou KA, e será comparada com a implementação original do simulador, aqui chamado de método dos *kernels* não agregados, ou KNA. Ambos os métodos serão empregados em um simulador cardíaco que utiliza: a) o modelo microscópico descrito na Seção 2.5 para descrever o tecido cardíaco, b) o modelo monodomínio para modelar a comunicação elétrica entre as células, c) a discretização descrita na Seção 2.6 para simular a passagem de tempo, e d) o modelo BDK, descrito na Seção 2.3, para simular matematicamente o comportamento do potencial de ação em cada célula.

4.2 Implementação do Método KNA

4.2.1 Implementação Paralela

A biblioteca MPI foi usada para implementar a resolução do sistema de EDPs descrito pela equação (2.16) em um *cluster* de CPUs. Essa implementação paralela usa a biblioteca PETSc para automatizar os processos da criação e utilização das matrizes e vetores, além da solução do sistema de equações lineares. O método do gradiente conjugado, pré-

condicionado com fatoração LU incompleta, foi utilizado para resolver o sistema linear associado à discretização das EDPs do modelo de monodomínio. Mais detalhes desta implementação paralela podem ser encontrados nas referências da versão que serviu de base para este trabalho [2, 3].

O método explícito de Euler foi usado para resolver o sistema não-linear de EDOs da equação (2.17), um problema embaraçosamente paralelo, uma vez que não existe dependência alguma entre as soluções dos diferente sistemas de EDOs em cada volume finito $Vol_{i,j}$. Por isso, é bem simples a implementação da versão paralela do código: cada processo MPI é responsável pela computação de uma fração N_P do número total de volumes da simulação, sendo N_P o número de processos envolvidos na computação.

4.2.2 Implementação Multi-GPU

Na implementação Multi-GPU [2, 3], o sistema linear associado à discretização das EDPs do modelo monodomínio é resolvido da mesma forma que na implementação em *cluster*, ou seja, usando a biblioteca PETSc para implementar o método do gradiente conjugado em paralelo, pré-condicionado com ILU(0) (com bloco de Jacobi em paralelo). A solução do sistema de EDOs da equação (2.17), porém, é acelerada utilizando múltiplas GPUs.

Dois *kernels* foram desenvolvidos para resolver os sistemas de EDOs relacionados ao modelo BDK. O primeiro *kernel* é responsável por definir as condições iniciais do sistema de EDOs, enquanto que o segundo integra os sistemas a cada passo de tempo.

Ambas implementações dos *kernels* foram otimizadas de diversas maneiras diferentes. As variáveis de estado de M células cardíacas foram armazenadas num vetor chamado SV de tamanho $M \times N_{eq}$ onde N_{eq} é o número de equações diferenciais do modelo iônico. Como foi utilizado o modelo BDK, N_{eq}, neste caso, é 41. SV foi organizado de forma que as primeiras M entradas correspondem à primeira variável de estado das M células, seguidos pelas segundas variáveis de estado, e assim por diante. Além disso, para todos os modelos iônicos, as primeiras M entradas do vetor SV correspondem ao potencial transmembrânico V. Durante a solução dos sistemas de EDPs, logo após a integração dos sistemas de EDOs, o potencial transmembrânico de cada nó deve ser passado para o PETSc, que irá solucioná-lo. Como as M primeiras entradas do vetor SV correspondem ao potencial transmembrânico de cada nó, a organização escolhida para o vetor SV torna esta uma tarefa simples, além de evitar transações de memória desnecessárias entre a

GPU e a CPU, melhorando o desempenho.

A rotina *cudaMallocPitch* da API do CUDA foi escolhida para alocar o vetor SV na memória global da GPU. Esta rotina garante que os endereços de memória correspondentes de qualquer dada linha continuarão a satisfazer os requisitos de alinhamento para as operações de coalescência executadas pelo *hardware*, o que gera impacto positivo no desempenho. Com isso, num conjunto de n *threads* numa coalescência estrita, uma *thread* j deve acessar $u[j]$ se a *thread* 0 acessou $u[0]$. Todo acesso a dados tem distância fixa de n elementos. Portanto, no primeiro *kernel*, para definir as condições iniciais, cada *thread* define o valor de todas as variáveis de estado. O segundo *kernel* opera de forma similar, onde cada *thread* computa e atualiza suas variáveis de estado, escrevendo o resultado na posição da memória correspondente às suas variáveis.

Na paralelização, o domínio foi dividido em N_p subdomínios sem sobreposição onde N_p é o número de processos MPI. A solução de cada EDP é implementada via PETSc, com cada núcleo de processamento p responsável por atualizar as variáveis associadas ao subdomínio T_p. No ambiente computacional onde os testes foram realizados, cada máquina ou nó possui mais núcleos de CPU, no caso 8, do que de GPU, no caso 2. Por isso, para resolver as EDOs, cada dispositivo GPU foi responsável por processar mais de uma tarefa. As tarefas são distribuídas entre as GPUs utilizando um esquema de *round-robin*. A Figura 4.1 ilustra a decomposição do domínio para um problema com $N_p = 16$ em duas máquinas com 8 núcleos e 2 dispositivos de GPU cada. Quatro tarefas serão atribuídas para cada dispositivo GPU, a saber, no nó 0, a GPU 0 processará as tarefas T_1, T_3, T_5 e T_7, enquanto que a GPU 1 processará as tarefas T_2, T_4, T_6 e T_8. A implementação deste método se encontra ilustrado na Figura 4.2.

4.3 Uso de Transferências Assíncronas

Conforme mencionado anteriormente, o uso de *streams* foi proposto para sobrepor as operações de transferências de dados entre CPUs e GPUs com a execução dos *kernels* [28]. Caso a execução dos *kernels* seja mais longa do que o tempo necessário para realizar as transferências de dados, do ponto de vista prático a sobreposição das operações "esconde"o tempo de cópia.

O método KNA faz a cópia dos dados de forma separada da chamada do *kernel* que

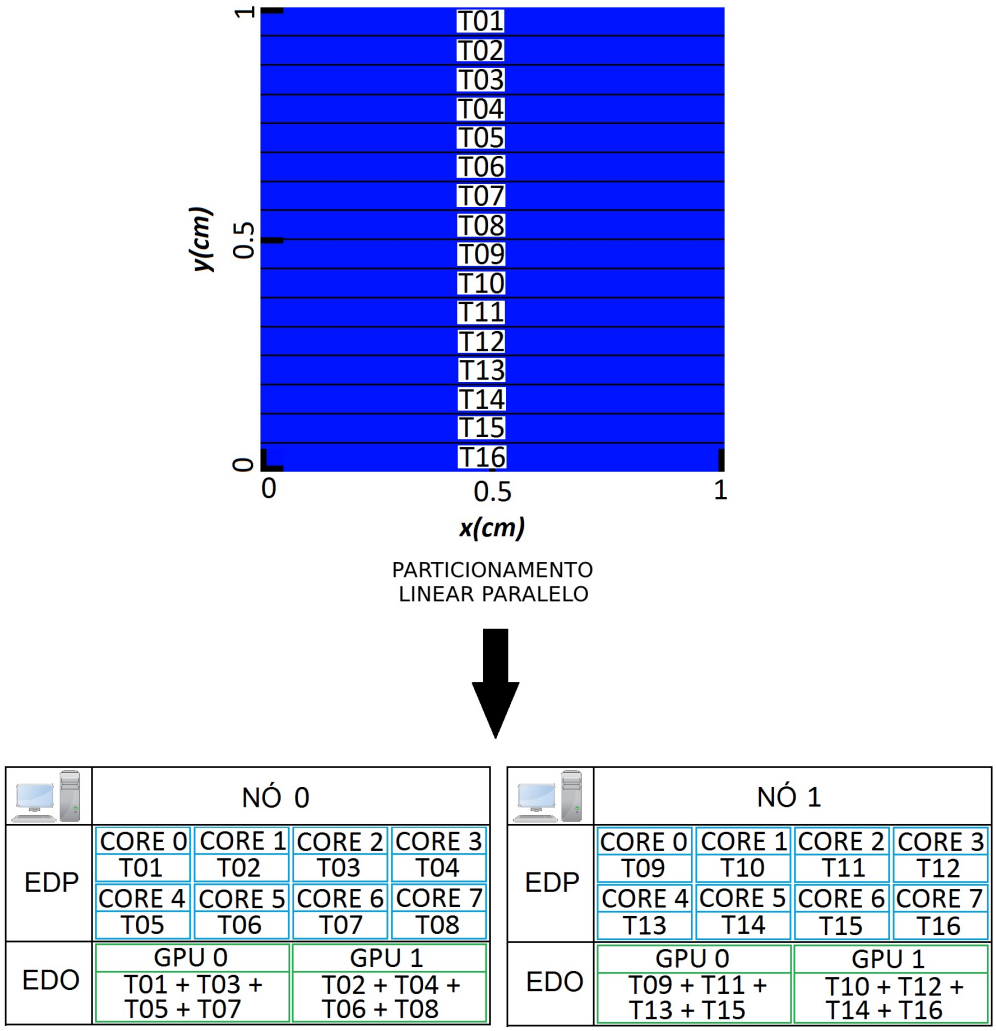

Figura 4.1: Decomposição paralela e linear de um tecido. Exemplo do caso de dois nós com 8 núcleos e 2 GPUs cada, totalizando 16 núcleos CPU e 4 GPUs. As tarefas atribuídas às CPUs foram distribuídas entre as GPUs utilizando-se o método *Round-Robin*. Adaptado de [2].

resolverá as EDOs. Com a utilização de *streams*, 2 ou 4 *kernels* serão chamados cuja função seja a cópia para a GPU, resolução das EDPs referentes à sua parte corresponde da malha e, logo depois, cópia dos dados para a CPU. A vantagem da utilização de transferências assíncronas é o *kernel* poder começar a resolução das EDOs imediatamente após sua cópia de dados ser finalizada, não tendo que esperar que as demais *streams* terminem suas próprias cópias.

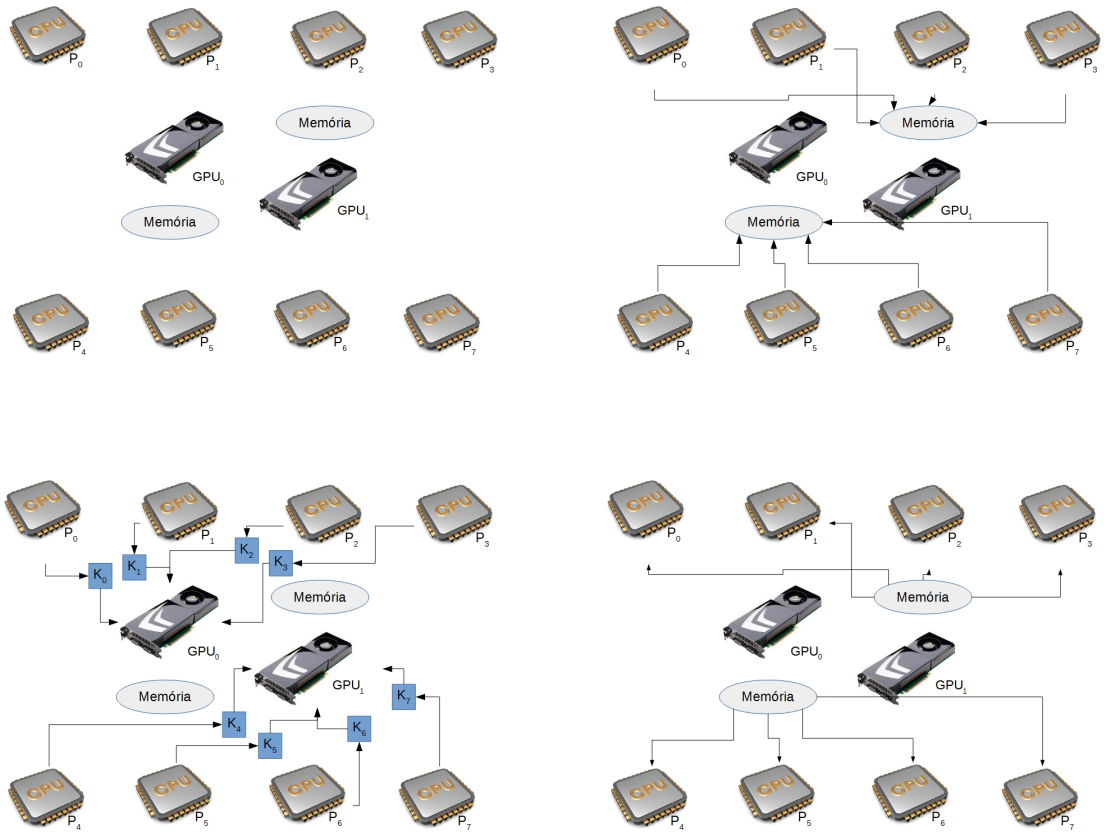

Figura 4.2: Ilustração do método multi-GPU. Todos os processos enviam seus dados para a memória da GPU. Os processos executam os *kernels* nas unidades gráficas e, logo depois, copiam os dados da memória das GPUs.

4.4 Implementação do Método KA

Como dito anteriormente, o modelo KA foi proposto para solucionar os problemas decorrentes dos acessos simultâneos dos vários processos MPI a uma mesma GPU, o que possivelmente, pode causar sobrecargas no Sistema Operacional, no barramento e até mesmo no *hardware* das GPUs.

Para implementar o método KA, os processos foram separados em grupos. O número de grupos é igual ao número de GPUs disponíveis em cada máquina, e o número de processos por grupo é igual ao número de núcleos dividido pelo número de GPUs. Caso a divisão não seja exata, os grupos poderão ter números distintos de participantes. Um dos processos de cada grupo, cujo identificador seja igual à zero (chamado processo 0), agrupará os dados dos demais elementos do mesmo grupo e os enviará para a GPU. Logo depois o processo 0 invocará um único *kernel*, ao invés dos múltiplos *kernels* que eram

chamados pelos demais processos do grupo. Do ponto de vista lógico, pode-se falar que as várias invocações de *kernels* foram agregadas em uma única chamada. Por fim, o processo 0 copiará da GPU os resultados de sua computação, e os enviará para os demais processos MPI. Esta implementação se encontra ilustrada na Figura 4.3.

Para a criação dos grupos foi utilizada a rotina *MPI_Comm_split*. Também fez-se necessário criar um novo comunicador, usando a rotina *MPI_Comm*. Dentro dos grupos um novo *size* e um novo *rank* serão definidos. As rotinas *MPI_Comm_rank* e *MPI_Comm_size* serão chamadas uma segunda vez, porém para o novo comunicador. Este processo é ilustrado no Algoritmo 2. É importante notar que o inteiro *membershipkey* definirá em que grupo ficará cada processo. No caso do Algoritmo 2, os processos foram separados usando um esquema *round-robin*.

Os processos 0 de cada grupo receberão os dados dos demais processos de seus grupos através da rotina *MPI_Gatherv*, os enviando para suas respectivas GPUs. O primeiro *kernel*, responsável por definir as condições iniciais, será então invocado por cada processo 0.

Antes da resolução das EDOs, será chamada a rotina *VecGetArray* do PETSc, rotina que, basicamente, retorna um ponteiro para a porção à qual o vetor paralelo deste processo é responsável. *MPI_Gatherv* é utilizada novamente para agregar as informações dos vetores dos processos no processo 0 do grupo, que atualizará o vetor SV na GPU e logo depois executará o segundo *kernel*, cuja função é resolver as EDOs. Após a resolução das EDOs o processo 0 terá seu vetor atualizado pelo vetor SV, atualizando, logo em seguida, os vetores dos demais processos através da rotina *MPI_Scatterv*.

Se tivermos 8 processos e 2 GPUs, no método KNA cada GPU receberá 4 atualizações diferentes em seu SV, terá que executar 4 *kernels* diferentes e depois será consultada por 4 processos na CPU sobre os resultados. Apesar do número de EDOs a serem resolvidas ser o mesmo no método KA, agora cada GPU recebe apenas uma atualização em seu SV, executa apenas um *kernel* e é consultado por apenas um processo na CPU sobre os resultados. A diferença entre as duas implementações pode ser vista nos pseudo-códigos 3 e 4.

Algoritmo 2: Separação em grupos de comunicação

```
1: #include <studio.h>
2: #include <string.h>
3: #include <mpi.h>
4:
5: const int num_groups = 2;
6:
7: int main(void){
8:     int     comm_sz;     /* Número de processos */
9:     int     my_rank;     /* Rank do processo atual */
10:
11:    MPI_Init(NULL, NULL);
12:    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
13:    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
14:
15:    MPI_Comm myComm;
16:    int     local_size;     /* Número de processos dentro do grupo */
17:    int     local_rank;     /* Rank do processo atual dentro do grupo */
18:
19:    int membershipkey = my_rank % num_groups;
20:
21:    MPI_Comm_split(MPI_COMM_WORLD, membershipkey, my_rank,
   &myComm);
22:
23:    MPI_Comm_size(myComm, &local_size);
24:    MPI_Comm_rank(myComm, &local_rank);
25:
26:       ...
27:
28:    MPI_Comm_free(&myComm);
29:    MPI_Finalize();
30: }
```

Algoritmo 3: Implementação do método KNA

```
   main
2:   ... inicializa computação preliminar ...
     ... define subdomínio no qual cada processo é responsável ...
4:   ... define os estímulos e os envia para a GPU ...
     while t ≤ final_time do
6:     ... atualiza vetor de estado na GPU ...
       ... resolve as EDOs na GPU ...
8:     ... atualiza vetor na CPU ...
       ... resolve EDPs na CPU ...
10:  end-while
   end-main
```

Figura 4.3: Ilustração do método KA. Os demais processos do grupo enviam seus dados para o processo 0 que, por sua vez, faz a cópia dos dados na memória da GPU. O processo 0 executa o *kernel* e, logo depois, copia o resultado da memória da GPU. Por fim os dados são espalhados entre os demais processos do grupo.

Algoritmo 4: Implementação do método KA

```
main
  ... inicializa computação preliminar ...
  ... define subdomínio no qual cada processo é responsável ...
  ... divide os processos em grupos ...
  ... define o estímulo ...
  if local_rank = 0 do
    ... junta os estímulos dos processos do grupo e manda-os para a GPU ...
  end-if
  while t ≤ final_time do
    if local_rank = 0 do
      ... junta os vetores dos processos do grupo e os envia-os para a GPU ...
      ... resolve EDOs na GPU ...
      ... distribui os resultados entre os processos do grupo na CPU ...
    end-if
    ... resolve EDPs na CPU ...
  end-while
end-main
```

5 EXPERIMENTOS E RESULTADOS

Este capítulo apresenta os resultados dos experimentos realizados para a avaliação de desempenho do modelo KA em comparação com o modelo KNA. Os testes foram realizados em distintos ambientes experimentais equipados com GPUs com arquiteturas Tesla, Kepler e Maxwell, esta última com suporte a tecnologia *Hyper-Q*. Também avaliou-se o impacto da utilização de *streams* no desempenho do simulador.

5.1 Ambiente Experimental

Para avaliar o desempenho do método descrito neste trabalho, três ambientes computacionais distintos foram utilizados.

O primeiro conjunto de testes foi realizado em dois nós de um *cluster*; cada máquina é equipada com dois processadores quad-core Intel E5620, com 12 GB de memória, duas GPUs Tesla C1060 (GT200B) com 240 CUDA *cores* e 4 GB de memória global. O sistema operacional Linux 2.6.32, driver CUDA versão 6.0, OpenMPI versão 1.6.2, nvcc 6.0 e gcc versão 4.4.7 foram usados nos experimentos. Este ambiente experimental será referenciado como arquitetura Tesla.

O segundo ambiente experimental é composto de um computador equipado com dois processadores quad-core Intel Core i7 4770, com 8 GB de memória, duas GPUs Kepler GTX 760 (GK104) com 1.152 CUDA *cores* e 2 GB de memória global. A máquina executa o sistema operacional Linux 4.4.33. O driver CUDA versão 7.5, OpenMPI versão 1.10.4, nvcc 7.5 e gcc versão 6.2.1 foram usados nos experimentos. Este ambiente experimental será referenciado como arquitetura Kepler.

O último ambiente experimental é composto de um computador equipado com um processador quad-core Intel Core i7 6700K, com 32 GB de memória, uma GPU Maxwell GTX 960 com 4 GB de memória global. Este ambiente experimental será referenciado como arquitetura Maxwell.

Todas as versões do simulador foram executadas pelo menos 5 vezes em cada

ambiente. Todas as execuções foram feitas de modo exclusivo, ou seja, nenhum outro processo executava simultaneamente em cada máquina, naturalmente excetuando-se aqueles relacionados ao sistema operacional. Foram observados desvios padrão abaixo de 2%. O tempo médio de execução, em segundos, foi utilizado para calcular o desempenho do método KA em relação ao KNA.

Foi utilizado na simulação o modelo microscópico com discretização espacial de $8\mu m$, num tecido cardíaco de $0,5cm \times 0,5cm$, estimulado no centro e executado por $10ms$. A execução por $10ms$ não é o suficiente para se obter um resultado numérico satisfatório para estudos mais elaborados, mas é capaz de avaliar se os resultados numéricos dos dois métodos são iguais e avaliar o tempo em cada um deles. A Figura 5.1 mostra que neste instante de tempo a propagação alcançou pouco mais da metade do tecido. Esta simulação, porém, é o suficiente para se obter o tempo de computação das EDOs e das EDPs com a finalidade de comparação entre os métodos KNA e KA. Os demais valores utilizados na simulação são apresentados na Tabela 5.1. Estes valores são idênticos aos utilizados em um estudo anterior [3].

Para a solução das EDOs, tanto para o código sequencial quanto para o paralelo (CUDA), foi usada precisão simples, seguindo-se o mesmo método do estudo anterior [3]. Para a solução das EDPs foi usada precisão dupla. Já foi anteriormente mostrado [31] que, no caso de simulações de monodomínio, o uso de precisão simples em CUDA não afeta a precisão numérica da solução.

Tabela 5.1: Valores dos parâmetros usados nas simulações.

Parâmetro	Valor
σ_c	0,4 $\mu S/\mu m$
G_p	0,5 μS
G_i	0,33 μS
G_c	0,062 μS
β	0,14 cm^{-1}
C_m	1,0 $\mu F/cm^2$
Δt_p	0,01 ms
Δt_o	0,0001 ms

5.2 Resultados numéricos

A Figura 5.1 mostra a propagação do estímulo central em um tecido de $0,5cm \times 0,5cm$ para diferentes instantes de tempo. Como esperado, macroscopicamente, a propagação aparenta ser suave e contínua. A propagação do potencial de ação é mais rápida no sentido das fibras do que quando as atravessa. Isso faz com que o pulso seja mais lento no sentido transversal do que no sentido longitudinal e, por isso, faz sentido que o pulso se propague de forma elipsoidal sobre o tecido. Os resultados numéricos são idênticos em todas as simulações, independente do método e da GPU utilizada.

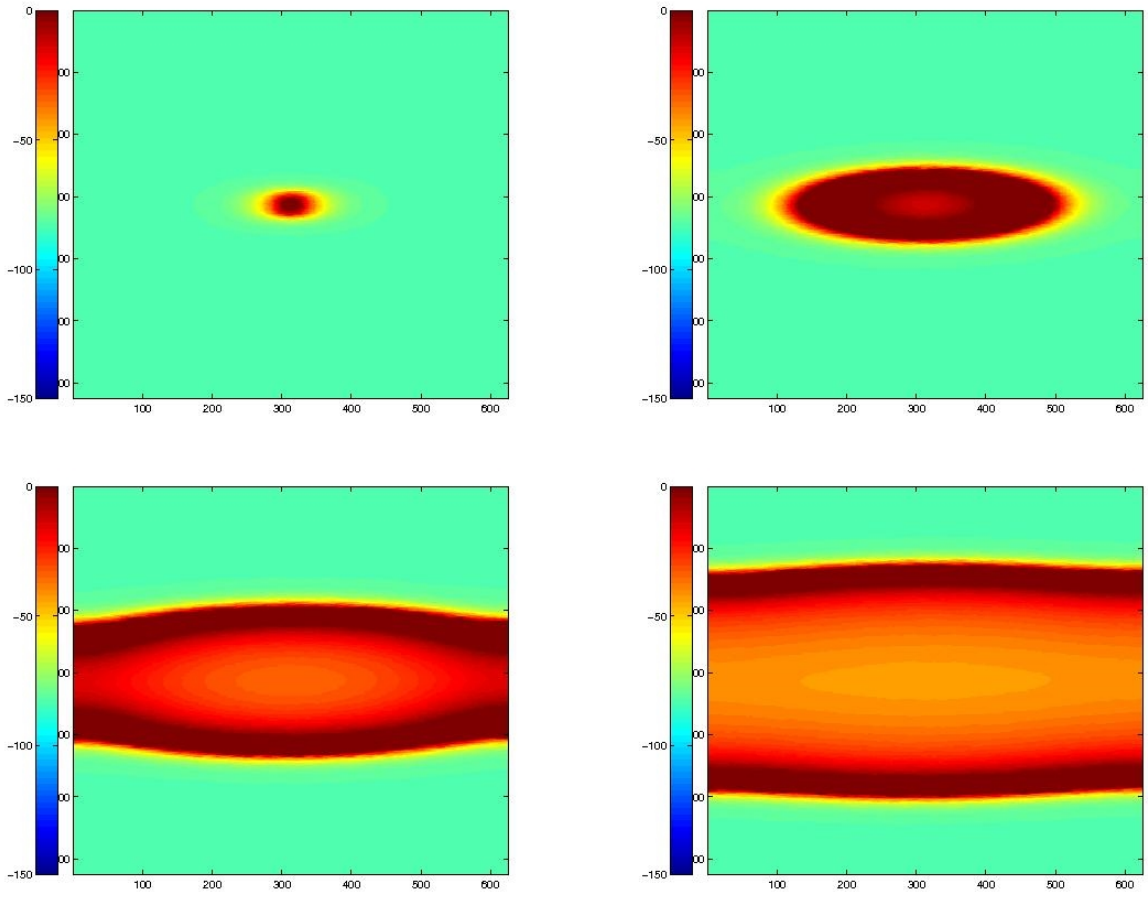

Figura 5.1: Potencial transmembrânico após 1ms (topo-esquerda), 5ms (topo-direita), 10ms (baixo-esquerda) e 15ms (baixo-direita) do estímulo central num tecido de $0,5$cm $\times 0,5$cm.

5.3 Experimentos com transferências assíncronas

Os testes utilizando transferências assíncronas foram feitos na arquitetura Tesla. Três configurações foram testadas: 1 *stream*, 2 *streams* e 4 *streams*, todas com 2 GPUs e 8 CPUs e usando o método KNA. Os resultados são apresentados na Tabela 5.2.

Tabela 5.2: Tempo de execução dos cálculos das EDOs, EDPs e tempo total de execução usando *streams*. Todos os tempos estão em segundos.

Número de streams	Número de GPUs	Número de CPUs	Tempo EDO	Tempo EDP	Tempo Total
1	2	8	806,1	2104,9	2915,6
2	2	8	833,2	2102,9	2940,4
4	2	8	917,5	2104,5	3030,2

Como foi dito na Seção 3.4.2, as chamadas de *kernels* já possuem um *stream* padrão, por isso o caso com um *stream* é equivalente ao modelo KNA. Pode-se verificar que de fato os mesmos são equivalentes ao se comparar o tempo de resolução das EDOs das Tabelas 5.2 e 5.4 (linha com 2 GPUs, 8 CPUs e método KNA). Os resultados ainda mostram que o método foi mal-sucedido pois verifica-se um aumento no tempo de execução a medida que o número de *streams* aumenta. O desempenho do método tem uma piora de 3% quando o número de *streams* sobe para 2 e de 14% quando o número sobe para 4. Verificou-se ainda, experimentalmente, que o uso de *streams* traria poucos ganhos no desempenho do simulador. Foi medido o tempo para transferência de dados (não mostrado na tabela), que representa menos de 0,2% do tempo total de execução. Após os resultados desses experimentos, o uso de *streams* foi abandonado neste trabalho.

5.4 Avaliação de Desempenho na arquitetura Tesla

A avaliação de desempenho na arquitetura Tesla, tanto para o método KNA quanto para o KA, foi feita usando 11 configurações diferentes de *hardware*, apresentadas na Tabela 5.3. Para cada esquema coletamos os tempos de computação das EDOs, das EDPs e o tempo total de execução, além dos tempos de comunicação entre as GPUs e as CPUs. Os resultados são apresentados na Tabela 5.4. Os tempos de comunicação entre GPUs e CPUs não são apresentados por representarem menos que $0,2\%$ do tempo total de

execução. O desvio padrão apresentado nos tempos de computação das EDOs foi abaixo de $0,4\%$ enquanto que nos tempos de computação das EDPs foi abaixo de $1,6\%$.

Tabela 5.3: Configurações de *Hardware* usadas nos experimentos.

Número de Nós	Número de GPUs	Número de CPUs
1	1	1, 2, 4 e 8
1	2	2, 4 e 8
2	2	8 e 16
2	4	4, 8 e 16

Como pode ser observado, os tempos de execução são praticamente idênticos em ambos os métodos para o caso em que foram utilizadas apenas uma GPU e uma CPU, o que era o comportamento esperado, uma vez que, nesta configuração, existe apenas um *kernel* a ser executado na GPU. O mesmo acontece quando temos 2 GPUs e 2 CPUs e quando temos 4 GPUs e 4 CPUs. Analogamente ao caso de 1 GPU e 1 CPU, em todas essas configurações existe apenas um *kernel* para cada GPU. Com isso cada grupo fica com apenas um processo MPI e dado algum será agregado por isso, nestes casos, os esquemas são equivalentes.

Os tempos de computação das EDPs também são praticamente os mesmos, o que também é esperado, uma vez que sua resolução é feita pelas CPUs, onde não houve qualquer mudança no código.

É possível notar a diferença à partir do momento em que o número de CPUs aumenta e número de GPUs se mantém constante. O código do método KA mantém o mesmo tempo para o cálculo das EDOs do caso em que o número de GPUs e CPUs é igual, enquanto que este tempo aumenta no código KNA. Temos, por exemplo, no caso de apenas uma GPU, um tempo de resolução das EDOs entre 1196.3 e 1196.7 segundos no método KA, enquanto as quantidade de CPUs aumentam de uma para duas, quatro e oito. Para o método KNA, entretanto, os tempos são de 1196.4, 1264.6, 1360.8 e 1620.2 para uma, duas, quatro e oito CPUs, respectivamente.

Observe que o tempo de solução das EDOs no código KA é reduzido linearmente com a quantidade de GPUs, 1196 segundos para uma GPU, 588 segundos para 2 GPUs e 293 para 4 GPUs. Isto ocorre no método KNA apenas quando o número de CPUs e GPUs é igual.

Tabela 5.4: Tempo médio de execução dos cálculos das EDOs, EDPs e tempo médio total de execução nos dois códigos usando as distintas configurações de *hardware*. Todos os tempos estão em segundos.

Número de GPUs	Número de CPUs	Método	Tempo EDO	Tempo EDP	Tempo Total
1	1	KNA	1196,4	3815,2	5013,9
1	1	KA	1196,7	3801,9	5002,2
1	2	KNA	1264,6	3011,5	4279,6
1	2	KA	**1196,3**	3003,7	4206,3
1	4	KNA	1360,8	2231,6	3597,9
1	4	KA	**1196,5**	2273,5	3477,1
1	8	KNA	1620,2	2057,1	3685,4
1	8	KA	**1196,5**	2095,5	3299,9
2	2	KNA	587,6	3014	3604,3
2	2	KA	588,8	2995,6	3587,9
2	4	KNA	680,4	2212,0	2897,0
2	4	KA	**587,3**	2272,3	2865,3
2	8	KNA	808,4	2058,8	2873,4
2	8	KA	**587,8**	2074,4	2668,9
2	16	KNA	1070,6	1471,4	2550,6
2	16	KA	**587,5**	1448,2	2042,2
4	4	KNA	293,6	1668,9	1964,9
4	4	KA	293,3	1670,8	1966,8
4	8	KNA	405,5	1537,1	1947,6
4	8	KA	**293,3**	1559,8	1858,1
4	16	KNA	534,9	1464,1	2005,6
4	16	KA	**293,5**	1459,7	1760,3

Os melhores resultados obtidos neste experimento foram no caso com 4 GPUs e 16 CPUs, onde o ganho de desempenho relativo à resolução das EDOs e o ganho relativo à resolução total foram de 100% e 17% respectivamente, e no caso com 2 GPUs e 16 CPUs onde os ganhos de desempenho relativos à resolução das EDOs e ao tempo total foram, respectivamente, de 82% e 25% conforme aponta a Figura 5.2. O ganho de desempenho relativo ao tempo total tende a diminuir conforme aumentamos o número de GPUs pois esta resolução fica cada vez mais rápida em relação à resolução das CPUs, levando à conclusão que este método seria melhor aproveitado em um código que faz maior uso das

GPUs que das CPUs.

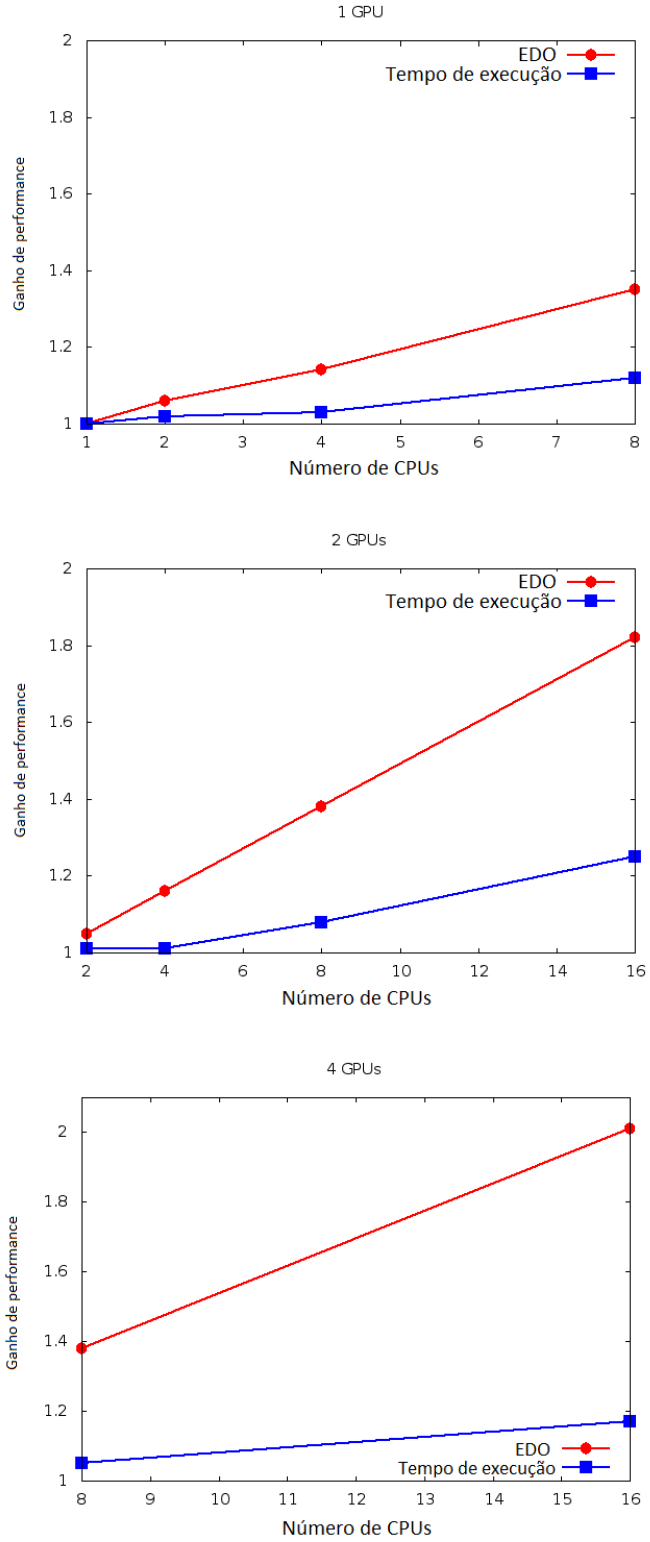

Figura 5.2: Ganho de desempenho obtido ao se comparar o código KA com o código KNA. As configurações foram agrupadas de acordo com o número de GPUs: 1 GPU (topo), 2 GPUs (meio) e 4 GPUs (baixo).

5.5 Avaliação de Desempenho nas Arquiteturas Kepler e Maxwell

O objetivo dos testes realizados em computadores com arquitetura Kepler e Maxwell foi o de avaliar o método proposto neste trabalho em arquiteturas mais atuais. A arquitetura Maxwell possui a tecnologia *Hyper-Q*, capaz de agrupar os dados de processos da CPU de maneira automática, teoricamente tornando o método KA dispensável.

Diferentemente dos testes na placa Tesla, estes testes não apresentaram uma diferença significativa nos tempos de resolução das EDOs.

A T abela 5.5 mostra que na arquitetura Kepler foi obtido um ganho de 10% no tempo de resolução das EDOs, quando são usadas uma GPU e 4 CPUS, e um ganho de 7%, no caso de 2 GPUs e 4 CPUs, o que representou em ambos os casos um ganho de quase 5% no tempo total de execução. É possível notar ainda que o tempo de resolução das EDOs, no método KNA, sofreu aumentos de 6% e 7%, respectivamente para 1 e 2 GPUs, quando o número de processos em CPU subiu de 2 para 4.

Tabela 5.5: Tempo de execução dos cálculos das EDOs, EDPs e tempo total de execução nos dois métodos na arquitetura Kepler. Todos os tempos estão em segundos.

Número de GPUs	Número de CPUs	Método	Tempo EDO	Tempo EDP	Tempo Total
1	2	KNA	1022,7	1366,9	2393
1	2	KA	988,9	1345,1	2338,7
1	4	KNA	1090,2	912,7	2006,4
1	4	KA	988,9	918,3	1911,9
2	2	KNA	513,5	1357,3	1874,2
2	2	KA	513,1	1342,9	1860,2
2	4	KNA	549,9	914,1	1500,5
2	4	KA	513	914,1	1432,6

Conforme esperado, nos testes no computador com arquitetura Maxwell, que possui suporte a *Hyper-Q*, não houve ganho algum na resolução das EDOs. Por outro lado, não há perda de desempenho para a implementação do método KA, implicando que seu uso não gera perdas significativas de desempenho em computadores com arquiteturas mais novas.

Tabela 5.6: Tempo de execução dos cálculos das EDOs nos dois métodos em um computador com apenas uma placa com arquitetura Maxwell. Todos os tempos estão em segundos.

Número de GPUs	Número de CPUs	Método	Tempo EDO	Tempo EDP	Tempo Total
1	2	KNA	2128,6	1282,2	3414,6
1	2	KA	2125,7	1325,1	3454,5
1	4	KNA	2129,4	792,8	2926,2
1	4	KA	2128,5	837,8	2970,2

6 CONCLUSÕES E TRABALHOS FUTUROS

O principal objetivo desta dissertação foi o de melhorar o desempenho de um simulador cardíaco que executa em ambientes compostos de múltiplas GPUs, usando para isso o agrupamento de dados e de *kernels*. A versão multi-GPU do simulador cardíaco foi projetada para executar em um ambiente de memória distribuída composto de vários nós, onde por sua vez cada nó é composto por múltiplos núcleos CPUs e múltiplas GPUs. Tipicamente o número de núcleos CPUs é superior ao número de GPUs. Os núcleos CPUs são utilizados no simulador para a computação das EDPs, enquanto as EDOs são computadas nas GPUs. A cada passo de tempo, são realizadas chamadas para resolver EDOs e EDPs. Neste cenário, uma única GPU pode ser responsável pelo processamento dos dados de várias CPUs, tendo em vista que cada CPU invoca um *kernel* CUDA para ser executado na GPU.

Para atingir o objetivo de melhorar o desempenho do simulador cardíaco, foram criados novos grupos e comunicadores MPI para permitir que todas as invocações a uma GPU fossem centralizadas em um único processo MPI. A hipótese é que este agrupamento possa reduzir os custos associados à resolução das EDOs nas GPUs, principalmente decorrentes da serialização na execução dos múltiplos *kernels* nas GPUs.

A implementação foi feita da seguinte forma. Dinamicamente são criados grupos em número igual ao número de GPUs disponíveis em cada máquina. Antes de cada chamada a um *kernel* CUDA faz-se necessário transferir os dados para o processo responsável pela invocação, e após a chamada os resultados devem ser repassados para todos os processos que fazem parte do grupo. O método foi chamado KA (*kernel aggregation*) por substituir múltiplas chamadas à *kernels* CUDA, que eram realizadas por cada processo MPI, por uma única chamada por GPU. Da mesma forma, ocorre um agrupamento de dados, visto que várias transferências de dados entre GPUs e CPUs são substituídas por uma única operação de transferência.

Arquiteturas de GPUs mais atuais possuem a tecnologia *Hyper-Q*, que permite que vários núcleos CPUs invoquem simultaneamente *kernels* em uma única GPU, reduzindo

ou mesmo eliminando a serialização entre *kernels* CUDA. Deste modo, a pergunta que esta dissertação tentou responder é: pode-se obter resultados equivalentes, em GPUs mais antigas, usando-se uma abordagem por software? Os resultados mostraram que o objetivo foi alcançado.

Os primeiros testes foram feitos em uma GPU antiga, com arquitetura Tesla. Com o uso do método KA, o *speedup* relativo ao método que não utiliza a técnica proposta nesta dissertação, KNA, chegou a 25% quando foram utilizados 2 GPUs e 16 CPUs. É interessante observar que a maior parte do tempo total de execução do simulador foi gasto na resolução das EDPs, ou seja, na parte cuja responsabilidade é das CPUs.

Os resultados para as arquiteturas de GPUs mais novas também foram satisfatórios. Mesmo em uma placa onde a tecnologia *Hyper-Q* está disponível, o simulador que emprega o método KNA não foi substancialmente mais rápido do que a versão do simulador que utiliza o método KA, o que mostra que o custo adicional para a implementação do método KA é baixo. Deve-se acrescentar que nem sempre as novas tecnologias estão disponíveis. O constante avanço tecnológico faz com que a completa substituição de sistemas antigos por novos seja muito oneroso, principalmente quando se usa um *cluster* composto por grande número de nós. Por este motivo a técnica apresentada neste trabalho é muito útil, mesmo que aplicada para gerações antigas de GPUs.

Neste trabalho também foi avaliado o uso de *streams* para obter ganho de desempenho nas operações de transferência de dados entre as CPUs e GPUs. Apesar de Xavier [28] ter obtido bons resultados em sua utilização, neste estudo não foi verificado ganhos de desempenho significativos, uma vez que o tempo de comunicação é muito baixo frente ao tempo total de execução. É possível que com malhas muito maiores, como as que representem um coração inteiro, o emprego da técnica faça diferença.

Trabalhos futuros incluem uma atenção maior às GPUs. Existe a possibilidade de aumentar ainda mais o desempenho dos cálculos das EDOs, visto que a ocupação atual da GPU é de apenas 33%. A causa da baixa ocupação durante a execução do segundo *kernel* está relacionada a quantidade de registradores necessários para sua execução por conta da grande quantidade de variáveis utilizadas. Outros trabalhos se relacionam com a comprovação da existência e medição da sobrecarga citada nas Seções 4.1 e 4.4 através da medição mais precisa e comparação dos tempos de execução dos métodos KNA e KA.

Seria também interessante dividir as resoluções das EDOs e das EDPs entre as GPUs

e CPUs. A resolução de uma é dependente da outra, o que faz com que enquanto as GPUs trabalham, as CPUs fiquem ociosas e vice-versa. Seria desejável que todos os recursos da máquina fossem utilizados simultaneamente na resolução tanto das EDOs quanto das EDPs. Um passo inicial e mais simples nesse sentido poderia ser o uso de somente GPUs nos cálculos das EDOs e EDPs. Atualmente a resolução das EDPs é feita utilizando a biblioteca PETSc, que nas suas versões mais novas oferece suporte para execução em GPUs. Isto por si só poderia ocasionar um grande ganho no tempo total de simulação.

REFERÊNCIAS

[1] OLIVEIRA, R. S., *Ajuste automático de modelos celulares apoiado por Algoritmos Genéticos*, Dissertação de Mestrado, Universidade Federal de Juiz de Fora, Juiz de Fora, MG, Brasil, Agosto 2008.

[2] BARROS, B. G., *Simulações computacionais de arritmias cardíacas em ambientes de computação de alto desempenho do tipo Multi-GPU*, Dissertação de Mestrado, Universidade Federal de Juiz de Fora, Juiz de Fora, MG, Brasil, Fevereiro 2013.

[3] BARROS, B. G., OLIVEIRA, R. S., JR., W. M., LOBOSCO, M., SANTOS, R. W., "Simulations of Complex and Microscopic Models of Cardiac Electrophysiology Powered by Multi-GPU Platforms", *Computational and Mathematical Methods in Medicine*, v. 2012, pp. 13, 2012.

[4] LINDHOLM, E., NICKOLLS, J., OBERMAN, S., MONTRYM, J., "NVIDIA Tesla: A unified graphics and computing architecture", *IEEE Micro*, v. 28, 2008.

[5] NVIDIA, "NVIDIA GeForce GTX 980", 2014.

[6] WHO, "Cardiovascular Deseases", http://http://www.who.int/mediacentre/factsheets/fs317/en/, 2016, acessado em: 20 de fevereiro de 2017.

[7] SHAW, R. M., RUDY, Y., "Ionic Mechanisms of Propagation in Cardiac Tissue", *Circulation Research*, v. 81, n. 5, pp. 727–741, 1997.

[8] DOS SANTOS, R., PLANK, G., BAUER, S., VIGMOND, E., "Preconditioning techniques for the bidomain equations." *Lecture Notes In Computational Science And Engineering*, v. 40, pp. 571–580, 2004.

[9] NVIDIA, "NVIDIA's Next Generation - CUDA Compute Architecture: Kepler GK110", *IEEE Micro*, 2012.

[10] BONDARENKO, V. E., SZIGETI, G. P., BETT, G. C. L., KIM, S. J., RASMUSSON, R. L., "Computer Model of Action Potential of Mouse Ventricular Myocytes", *American Journal of Physiology - Heart and Circulatory Physiology*, v. 287, pp. H1378–H1403, 2004.

[11] SUNDNES, J., LINES, G. T., CAI, X., NIELSEN, B. F., MARDAL, K. A., TVEITO, A., *Computing the Electrical Activity in the Heart*. ed. Springer-Verlag: New York, 2006.

[12] GUYTON, A. C., HALL, J. E., *Tratado de Fisiologia Médica*. 12ª ed. Elsevier Editora Ltda: Rio de Janeiro, 2011.

[13] 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*, v. 117, pp. 500–557, 1952.

[14] KEENER, J., SNEYD, J., *Mathematical Physiology*. ed., v. 8. Springer, 1998.

[15] HILLE, B., *Ionic Channels of Excitable Membranes*. 3 ed. Sinauer Associates: Massachusetts, 2001.

[16] TUNG, L., *A Bi-domain model for describing ischemic myocardial D-C potentials*, Tese de Doutorado, MIT, Cambridge, MA, USA, 1978.

[17] ROCHA, B. M., *Um modelo de acoplamento eletromecânico do tecido cardíaco através do método dos elementos finitos*, Dissertação de Mestrado, Universidade Federal de Juiz de Fora, Juiz de Fora, MG, Brasil, Abril 2008.

[18] SPACH, M. S., HEIDLAGE, J. F., "The Stochastic Nature of Cardiac Propagation at a Microscopic Level: Electrical Description of Myocardial Architecture and Its Application to Conduction", *Circulation Research*, v. 76, pp. 366–380, 1995.

[19] HAIRER, E., WANNER, G., *Solving Ordinary Differential Equations II*. Springer, 1996.

[20] RUSH, S., LARSEN, H., "A practical algorithm for solving dynamic membrane equations", *Biomedical Engineering, IEEE Transactions on*, v. 4, pp. 389–392, 1978.

[21] HARRILD, D. M., HENRIQUEZ, C. S., "A finite volume model of cardiac propagation", *Annals of biomedical engineering*, v. 25(2), pp. 315–334, 1997.

[22] COUDIERE, Y., PIERRE, C., TURPAULT, R., "A 2d/3d finite volume method used to solve the bidomain equations of electrocardiology", *Algoritmy, Conference on Scientic Computing*, pp. 1–10, 2009.

[23] WILKINSON, B., ALLEN, M., *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel*. 2 ed. Pearson Prentice Hall, 2004.

[24] PACHECO, P., *An Introduction to Parallel Programming*. ed. Morgan Kaufmann: Massachusetts, 2011.

[25] BALAY, S., ABHYANKAR, S., ADAMS, M. F., BROWN, J., BRUNE, P., BUSCHELMAN, K., DALCIN, L., EIJKHOUT, V., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., RUPP, K., SMITH, B. F., ZAMPINI, S., ZHANG, H., *PETSc Users Manual*, Tech. Rep. ANL-95/11 - Revision 3.7, Argonne National Laboratory, 2016.

[26] BALAY, S., ABHYANKAR, S., ADAMS, M. F., BROWN, J., BRUNE, P., BUSCHELMAN, K., DALCIN, L., EIJKHOUT, V., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., RUPP, K., SMITH, B. F., ZAMPINI, S., ZHANG, H., "PETSc Web page", `http://www.mcs.anl.gov/petsc`, 2016.

[27] COOK, S., *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. 1 ed. Morgan Kaufmann: San Francisco, 2012.

[28] XAVIER, M. P., *Implementac ao Paralela em um Ambiente de Multiplas GPUs de um Modelo 3D do Sistema Imune Inato*, Dissertação de Mestrado, Universidade Federal de Juiz de Fora, Juiz de Fora, MG, Brasil, Agosto 2013.

[29] DOS SANTOS, R., PLANK, G., BAUER, S., VIGMOND, E., "Parallel multigrid preconditioner for the cardiac bidomain model." *IEEE Trans Biomed Eng*, v. 51(11), 2004.

[30] XAVIER, C., OLIVEIRA, R., VIEIRA, V. F., DOS SANTOS, R. W., MEIRA, W., "Multi-level parallelism for the cardiac bidomain equations." *International Journal of Parallel Programming*, v. 37, pp. 572–592, 2009.

[31] OLIVEIRA, R. S., *Algoritmos paralelos e adaptativos no tempo e no espaço para simulação numérica da eletrofisiologia do coração*, Tese de Doutorado, UFMG, Belo Horizonte, MG, Brasil, Fevereiro 2013.