Universidade Federal de Juiz de Fora
Programa de Mestrado em Modelagem Computacional

Solução das Equações do Bidomínio em Processadores Gráficos

Por

Ronan Mendonça Amorim

JUIZ DE FORA, MG - BRASIL
FEVEREIRO DE 2009

Amorim, Ronan Mendonça

Solução das Equações do Bidomínio em Processadores Gráficos/ Ronan Mendonça Amorim;– 2009.

151 f. :il.

Dissertação (Mestrado em Modelagem Computacional) – Universidade Federal de Juiz de Fora, Juiz de Fora, 2009.

1. Engenharia biomédica. 2. Modelos matemáticos. 3. Simulação de sistemas. I. Título

CDU 61:573

SOLUÇÃO DAS EQUAÇÕES DO BIDOMÍNIO EM PROCESSADORES GRÁFICOS

Ronan Mendonça Amorim

DISSERTAÇÃO SUBMETIDA AO PROGRAMA DE PÓS GRADUAÇÃO EM MODELAGEM COMPUTACIONAL DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS (M.SC.) EM MODELAGEM COMPUTACIONAL.

Aprovada por:

Prof. Rodrigo Weber Dos Santos, D.Sc.
(Orientador)

Prof. Wagner Meira Júnior, Ph.D.

Profa. Denise Burgarelli Duczmal, D.Sc.

Prof. Sócrates de Oliveira Dantas, D.Sc.

JUIZ DE FORA, MG - BRASIL
FEVEREIRO DE 2009

AGRADECIMENTOS

Meus sinceros agradecimentos à minha noiva Elisa que esteve ao meu lado durante todo esse doloroso processo me apoiando, ajudando e não me deixando desistir.

À minha família que sempre me apoiou e incentivou durante toda a vida.

Agradeço também ao meu orientador Rodrigo pelo conhecimento transmitido, pela paciência e oportunidades oferecidas.

A todos os amigos, tanto os que tive a oportunidade de fazer durante este processo quanto os outros, pelos ótimos momentos e experiências compartilhadas.

A todos os professores, em especial os professores da graduação e Mestrado, que direta ou indiretamente contribuíram para que tudo isso fosse possível.

Também agradeço ao professor Gundolf Haase e Manfred Liebmann pela oportunidade de estadia na Áustria e pelo apoio que me ofereceram bem como o grande conhecimento que me transmitiram durante tão pouco tempo.

À FAPEMIG pelo apoio financeiro através da bolsa de mestrado.

Finalmente agradeço a todos que contribuíram de alguma forma para que este trabalho fosse possível.

Resumo da Dissertação apresentada à UFJF como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

SOLUÇÃO DAS EQUAÇÕES DO BIDOMÍNIO EM PROCESSADORES GRÁFICOS

Ronan Mendonça Amorim

Fevereiro/2009

Orientador : Rodrigo Weber Dos Santos

A modelagem computacional do coração tem se mostrado uma ferramenta de destaque da bioinformática funcional. Os modelos, cada vez mais realistas, oferecem uma melhor compreensão dos complexos fenômenos biofísicos associados à atividade elétrica do coração, como por exemplo, das arritmias cardíacas. Ao mesmo tempo, a complexidade crescente dos modelos tem gerado grandes desafios para a computação de alto desempenho. Este trabalho apresenta de forma inédita um simulador do coração baseado nas Equações do Bidomínio que explora a arquitetura vetorial das unidades de processamento gráfico (GPU) de uso geral. Os resultados iniciais são bastante promissores. O uso da GPU acelerou a execução do simulador cardíaco em aproximadamente 6 vezes se comparado ao melhor desempenho obtido em um processador de uso geral (CPU).

Abstract of Dissertation presented to UFJF as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

SOLVING THE BIDOMAIN EQUATIONS USING GRAPHICS PROCESSING UNITS

Ronan Mendonça Amorim

Fevereiro/2009

Supervisor: Rodrigo Weber Dos Santos

The computational modelling of the heart has shown to be a very useful tool in the functional bioinformatics field. The models, becoming more realistic each day, provide a better understanding of the complex biophysical processes related to the electrical activity in the heart as in the case of cardiac arrhythmias. However, the increasing complexity of the models challenges high performance computing in many aspects. This work presents a cardiac simulator based on the bidomain equations that exploits the vectorial architecture of the graphics processing units (GPU). The initial results are promising. The use of the GPU has accelerated the cardiac simulator in about 6 times compared to the best performance obtained in a general purpose processor (CPU).

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

De acordo com a Organização Mundial de Saúde, as doenças cardíacas são responsáveis por um terço do total de mortes no mundo. Pesquisas indicam que mais de 300 mil pessoas morrem por ano no Brasil vítimas de anomalias relacionadas principalmente à atividade elétrica do coração. A falha abrupta da função do coração, que é a causa da morte súbita, pode ser causada por arritmias, que são disfunções que afetam o sistema elétrico do músculo cardíaco produzindo ritmos anormais para o batimento deste. Este funcionamento irregular do ritmo cardíaco pode fazer com que os impulsos elétricos que provocam a contração dos músculos das paredes cardíacas se tornem mais rápidos, a chamada taquicardia, ou se comportem de forma caótica, a chamada fibrilação. Muitos esforços têm sido feitos para entender as causas das doenças cardíacas na esperança de se desenvolver curas.

A modelagem computacional tem se tornado uma importante ferramenta para o estudo da propagação elétrica no coração possibilitando testar o efeito de drogas e estudar doenças de maneira não-invasiva. A propagação elétrica no coração compreende um conjunto de processos biofísicos não-lineares complexos. Sua natu-

reza multi-escalar abrange desde processos nanométricos como movimentos iônicos e dinâmica de estruturas proteicas, até fenômenos na escala dos centímetros como a estrutura do coração e sua contração. Modelos computacionais (HODGKIN e HUXLEY, 1952) se tornaram ferramentas importantes no estudo e compreensão dos complexos fenômenos envolvidos, por permitirem que diferentes informações obtidas de diferentes escalas físicas e experimentos sejam combinadas para se obter uma melhor compreensão da funcionalidade do sistema como um todo. Não é de se surpreender que a alta complexidade dos processos biofísicos seja refletida nos modelos matemáticos e computacionais. Infelizmente, simulações em larga escala, como as resultantes da discretização de um coração inteiro, continuam sendo um desafio computacional. Apesar das dificuldades e da complexidade associadas à implementação e ao uso destes modelos, os benefícios e aplicações justificam sua utilização. Modelos computacionais têm sido utilizados para testes de novas drogas, desenvolvimento de novos dispositivos médicos e novas técnicas de diagnósticos não invasivos para diversas doenças cardíacas.

As células cardíacas são conectadas entre si por junções chamadas *gap* que permitem o fluxo elétrico através de íons entre as células. Isso permite que um estímulo elétrico se propague por todas as células do coração causando a contração do órgão. A modelagem da propagação elétrica no coração envolve dois componentes, um que descreve o fluxo iônico através da membrana celular e o outro que é um modelo elétrico para o tecido que descreve como as correntes de uma região da membrana interagem com as outras. O modelo do bidomínio descreve essa atividade elétrica no coração e é considerado atualmente como o mais completo modelo matemático para descrever a propagação elétrica cardíaca. Ele é dado por um sistema não-linear

de equações diferenciais parciais (EDPs). Infelizmente, a resolução deste sistema não-linear de equações requer um grande esforço computacional. Por isso, podemos encontrar na literatura diversos trabalhos que apresentam o desenvolvimento e a avaliação de sofisticados métodos numéricos para a resolução deste sistema não-linear de EDPs (VIGMOND *et al.* (2002),SANTOS *et al.* (2004)).

Na última década, visando acelerar ainda mais a resolução dessas equações, muitos trabalhos avaliaram algoritmos paralelos e plataformas de alto desempenho baseadas em cluster de computadores (YEO *et al.*, 2006). Atualmente, novas tecnologias e arquiteturas estão surgindo no cenário de computação de alto-desempenho, como as FPGAs (BROWN, 1996), grids computacionais (FOX e GANNON, 2001), processadores multicores homogêneos e heterogêneos (como o Cell processor) e GPUs. Para cada uma dessas tecnologias podemos encontrar na literatura exemplos de sucesso, isto é, de aplicações que se beneficiaram muito com a implementação em algumas dessas arquiteturas. Nem toda aplicação tem sua execução acelerada quando implementada em uma dessas novas tecnologias. Cada uma dessas novas arquiteturas atende melhor um conjunto específico de aplicações que compartilham certas características. Quando uma aplicação não possui certos pré-requisitos sua execução pode ser até prejudicada.

Devido à complexidade das aplicações computacionais de hoje e à complexidade das novas arquiteturas de alto-desempenho em estágio ainda recente de desenvolvimento, o mapeamento ótimo entre as classes de aplicação e as novas tecnologias ainda não é claro.

Neste trabalho, avaliamos como as aplicações de modelagem computacional do

coração, em particular, aquelas que se baseiam nas equações do bidomínio podem se beneficiar das novas arquiteturas de processamento gráfico (GPUs).

Os avanços tecnológicos recentes e a natureza paralela das operações envolvidas na renderização 3D em tempo real transformaram as placas gráficas atuais em máquinas com grande poder computacional paralelo. Os hardwares de processamento gráfico, conhecidos como *Graphics Processor Units* ou GPUs são, provavelmente, os hardwares com a melhor relação entre custo e desempenho da atualidade. Neste trabalho são propostas implementações em GPU para a solução das equações do bidomínio. Em particular, foram implementados em GPU o método dos gradientes conjugados precondicionado, um precondicionador multigrid para a equação elíptica, um precondicionador Jacobi para a equação parabólica, e o método de Euler explícito para as equações diferenciais não-lineares. A implementação em GPU foi comparada à implementação sequencial clássica em CPU e a uma implementação paralela baseada em passagem de mensagem que é executada em um cluster de computadores.

Este trabalho está organizado da seguinte forma: uma apresentação dos aspectos gerais sobre a propagação elétrica no coração é feita no Capítulo 2; os métodos numéricos utilizados para a resolução das equações do bidomínio são apresentados no Capítulo 3; o Capítulo 4 apresenta uma introdução da programação em GPU para problemas de propósito geral; no Capítulo 5 serão apresentadas as implementações em CPU e GPU para solução das equações do bidomínio; no Capítulo 6 será apresentada a metodologia dos testes; no Capítulo 7 serão apresentados os resultados dos testes realizados com implementações em CPU e GPU; no Capítulo 8 discutiremos o

desempenho de ambas as implementações e idéias de trabalhos futuros; finalmente, no Capítulo 9 serão apresentadas as conclusões sobre o trabalho.

Capítulo 2

Modelagem Cardíaca

O coração é uma bomba que fornece sangue e nutrientes para os órgãos do corpo. A contração do coração é disparada por mudanças nas propriedades elétricas das células cardíacas. Sob circunstâncias normais, os impulsos elétricos são gerados espontâneamente no nódulo sinoatrial que é localizado logo abaixo da veia cava superior no átrio direito. Estes impulsos são transmitidos através do átrio pelas células atriais (KEENER e SNEYD, 2001). Durante este processo, as células são estimuladas pelos impulsos, o potencial elétrico da membrana celular é modificado drasticamente de forma que as células vizinhas são excitadas e os impulsos elétricos são propagados.

A membrana celular geralmente mantém um potencial negativo no estado de repouso. Quando o potencial é elevado acima de um limiar, um aumento súbito no potencial da membrana irá acontecer (despolarização) e será seguido de um platô de potencial positivo antes de retornar gradualmente ao potencial de repouso (repolarização). Esta mudança no potencial da membrana é chamada de potencial de ação. As células musculares que compõem o tecido cardíaco, chamadas de miócitos,

pertencem a uma classe de células conhecidas como células excitáveis, as quais têm a capacidade de responder a um estímulo elétrico com um potencial de ação. O potencial de ação segue uma relação fixa entre tempo e voltagem de acordo com o tipo de célula. O potencial de ação é de interesse dos eletrofisiologistas não apenas porque induz a contração muscular mas também porque as mudanças no potencial de ação podem provocar arritmias (SACHSE, 2004). A Figura 2.1 mostra o potencial de ação típico para uma célula do ventrículo humano (TEN TUSSCHER *et al.*, 2004).

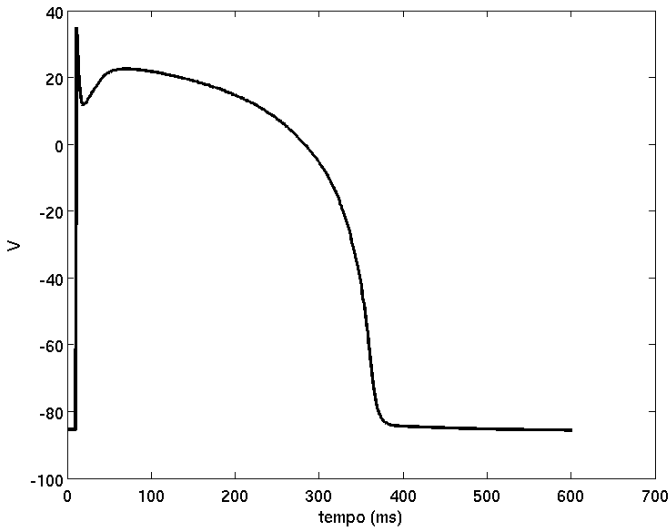

Figura 2.1: Potencial de ação para um célula do ventrículo humano.

As células cardíacas são excitáveis e contráteis. Elas são excitáveis, permitindo que os potenciais de ação propaguem, e o potencial de ação faz com que as células se contraiam em uma ação coletiva, tornando possível o bombeamento de sangue.

As células cardíacas são conectadas entre si por junções *gap* formando um canal entre células adjacentes, o que permite o fluxo de corrente elétrica na forma de íons e portanto funciona como uma conexão elétrica entre células vizinhas. Desta forma,

uma célula estimulada transmite o sinal elétrico para as células vizinhas, permitindo que o estímulo elétrico de uma parte do coração se propague e ative a contração do coração inteiro (ROCHA, 2008).

Existem dois componentes na modelagem da propagação elétrica do coração. O primeiro é um modelo que descreve o fenômeno de potencial de ação proposto por HODGKIN e HUXLEY (1952) em seu trabalho com células nervosas. O segundo é um modelo elétrico para o tecido que descreve como as correntes de uma região da membrana interagem com as outras.

O conjunto de equações do bidomínio é, atualmente, um dos modelos matemáticos mais completos para descrever a atividade elétrica no coração. O sistema não-linear de equações diferenciais parciais (EDPs) modela os domínios intracelular e extracelular do tecido cardíaco de um ponto de vista eletrostático. O acoplamento dos dois domínios é realizado através dos modelos não-lineares que descrevem o potencial de ação na célula.

O conjunto de equações do monodomínio é um modelo simplificado da propagação elétrica no coração onde, diferentemente do modelo do bidomínio, apenas o domínio intracelular é considerado.

Na Seção 2.1 será discutida a modelagem celular e nas Seções 2.7 e 2.8 serão apresentados os modelos elétricos para o tecido cardíaco.

2.1 Modelo Celular

A célula possui uma membrana que separa o meio intracelular do meio extracelular. Esta membrana controla o fluxo de substâncias que entram e saem do citoplasma e é constituída por uma bi-camada fosfolipídica, na qual cada lipídio contém duas caudas hidrofóbicas ligadas a uma cabeça hidrofílica. Em um meio aquoso as caudas estão alinhadas para dentro, repelidas pela água, enquanto a cabeça se encontra na superfície da bi-camada, formando uma barreira para as moléculas carregadas.

Na membrana celular existem arranjos especiais de proteínas que formam os canais iônicos. Os canais iônicos são responsáveis pelo mecanismo de transferência de determinados íons para dentro e para fora da célula. Os canais são especializados e apenas um determinado tipo de substância ou grupo de íons pode passar através de um canal em particular (OLIVEIRA, 2008).

A membrana celular impede o fluxo destes íons para dentro e para fora da célula, mantendo assim uma diferença de concentração de íons. Esta diferença de concentração de íons gera uma diferença de potencial através da membrana celular. A Figura 2.2 ilustra a membrana celular.

2.2 Difusão

Seja u a quantidade de íons em um domínio Ω. A lei de conservação nos diz que a taxa de variação temporal de u é igual a produção local de u mais o acúmulo de u devido ao seu transporte. Esta lei de conservação pode ser escrita matematicamente como

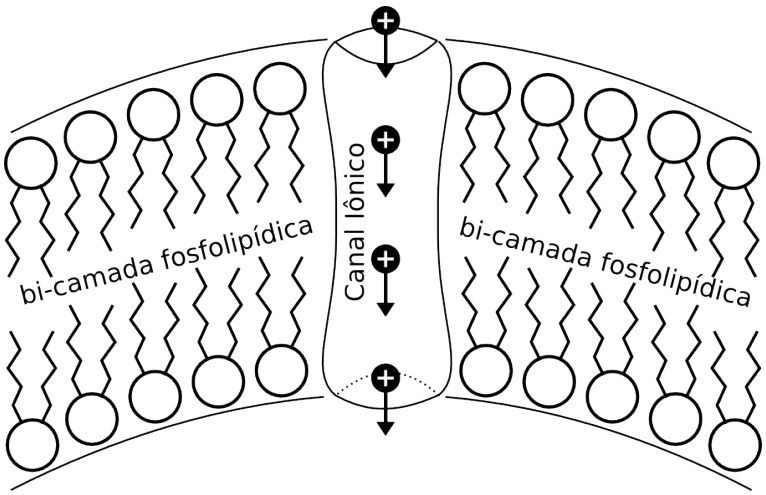

Figura 2.2: Membrana celular com a bi-camada fosfolipídica e um canal iônico permitindo a passagem de alguns íons.

$$\frac{d}{dt}\int_{\Omega} u dV = \int_{\Omega} f dV - \int_{\partial\Omega} \boldsymbol{J}\cdot\boldsymbol{n} dA, \tag{2.1}$$

onde $\partial\Omega$ é a borda da região Ω, $\boldsymbol{n}$ é o vetor unitário normal à borda de Ω, f representa a produção local de u por unidade de volume, e $\boldsymbol{J}$ é o fluxo de íons através da borda. De acordo com o teorema da divergência, se $\boldsymbol{J}$ é suave, então

$$\int_{\partial\Omega} \boldsymbol{J}\cdot\boldsymbol{n} dA = \int_{\Omega} \nabla\cdot\boldsymbol{J} dV, \tag{2.2}$$

Substituindo 2.2 em 2.1 obtemos

$$\frac{d}{dt}\int_{\Omega} u dV = \int_{\Omega} f dV - \int_{\Omega} \nabla\cdot\boldsymbol{J} dV, \tag{2.3}$$

tal que se o volume no qual u está sendo medido é fixo mas arbitrário, as integrais podem ser canceladas resultando em

$$\frac{\partial u}{\partial t} = f - \nabla \cdot \boldsymbol{J}. \tag{2.4}$$

Esta é a equação de conservação que relaciona a taxa de variação da concentração de u com sua produção local e seu acúmulo devido ao transporte.

2.3 Lei de Fick

A lei de Fick diz que existe um fluxo de uma dada espécie ou partícula entre uma região com uma alta concentração e uma região de baixa concentração. Este fluxo tem magnitude proporcional ao gradiente de concentração. A lei de Fick não é realmente uma lei, mas um aproximação macroscópica se a concentração de certa espécie química, como a concentração iônica, não é muito alta. A lei de Fick é mostrada na Equação 2.5.

$$\boldsymbol{J} = -D\nabla u. \tag{2.5}$$

O escalar D é o coeficiente de difusão e é uma característica do soluto e fluido nos quais os íons estão dissolvidos. Aplicando a lei de Fick a equação de conservação se torna a equação de reação e difusão

$$\frac{\partial u}{\partial t} = \nabla \cdot (D\nabla u) + f, \tag{2.6}$$

ou, se D é uma constante, temos

$$\frac{\partial u}{\partial t} = D\nabla^2 u + f. \tag{2.7}$$

2.4 O Potencial da Membrana

Os íons são transportados através da membrana celular com o objetivo de regular a composição iônica intracelular. Existem alguns mecanismos para o transporte de moléculas através da membrana. O transporte pode ser passivo ou ativo. O transporte passivo é dado pelo movimento aleatório das moléculas, e o transporte ativo tem um gasto de energia envolvido para transportar as moléculas através da membrana celular.

As diferenças de concentração são mantidas por um mecanismo ativo que utiliza energia para bombear íons contra o seu gradiente de concentração. As diferenças nas concentrações iônicas criam uma diferença de potencial através da membrana celular. A manutenção da diferença de concentração é essencial para a sobrevivência da célula.

2.4.1 O potencial de equilíbrio de Nernst

Uma das mais importantes equações em eletrofisiologia é a equação de Nernst, que descreve como a diferença na concentração iônica entre duas fases pode resultar em uma diferença de potencial entre as mesmas.

Suponha que existem dois reservatórios contendo um mesmo íon S, mas com diferentes concentrações. Os reservatórios são separados por uma membrana semi-permeável. As soluções em cada lado da membrana são consideradas eletricamente neutras e desta forma um íon S é balanceado com um contra-íon S' com sinal oposto em cada reservatório.

Se a membrana é permeável à S mas não a S', a diferença de concentração através da membrana resultará em um fluxo de S de um lado para o outro. No entanto, S' não pode atravessar a membrana causando um acúmulo de carga em um reservatório. Este desbalanceamento de carga gera um campo elétrico que se opõe a uma difusão maior de S através da membrana. O equilíbrio é atingido quando a difusão de S é exatamente balanceada pelo campo elétrico. Em geral, o fluxo de íons através da membrana é dado pelo gradiente de concentração e pelo campo elétrico. A contribuição do campo elétrico para o fluxo iônico é dado pela equação de *Planck*

$$\boldsymbol{J} = -\nu \frac{z}{|z|} c \nabla \phi, \tag{2.8}$$

onde ν é a mobilidade do íon, definido como a velocidade do íon em um campo elétrico constante; z é a valência do íon, tal que $z/|z|$ é o sinal da força no íon; c é a concentração de S; ϕ é o potencial elétrico, tal que $-\nabla\phi$ é o campo elétrico.

Existe uma relação, determinada por Einstein, entre a mobilidade iônica u e a constante de difusão de *Fick*:

$$D = \frac{\nu RT}{|z|F}, \tag{2.9}$$

onde R é a constante universal do gás, T é a temperatura absoluta e F é a constante de Faraday. Quando os efeitos dos gradientes de concentração e dos gradientes elétricos são combinados, obtemos a equação de *Nernst-Planck*

$$\boldsymbol{J} = -D\left(\nabla c + \frac{zF}{RT}c\nabla\phi\right). \tag{2.10}$$

Se o fluxo íons e campo elétrico forem transversais à membrana, podemos ver a Equação 2.10 como a relação em uma dimensão

$$J = -D\left(\frac{dc}{dx} + \frac{zF}{RT}c\frac{d\phi}{dx}\right). \tag{2.11}$$

Equação de Nernst

A equação de Nernst pode ser derivada da equação da eletrodifusão Nernst-Planck 2.11. Quando o fluxo J é zero, encontramos

$$-D\left(\frac{dc}{dx} + \frac{zF}{RT}c\frac{d\phi}{dx}\right) = 0, \tag{2.12}$$

tal que

$$\frac{1}{c}\frac{dc}{dx} + \frac{zF}{RT}\frac{d\phi}{dx} = 0. \tag{2.13}$$

Agora suponha que a membrana celular se estende de $x = 0$ (parte interior) até

$x = L$ (parte exterior), e os subscritos i e e denotam as quantidades no interior e exterior respectivamente. Então, integrando de $x = 0$ até $x = L$

$$ln(c)|_{c_i}^{c_e} = \frac{zF}{RT}(V_i - V_e). \tag{2.14}$$

A diferença de potencial através da membrana, $V_{Nernst} = V_i - V_e$ é dado por

$$V_{Nernst} = \frac{RT}{zF} ln\left(\frac{c_e}{c_i}\right), \tag{2.15}$$

que é a equação de Nernst e V_{Nernst} é o potencial de equilíbrio quando o fluxo é nulo.

2.5 Modelo Elétrico da Membrana Celular

Uma vez que a membrana celular separa cargas, ela pode ser vista como um capacitor. A capacitância pode ser definida como a razão da carga armazenada e a voltagem necessária para manter esta carga, e é denotado por

$$C_m = \frac{Q}{V}. \tag{2.16}$$

Os diversos modelos existentes do comportamento elétrico das células cardíacas são baseados no trabalho de HODGKIN e HUXLEY (1952), portanto, têm em comum dois elementos básicos, a membrana celular e os canais iônicos que a permeiam. A membrana plasmática é modelada eletricamente como um dielétrico de

capacitância C_m. Já os canais iônicos são modelados como resistências não-lineares (não ôhmicos), que dependem da diferença de potencial sobre a membrana, e da concentração de diversos íons. A Figura 2.3 mostra o esquema elétrico do modelo celular.

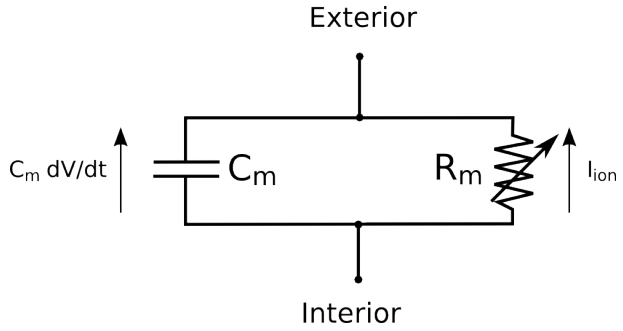

Figura 2.3: Circuito elétrico para o modelo celular.

Considerando a variação de carga no tempo, tem-se

$$\frac{dQ}{dt} = \frac{d(C_m V)}{dt} = C_m \frac{dV}{dt}, \tag{2.17}$$

onde $\frac{dQ}{dt}$ é a variação da carga no tempo correspondendo a uma contribuição para a corrente chamada corrente capacitiva. A corrente total através da membrana celular é definida como a soma da corrente capacitiva com a corrente iônica que atravessa os canais iônicos,

$$I_m = I_{ion} + I_c. \tag{2.18}$$

Substituindo 2.17

$$I_m = I_{ion} + C_m \frac{dV}{dt}. \tag{2.19}$$

Se o circuito da Figura 2.3 for fechado, pela conservação da corrente temos que

$$I_{ion} + C_m \frac{dV}{dt} = 0. \tag{2.20}$$

Quando o potencial através da membrana é diferente do potencial de equilíbrio de Nernst, uma corrente de íons passa pelos canais iônicos e é representada na equação como I_{ion}. A corrente iônica em cada canal iônico pode ser modelada por uma relação linear entre a corrente e a voltagem como,

$$I = G(V - V_{Nernst}), \tag{2.21}$$

onde G é a condutância, definida como o inverso da resistência $\frac{1}{R}$ do canal. A condutância dependendo do tipo de canal iônico pode ser constante ou dependente do tempo, do potencial elétrico e até mesmo da concentração iônica.

Um outro modelo para a corrente iônica parte da hipótese de que o campo elétrico é constante na membrana. Maiores detalhes podem ser encontrados em KEENER e SNEYD (2001). Conhecido como equação de Goldman-Hodgkin-Katz (GHK), o modelo considera uma relação não-linear entre o potencial transmembrânico e a corrente, definida por

$$I = P_S \frac{z^2 F^2}{RT} V \frac{c_i - c_e\, exp(\frac{-zFV}{RT})}{1 - exp(\frac{-zFV}{RT})} \tag{2.22}$$

onde $P_S = \frac{D}{L}$ é a permeabilidade da membrana ao íon considerado e c_i e c_e são as concentrações interna e externa deste íon.

Ambos os modelos satisfazem o potencial de equilíbrio de Nernst, ou seja, quando o potencial transmembrânico é igual ao potencial de Nernst não há fluxo.

A condutância dos canais iônicos pode variar no tempo em resposta a mudanças no potencial transmembrânico. Do ponto de vista da modelagem os canais iônicos são representados como um conjunto de sub-unidades que podem estar abertas, permitindo a passagem de íons, ou fechadas. A taxa com que os canais se abrem é

$$\frac{d[A]}{dt} = \alpha(V)[F] - \beta(V)[A] \tag{2.23}$$

onde $[A]$ e $[F]$ são a concentração de canais abertos e fechados, respectivamente; α a taxa que os canais se abrem e β a taxa que os canais se fecham ($A \underset{\alpha}{\overset{\beta}{\rightleftarrows}} F$). Se dividirmos a Equação 2.23 pela concentração total $[A] + [F]$ obtemos

$$\frac{dg}{dt} = \alpha(V)(1 - g) - \beta(V)g, \tag{2.24}$$

onde $g = \frac{[A]}{[A]+[F]}$. A Equação 2.24 pode ser vista como a probabilidade de cada uma das sub-unidades que compõem o canal estar aberta. Portanto, para um canal com n sub-unidades iguais, as quais abrem e fecham independentes umas das outras, a probabilidade A do canal estar aberto é

$$A = g^n \tag{2.25}$$

Entretanto os canais iônicos podem ser formados por diferentes sub-unidades. Um canal formado por dois tipos de sub-unidades g e h, e com m e n sub-unidades de cada respectivamente, tem a probabilidade de estar aberto dada por

$$A = g^m h^n \tag{2.26}$$

Finalmente a condutividade pode ser calculada pelo produto da condutividade máxima G_{max} quando todos os canais estão abertos, pela proporção de canais abertos. Desta maneira a corrente iônica para um canal p é descrito da forma:

$$\begin{aligned} I_p &= G_{max} A (V - V_{Nernst(p)}) && (2.27)\\ A &= g^m h^h && (2.28)\\ \frac{dg}{dt} &= \alpha_g(V)(1-g) - \beta_g(V) g && (2.29)\\ \frac{dh}{dt} &= \alpha_h(V)(1-h) - \beta_h(V) h && (2.30) \end{aligned}$$

onde $\alpha_g(V)$, $\alpha_h(V)$, $\beta_g(V)$ e $\beta_h(V)$ são funções não-lineares que dependem de V e são obtidas experimentalmente.

2.6 Modelo Celular do Ventrículo Humano

O modelo celular para o ventrículo humano é baseado em dados experimentais recentes da maior parte das correntes iônicas mais importantes. O modelo inclui também uma dinâmica de íons cálcio simples e permite a reprodução de potenciais de ação de células do epicárdio, endocárdio e célula M do ventrículo humano. O modelo é capaz de reproduzir dados experimentais importantes para o estudo de arritmias reentrantes no tecido cardíaco ventricular humano (TEN TUSSCHER *et al.*, 2004).

A membrana celular é modelada como um capacitor conectado em paralelo com resistências variáveis e baterias representando diferentes correntes iônicas e bombas. As equações diferenciais que descrevem esse comportamento são apresentadas no modelo a seguir

$$\begin{aligned} C_m \frac{dV}{dt} = -(&I_{K1} + I_{to} + I_{Kr} + I_{Ks} + I_{CaL} + \\ &I_{NaK} + I_{Na} + I_{b_{Na}} + I_{NaCa} +, \\ &I_{b_{Ca}} + I_{p_K} + I_{p_{Ca}} + I_{Stim}) \end{aligned} \tag{2.31}$$

onde cada um dos termos do lado direito da Equação 2.31 representam um tipo de corrente iônica diferente como I_{K1}, I_{to}, I_{Kr}, I_{Ks} e I_{p_K} que são diferentes tipos de correntes de potássio; e I_{CaL}, $I_{b_{Ca}}$ e $I_{p_{Ca}}$ correntes de cálcio; I_{Na} e $I_{b_{Na}}$ correntes de sódio; I_{NaK} e I_{NaCa} são bombas; e I_{Stim} uma corrente de estímulo que pode ser aplicada à célula. Cada corrente é descrita por equações do tipo da Equação 2.27.

A modelagem das concentrações intracelular de cálcio (Ca_i), cálcio no retículo sarcoplasmático (Ca_{SR}), sódio (Na_i) e potássio (K_i) são dadas por

$$\frac{d(Ca_i)}{dt} = Ca_{I_{bufc}}(((I_{leak} - I_{up}) + I_{rel}) - \frac{((I_{CaL} + I_{b_{Ca}} + I_{p_{Ca}}) - 2I_{NaCa})}{21V_cF}C_m) \tag{2.32}$$

$$\frac{dCa_{SR}}{dt} = \frac{Ca_{sr_{bufsr}}V_c}{V_{sr}}(I_{up} - (I_{rel} + I_{leak})) \tag{2.33}$$

$$\frac{dNa_i}{dt} = \frac{-(I_{Na} + I_{b_{Na}} + 3I_{NaK} + 3I_{NaCa})C_m}{1V_cF} \tag{2.34}$$

$$\frac{dK_i}{dt} = \frac{-1}{1V_cF}((I_{K1} + I_{to} + I_{Kr} + I_{Ks} + I_{p_K} + I_{Stim}) - 2I_{NaK})C_m, \tag{2.35}$$

onde V_{sr}, V_c são os volumes do retículo sarcoplasmático e citoplasma, respectivamente, F a constante de Faraday e C_m a capacitância da célula por unidade de área. O modelo completo possui um total 17 equações diferenciais. A maior parte dessas equações diferenciais ordinárias modelam as sub-unidades dos canais. Uma descrição detalhada deste modelo das células do ventrículo humano pode ser encontrado em TEN TUSSCHER *et al.* (2004).

Desta forma, um modelo celular que inclui as correntes iônicas de vários canais é modelado por um sistema de equações diferenciais ordinárias e pode ser generalizado da seguinte forma

$$0 = C_m\frac{dV}{dt} + f(V, \boldsymbol{n}) \tag{2.36}$$

$$\frac{d\boldsymbol{n}}{dt} = g(V, \boldsymbol{n}), \tag{2.37}$$

onde o vetor $\boldsymbol{n}$ representa os efeitos dos diversos íons envolvidos na geração do potencial de ação (Na^+, K^+, Ca^{2+}, Cl^-). Utilizaremos as Equações 2.36 e 2.37, para representar os modelos celulares no restante desse trabalho.

2.7 O Modelo Monodomínio

As células cardíacas encontram-se conectadas por junções *gap*, que permitem a passagem livre de íons entre os meios intracelulares das células. Este meio intracelular interconectado pode ser modelado por uma distribuição de resistência R que varia espacialmente, pois a resistência do citoplasma é tipicamente bem menor que a da junção *gap*. Supondo que o meio extracelular não afeta a atividade elétrica, obtemos o esquema de conexão intercelular ilustrado na Figura 2.4, onde o potencial extracelular é o terra $V_e = 0$, V igual ao potencial intracelular V_i, $J_m dx$ é a corrente transmembrânica, J é a corrente intracelular e Rdx é a resistência intracelular. Aplicando a Lei dos Nós e a Lei de Ohm, obtemos:

$$J(x + dx) - J(x) + J_m(x)dx = 0 \tag{2.38}$$

$$J_m(x) = -\frac{J(x + dx) - J(x)}{dx} = -\frac{dJ(x)}{dx} \tag{2.39}$$

$$V(x - dx) - V(x) = R(x)dxJ(x) \tag{2.40}$$

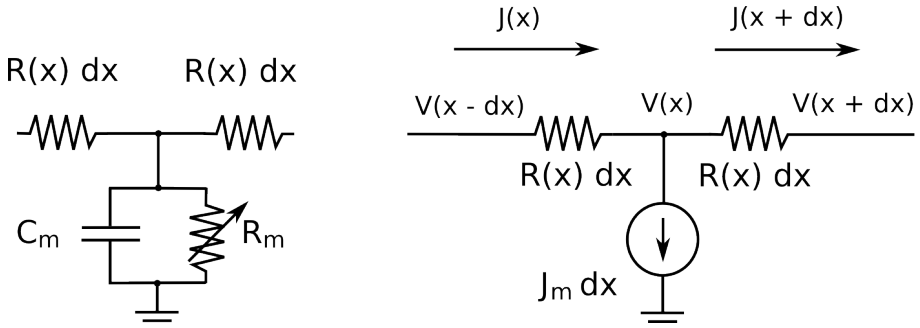

Figura 2.4: Modelo Monodomínio.

$$J(x) = \frac{1}{R(x)} \frac{V(x-dx) - V(x)}{dx} = -\frac{1}{R(x)} \frac{dV}{dx} \tag{2.41}$$

$$J_m(x) = -\frac{dJ(x)}{dx} = \frac{d}{dx}\left(\frac{1}{R(x)} \frac{dV}{dx}\right) \tag{2.42}$$

Substituindo J_m pelo sistema 2.36-2.37, obtemos o modelo do monodomínio em uma dimensão:

$$\frac{d}{dx}\left(\frac{1}{R(x)} \frac{dV}{dx}\right) = \chi\left(C_m \frac{dV}{dt} + f(V, \boldsymbol{n})\right) \tag{2.43}$$

$$\frac{d\boldsymbol{n}}{dt} = g(V, \boldsymbol{n}) \tag{2.44}$$

onde χ é a razão superfície-volume, que converte unidades de distribuição de corrente por unidade de área para corrente por unidade de volume.

O modelo descrito na Figura 2.4 depende de uma escala espacial microscópica, pois a extensão das junções *gap* está em torno de $2nm$. Uma simplificação normalmente adotada é a substituição de $\frac{1}{R(x)}$ por uma condutividade média efetiva

σ, constante. Esta condutividade média pode ser obtida através de técnicas de homogenização (KEENER e SNEYD, 2001). Essa simplificação modela as regiões isotrópicas, que são aquelas em que a condutividade é igual em todas as direções. Porém o tecido cardíaco não é isotrópico. O tecido cardíaco é basicamente composto por células cardíacas (miócitos cardíacos) e colágeno (tecido conjuntivo). As células cardíacas estão normalmente conectadas longitudinalmente e organizadas em uma estrutura tubular, denominada de fibra cardíaca. As fibras cardíacas, por sua vez, estão conectadas pelo tecido conjuntivo. Porém, esta conexão é estruturada. Um tipo de tecido conjuntivo conecta diferentes fibras cardíacas para a formação das folhas do coração. Um segundo tipo de tecido conjuntivo conecta as diferentes folhas cardíacas. Esta estrutura folha-fibra tem implicações importantes na condução do impulso elétrico no coração.

Para este tipo de tecido o modelo adequado é o ortotrópico, onde a condutividade elétrica do tecido depende tanto da orientação das fibras como da orientação das folhas, que são um conjunto de fibras fortemente conectadas por tecido conjuntivo (Figura 2.5). Nesse caso o tensor σ para folhas e fibras orientadas na direção do eixo x ficaria da forma mostrada na Equação 2.45:

$$\boldsymbol{\sigma} = \begin{pmatrix} \sigma_l & 0 & 0 \\ 0 & \sigma_t & 0 \\ 0 & 0 & \sigma_o \end{pmatrix}, \tag{2.45}$$

onde σ_l, σ_t e σ_o são as condutividades nas direções longitudinal à fibra, transversal às fibras na mesma folha e ortogonal à folha, respectivamente, e $\sigma_l > \sigma_t > \sigma_o$.

Para o cálculo do tensor de condutividade em regiões ortotrópicas utiliza-se a

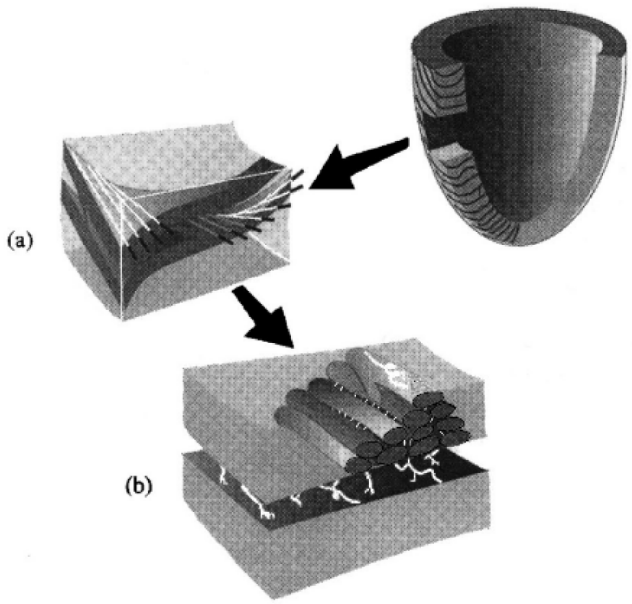

Figura 2.5: Esquema da micro-estrutura cardíaca. (a) Corte da parede ventricular mostra a orientação das fibras dentro de uma folha do miocárdio. (b) Os miócitos são mostrados com três a quatro células de espessura em uma folha. Adaptado de SCHULTE *et al.* (2000).

Equação 2.46.

$$\boldsymbol{\sigma} = T\,\bar{\sigma}\,T^T \tag{2.46}$$

Onde T é uma matriz de mudança de base que leva a base canônica para a base formada pelos vetores ortogonais fibra, folha e o vetor $fibra \otimes folha$, $\boldsymbol{\sigma}$ é o tensor de condutividade canônico e T^T é a transformação inversa.

A generalização do modelo do monodomínio para três dimensões é da forma:

$$\nabla \cdot (\boldsymbol{\sigma} \nabla V) = \chi \left(C_m \frac{\partial V}{\partial t} + f(V, \boldsymbol{n}) \right) \tag{2.47}$$

$$\frac{d\boldsymbol{n}}{dt} = g(V, \boldsymbol{n}) \tag{2.48}$$

2.8 O Modelo Bidomínio

Alguns fenômenos elétricos observados experimentalmente ou clinicamente não são explicados ou simulados pelas equações do modelo monodomínio. Recentemente, o modelo bidomínio, que inclui os efeitos do meio extracelular, possibilitou a simulação de alguns destes fenômenos, como por exemplo, os relacionados aos padrões complexos de arritmias e ao processo de desfibrilação cardíaca (SANTOS, 2002).

O esquema elétrico do modelo do bidomínio está apresentado na Figura 2.6. Através de uma análise análoga à descrita para o monodomínio, obtemos o modelo bidomínio unidimensional.

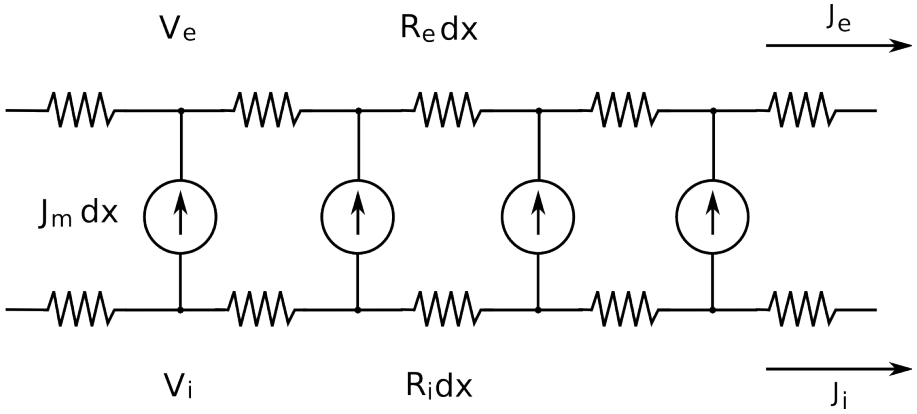

Figura 2.6: Esquema elétrico do modelo do bidomínio em 1D.

$$\frac{d}{dx}\left(\frac{1}{R_i(x)}\frac{dV_i}{dx}\right) = \chi\left(C_m\frac{dV}{dt} + f(V,\boldsymbol{n})\right) \tag{2.49}$$

$$-\frac{d}{dx}\left(\frac{1}{R_e(x)}\frac{dV_e}{dx}\right) = \chi\left(C_m\frac{dV}{dt} + f(V,\boldsymbol{n})\right) \tag{2.50}$$

$$\frac{d\boldsymbol{n}}{dt} = g(V,\boldsymbol{n}) \tag{2.51}$$

$$V = V_i - V_e, \tag{2.52}$$

onde V_i e V_e são os potenciais intra e extracelular, respectivamente. No modelo bidomínio tridimensional, os potenciais V_i e V_e estão definidos em todo o espaço. Ou seja, os meios intracelular e extracelular estão espacialmente entrelaçados. O modelo bidomínio tridimensional é da forma:

$$\nabla\cdot(\sigma_i\nabla V_i) = \chi\left(C_m\frac{\partial V}{\partial t} + f(V,\boldsymbol{n})\right) \tag{2.53}$$

$$-\nabla\cdot(\sigma_e\nabla V_e) = \chi\left(C_m\frac{\partial V}{\partial t} + f(V,\boldsymbol{n})\right) \tag{2.54}$$

$$\frac{d\boldsymbol{n}}{dt} = g(V,\boldsymbol{n}) \tag{2.55}$$

$$V = V_i - V_e \tag{2.56}$$

onde $\boldsymbol{\sigma_i}$ e $\boldsymbol{\sigma_e}$ são os tensores de condutividade dos meios intracelular e extracelular, respectivamente.

Capítulo 3

Métodos Numéricos

O sistema de equações do bidomínio como apresentado anteriormente é de difícil resolução. Neste capítulo será apresentada uma abordagem para desacoplar as equações permitindo que diferentes métodos sejam aplicados para cada equação.

3.1 Discretização

O operador *splitting* (STRIKWERDA, 1989) é utilizado a fim de desacoplar o sistema não-linear de equações cuja resolução é, em geral, custosa computacionalmente. Ao aplicar este operador, a solução numérica se reduz a um esquema de três passos que envolve a solução de uma EDP parabólica, uma EDP elíptica e um sistema não-linear de equações diferenciais ordinárias (EDOs) a cada passo de tempo. A seguir o modelo do bidomínio é novamente apresentado seguido das etapas para se obter o esquema de três passos para a solução das equações do bidomínio.

$$\nabla \cdot (\sigma_i \nabla V_i) = \chi \left(C_m \frac{\partial V}{\partial t} + f(V(n)) \right) \tag{3.1}$$

$$-\nabla \cdot (\sigma_e \nabla V_e) = \chi \left(C_m \frac{\partial V}{\partial t} + f(V(n)) \right) \tag{3.2}$$

$$\frac{d\boldsymbol{n}}{dt} = g(V(\boldsymbol{n})) \tag{3.3}$$

$$V = V_i - V_e; \qquad x \in \Omega \quad t \in [0, t_f] \tag{3.4}$$

Rescrevendo a Equação 3.1 usando $V_i = V + V_e$ obtemos:

$$(\chi C_m)\frac{\partial V}{\partial t} = \nabla \cdot (\sigma_i \nabla V_e) + \nabla \cdot (\sigma_i \nabla V) - \chi f(V, \boldsymbol{n}) \tag{3.5}$$

Somando a Equação 3.5 com a Equação 3.2 obtemos:

$$-\nabla \cdot ((\sigma_i + \sigma_e)\nabla V_e) = \nabla \cdot (\sigma_i \nabla V) \tag{3.6}$$

O sistema de equações do bidomínio é então reescrito da seguinte forma

$$(\chi C_m)\frac{\partial V}{\partial t} = \nabla \cdot (\sigma_i \nabla V_e) + \nabla \cdot (\sigma_i \nabla V) - \chi f(V, \boldsymbol{n}) \tag{3.7}$$

$$-\nabla \cdot ((\sigma_i + \sigma_e)\nabla V_e) = \nabla \cdot (\sigma_i \nabla V) \tag{3.8}$$

$$\frac{d\boldsymbol{n}}{dt} = g(V(\boldsymbol{n})) \tag{3.9}$$

Aplicando o operador *splitting* ao sistema de equações anterior, a solução numérica se reduz a um esquema de três passos que envolve a solução de uma EDP parabólica, uma EDP elíptica e um sistema não-linear de EDOs a cada passo de tempo. A EDP parabólica é discretizada no tempo através do método de Crank-Nicholson (CRANK e NICOLSON, 1947). A discretização temporal da Equação 3.7 é mostrada na Equação 3.10. Já para o sistema de EDOs o método de Euler-Explícito (PUWAL e ROTH, 2007) é utilizado, como mostrado nas Equações 3.11 e 3.12.

$$\left(1 - \frac{\Delta t}{2}\frac{\nabla \cdot (\sigma_i \nabla)}{\chi C_m}\right)\varphi^{k+1/2} = \left(1 + \frac{\Delta t}{2}\frac{\nabla \cdot (\sigma_i \nabla)}{\chi C_m}\right)\varphi^{k} + \frac{\Delta t}{\chi C_m}\nabla \cdot (\sigma_i \nabla(\varphi_e)^k) \quad (3.10)$$

$$\varphi^{k+1} = \varphi^{k+1/2} - \frac{\Delta t}{C_m} f(\varphi^{k+1/2}, (\boldsymbol{\eta})^k) \quad (3.11)$$

$$(\boldsymbol{\eta})^{k+1} = (\boldsymbol{\eta})^k + \Delta t g(\varphi^{k+1/2}, (\boldsymbol{\eta})^k) \quad (3.12)$$

$$\nabla \cdot ((\sigma_i + \sigma_e)\nabla(\varphi_e)^{k+1}) = -\nabla \cdot (\sigma_i \nabla \varphi^{k+1}) \quad (3.13)$$

onde φ^k, φ_e^k e $\boldsymbol{\eta}^k$ são as discretizações de V, V_e e $\boldsymbol{n}$ no passo de tempo k. Podemos reescrever as equações anteriores da seguinte forma

$$(1 - A_i)\,\varphi^{k+1/2} = (1 + A_i)\,\varphi^k + \Delta t A_i (\varphi_e)^k \quad (3.14)$$

$$\varphi^{k+1} = \varphi^{k+1/2} - \frac{\Delta t}{C_m} f(\varphi^{k+1/2}, (\boldsymbol{\eta})^k) \tag{3.15}$$

$$(\boldsymbol{\eta})^{k+1} = (\boldsymbol{\eta})^k + \Delta t\, g(\varphi^{k+1/2}, (\boldsymbol{\eta})^k) \tag{3.16}$$

$$(A_i + A_e)\varphi_e^{k+1} = -A_i\varphi^{k+1}, \tag{3.17}$$

onde A_i e A_e são as discretizações dos operadores $\nabla \cdot (\boldsymbol{\sigma_i} \nabla / \chi C_m)$ e $\nabla \cdot (\boldsymbol{\sigma_e} \nabla / \chi C_m)$, respectivamente. A discretização no espaço é realizada através do método dos elementos finitos em um grid 2D estruturado composto de elementos quadrados com interpolação bilinear. Para mais detalhes sobre o método dos elementos finitos veja GOCKENBACH (2007).

O primeiro passo do esquema (Equação 3.14) está relacionado com a resolução da Equação 3.7 que é uma EDP parabólica linear. O segundo passo do esquema (Equação 3.15 e 3.16) resolve a parte não-linear, Equação 3.9, que é um sistema não-linear de EDOs. O terceiro passo do esquema (Equação 3.17) está relacionado com a resolução da parte linear da Equação 3.8 que é uma EDP elíptica linear.

A utilização do operador *splitting* permite que diferentes métodos numéricos sejam utilizados em cada passo, tornando possível maximizar o desempenho computacional, bem como eliminar dependências complexas entre variáveis. A discretização das EDPs fornece dois sistemas lineares a serem resolvidos. Um sistema linear para a EDP parabólica e um para a EDP elíptica. Na Seção 3.3 serão apresentados os métodos para a resolução de sistemas lineares utilizados neste trabalho. Também será

apresentado na Seção 3.2 o método de Euler explícito para a resolução do sistema de EDOs.

3.2 Método de Euler Explícito

Como visto anteriormente a solução das equações do bidomínio envolve a solução de um sistema de EDOs. O método de Euler explícito é um método numérico para resolver equações diferenciais ordinárias. Suponha que queremos resolver a seguinte EDO:

$$y'(t) = f(t, y), \qquad 0 \leq t \leq b \tag{3.18}$$

$$y(0) = c. \tag{3.19}$$

E desejamos aproximar a solução em um intervalo discreto com $N + 1$ pontos igualmente espaçados

$$0 = t_0 < t_1 < t_2 < \cdots < t_{N-1} < t_N = b. \tag{3.20}$$

Seja $h = t_n - t_{n-1}$ o espaçamento entre os pontos discretos. Para construirmos um método discreto consideremos a série de Taylor

$$y(t) = y(t) + +h\, y'(t) + h^2\, \frac{y''(t)}{2!} + h^3\, \frac{y'''(t)}{3!} + \cdots, \tag{3.21}$$

que também pode ser escrito como

$$y(t_n) = y(t_{n-1}) + h\, y'(t_{n-1}) + O(h^2) \tag{3.22}$$

Se desconsiderarmos os termos com ordem maior ou igual a 2 e substituirmos a Equação 3.18 na Equação 3.22 obtemos

$$y(t_n) = y(t_{n-1}) + h\, f(t_{n-1}, y(t_{n-1})) \tag{3.23}$$

que é o método de Euler explícito, começando com $y(0) = c$. A cada passo do método uma aproximação de $y(t_n)$ é obtida, baseando-se apenas na aproximação para o passo anterior $y(t_{n-1})$. A precisão do método é de primeira ordem pois o erro cometido pela aproximação da série de Taylor (Equação 3.21) é de ordem O(h).

3.3 Resolução de Sistemas Lineares

A discretização das equações do bidomínio como descrita na Seção 3.1 produz dois sistemas lineares a serem resolvidos derivados das EDPs parabólica e elíptica.

Existem diferentes abordagens para a solução de sistemas lineares. Uma primeira idéia é a utilização de métodos diretos que consistem em construir a inversa da matriz de coeficientes ou decompô-la de maneira que a obtenção da matriz inversa seja facilitada, como matrizes triangulares. Exemplos de métodos diretos incluem eliminação de Gauss e decomposição LU. No entanto, os métodos diretos geralmente

necessitam de sistemas computacionais com uma grande quantidade de memória disponível para que seja possível armazenar as matrizes inversas, o que se torna inviável na solução das equações do bidomínio onde geralmente os sistemas lineares possuem milhões de incógnitas.

A utilização de métodos iterativos é uma alternativa aos métodos diretos. A idéia dos métodos iterativos é utilizar aproximações sucessivas para se obter soluções mais precisas do sistema linear a cada passo. A taxa de convergência dos métodos iterativos depende muito das propriedades espectrais da matriz de coeficientes. Por esta razão, geralmente são utilizados precondicionadores, que transformam a matriz de coeficientes em uma outra matriz com um espectro mais favorável. A matriz de transformação, chamada de precondicionador, melhora a convergência do método iterativo suficientemente para sobrepujar os custos computacionais de sua construção e aplicação (BARRETT *et al.*, 1994).

Diversos métodos iterativos e precondicionadores têm sido utilizados para a solução das equações do bidomínio. Uma vantagem dos métodos iterativos é que necessitam de muito menos espaço de armazenamento que os métodos diretos tornando possível que grandes sistemas lineares sejam resolvidos. A escolha de um precondicionador apropriado ao tipo de problema pode tornar o desempenho dos métodos iterativos compatível ao desempenho dos métodos diretos. De acordo com SANTOS *et al.* (2004) dos vários métodos testados, o método dos Gradientes Conjugados Precondicionado têm se tornando uma escolha padrão. As próximas seções apresentarão o método dos gradientes conjugados, bem como os precondicionadores utilizados neste trabalho.

3.3.1 Método dos gradientes conjugados

O método dos Gradientes Conjugados é um método iterativo que converge em no máximo n passos, onde n é o número de incógnitas, se nenhum erro de arredondamento é encontrado. É um dos métodos iterativos mais conhecidos para resolver sistemas lineares cujas matrizes de coeficientes são esparsas, simétricas e positivo definidas. Suponha que desejamos resolver o seguinte sistema:

$$A\boldsymbol{x} = \boldsymbol{b} \tag{3.24}$$

onde A é a matriz de coeficientes ou matriz de rigidez, $\boldsymbol{x}$ o vetor de incógnitas e $\boldsymbol{b}$ um vetor conhecido ou vetor de carga. Define-se que um sistema é simétrico se $A^T = A$ e positivo definido se $\boldsymbol{x}^T A\boldsymbol{x} > 0$ para todo $\boldsymbol{x} \in R^n$ tal que $\boldsymbol{x} \neq \boldsymbol{0}$. Começando com uma estimativa inicial $\hat{\boldsymbol{x}}_0$ da solução $\boldsymbol{x}$, o método determina aproximações sucessivas $\hat{\boldsymbol{x}}_1$, $\hat{\boldsymbol{x}}_2$, ... de $\boldsymbol{x}$, sendo a estimativa $\hat{\boldsymbol{x}}_{i+1}$ mais próxima da solução exata $\boldsymbol{x}$ do que $\hat{\boldsymbol{x}}_i$. A cada passo o resíduo $\boldsymbol{r}_i = \boldsymbol{b} - A\hat{x}_i$ é calculado.

Dizemos que um vetor $\boldsymbol{u}$ é conjugado ao outro $\boldsymbol{v}$, em relação a A, se

$$\boldsymbol{u}^T A\boldsymbol{v} = 0 \tag{3.25}$$

Uma vez que A é simétrico positivo definido, o lado esquerdo da Equação 3.25 define um produto interno:

$$\langle \boldsymbol{u}, \boldsymbol{v} \rangle_A = \langle \boldsymbol{u}^T A, \boldsymbol{v} \rangle = \langle A\boldsymbol{u}, \boldsymbol{v} \rangle = \langle \boldsymbol{u}, A\boldsymbol{v} \rangle = \boldsymbol{u}^T A\boldsymbol{v} \tag{3.26}$$

Então dois vetores são conjugados se são ortogonais em relação a este produto interno. Se $\boldsymbol{u}$ é conjugado a $\boldsymbol{v}$, então $\boldsymbol{v}$ é conjugado a $\boldsymbol{u}$.

Suponha que $\boldsymbol{p}_k$ é uma sequência de n vetores mutuamente conjugados. Logo formam uma base para R^n. Então podemos representar o vetor solução $\boldsymbol{x}$ como:

$$\boldsymbol{x} = \sum_{i=1}^{n} \alpha_i \boldsymbol{p}_i \tag{3.27}$$

Os coeficientes α_i são dados por:

$$A\boldsymbol{x} = \sum_{i=1}^{n} \alpha_i A\boldsymbol{p}_i = \boldsymbol{b} \tag{3.28}$$

$$\boldsymbol{p}_k^T A\boldsymbol{x} = \sum_{i=1}^{n} \alpha_i \boldsymbol{p}_k^T A\boldsymbol{p}_i = \boldsymbol{p}_k^T \boldsymbol{b} \tag{3.29}$$

Como $\boldsymbol{p}_k^T A\boldsymbol{p}_i = \boldsymbol{0}$ para $k \neq i$ temos:

$$\alpha_k = \frac{\boldsymbol{p}_k^T \boldsymbol{b}}{\boldsymbol{p}_k^T A\boldsymbol{p}_k} = \frac{\langle \boldsymbol{p}_k, \boldsymbol{b} \rangle}{\langle \boldsymbol{p}_k, A\boldsymbol{p}_k \rangle} = \frac{\langle \boldsymbol{p}_k, \boldsymbol{b} \rangle}{\langle \boldsymbol{p}_k, \boldsymbol{p}_k \rangle_A}, \quad k = 1, 2, ..., n \tag{3.30}$$

Desta forma, se encontrarmos uma sequência de n direções conjugadas e calcularmos os coeficientes α_k podemos encontrar a solução para o sistema $A\boldsymbol{x} = \boldsymbol{b}$.

No entanto, se escolhermos os vetores conjugados $\boldsymbol{p}_k$ cuidadosamente, não necessitaremos de todos os n vetores para obter uma boa aproximação da solução $\boldsymbol{x}$. Desta maneira o método dos gradientes conjugados se torna um método iterativo.

Consideremos a forma quadrática auxiliar:

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T A\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x} \tag{3.31}$$

A Equação 3.31 transforma vetores em números reais. Como $f(\boldsymbol{x})$ é quadrática, há apenas um vetor que a minimiza. Este vetor é exatamente o ponto crítico para a solução de:

$$\nabla f(\boldsymbol{x}) = \boldsymbol{0} \tag{3.32}$$

onde

$$\nabla f(\boldsymbol{x}) = A\boldsymbol{x} - \boldsymbol{b} \tag{3.33}$$

Desta forma, se encontrarmos o ponto de mínimo da Equação 3.31 ele será a solução do sistema linear $A\boldsymbol{x} = \boldsymbol{b}$. Vale notar que o resíduo é o negativo do gradiente da função sendo minimizada.

$$\boldsymbol{r}_k = -\nabla f(\hat{\boldsymbol{x}}_k) = \boldsymbol{b} - A\hat{\boldsymbol{x}}_k \tag{3.34}$$

Logo o resíduo aponta para a região de descrescimento da função $f(\boldsymbol{x})$, levando para o ponto de mínimo da mesma, fazendo $f(\boldsymbol{x}_{k+1}) < f(\boldsymbol{x}_k)$.

Vamos considerar os métodos iterativos de otimização, com $\boldsymbol{p}_k$ sendo as direções de descida de $f(\boldsymbol{x})$:

$$\hat{\boldsymbol{x}}_{k+1} = \hat{\boldsymbol{x}}_k + \alpha_k \boldsymbol{p}_k \tag{3.35}$$

A Equação 3.35 nos diz que a nossa nova aproximação $\hat{\boldsymbol{x}}_{k+1}$ é dado pela última aproximação $\hat{\boldsymbol{x}}_k$ somado da direção de descida de $f(\hat{\boldsymbol{x}}_k)$ multiplicado pelo escalar α_k.

O algoritmo dos gradientes conjugados define direções de busca sucessivas para satisfazer a propriedade de que cada novo passo preserva a otimalidade dos passos anteriores. É bastante complicado derivar o algoritmo dos gradientes conjugados, por esta razão nos limitaremos a apresentar apenas o algoritmo, uma vez que uma idéia básica do método já foi mostrada. Mais detalhes a respeito da derivação do algoritmo podem ser encontrados em HESTENES e STIEFEL (1952). A seguir ilustramos os passos básicos do algoritmo.

procedimento *gradientes_conjugados*($Matriz\ A, vetor\ \boldsymbol{x}, vetor\ \boldsymbol{b}$)

$\boldsymbol{r}_0 \leftarrow \boldsymbol{b} - A\boldsymbol{x}_0$

$\boldsymbol{p}_0 \leftarrow \boldsymbol{r}_0$

para $k \leftarrow 0, 1, ...$ até convergência **faça**

$\alpha_k \leftarrow \frac{\boldsymbol{r}_k^T \boldsymbol{r}_k}{\boldsymbol{p}_k^T A \boldsymbol{p}_k}$

$\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$

$\boldsymbol{r}_{k+1} \leftarrow \boldsymbol{r}_k - \alpha_k A \boldsymbol{p}_k$

$\beta_{k+1} \leftarrow \frac{\boldsymbol{r}_{k+1}^T \boldsymbol{r}_{k+1}}{\boldsymbol{r}_k^T \boldsymbol{r}_k}$

$\boldsymbol{p}_{k+1} \leftarrow \boldsymbol{r}_{k+1} + \beta_{k+1} \boldsymbol{p}_k$

fim para

3.4 Precondicionadores

O precondicionamento de sistemas lineares é uma técnica que visa melhorar o condicionamento da matriz de coeficientes. O condicionamento da matriz mede o quão sensível é a solução do sistema linear aos erros nos dados. A taxa de convergência dos métodos iterativos diminui à medida que o número de condicionamento da matriz cresce. O número de condicionamento para uma matriz A é dado por $\kappa(A) = \|A\|\|A^{-1}\|$.

Supondo que desejamos resolver o sistema linear:

$$A\boldsymbol{x} = \boldsymbol{b} \tag{3.36}$$

Supondo também que M é uma matriz simétrica positiva definida que aproxima A e cuja inversa é muito mais simples de ser obtida. Podemos resolver o sistema 3.36 muito mais facilmente fazendo

$$M^{-1}A\boldsymbol{x} = M^{-1}\boldsymbol{b} \tag{3.37}$$

Se o condicionamento $\kappa(M^{-1}A) \ll \kappa(A)$ então podemos resolver a Equação 3.37 com menos iterações do que a equação original (SHEWCHUK, 1994).

Uma escolha cuidadosa de M, que depende do problema a ser resolvido, geralmente pode fazer com que o número de condicionamento de $M^{-1}A$ seja bem menor que o número de condicionamento de A e desta forma acelerar a convergência drasticamente.

3.5 Gradientes Conjugados Precondicionado

Não podemos aplicar o método dos gradientes conjugados diretamente ao sistema $M^{-1}A\boldsymbol{x} = M^{-1}\boldsymbol{b}$ porque geralmente $M^{-1}A$ não é simétrico positivo definido. Para preservar a simetria podemos observar que $M^{-1}A$ é auto-adjunto para o M-produto-interno.

$$(\boldsymbol{x}, \boldsymbol{y})_M \equiv (M\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x}, M\boldsymbol{y})$$

uma vez que

$$(M^{-1}A\boldsymbol{x}, \boldsymbol{y})_M = (A\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x}, A\boldsymbol{y}) = (\boldsymbol{x}, M(M^{-1}A)\boldsymbol{y}) = (\boldsymbol{x}, M^{-1}A\boldsymbol{y})_M$$

Por esta razão, uma alternativa é substituir o produto interno Euclidiano do método dos gradientes conjugados pelo M-produto-interno.

Se o algoritmo dos gradientes conjugados for reescrito utilizando o novo produto

interno, denotando $\boldsymbol{r}_k = \boldsymbol{b} - A\hat{\boldsymbol{x}}_k$ como o resíduo original e $\boldsymbol{z}_k = M^{-1}\boldsymbol{r}_k$ o resíduo para o sistema precondicionado obtemos o novo algoritmo:

procedimento *gradientes_conjugados_precondicionado*($Matriz\ A, Matriz\ M^{-1}, vetor\ x, vetor\ b$)

$\boldsymbol{r}_0 \leftarrow \boldsymbol{b} - A\boldsymbol{x}_0$

$\boldsymbol{z}_0 \leftarrow M^{-1}\boldsymbol{r}_0$

$\boldsymbol{p}_0 \leftarrow \boldsymbol{z}_0$

para $k \leftarrow 0, 1, ...$ até convergência **faça**

$\alpha_k \leftarrow \frac{\boldsymbol{z}_k^T \boldsymbol{r}_k}{\boldsymbol{p}_k^T A \boldsymbol{p}_k}$

$\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$

$\boldsymbol{r}_{k+1} \leftarrow \boldsymbol{r}_k - \alpha_k A \boldsymbol{p}_k$

$\boldsymbol{z}_{k+1} \leftarrow M^{-1}\boldsymbol{r}_{k+1}$

$\beta_{k+1} \leftarrow \frac{\boldsymbol{z}_{k+1}^T \boldsymbol{r}_{k+1}}{\boldsymbol{z}_k^T \boldsymbol{r}_k}$

$\boldsymbol{p}_{k+1} \leftarrow \boldsymbol{z}_{k+1} + \beta_{k+1} \boldsymbol{p}_k$

fim para

3.5.1 Precondicionador *Jacobi*

O precondicionador mais simples e não muito eficaz é conhecido como *Jacobi* e consiste em escolher a matriz M como sendo a diagonal principal da matriz A. Desta maneira, a matriz M^{-1} é muito fácil de ser obtida por ser uma matriz diagonal. Este precondicionador tem a vantagem de ser totalmente paralelizável. Seja a matriz A formada por $(L + D + U)$ onde D, L e U são matrizes formadas pelas diagonais principal, inferiores e superiores da matriz A, respectivamente. Desta maneira o

precondicionador *Jacobi* é dado por:

$$M = \begin{bmatrix} d_{1,1} & 0 & 0 & \cdots & 0 \\ 0 & d_{2,2} & 0 & \cdots & 0 \\ 0 & 0 & \ddots & & 0 \\ \vdots & \vdots & & & \vdots \\ 0 & 0 & 0 & 0 & d_{n,n} \end{bmatrix} \tag{3.38}$$

$$M^{-1} = \begin{bmatrix} \frac{1}{d_{1,1}} & 0 & 0 & \cdots & 0 \\ 0 & \frac{1}{d_{2,2}} & 0 & \cdots & 0 \\ 0 & 0 & \ddots & & 0 \\ \vdots & \vdots & & & \vdots \\ 0 & 0 & 0 & 0 & \frac{1}{d_{n,n}} \end{bmatrix}, \tag{3.39}$$

onde $d_{i,j}$ são os elementos da matriz D.

3.6 Precondicionador ILU

O precondicionador ILU é baseado no método direto LU. O método LU decompõe a matriz (A) do problema em uma matriz triangular inferior (L) e uma matriz triangular superior (U) de maneira que $A = LU$. Seja o sistema linear

$$A\boldsymbol{x} = \boldsymbol{b} \tag{3.40}$$

como $A = LU$ temos que $LU\boldsymbol{x} = \boldsymbol{b}$. Se assumirmos $U\boldsymbol{x} = \boldsymbol{y}$ temos que

$$L\boldsymbol{y} = \boldsymbol{b} \tag{3.41}$$

$$U\boldsymbol{x} = \boldsymbol{y} \tag{3.42}$$

Após a decomposição LU é necessária a solução de dois sistemas lineares para encontrarmos a solução do sistema $A\boldsymbol{x} = \boldsymbol{b}$. No entanto, as matrizes L e U são matrizes triangulares. Desta forma, os sistemas 3.41 e 3.42 são resolvidos através de substituição e retro-substituição, o que é bem simples e direto.

Existem vários métodos para realizar a decomposição das matrizes em LU, como a decomposição Cholesky para matrizes simétricas positivas definidas. Porém as matrizes L e U geralmente perdem as propriedades de esparsidade da matriz original A. Este é o caso da matriz do problema bidomínio em 2D descrita neste trabalho, que possui valores não nulos apenas em 9 diagonais. Após a decomposição as matrizes L e U são menos esparsas, possuindo poucos elementos não nulos. A Figura 3.1 ilustra como a decomposição de uma matriz esparsa com 9 diagonais gera matrizes com menor esparsidade para um problema com dimensão 6×6. Para problemas maiores as diagonais inferiores e superiores se encontrariam cada vez mais afastadas da diagonal principal aumentando ainda mais o número de elementos não-zero das matrizes.

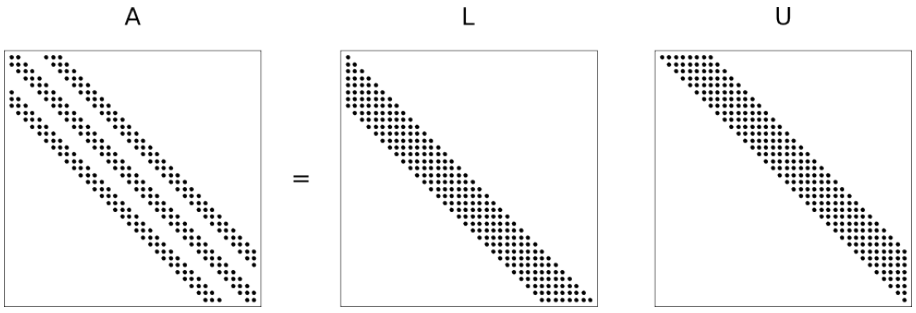

Figura 3.1: Ilustração de uma descomposição LU por Cholesky. No caso da decomposição Cholesky, como A é simétrica positiva definida, $U = L^T$.

O precondicionador ILU (Incomplete LU factorization) é baseado em uma decomposição LU incompleta onde a matriz resíduo $R = LU - A$, satisfaz algumas

restrições. A decomposição incompleta gera matrizes L e U que são próximas das que seriam geradas por uma fatoração completa. No precondicionador ILU(0), que será utilizado neste trabalho, as matrizes LU possuem a mesma esparsidade da matriz A. A Figura 3.2 mostra a esparsidade das matrizes L e U para o precondicionador ILU(0).

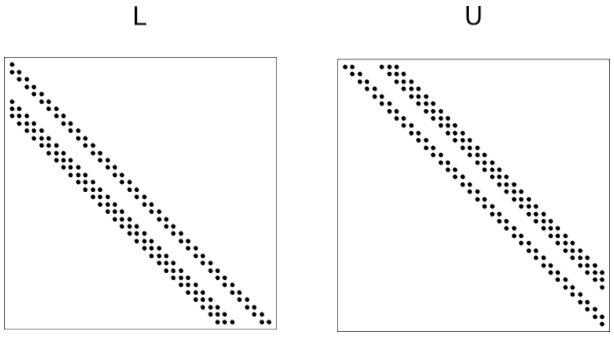

Figura 3.2: Ilustração de uma descomposição LU incompleta com a mesma esparsidade de A.

Como a fatoração incompleta gerou matrizes tais que $LU \approx A$, podemos utilizar esta decomposição como precondicionador para o problema original.

3.7 Precondicionador Block-ILU

A paralelização do precondicionador ILU pode ser realizada dividindo a matriz do precondicionador em blocos independentes para cada processo paralelo. Para isso, cada bloco utiliza apenas os elementos da matriz que pertencem ao bloco para a aplicação do precondicionador ILU. A Figura 3.3 mostra como a matriz do precondicionador fica dividida após a partição em blocos paralelos.

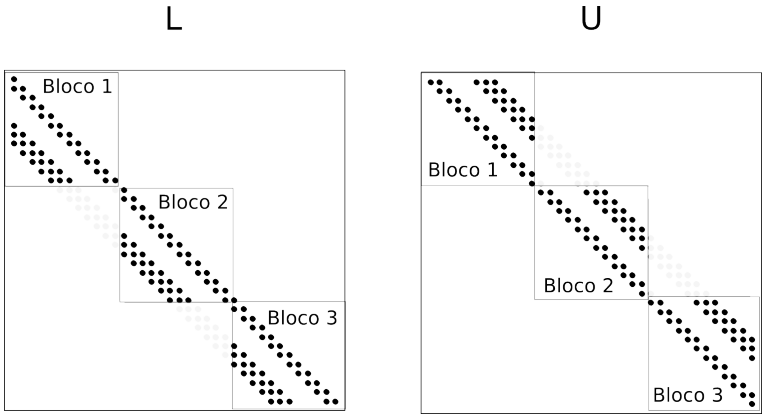

Figura 3.3: Divisão da matriz do precondicionador ILU(0) em 3 blocos.

3.8 Precondicionador Multigrid

Os métodos multigrid foram originalmente aplicados a problemas de valor de contorno que advém de aplicações físicas (BRIGGS *et al.*, 2000). Atualmente os princípios dos métodos multigrid não são apenas utilizados para a solução de equações diferenciais mas estão sendo aplicados em diversas áreas como teoria de controle, otimização, reconhecimento de padrões e tomografia computacional (WESSELING, 1992). A idéia básica atrás dos métodos multigrid é que a taxa de convergência dos métodos iterativos pode ser melhorada através de sua utilização. Os métodos multigrid podem também ser utilizados como precondicionadores aplicando um pequeno número de iterações do mesmo com a matriz original do problema.

Nesta seção, o principio do método multigrid será apresentado utilizando como exemplo uma equação diferencial parcial em uma dimensão (1D). Geralmente esse tipo de problema pode ser resolvido analiticamente mas, em 1D, o método multigrid pode ser mais facilmente demonstrado.

$$-u''(x) + \sigma u(x) = f(x), \qquad 0 < x < 1, \quad \sigma \geq 0, \tag{3.43}$$

$$u(0) = u(1) = 0 \tag{3.44}$$

Várias abordagens são possíveis para a discretização numérica desta equação. A mais simples entre elas é o método de diferenças finitas. O domínio do problema é particionado em n sub-intervalos introduzindo os pontos $x_j = jh$ no grid, onde $h = 1/n$ é a largura constante dos sub-intervalos. Isto estabelece o grid mostrado na Figura 3.4, que denotamos por Ω_h.

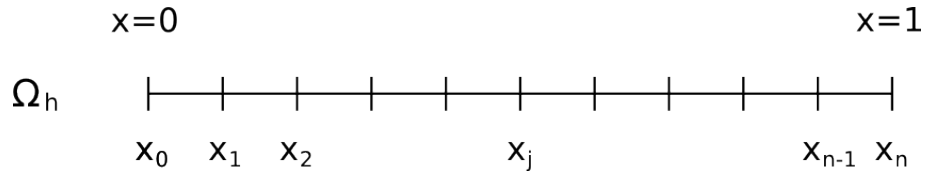

Figura 3.4: Grid de 1 dimensão no intervalo $0 \leq x \leq 1$.

A cada um dos $n-1$ pontos interiores do grid, a equação diferencial original (3.43), é substituída por uma aproximação de diferenças finitas de segunda ordem. Fazendo essa aproximação introduzimos $\hat{u}_j$ como uma aproximação da solução $u(x_j)$. Essa aproximação pode ser representada pelo vetor $\hat{\boldsymbol{u}} = (\hat{u}_1, \hat{u}_2, ..., \hat{u}_{n-1})^T$, no qual os componentes satisfazem as $n-1$ equações lineares.

$$\frac{-\hat{u}_{j-1} + 2\hat{u}_j - \hat{u}_{j+1}}{h^2} + \sigma \hat{u}_j = f(x_j), \qquad 1 \leq j \leq n-1,$$

$$\hat{u}_0 = \hat{u}_n = 0 \tag{3.45}$$

Definindo como $\boldsymbol{f} = (f(x_1), \cdots, f(x_{n-1}))^T = (f_1, \cdots, f_{n-1})^T$, o vetor de valores do lado direito da equação.

Agora vamos considerar como tratar esse problema proposto. Primeiro vamos dizer que o sistema que queremos resolver é dado pelo sistema linear:

$$A\boldsymbol{u} = \boldsymbol{f}$$

Vamos supor que $A\boldsymbol{u} = \boldsymbol{f}$ tem uma única solução e que $\hat{\boldsymbol{u}}$ é uma aproximação da solução exata $\boldsymbol{u}$. Existem dois tipos de medidas importantes de $\hat{\boldsymbol{u}}$ como aproximação de $\boldsymbol{u}$. Um é o erro (ou erro algébrico) e é dado por:

$$\boldsymbol{e} = \boldsymbol{u} - \hat{\boldsymbol{u}}$$

Infelizmente o erro é tão desconhecido quanto a solução exata. No entanto, uma medida que pode ser calculada é o resíduo.

$$\boldsymbol{r} = \boldsymbol{f} - A\hat{\boldsymbol{u}}$$

O resíduo é simplesmente a quantidade que a aproximação $\hat{\boldsymbol{u}}$ falha em satisfazer o problema original $A\boldsymbol{u} = \boldsymbol{f}$.

Lembrando que $A\boldsymbol{u} = \boldsymbol{f}$ e usando as definições de $\boldsymbol{e}$ e $\boldsymbol{r}$, podemos derivar uma relação muito importante entre o erro e o resíduo.

$$A\boldsymbol{e} = \boldsymbol{r} \tag{3.46}$$

Essa relação é chamada de equação residual. E diz que o erro satisfaz o mesmo conjunto de equações que a equação original substituindo $\boldsymbol{f}$ por $\boldsymbol{r}$. A equação

residual é muito importante nos métodos multigrid. Suponha que uma aproximação $\hat{\boldsymbol{u}}$ foi computada por algum método. O resíduo pode ser facilmente calculado por $\boldsymbol{r} = \boldsymbol{f} - A\hat{\boldsymbol{u}}$. Para melhorar a aproximação $\hat{\boldsymbol{u}}$, poderíamos resolver a equação residual para $\boldsymbol{e}$ e depois calcular uma nova aproximação usando a definição do erro:

$$\boldsymbol{u} = \hat{\boldsymbol{u}} + \boldsymbol{e} \tag{3.47}$$

A idéia da correção residual é muito importante em tudo que seguirá.

Agora vamos tratar os métodos de relaxação, como Jacobi e ω-Jacobi, para o problema descrito na Equação 3.45 com $\sigma = 0$. Multiplicando equação por h^2 por conveniência, o problema discreto se torna:

$$\begin{aligned} -\hat{u}_{j-1} + 2\hat{u}_j - \hat{u}_{j+1} &= h^2 f(x_j), \qquad 1 \leq j \leq n-1, \\ \hat{u}_0 = \hat{u}_n &= 0 \end{aligned} \tag{3.48}$$

Método iterativo de Jacobi

Um dos esquemas iterativos mais simples é chamado método de *Jacobi*. É produzido ao resolvermos a j-ésima equação de (3.48) para o j-ésima incógnita utilizando as aproximações atuais para as incógnitas $\hat{u}_{j-1}$ e $\hat{u}_{j+1}$. Aplicado ao vetor de aproximações atual produz o seguinte esquema iterativo:

$$\hat{u}_j^{(k+1)} = \frac{1}{2}(\hat{u}_{j-1}^{(k)} + \hat{u}_{j+1}^{(k)} + h^2 f_j), \qquad 1 \leq j \leq n-1 \tag{3.49}$$

onde $k+1$ é nova aproximação para j-ésima incógnita de $\boldsymbol{u}$. Esse processo é repetido para todas as equações de (3.48) e teremos uma nova aproximação para cada incógnita. Vale a pena lembrar que apenas os valores antigos são utilizados nas novas aproximações, ou seja, cada iteração do método de *Jacobi* pode ser totalmente paralelizada uma vez que não existe nenhuma dependência entre as incógnitas.

Método iterativo ω-Jacobi

Existe uma simples porém importante modificação que pode ser feita no método iterativo de *Jacobi*. Da mesma forma anterior computamos:

$$\hat{u}_j^{(*)} = \frac{1}{2}(\hat{u}_{j-1}^{(k)} + \hat{u}_{j+1}^{(k)} + h^2 f_j), \qquad 1 \leq j \leq n-1 \tag{3.50}$$

No entanto, $\hat{u}_j^*$ é apenas um valor intermediário. A nova iteração é dada pela média ponderada:

$$\hat{u}_j^{(k+1)} = (1-\omega)\hat{u}_j^{(k)} + w\hat{u}_j^{(*)}, \qquad 1 \leq j \leq n-1 \tag{3.51}$$

onde $\omega \in \mathbb{R}$ é um peso que deve ser escolhido. Isto gera uma família inteira de métodos iterativos chamados de método ω-*Jacobi*. Note que se escolhermos $\omega = 1$ obtemos o método iterativo *Jacobi* original. Da mesma maneira que o método *Jacobi* original, o método ω-*Jacobi* computa todos os valores da nova aproximação antes de utilizar qualquer um deles.

Análise dos métodos iterativos

Apresentamos anteriormente dois métodos iterativos para a solução de sistemas lineares. No entanto, existem inúmeros outros métodos iterativos, como o método *Gauss-Seidel* que se assemelha ao método de *Jacobi*, porém as novas aproximações computadas para cada incógnita são utilizadas no cálculo das próximas incógnitas na mesma iteração.

Agora analisaremos os métodos apresentados. Para a análise é suficiente trabalhar apenas com o sistema linear homogêneo $A\boldsymbol{u} = \mathbf{0}$ e utilizar uma aproximação inicial arbitrária. Uma razão para isso é que a solução exata é conhecida e bem simples ($\boldsymbol{u} = \mathbf{0}$) e o erro para uma aproximação $\hat{\boldsymbol{u}}$ é simplesmente $-\hat{\boldsymbol{u}}$. Desta maneira retornamos ao problema inicial com $\boldsymbol{f} = \mathbf{0}$.

$$\begin{aligned} -\hat{u}_{j-1} + 2\hat{u}_j - \hat{u}_{j+1} &= 0, \qquad 1 \leq j \leq n-1, \\ \hat{u}_0 = \hat{u}_n &= 0 \end{aligned} \tag{3.52}$$

Podemos obter resultados bastante interessantes se aplicarmos os métodos iterativos com uma aproximação inicial dada pelos vetores ou modos de *Fourier*

$$\hat{u}_j = sen\left(\frac{jk\pi}{n}\right), \qquad 0 \leq j \leq n, \quad 1 \leq k \leq n-1 \tag{3.53}$$

Lembrando que j denota o componente do vetor $\hat{\boldsymbol{u}}$. O inteiro k é chamado de número de onda, ou frequência, e indica o número de semi-ondas de seno que constituem $\hat{\boldsymbol{u}}$ no domínio do problema. A Figura 3.5 ilustra as aproximações iniciais $\hat{\boldsymbol{u}}_1$, $\hat{\boldsymbol{u}}_3$ e $\hat{\boldsymbol{u}}_6$, onde os subscritos representam o valor de k. Repare que os modos com valores de k

pequenos correspondem a ondas suaves, enquanto que valores grandes correspondem a ondas que oscilam mais. Agora vamos explorar o comportamento dos modos de *Fourier* sobre as iterações.

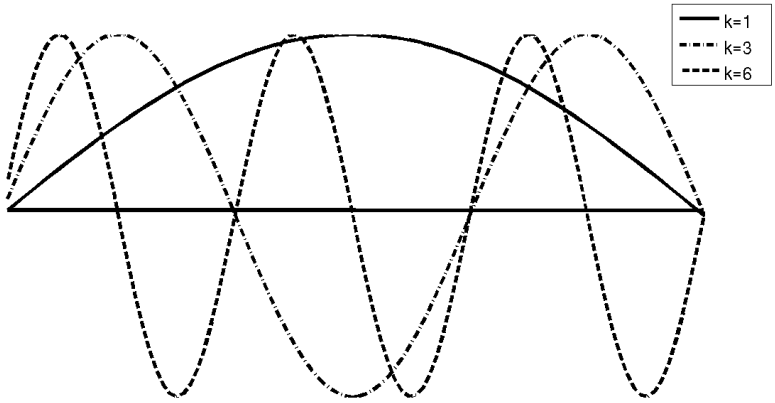

Figura 3.5: Os vetores como números de onda $k = 1, 3, 6$. Onde o k-ésimo modo consiste de $k/2$ ondas de seno completas no intervalo. Adaptado de BRIGGS *et al.* (2000)

Vamos primeiramente aplicar o método ω-*Jacobi* com $\omega = \frac{2}{3}$ ao problema 3.52 em um grid de 64 pontos. Utilizando as aproximações iniciais $\hat{u}_1$, $\hat{u}_3$ e $\hat{u}_6$, a iteração é aplicada 100 vezes. Utilizando a seguinte norma para o erro:

$$\|\boldsymbol{e}\|_\infty = \max_{1 \leq j \leq n} |e_j|$$

Relembrando que o vetor erro é dado por $-\hat{\boldsymbol{u}}$. A Figura 3.6 mostra o gráfico da norma-∞ do erro com relação ao número da iteração.

Podemos observar na Figura 3.6 que o erro decresce em cada iteração e que a taxa de decrescimento desse erro é maior para as aproximações iniciais com maior frequência. A Figura 3.7 mostra como as novas aproximações ficam após as 100 iterações.

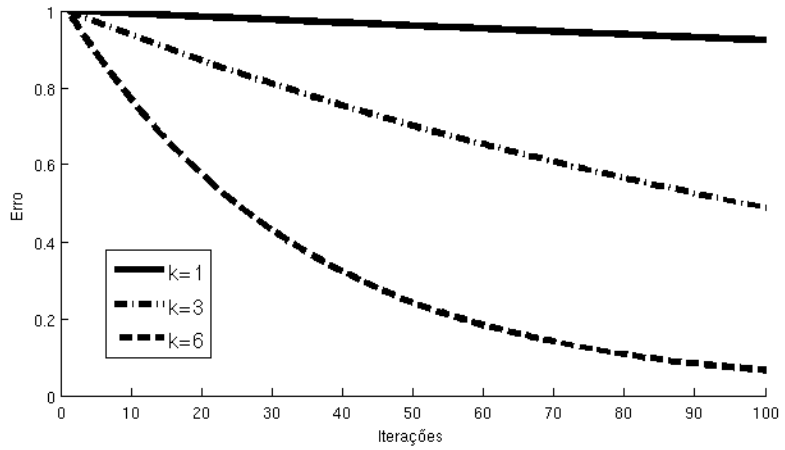

Figura 3.6: Iteração com ω-*Jacobi* com $\omega = \frac{2}{3}$ aplicado ao problema modelo com $n = 64$ pontos e aproximações iniciais $\hat{\boldsymbol{u}}_1$, $\hat{\boldsymbol{u}}_3$ e $\hat{\boldsymbol{u}}_6$. A norma $\|e\|_\infty$, é mostrada em relaxação ao número de 100 iterações. Adaptado de BRIGGS *et al.* (2000)

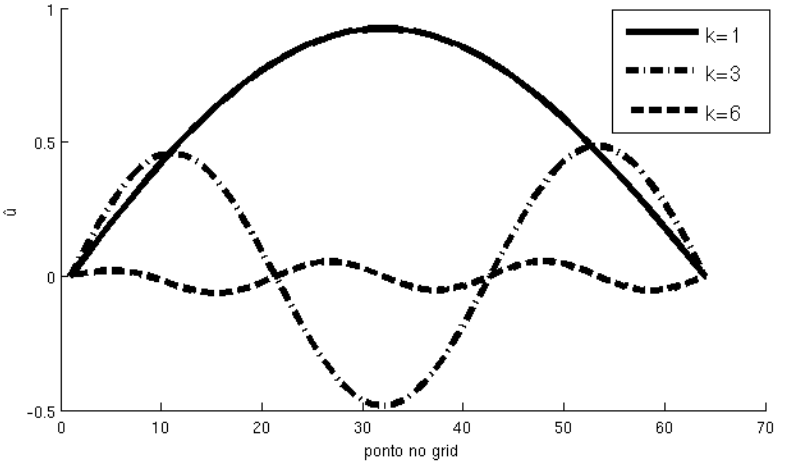

Figura 3.7: Iteração com ω-*Jacobi* com $\omega = \frac{2}{3}$ aplicado ao problema modelo com $n = 64$ pontos e aproximações iniciais $\hat{\boldsymbol{u}}_1$, $\hat{\boldsymbol{u}}_3$ e $\hat{\boldsymbol{u}}_6$. As novas aproximações $\hat{\boldsymbol{u}}_1$, $\hat{\boldsymbol{u}}_3$ e $\hat{\boldsymbol{u}}_6$ são mostradas após 100 iterações.

Em geral, a maioria das aproximações iniciais não consistem de apenas um único modo. Vamos considerar uma situação um pouco mais realista onde a aproximação inicial consiste de dois modos

$$w_j = \frac{1}{2}\left(sen\left(\frac{jk_1\pi}{n}\right) + sen\left(\frac{jk_2\pi}{n}\right)\right), \qquad 0 \leq j \leq n \tag{3.54}$$

k_1 e k_2 são os dois modos e w a aproximação inicial com dois modos. Realizamos a implementação do método de ω-Jacobi e utilizamos w como aproximação inicial. A Figura 3.8 mostra a aproximação inicial com $k_1 = 2$ e $k_2 = 16$.

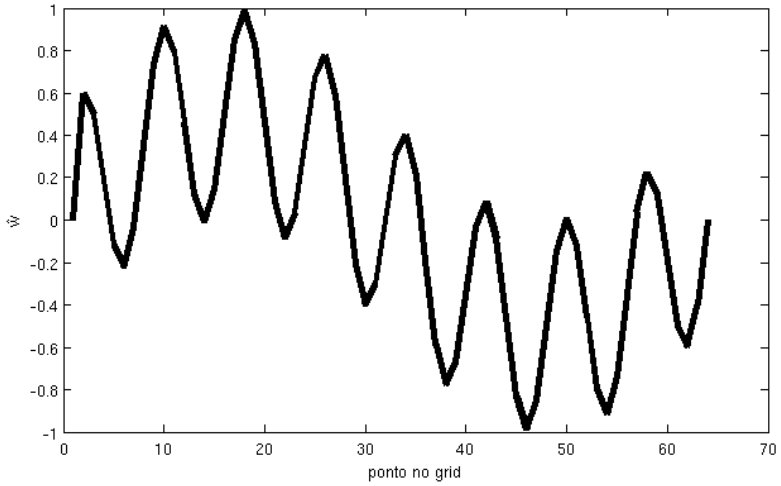

Figura 3.8: Aproximação inicial para o problema modelo com 64 pontos no grid dado pela Equação 3.54. Com $k_1 = 2$ e $k_2 = 16$.

A Figura 3.9 mostra o decrescimento do erro e nova aproximação da solução em apenas 10 iterações do método ω-*Jacobi* implementado com $\omega = \frac{2}{3}$. Podemos também observar como a frequência alta foi suprimida bem mais rapidamente que a frequência baixa que permaneceu quase sem alterações.

Através de alguns experimentos examinamos o método ω-*Jacobi*. Porém, este método compartilha várias propriedades com outros métodos iterativos. Foi possível perceber que os métodos iterativos possuem uma propriedade de suavização. Esta propriedade faz com que esses métodos sejam bastante eficientes para eliminar altas

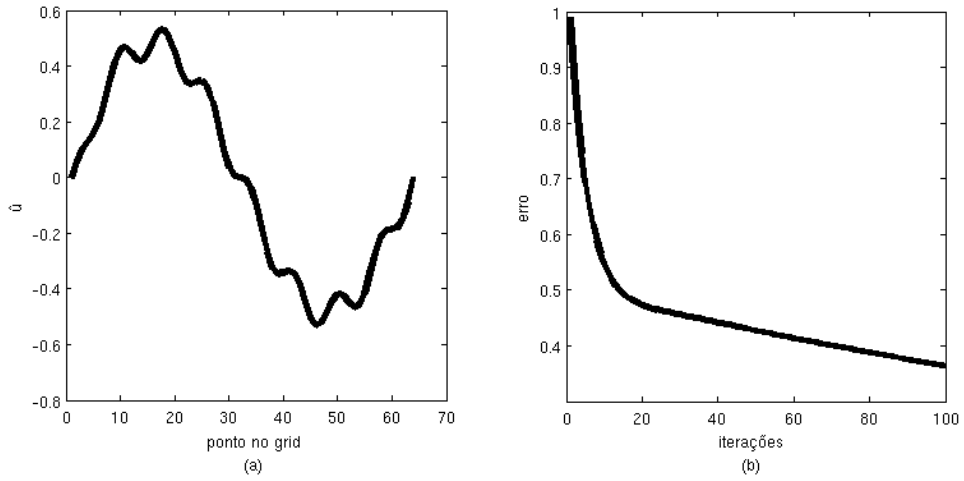

Figura 3.9: Iteração com ω-*Jacobi* com $\omega = \frac{2}{3}$ aplicado ao problema modelo com $n = 64$ pontos e aproximação inicial dada pela Equação 3.54 com $k_1 = 2$ e $k_2 = 16$. A nova aproximação é mostrada em (a) após 10 iterações e o erro $\|e\|_\infty$ em (b) em 100 iterações.

frequências do erro, enquanto deixa os componentes de baixa frequência quase sem mudança (BRIGGS *et al.*, 2000).

Grids

Utilizamos anteriormente para nosso problema modelo uma discretização em um grid com 64 pontos. No entanto, poderíamos escolher uma discretização com mais ou menos pontos. Se escolhêssemos um grid com 32 pontos teríamos um grid que diríamos ser mais grosseiro que o de 64 pontos. Apesar de escolhermos discretizações diferentes estaríamos resolvendo o mesmo problema com a diferença que agora o sistema linear teria menos equações. Vamos assumir que um esquema de relaxação como ω-*Jacobi* foi aplicado até que apenas as componentes suaves do erro permanecessem. Agora queremos saber o que aconteceria com essas componentes suaves no grid mais grosseiro. A Figura 3.10 nos mostra a resposta. Uma curva

suave com $k = 4$ em um grid Ω^h com 12 pontos foi projetado em um grid Ω^{2h} com 6 pontos. No grid grosseiro, a onda original ainda possui o número de onda k que é igual a 4, porém parece ser mais oscilatório em Ω^{2h}.

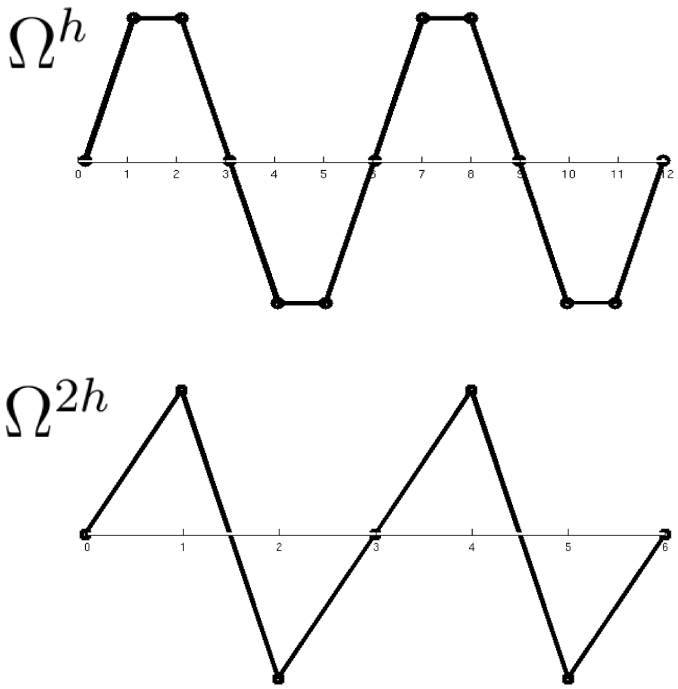

Figura 3.10: Transferência entre grids.

A parte importante é que os modos que são suaves no grids mais finos parecem menos suaves nos grids mais grosseiros. O que sugere que quando a convergência começa a estagnar, e os modos de erros suaves predominam, é aconselhável transferir para o grid grosseiro onde os modos dos erros parecem ser mais oscilatórios e os métodos iterativos serão mais eficientes. O problema agora é como transferir para o grid grosseiro. Como visto anteriormente temos uma equação para erro, a equação residual. Se $\hat{\boldsymbol{u}}$ é uma aproximação para a solução exata $\boldsymbol{u}$, então o erro $\boldsymbol{e} = \boldsymbol{u} - \hat{\boldsymbol{u}}$ satisfaz

$$A\boldsymbol{e} = \boldsymbol{r} = \boldsymbol{f} - A\hat{\boldsymbol{u}}$$

que diz que podemos relaxar diretamente no erro utilizando a equação residual. A relaxação no problema original $A\boldsymbol{u} = \boldsymbol{f}$ com uma aproximação inicial $\hat{\boldsymbol{u}}$ é equivalente a relaxar na equação residual $A\boldsymbol{e} = \boldsymbol{r}$ com aproximação inicial específica de $\boldsymbol{e} = \boldsymbol{0}$.

Após resolver $A\boldsymbol{e} = \boldsymbol{r}$ no grid mais grosseiro podemos transferir o erro para o grid mais fino e corrigir a solução

$$\hat{\boldsymbol{u}}^h = \hat{\boldsymbol{u}}^h + \boldsymbol{e}^{2h} \tag{3.55}$$

Esse esquema é chamado de esquema de correção. Em nossa discussão de transferências entre grids consideramos apenas o caso onde o grid mais grosseiro possui o dobro do espaçamento do grid mais fino. O processo de transferir do grid mais grosseiro para o grid mais fino é chamado interpolação ou prolongamento. Vamos considerar apenas a interpolação linear que apesar de simples é bem eficaz. O operador de interpolação linear será denotado por I_{2h}^h. Que transfere vetores no grid grosseiro para o grid fino de acordo com a regra $I_{2h}^h \boldsymbol{v}^{2h} = \boldsymbol{v}^h$, onde

$$v_{2j}^h = v_j^{2h}, \tag{3.56}$$

$$v_{2j+1}^h = \frac{1}{2}\left(v_j^{2h} + v_{j+1}^{2h}\right), \quad 0 \leq j \leq \frac{n}{2} - 1 \tag{3.57}$$

Para o caso $n = 8$ teríamos este operador na forma matricial

$$I_{2h}^{h}\hat{\boldsymbol{u}}^{2h} = \frac{1}{2}\begin{bmatrix} 1 & & \\ 2 & & \\ 1 & 1 & \\ & 2 & \\ & 1 & 1 \\ & & 2 \\ & & 1 \end{bmatrix}\begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \end{bmatrix}_{2h} = \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \\ \hat{u}_4 \\ \hat{u}_5 \\ \hat{u}_6 \\ \hat{u}_7 \end{bmatrix}_{h} = \hat{\boldsymbol{u}}^{h} \tag{3.58}$$

Para problemas em 2 dimensões, o operador de interpolação pode ser definido de uma maneira parecida. Seja $I_{2h}^{h}\hat{u}^{2h} = \hat{u}^{h}$ então os componentes de $\hat{u}^{h}$ são dados por:

$$\begin{aligned}
\hat{u}_{2i,2j}^{h} &= \hat{u}_{i,j}^{2h}, \\
\hat{u}_{2i+1,2j}^{h} &= \frac{1}{2}(\hat{u}_{i,j}^{2h} + \hat{u}_{i+1,j}^{2h}), \\
\hat{u}_{2i,2j+1}^{h} &= \frac{1}{2}(\hat{u}_{i,j}^{2h} + \hat{u}_{i,j+1}^{2h}), \\
\hat{u}_{2i+1,2j+1}^{h} &= \frac{1}{4}(\hat{u}_{i,j}^{2h} + \hat{u}_{i+1,j}^{2h} + \hat{u}_{i,j+1}^{2h} + \hat{u}_{i+1,j+1}^{2h}), \qquad 0 \leq i,j \leq \frac{n}{2} - 1
\end{aligned}$$

A segunda classe de operadores de transferência entre grids envolve transferir vetores de um grid fino para um grid grosseiro. São geralmente chamados de operadores de restrição e são denotados por I_{h}^{2h}. Um tipo de operador de restrição é o *full-weighting* definido por $I_{h}^{2h}\hat{u}^{h} = \hat{u}^{2h}$, onde

$$\hat{u}_{j}^{2h} = \frac{1}{4}(\hat{u}_{2j-1}^{h} + 2\hat{u}_{2j}^{h} + \hat{u}_{2j+1}^{h}), \qquad 1 \leq j \leq \frac{n}{2} - 1$$

O operador *full-weighting* para o caso $n = 8$ tem a forma:

$$I_h^{2h}\hat{u}^h = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 & & & & \\ & & 1 & 2 & 1 & & \\ & & & & 1 & 2 & 1 \end{bmatrix}\begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \\ \hat{u}_4 \\ \hat{u}_5 \\ \hat{u}_6 \\ \hat{u}_7 \end{bmatrix}_h = \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \end{bmatrix}_{2h} = \hat{u}^{2h} \tag{3.59}$$

Um motivo para escolher o *full-weighting* como operador de restrição é o importante fato de que

$$I_{2h}^h = c(I_h^{2h})^T, \qquad c \in \mathbb{R}$$

O fato de que os operadores de restrição e de interpolação são transpostos um do outro a uma constante é chamada propriedade variacional (BRIGGS *et al.*, 2000) e é importante para o método multigrid.

Para problemas em 2 dimensões o operador de restrição *full-weighting* é dado como uma média dos vizinhos mais próximos no grid fino. Seja $I_h^{2h}\boldsymbol{v}^h = \boldsymbol{v}^{2h}$, temos que

$$\begin{aligned} \hat{u}_{i,j}^{2h} = \quad & \tfrac{1}{16}[\hat{u}_{2i-1,2j-1}^h + \hat{u}_{2i-1,2j+1}^h + \hat{u}_{2i+1,2j+1}^h + \\ & 2(\hat{u}_{2i,2j-1}^h + \hat{u}_{2i,2j+1}^h + \hat{u}_{2i-1,2j}^h + \hat{u}_{2i+1,2j}^h) + \\ & 4\hat{u}_{2i,2j}^h], \qquad 1 \leq i,j \leq \tfrac{n}{2} - 1. \end{aligned}$$

Agora temos os dois operadores de transferência entre os grids grosseiro e fino. Podemos então finalizar o algoritmo multigrid.

Algoritmo Multigrid

O algoritmo multigrid que vamos definir é um esquema de correção com os ciclos em V. Existem outros tipos de ciclos como ciclos em W, porém aqui iremos apenas definir o ciclo em V por ser o utilizado no trabalho. No algoritmo que vamos descrever, a *relaxação* se refere aos métodos iterativos como ω-*Jacobi* descrito.

---

procedimento $Multigrid(A^{h_{grid}}, \hat{u}^{h_{grid}}, r^{h_{grid}})$

se ($grid \neq$ mais grosseiro) **então**

se ($grid \neq$ mais fino) **então**

$\hat{u}^{h_{grid}} \leftarrow 0$

fim se

Relaxar $(A^{h_{grid}}, \hat{u}^{h_{grid}}, f^{h_{grid}}, v_{times})$;

$r^{h_{grid}} \leftarrow f^{h_{grid}} - A^{h_{grid}} \hat{u}^{h_grid}$

$r^{h_{grid+1}} \leftarrow I_{h_{grid}}^{h_{grid+1}} r^{h_{grid}}$

Multigrid$(A^{h_{grid+1}}, \hat{u}^{h_{grid+1}}, r^{h_{grid+1}})$

$\hat{u}^{h_{grid}} \leftarrow \hat{u}^{h_{grid}} + I_{h_{grid+1}}^{h_{grid}} \hat{u}^{h_{grid+1}}$

Relaxar $(A^{h_{grid}}, \hat{u}^{h_{grid}}, f^{h_{grid}}, v_{times})$;

senão

$\hat{u}^{h_{grid}} \leftarrow (A^{h_{grid}})^{-1} f^{h_{grid}}$

fim se

O método multigrid pode ser utilizado também como um precondicionador para o gradiente conjugado precondicionado que descrevemos anteriormente. Basta aplicarmos o método multigrid uma ou mais vezes com a matriz original e com o resíduo no lado direito da equação como descrito na Seção 3.5 e utilizarmos a aproximação obtida como solução da equação $z = M^{-1}r$.

Capítulo 4

Programação em GPU

Os avanços tecnológicos recentes e a natureza paralela das operações envolvidas na renderização 3D em tempo real transformaram as placas gráficas atuais em máquinas com grande poder computacional paralelo. Os hardwares de processamento gráfico, conhecidos como *Graphics Processor Units* ou GPUs, são, provavelmente, os hardwares com a melhor relação entre custo e desempenho da atualidade.

As GPUs antigas possuiam um conjunto de funções fixas distribuídas em uma *pipeline*. Recentemente, alguns estágios dessa *pipeline* se tornaram programáveis permitindo que problemas computacionais comuns possam ser resolvidos utilizando seu grande poder computacional. A utilização de GPUs para a computação de propósito geral é conhecida como GPGPU (*General-Purpose Computing on the GPU*).

As GPUs têm tido um aumento de desempenho superior ao obtido pelas CPUs ao longo dos anos. Isso se deve à diferenças cruciais na arquitetura das duas plataformas. As CPUs são otimizadas para obter desempenho em código sequencial extraindo performance no paralelismo de instruções, enquanto as GPUs obtêm seu desempenho no paralelismo de dados, utilizando mais transistores diretamente na

computação.

Muitos pesquisadores têm aplicado esta nova tecnologia a problemas que eram resolvidos anteriormente em CPUs. As GPUs consistem de um conjunto de multiprocessadores desenvolvidos para obter um ótimo desempenho em computações gráficas. Apesar da programação de GPUs para a computação de propósito geral estar se tornando mais popular em virtude de sua promessa de computação altamente paralela, extrair um bom desempenho geralmente não é uma tarefa fácil pois existem limitações onde as GPUs não obtêm um desempenho satisfatório. Por exemplo, este é o caso das operações de escrita do tipo *"scatter"*, isto é, uma operação de escrita em qualquer local da memória. Outra limitação é o suporte à precisão dupla, encontrado apenas nos hardwares gráficos de última geração. Ainda assim, este suporte só é possível utilizando a linguagem para GPGPU do fabricante do hardware.

Foram desenvolvidas várias linguagens para a programação de GPUs tanto para propósitos gráficos como para propósitos mais gerais. Neste trabalho foi utilizada uma abordagem com a OpenGL que é uma interface de software para o hardware gráfico (SHREINER *et al.*, 2005) utilizando a linguagem GLSL (ROST, 2005) para a programação dos processadores da GPU. No entanto, existem outras linguagens e abordagens, que não serão discutidas aqui, mas valem a pena serem citadas como a linguagem BROOK (BUCK *et al.*, 2004) desenvolvida para ser portável para todos os hardwares gráficos programáveis, e o modelo de programação CUDA (NVIDIA, 2008) que é uma extensão da linguagem de programação C, para os hardwares gráficos da NVIDIA.

4.1 Arquitetura da GPU

O domínio das aplicações 3D em tempo real possui várias características que o diferencia de domínios computacionais mais gerais. Aplicações 3D necessitam de altas taxas de computação e paralelismo. A construção de um hardware que utiliza o paralelismo intrínseco da aplicação permite o alto desempenho das aplicações gráficas. A Figura 4.1 mostra a evolução das CPUs e GPUs em relação ao número de operações em ponto flutuante por segundo (GFLOPS).

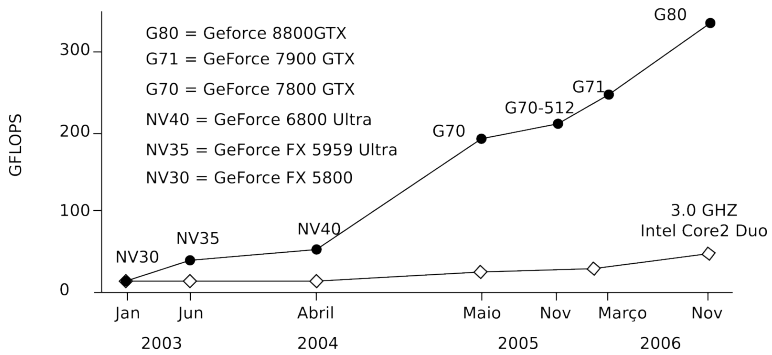

Figura 4.1: Comparação entre a evolução da GPU e CPU ao longo dos últimos anos em relação aos GFLOPS.

O objetivo das CPUs é processar uma tarefa o mais rápido possível enquanto as GPUs devem ser capazes de processar o máximo de tarefas em uma grande quantidade de dados. As prioridades de ambas não são as mesmas, suas respectivas arquiteturas mostram isso.

A Figura 4.2 mostra as diferenças entre as duas arquiteturas. Grande parte da CPU é destinada a uma unidade de controle mais elaborada e uma memória cache maior enquanto a GPU tem um grande número de unidades de processamento. Dessa forma, enquanto na CPU grandes caches são utilizadas para diminuir a latência do acesso à memória, nas GPUs isso é obtido através da sobreposição de computação

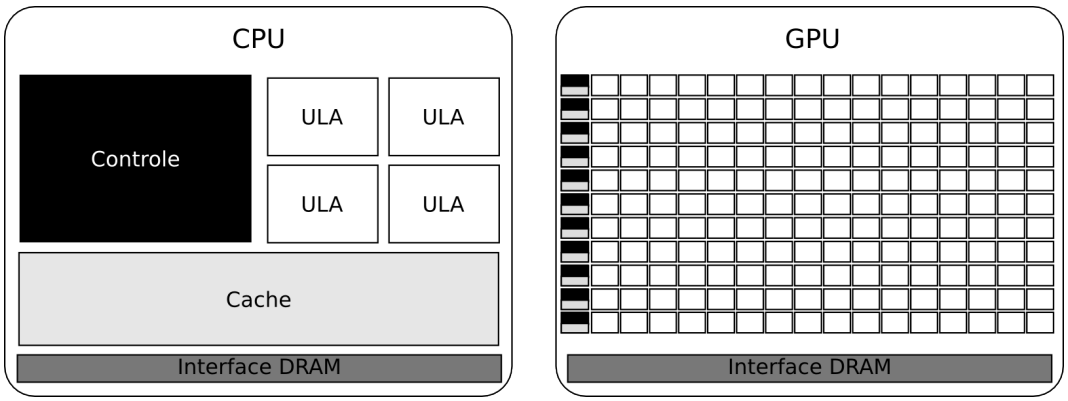

Figura 4.2: Arquiteturas da CPU e GPU.

e uma grande largura de banda de memória.

4.1.1 *Pipeline* Gráfica

Todas as GPUs atuais tem sua computação organizada na chamada *pipeline* gráfica. Esta *pipeline* é desenvolvida para permitir que implementações de hardware tenham altas taxas de computação através do paralelismo. A *pipeline* é dividida em vários estágios. Todas as primitivas geométricas passam por todos os estágios. Na GPU cada estágio é implementado como uma peça separada de hardware. A Figura 4.3 mostra os estágios de GPUs atuais (OWENS *et al.*, 2007).

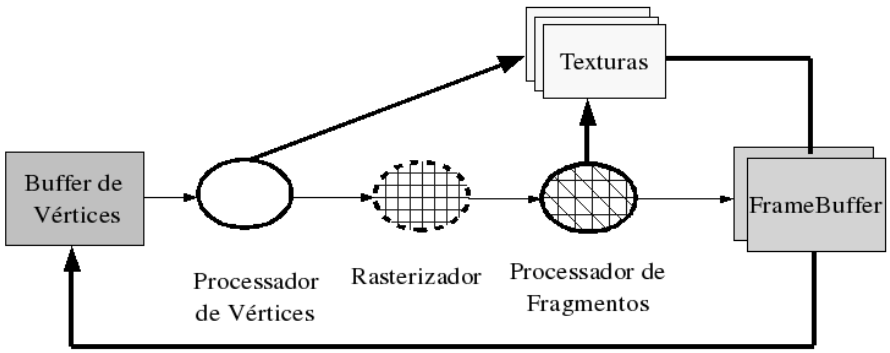

Figura 4.3: *Pipeline* Gráfica moderna. Os processadores de vértices e fragmentos são ambos programáveis.

A entrada da *pipeline* é uma lista de vértices e a saída, uma imagem no *framebuffer*. No primeiro estágio da *pipeline* o processador de vértices transforma os vértices em triângulos, e geralmente faz cálculos da iluminação em cada vértice. A saída deste estágio são triângulos no espaço da tela. O próximo estágio, a rasterização, determina as posições na tela cobertas por cada triângulo e interpola os parâmetros dos vértices no triângulo. O resultado do estágio de rasterização é um fragmento por cada pixel coberto pelo triângulo. O terceiro estágio, processador de fragmentos, computa a cor de cada fragmento, usando os valores interpolados anteriormente. Esta computação pode utilizar valores da memória global que geralmente fazem parte de uma imagem chamada textura. E finalmente os fragmentos são montados em uma imagem de pixels, geralmente escolhendo os fragmentos mais próximos da câmera para cada pixel.

O processador de vértices e de fragmentos, nos hardwares atuais, são programáveis. Estes estágios possuem múltiplos processadores. A Figura 4.4 mostra os processadores de vértices e de fragmentos.

4.2 Programação em GPU

Nesta seção será apresentada de forma simples uma comparação entre a programação em GPU para propósitos gráficos, CPU, e GPU para propósitos gerais (GPGPU).

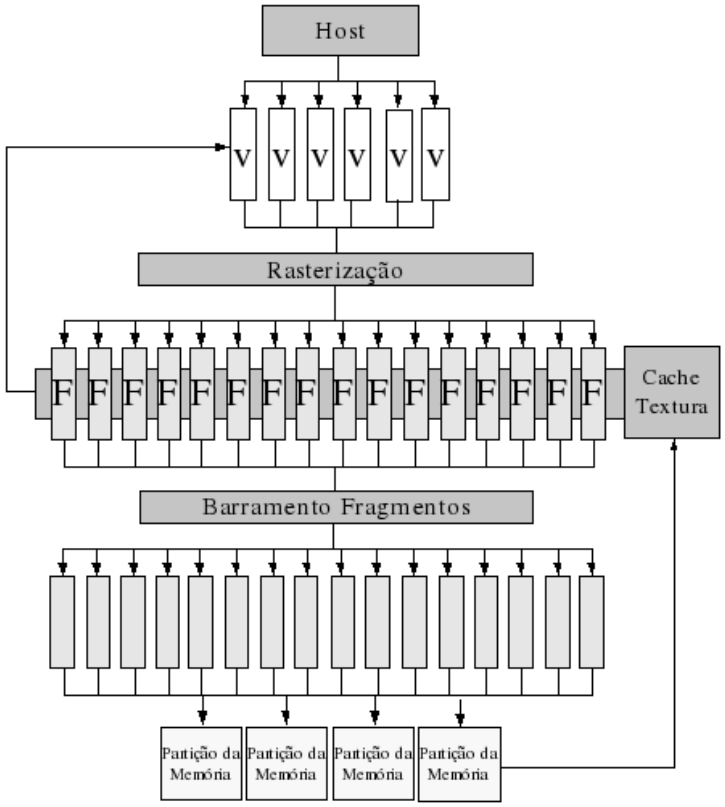

Figura 4.4: Arquitetura de uma GPU atual.

4.2.1 Desenhando Quadrados com OpenGL

Vamos começar com um exemplo de como desenhar dois quadriláteros coloridos usando OpenGL.

A Figura 4.5 mostra um trecho de código OpenGL para desenhar dois quadriláteros. O trecho de código mostra apenas a parte onde os vértices são enviados para a *pipeline*. Algumas configurações e inicializações são necessárias para que o código exibido funcione, como a criação da janela gráfica, a configuração da área desenhável da tela e a configuração do tipo de projeção a ser utilizado. A Figura 4.6 mostra o processamento dos vértices até serem renderizados.

Na Figura 4.6 na primeira parte (1) os vértices são definidos com as respectivas

```
glBegin(GL_QUADS);
    // vertices cor verde
    glColor3f(0,1,0);
    //vertices de 1 a 4
    glVertex3f(0,0,0);
    glVertex3f(6,0,0);
    glVertex3f(6,6,0);
    glVertex3f(0,6,0);
glEnd();


glBegin(GL_QUADS);
    // vertice 1 cor vermelha
    glColor3f(1,0,0);
    glVertex3f(8,4,0);
    // vertice 2 cor azul
    glColor3f(0,0,1);
    glVertex3f(12,4,0);
    // vertice 3 cor vermelha
    glColor3f(1,0,0);
    glVertex3f(12,8,0);
    // vertice 4 cor azul
    glColor3f(0,0,1);
    glVertex3f(8,8,0);
glEnd();
```

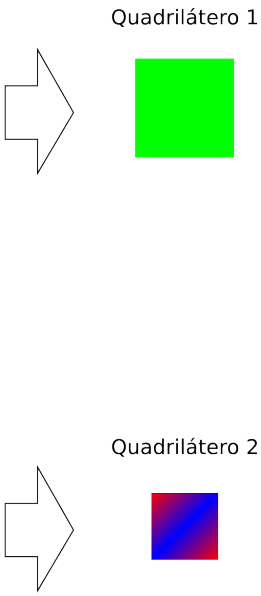

Figura 4.5: Trecho de código OpenGL para desenhar dois quadriláteros na tela. O quadrilátero 1 é todo verde, enquanto o quadrilátero 2 é interpolado com as cores vermelho e azul.

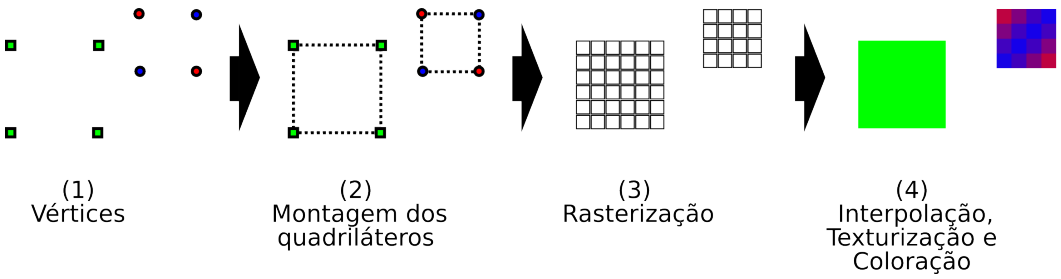

Figura 4.6: Processamento dos vértices enviados para a *pipeline*.

cores e enviados para o próximo estágio da *pipeline* mostrada na Figura 4.3. Em (2) os vértices estão no estágio do *processador de vértices* da *pipeline*, onde os vértices se tornam os quadriláteros. Em (3), *rasterização*, são gerados os fragmentos que são enviados para (4) o *processador de fragmentos* que colore os fragmentos de acordo com informações de cor e textura.

4.2.2 Somando vetores na GPU

Antes de mostrarmos a soma de vetores em GPU ilustraremos como esta operação é realizada em CPU. A Figura 4.7 mostra o loop contendo a soma dos vetores em CPU. O trecho de código está na linguagem de programação C.

```
// loop para a soma de dois vetores
for(int i=0; i < tamanho_vetor; i++)
{
        vetor_saida[i] = vetor_1[i] + vetor_2[i];
}
```

Figura 4.7: Loop para a soma de dois vetores na CPU.

A computação na CPU é feita em série, ou seja, apenas um elemento do vetor é computado por cada iteração do loop.

No entanto, a soma dos vetores na GPU é o que realmente nos interessa por desejarmos utilizar a GPU para realizar computações de propósito geral. Diferentemente do que acontece na CPU a computação da soma dos vetores na GPU é feita de forma totalmente parelela. Todos os elementos dos vetores são somados em paralelo. Na verdade, como o número de multiprocessadores da GPU é limitado, a computação pode não ser realizada completamente em paralelo, dependendo do tamanho dos vetores a serem somados. No entanto, a computação aparenta ser totalmente paralela uma vez que o hardware torna o processo transparente para o programador.

Para utilizar a GPU para a computação de propósito geral, é necessário que algumas configurações sejam realizadas de forma exata para que não haja nenhum erro no cálculo da soma. Os detalhes dessa configuração são dados no Apêndice A. Agora vamos nos ater apenas no programa do processador de fragmentos. O

programa mostrado na Figura 4.8 é chamado *shader* e é um programa para o *processador de fragmentos* escrito na linguagem GLSL, que computa a soma de dois vetores. Este programa é executado em paralelo para cada fragmento e o resultado é escrito em uma textura ao invés da tela como será explicado no Apêndice A.

```
#extension GL_ARB_texture_rectangle : enable

uniform sampler2DRect texture_y;
uniform sampler2DRect texture_x;

void main(void){
        // carrega os vetores em x e y
        float y = texture2DRect(texture_y, gl_FragCoord.xy).x;
        float x = texture2DRect(texture_x, gl_FragCoord.xy).x;
        // realiza a soma
        gl_FragColor.x =  x + y;
}
```

Figura 4.8: Programa completo para a soma de dois vetores para processador de fragmentos.

Nas próximas seções iremos explicar alguns aspectos teóricos da programação em GPU e no Apêndice A é apresentado um exemplo prático completo de como realizar a soma dos vetores em GPU.

4.2.3 Modelo de Programação

Os modelos de programação para CPUs são geralmente seriais e o seu hardware reflete este modelo de programação. No caso geral, a CPU explora muito bem o paralelismo de instruções mas processa um dado por vez, não explorando o paralelismo de dados. Recentemente, a arquitetura dos processadores teve o conjunto de instruções incrementado para que houvesse algum paralelismo de dados, como por exemplo a tecnologia SSE (Streaming SIMD Extensions) da Intel e AltiVec do PowerPC. Mas o grau de paralelismo é muito menor que o de uma GPU (LEFEBVRE *et al.*, 2005).

As CPUs, por terem como objetivo executar programas de propósito geral, não contêm hardware especializado para funções especificas. As GPUs, no entanto, podem implementar hardware de propósito especial para tarefas particulares, o que é muito mais eficiente (LEFEBVRE *et al.*, 2005).

GPU *Stream* Programming

As GPUs utilizam o modelo de programação *stream*. O modelo de programação *stream* captura a localidade computacional, que não é encontrada em modelos vetoriais, através do uso de *streams* e *kernels*. Uma *stream* é uma coleção de registros que necessitam de uma computação similar e os *kernels* são funções aplicadas a cada elemento da *stream*. Um processador *stream* executa um *kernel* em todos os elementos de uma *stream* de entrada, colocando os resultados em uma *stream* de saída (BUCK *et al.*, 2004). Processadores *stream* são capazes de executar programas inteiros em cada elemento da *stream*, diferente do que acontece com processadores vetoriais onde apenas operações simples são realizadas em cada elemento. Nas arquiteturas vetoriais os resultados intermediários são armazenados no registrador vetorial, enquanto que nos processadores *stream* os valores intermediários são armazenados localmente e, portanto, acessados em um tempo muito menor.

4.3 Analogias entre CPU e GPU

4.3.1 Mapeamento de Recursos Computacionais em GPUs

Até mesmo para programadores experientes, começar a programar em GPU pode ser complicado sem algum conhecimento de programação gráfica. Nesta seção serão

feitas algumas analogias básicas entre conceitos tradicionais das CPUs com GPUs.

Streams: Texturas da GPU = *Arrays* da CPU

As estruturas de dados fundamentais em GPUs são texturas e *arrays* (vetores) de vértices. Os processadores de fragmentos tendem a ser mais úteis para a programação de propósito geral do que os processadores de vértices. Desta maneira, onde se utilizaria um *array* de dados na CPU, pode-se utilizar uma textura na GPU.

Kernels: Programas de Fragmento na GPU = Loop interno da CPU

Os vários processadores paralelos de uma GPU realizam a computação do *kernel* em *streams* de dados. Na CPU, utilizaríamos um loop para iterar em todos os elementos de uma *stream* (armazenado em um *array*), processando-os sequencialmente. No caso da CPU, as instruções dentro do loop são o *kernel*. Na GPU, implementa-se instruções semelhantes dentro dos programas de fragmento, que são aplicados a todos os elementos da *stream*. A quantidade de paralelismo nesta computação depende do número de processadores fornecidos pela GPU.

Renderizar na Textura = Atribuição

A atribuição é simples de ser implementada em CPU devido ao seu modelo de memória unificado, no qual a memória pode ser lida ou escrita em qualquer parte do programa. Já na GPU, isto não é tão simples. Para conseguir resultados semelhantes na GPU é necessário renderizar para uma textura os resultados de um programa de fragmentos para que estes possam ser utilizados como entrada nos próximos programas. Tudo isso acontece porque na GPU uma textura é de apenas

escrita ou apenas leitura.

Rasterização da Geometria = Invocação da Computação

As seções anteriores deram uma idéia das analogias na representação dos dados. No entanto, para executar um programa, é necessário saber como invocar essa computação. Os *kernels* são programas de fragmentos, logo, é necessário desenhar uma geometria. Os processadores de vértices irão transformar essa geometria, o rasterizador determinará quais pixels no buffer (ou textura) de saída serão cobertos e gerar um fragmento para cada um.

Na computação de propósito geral em GPUs (GPGPU), a saída da computação é armazenada em uma textura 2D. Desta maneira, a invocação mais comum na programação GPGPU é um quadrilátero simples que preencha essa textura.

Coordenadas das Texturas = Domínio Computacional

Cada *kernel* (programa de fragmentos) que executa na GPU recebe como entrada um número de *streams* (texturas) e geralmente gera apenas uma *stream* de saída que é armazenada em uma textura. No entanto, GPUs mais recentes suportam múltiplas *streams* de saída fornecendo mais flexibilidade. Qualquer computação tem um domínio de entrada e uma saída. Em vários casos, o domínio de computação na GPU pode ter dimensões diferentes que as *streams* de entrada, resultando em uma *stream* de saída com uma dimensão menor, por exemplo.

Redução

Nem todas as operações são puramente paralelas. Por exemplo, em alguns casos é necessário reduzir um grande vetor de valores para um vetor menor, ou até mesmo para um único valor. Este é o caso para o cálculo do produto interno de dois vetores. Este tipo de computação é chamado de redução paralela.

Em GPUs, reduções podem ser realizadas alternando a escrita e a leitura em um par de buffers ou texturas. Em cada passo, o tamanho da saída é reduzido por uma fração. Para produzir um elemento da saída, dois ou mais elementos são lidos e o novo é computado usando o operador de redução, como a adição no caso do produto interno. Esses passos continuam até que a saída é reduzida a apenas um elemento, onde se obtém o resultado reduzido. Em geral esse processo leva O(log n) passos, onde n é o numero de elementos a reduzir. A Figura 4.9 ilustra este procedimento.

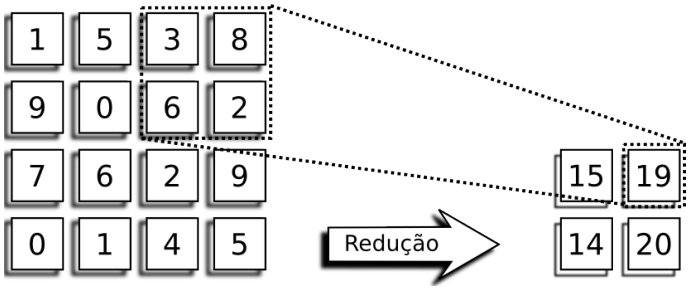

Figura 4.9: Um passo de redução de soma. Cada conjunto de 2x2 células é reduzido em apenas uma célula contendo a soma das 4 células no buffer de saída.

4.4 Linguagem GLSL

Como foi visto na Seção 4.1, o hardware gráfico é implementado em forma de uma *pipeline*. Cada estágio desta *pipeline* tem uma função e dois deles são pro-

gramáveis, a saber, o estágio do processador de vértices e o estágio do processador de fragmentos. Essa programação em GPU segue o modelo *Stream Programming* onde a saída de cada *kernel* é função apenas de sua própria entrada e, em um *kernel*, a computação em um elemento da *stream* nunca depende da computação em outro elemento. Existem várias linguagens para a descrição dos programas para os processadores da GPU, como a linguagem GLSL do padrão OpenGL, HLSL do DirectX, Cg da NVIDIA, entre outras, além de linguagens de mais alto nível que estão sendo desenvolvidas principalmente para computação de propósito geral em GPUs. A linguagem escolhida para a implementação deste trabalho foi a GLSL por ser do padrão OpenGL, o que garante uma maior portabilidade do código implementado.

A linguagem GLSL (*OpenGL Shading Language*) é utilizada para a implementação de códigos para os processadores de fragmentos e vértices. O código implementado em GLSL é chamado de *shader*. O termo OpenGL Shader é algumas vezes usado para diferenciar um shader escrito em GLSL de um shader escrito em outra linguagem *shading*. Por existirem dois processadores programáveis definidos em OpenGL, existem dois tipos de shaders: *fragment shaders* e *vertex shaders*.

A linguagem GLSL tem suas raízes na linguagem C. A linguagem possui um conjunto de tipos rico, incluindo vetores e matrizes e alguns mecanismos de C++, como sobrecarga de funções. A linguagem inclui também suporte para loops, chamadas de subrotinas, expressões condicionais, entre outros (ROST, 2005). A Figura 4.10 mostra um exemplo de um *fragment shader* utilizado na implementação da soma de um vetor multiplicado por um escalar com um outro vetor.

No código demonstrado na Figura 4.10, as varíaveis com o modificador *uniform*

```
#extension GL_ARB_texture_rectangle : enable

uniform sampler2DRect texture_y;
uniform sampler2DRect texture_x;
uniform float alpha;

void main(void){
        float y = texture2DRect(texture_y, gl_FragCoord.xy).x;
        float x = texture2DRect(texture_x, gl_FragCoord.xy).x;
        gl_FragColor.x =  x + alpha*y;
}
```

Figura 4.10: *Fragment shader* utilizado na implementação do trabalho.

são passadas para o shader através do programa principal na CPU por meio de texturas como é o caso das variáveis do tipo *sampler2DRect* ou dados simples como a variável *alpha*. É interessante observar o modelo de programação *stream* explicitado neste shader, onde este código é executado em todos os elementos da textura resultando na operação $\boldsymbol{x} + \alpha.\boldsymbol{y}$. O comando *gl_ FragCoord.xy* retorna a coordenada 2D (x, y) da textura no *kernel* para que a operação seja executada consistentemente entre as duas texturas.

Um código para a criação de um programa GPGPU completo é apresentado no Apêndice A.

Capítulo 5

Implementações

Neste capítulo serão apresentadas as implementações para a solução das equações do bidomínio em CPU e GPU.

5.1 Implementação em CPU - IMP-CPU

Os métodos para a resolução das equações do bidomínio foram descritos no Capítulo 3. Nesta seção será descrita a implementação desses métodos na CPU. A implementação dos métodos numéricos foi realizada utilizando a linguagem de programação C. Foi implementada a resolução dos sistemas lineares provenientes da discretização da EDP parabólica através do método dos gradientes conjugados precondicionado por Jacobi, e da EDP elíptica através do método dos gradientes conjugados precondicionado por multigrid. Também foi implementada a resolução das EDOs dos modelos celulares através do método de Euler Explícito (PUWAL e ROTH, 2007).

Primeiramente será apresentada a implementação da solução do sistema de EDOs,

depois a implementação do método dos gradientes conjugados precondicionado para a solução das EDPs seguido dos precondicionadores Jacobi, ILU, block-ILU e multigrid.

5.1.1 Resolução do Sistema de EDOs

O sistema de EDOs é proveniente dos modelos para as células cardíacas. Neste trabalho foi implementado o modelo celular do ventrículo humano (TEN TUSSCHER *et al.*, 2004) descrito no Capítulo 2. Este modelo compreende a resolução de um sistema de EDOs com dezessete incógnitas. Como estamos interessados em simular um tecido cardíaco discretizado, iremos resolver esse sistema de EDOs separadamente para cada ponto do tecido.

Seja a equação a seguir:

$$\frac{dV(x,t)}{dt} = \frac{-I_{total}(V,x,t)}{Cm}$$

que é uma das dezessete equações do modelo celular. O código na linguagem de programação C, para a solução de um passo de tempo da equação citada acima, é o que segue.

```
V[i][j] = dt*(-I_total_antigo[i][j] / Cm) + V_antigo[i][j];
```

onde V é a voltagem na membrana celular e I_total é a soma de todas correntes da membrana. No código os índices i, j representam a posição da célula no tecido cardíaco 2D discretizado e a variável V_antigo contém o valor da aproximação da

incógnita V na iteração anterior. Como cada incógnita do sistema influencia a computação de outras, é necessário no código, uma variável que guarde o valor da última aproximação, caso contrário, como a computação é realizada de forma serial, as novas aproximações já calculadas para algumas incógnitas seriam utilizadas no cálculo das outras em uma mesma iteração.

Podem existir regiões na simulação que não são modeladas por células cardíacas. Por exemplo, regiões passivas que descrevem o banho extracelular ou mesmo o sangue. Para estar regiões nenhum cálculo é realizado na etapa da resolução do sistema de EDOs. Desta maneira, o trecho de código para a resolução dos sistemas de EDOs para cada ponto no tecido é

```
for(int i=0; i<dimensao_y_tecido; i++){
    for(int j=0; j< dimensao_x_tecido; j++){
        if(mascara[i][j] == tecido){
            resolve_sistema_EDOs(incognitas[i][j]);
        }
    }
}
```

onde *mascara* possui o mapeamento da região que possui tecido ou não. Caso o ponto seja tecido cardíaco e não banho, o sistema de EDOs é resolvido, caso contrário, nada é feito. Os dois loops aninhados iteram por todo o tecido e banho.

5.1.2 Resolução das EDPs

Como já foi dito no Capítulo 3 a solução das equações do bidomínio envolve a solução de sistemas lineares derivados da discretização das EDPs parabólica e elíptica. Esses sistemas lineares são resolvidos utilizando o métodos dos gradientes conjugados precondicionado. Para o sistema linear resultante da resolução da

EDP parabólica o melhor precondicionador é o baseado na fatoração incompleta LU (SANTOS *et al.*, 2004). No entanto, como será mostrado no capítulo 5.2, para a resolução em GPU esse precondicionador não é eficiente pois envolve tarefas seriais. Por isso, o precondicionador mais indicado para a EDP parabólica seria o Jacobi por ser facilmente paralelizável (AMORIM e W., 2008). Desta maneira, foi realizada a implementação do precondicionador Jacobi para CPU para uma comparação direta com o método em GPU. Uma implementação utilizando a biblioteca PETSc (BALAY *et al.*, 2002) com o precondicionador ILU também é fornecida para a realização de comparações, no entanto não será detalhada aqui. Já para o sistema linear resultante da resolução da EDP elíptica o melhor precondicionador é o multigrid e sua implementação será detalhada neste capítulo.

Gradientes Conjugados Precondicionado

A implementação do método dos gradientes conjugados precondicionado segue o algoritmo dado na Seção 3.5. Esse algoritmo compreende 3 funções básicas que são: a multiplicação matriz vetor, soma de um vetor multiplicado por um escalar com um outro vetor (SAXPY) e o produto interno. Iremos apenas mostrar a implementação destas funções.

Multiplicação matriz vetor

A matriz do sistema linear resultante da discretização por elementos finitos bilineares das EDPs do bidomínio é simétrica em bandas. A matriz é composta por três bandas, cada uma com 3 diagonais, resultando em um total de 9 diagonais. Por se tratar de um sistema simétrico apenas 5 das diagonais precisam ser armazena-

das e uma indexação diferente precisa ser realizada para as diagonais superiores e inferiores. Desta forma, a matriz é armazenada por 5 *arrays* 1D que contêm essas diagonais. Como o objetivo é realizar a multiplicação matriz vetor, e a matriz está armazenada em *arrays* 1D, um loop por todos elementos do vetor sendo multiplicado é realizado. O código para a multiplicação matriz por vetor é o seguinte:

```
for(int i=0; i<L*A; i++){
  vs[i] =
      v[i-L-1]*d[4][i]     + v[i-L]*d[3][i]   + v[i-L+1]*d[2][i]   +
      v[i-1]  *d[1][i]     + v[i]  *d[0][i]   + v[i+1]  *d[1][i+1] +
      v[i+L-1]*d[2][i+L-1] + v[i+L]*d[3][i+L] + v[i+L+1]*d[4][i+L+1];
}
```

onde L e A são a largura e altura do tecido, respectivamente. E vs é o *array* que recebe o cálculo do vetor resultante da multiplicação da matriz contida nas diagonais d pelo vetor contido no *array* v. Uma observação detalhada do código acima revela um problema na indexação tanto dos vetores quanto das diagonais, indicando um possível acesso fora dos limites dos *arrays* da CPU. No entanto, isto é feito propositalmente para que não fosse necessário um teste muito custoso para identificar os elementos que pertencessem as bordas do tecido simulado e ainda assim obter um código legível e com bom desempenho. Para resolver isso, tanto o *array* de entrada quanto as diagonais são expandidas para que não haja nenhum acesso fora da área de memória do *array*. A Fig. 5.1 ilustra essa expansão para uma matriz simplificada com um total de apenas 3 diagonais ao invés de 9, e como os valores das diagonais são preenchidos com zeros para garantir um cálculo correto.

A Figura 5.1 mostra apenas duas diagonais distintas. A diagonal que aparece abaixo da diagonal principal é a mesma que aparece acima da diagonal principal. Pode se notar que o *array* que contém essa diagonal inferior e superior foi estendido

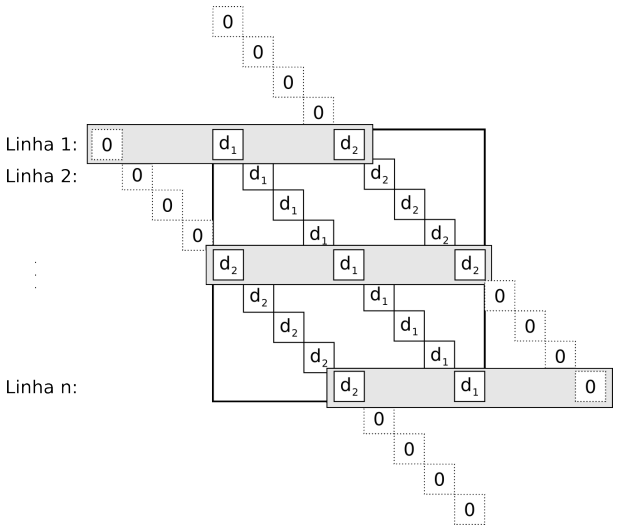

Figura 5.1: Indexação e preechimento por zeros das diagonais da matriz simplificada com apenas 3 diagonais.

com zeros para que não houvesse erro nos cálculos da multiplicação matriz por vetor. Mesmo precisando estender as diagonais com elementos nulos ainda assim esta abordagem é vantajosa pois reduz consideravelmente o número de elementos a serem armazenados. A fórmula para o cálculo do espaço requerido neste tipo de armazenamento é dado a seguir

$$E_5 = 5LA + 3L + 1 \tag{5.1}$$

onde L é a largura do tecido e A a altura. Para uma abordagem armazenando todas as 9 diagonais a fórmula seria

$$E_9 = 9LA \tag{5.2}$$

Como exemplo aplicaremos as fórmulas em um tecido de dimensão 513×513. Para

o armazenamento com 5 diagonais teríamos $E_5 = 1317385$, já utilizando o armazenamento com 9 diagonais teríamos $E_9 = 2368521$. Para este caso o armazenamento com 5 diagonais utiliza apenas 56% do necessário para o armazenamento de 9 diagonais. De fato podemos verificar que armazenar 5 diagonais é sempre mais eficiente em termos de memória como demonstrado a seguir

$$\begin{aligned} \text{Se} \quad E_9 &\leq E_5 \quad \Rightarrow \\ 9LA &\leq 5LA + 3L + 1 \\ 4LA - 3L &\leq 1 \\ L(4A - 3) &\leq 1 \\ \Rightarrow \quad L &\leq \frac{1}{4A - 3} \end{aligned}$$

como L e A representam as dimensões do tecido eles são inteiros positivos, logo o único caso que satisfaz a inequação é quando $L = 1$ e $A = 1$. Portanto para todos os outros casos $E_5 < E_9$.

SAXPY

A função SAXPY é responsável por realizar o cálculo $\alpha\boldsymbol{x} + \boldsymbol{y}$. A implementação em C para essa operação é bem simples e consiste em um loop iterando por todos os elementos do vetor da seguinte maneira:

```
for(int i=0; i<L*A; i++){
    v_s[i] = alpha*x[i] + y[i]
}
```

onde L e A são a largura e altura do tecido e sua multiplicação é o tamanho do vetor. Em cada iteração do loop um elemento do novo vetor é calculado.

Produto interno

O cálculo do produto interno de dois vetores é simples de ser implementado em CPU. É simplesmente a soma da multiplicação ponto a ponto dos dois vetores. O código em C para o produto interno dos vetores x e y é

```
double prod_int = 0.0;
for(int i=0; i<L*A; i++){
    prod_int += x[i]*y[i];
}
```

onde L e A são a largura e a altura do tecido.

Precondicionador Jacobi

Como foi descrito na Seção 3.4 um precondicionador é uma matriz que é facilmente inversível que multiplicada pelo sistema original melhora as propriedades espectrais da matriz original fazendo com que o método iterativo diminua o número de iterações necessárias para a convergência requerida. Logo, pelo algoritmo dos gradientes conjugados precondicionado mostrado na Seção 3.5 precisamos resolver $M^{-1}z = r$.

O precondicionador Jacobi consiste em escolher a matriz M como sendo a matriz diagonal formada pela diagonal principal de A. Desta maneira a implementação do precondicionador Jacobi em C fica da forma:

```
for(int i=0 i<L*A; i++){
   z[i] = r[i]/d[0][i];
}
```

Precondicionador Multigrid

A aplicação do método multigrid como precondicionador consiste em aplicar algumas iterações do método multigrid com a matriz original. O método multigrid foi apresentado na Seção 3.8 e consiste basicamente de uma sequência de três operações importantes: relaxação, restrição e interpolação (ou prolongamento). As operações de relaxação implementada em nosso trabalho foi o método de ω-Jacobi e as operações de restrição e interpolação são bilineares.

Implementação do método ω-Jacobi

O método de ω-Jacobi é bem simples de ser implementado como foi descrito na Seção 3.8. O código em C para o método ω-Jacobi é dado por:

```
for(int i=0; i<L*A; i++){
  vs[i] = (1-w)*v[i] +
          w*(v[i-L-1]*d[4][i] + v[i-L]*d[3][i] +
          v[i-L+1]*d[2][i] + v[i-1]*d[1][i] +
          v[i+1]*d[1][i+1] + v[i+L-1]*d[2][i+L-1] +
          v[i+L]*d[3][i+L] + v[i+L+1]*d[4][i+L+1])/d[0][i];
}
```

Interpolação

A operação de prolongamento ou interpolação transfere os pontos em um grid grosseiro para um grid fino. Uma interpolação bilinear é realizada para o cálculo

dos pontos no grid mais fino. A Figura 5.2 ilustra o procedimento de interpolação para um ponto do grid fino.

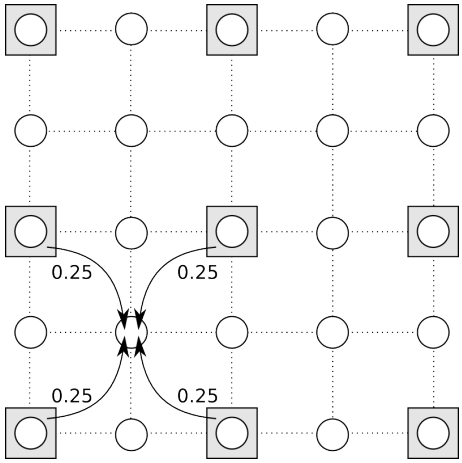

Figura 5.2: Interpolação para um ponto do grid fino em uma malha 5x5.

A interpolação é realizada baseada na distância do ponto no grid fino para o grid grosseiro. O grid fino mostrado na Figura 5.2 possui a propriedade de ter as dimensões dadas por $2n+1$ onde n, neste caso, é igual a 2. Isto garante que os pontos do grid fino estejam sempre a uma mesma distância dos pontos do grid grosseiro, sendo desta forma fácil obter uma regra para o cálculo desses pontos. No entanto, se o grid fino não possuir tal propriedade as distâncias entre os pontos podem variar e um algoritmo mais robusto foi implementado para este propósito. Apresentamos a seguir um exemplo de código para a realização da interpolação onde o grid fino tem dimensão $l \times a$ (l e a quaisquer) e o grid grosseiro tem dimensão $L \times A$ (PRESS *et al.*, 1992).

```
// calculos dos espaçamentos entre
// pontos de um mesmo grid.
hxf = 1.0/(l-1.0);
hyf = 1.0/(a-1.0);
hxg = 1.0/(L-1.0);
hyg = 1.0/(A-1.0);
```

```
// coordenadas no tecido grosseiro
int I,J;

// iteração sobre pontos do grid fino
for(int i=0; i<A; i++){
   // calculo posição y no grid fino
   yf = i*hyf;
   // identifica coord. base I no grid grosseiro
   I = (i-1)/2;
   for(int j=0; j<L; j++)
   {
       // calculo da posição x no grid fino
       xf = j*hxf;
       // identifica coord. base J no grid grosseiro
       J = (j-1)/2;
       // calculo das posições no grid grosseiro
       xg = J*hxg;
       yg = (I)*hyg;
       // cálculo da distância entre ponto
       // do grid fino e grosseiro
       t = (xf-xg)/hxg;
       u = (yf-yg)/hyg;
       // interpolação bilinear
       xf[i][j] = (1.0-t)*(1.0-u)* xg[I][J] + t*(1.0-u)* xg[I][J+1] +
                        (1.0-t)*u* xg[I+1][J] +     t*u* xg[I+1][J+1];
    }
}
```

Onde $xf[i][j]$ é um ponto no grid fino com coordenadas (i, j) e $xg[I][J]$ um ponto no grid grosseiro com coordenadas (I, J).

Restrição

A matriz da operação que transfere um grid fino para um grid grosseiro precisa ser a transposta da matriz da operação de interpolação. Essa restrição, como na interpolação, é realizada de maneira que os grids finos não precisem ser da forma $2n + 1$. Quando o grid fino tem a forma $2n + 1$ os pontos no grid fino coincidem com os pontos do grid grosseiro e também com a posição central entre os pontos

do grid grosseiro como é mostrado na Figura 5.3. A Figura 5.3 também mostra os pesos utilizados nos pontos do grid fino para a realização da restrição em um ponto no centro do grid grosseiro.

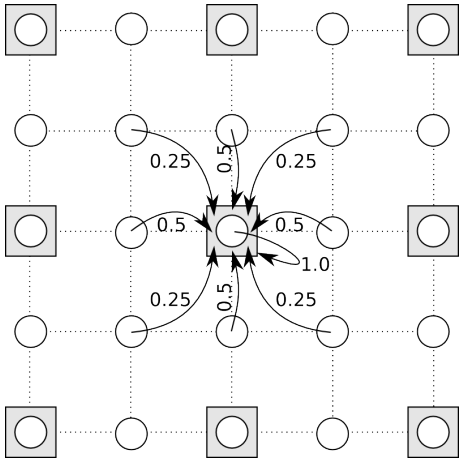

Figura 5.3: Restrição em um grid fino com dimensão igual a 5x5 $((2n+1) \times (2n+1), \quad n = 2)$. Os quadrados em cinza representam os pontos no grid grosseiro enquanto os círculos em branco os pontos no grid fino.

As dimensões do grid grosseiro são dependentes das dimensões do grid fino e são dadas por $nx_g = \lfloor (nx_f + 1)/2 \rfloor$. A Figura 5.4 mostra como a restrição é operada na CPU para o caso dos grids com tamanhos diferentes de $2n + 1$. O algoritmo de restrição itera pelos pontos do grid grosseiro acumulando a contribuição dos pontos do grid fino que estão entre o ponto do grid grosseiro sendo computado e seus vizinhos no grid grosseiro.

Tanto a restrição como a interpolação são implementadas em forma de funções onde uma iteração sobre os pontos finos ou grosseiros é realizada e, desta maneira, as matrizes das operações não precisam ser montadas. Apresentamos a seguir um exemplo de código para a realização da restrição onde o grid fino tem dimensão $l \times a$ (l e a qualquer) e o grid grosseiro tem dimensão $L \times A$.

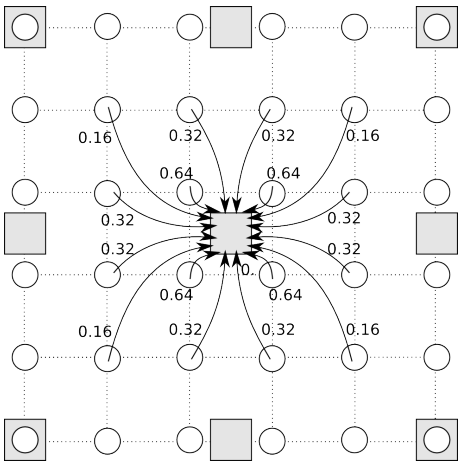

Figura 5.4: Restrição em um grid fino com dimensão 6x6. A restrição em um ponto no grid grosseiro é realizada com a contribuição de cada ponto ao seu redor no grid fino.

```
// calculos dos espacamentos entre
// pontos de um mesmo grid.
hxf = 1.0/(l-1);
hyf = 1.0/(a-1);
hxg = 1.0/(L-1);
hyg = 1.0/(A-1);

// teste de dimensão para identificação
// do numero de pontos finos entre os
// pontos do grid grosseiro
// se largura é par
if(l % 2 == 0) num_nos_finos_x = 4;
else           num_nos_finos_x = 3;
// se altura é par
if(a % 2 == 0) num_nos_finos_y = 4;
else           num_nos_finos_y = 3;

// iterando sobre pontos interiores do grid grosseiro
for(int I=1; I<A-1; I++){
  for(int J=1; J<L-1; J++){
    // iterando sobre os pontos do grid fino
    // que contribuem para o cálculo
    i_ini = 2*I-1;
    for(int i=i_ini; i< (i_ini+num_nos finos_y); i++){
      yg = I*hyg;
      yf = i*hyf;
      // calculo do peso da contribuição em y
      u = (yg>yf)?(1.0-(yg-yf)/hyg):(1.0-(yf-yg)/hyg);
```

```
      j_ini = 2*J-1;
      for(int j=j_ini; j<(j_ini+num_nos finos_y); j++){
        xg = J*hxg;
        xf = q*hxf;
        // calculo do peso da contribuição em x
        t = (xg>xf)?(1.0-(xg-xf)/hxg):(1.0-(xf-xg)/hxg);
        // computando a contribuicao do ponto fino i,j
        xg[I][J] += t*u*xf[i][j];
      }
    }
  }
// restrição das bordas do tecido ...
(...)
```

Onde $xg[I][J]$ é um ponto no grid grosseiro na posição (I, J) e $xf[i][j]$ um ponto no grid fino na posição (i, j). A implementação mostra apenas a restrição para os pontos interiores do grid grosseiro. A primeira parte do algoritmo calcula o espaçamento entre os pontos do grid fino e o espaçamento entre os pontos do grid grosseiro. Logo em seguida, é realizada a identificação da quantidade de pontos finos ao redor dos pontos do grid grosseiro em cada direção. Finalmente o valor de cada ponto do grid grosseiro é calculado somando as contribuições dos pontos do grid fino ao redor.

5.1.3 Implementação Paralela - IMP-Cluster

A implementação paralela da solução das equações do bidomínio em CPU é baseada nos mesmos métodos numéricos descritos neste capítulo. A paralelização é realizada através da divisão do tecido simulado entre diferentes processadores. O tecido é particionado apenas em sua altura, ou seja, cada processador é responsável pela computação em uma região do tecido com a mesma largura do tecido original. A Figura 5.5 ilustra este particionamento.

A implementação dos métodos númericos em paralelo é feita utilizando a biblio-

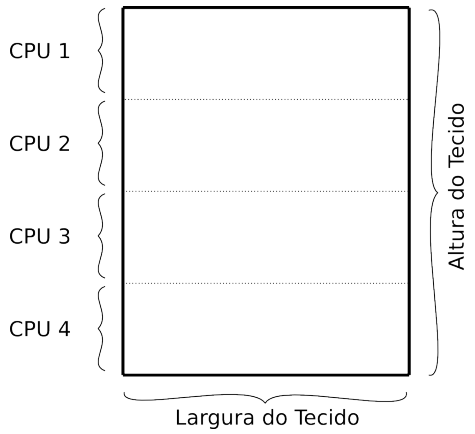

Figura 5.5: Tecido particionado entre 4 processadores.

teca para computação científica PETSc (BALAY *et al.*, 2002) para a resolução dos sistemas lineares provenientes da discretização das EDPs e MPI para toda comunicação por passagem de mensagens (MPI, 1994).

Como o domínio é particionado entre os processadores e existem dependências entre pontos em diferentes processadores, o método númerico não é totalmente paralelizado mas sim as operações que o compõem.

Como descrito no Capítulo 3 o método dos gradientes conjugados é composto por 3 operações básicas que são a multiplicação matriz vetor, produto interno e SAXPY.

Devido ao método de discretização utilizado, existe uma dependência entre pontos pertencentes a dois processadores consecutivos para a operação de multiplicação matriz vetor. As linhas que fazem interface no particionamentos precisam ser trocadas entre processadores consecutivos como mostra a Figura 5.6.

Para o cálculo da multiplicação matriz vetor, os trechos dos vetores correspondentes as linhas precisam ser trocadas entre processadores consecutivos antes de

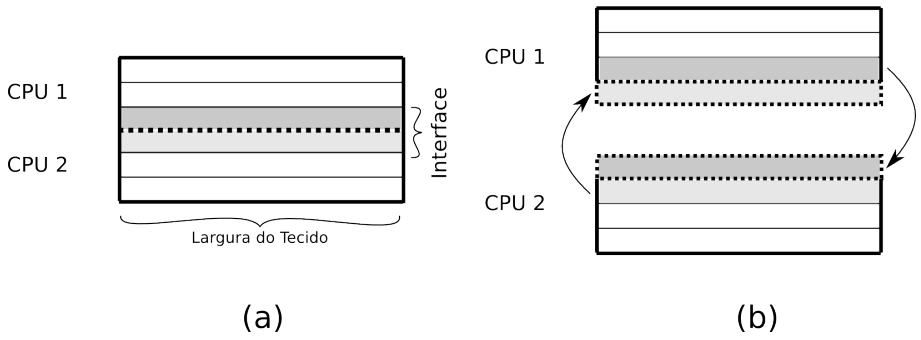

Figura 5.6: Troca de linhas entre dois processadores. Em (a) o tecido inteiro dividido entre dois processadores. Em (b) trechos do tecido armazenados em cada processador, inclusive as linhas da interface sendo trocadas.

cada processador realizar sua parte da multiplicação.

Para o cálculo do produto interno cada processador realiza o cálculo separadamente e envia o resultado do seu produto interno parcial aos outros processadores. Desta maneira, cada processador recebe os produtos internos parciais de todos os outros processadores e os soma com o seu produto interno parcial para a obtenção do produto interno total.

Os métodos multigrid também seguem o mesmo princípio e apenas as linhas nas interfaces precisam ser trocadas. A paralelização das EDOs é trivial, onde cada processador resolve o sistema de EDOs para as células que compõem sua parte de tecido.

5.2 Implementações em GPU - IMP-GPU

O objetivo principal deste trabalho é a implementação da solução das equações do bidomínio em GPU. Nesta seção serão apresentadas as implementações em GPU para o método de Euler Explícito para a resolução do sistema de EDOs e o método

dos gradientes conjugados precondicionado por Jacobi e multigrid para a solução das EDPs parabólica e elíptica.

No Capítulo 4 foram apresentados detalhes sobre a programação em GPU para propósito geral através de um exemplo de como realizar uma soma de vetores. Os conceitos utilizados neste exemplo serão utilizados nesta seção, porém algumas implementações descritas neste capítulo são mais complicadas, como o produto interno de dois vetores. Detalhes sobre como configurar o ambiente para a computação GPGPU não serão repetidos pois permanecem basicamente os mesmos. Com o propósito de facilitar a implementação dos métodos em GPU e de produzir um código mais legível, foi criada uma biblioteca para auxiliar a implementação. Desta maneira foram criadas funções que criam e configuram as texturas, inicializam a GPU, configuram a renderização fora da tela, definem a textura ou texturas que serão utilizadas para receber a computação, copiam dados da CPU para a GPU e vice-versa, entre outras funções. A biblioteca criada fornece uma maneira mais simples de se criar código GPGPU e torna o código menos suscetível a erros de implementação. A biblioteca pode ser vista no Apêndice B.

Antes de falarmos sobre a implementação dos métodos numéricos em GPU precisamos explicar uma técnica utilizada em GPGPU conhecida como *ping-pong*. Nesta técnica um par de texturas é alternado entre escrita e leitura a cada aplicação do shader. Desta maneira, os resultados obtidos na última aplicação de um shader são utilizados como entrada da próxima aplicação, e a textura utilizada como entrada anteriormente é utilizada como saída. Esta técnica será importante em algumas das funções implementadas.

5.2.1 Resolução do Sistema de EDOs

A solução do sistema de EDOs em GPU é realizada basicamente no shader do processador de fragmentos. O código apresentado para a soma de dois vetores na GPU fornece uma boa base para a implementação da solução por Euler Explícito do sistema de EDOs em GPU. Como dito no Capítulo 5.1 o modelo celular utilizado neste trabalho é para células do ventrículo humano (TEN TUSSCHER *et al.*, 2004), um sistema de EDOs com dezessete incógnitas como descrito no Capítulo 2. A solução dos sistemas de EDOs na GPU é realizada em paralelo para todos os pontos do tecido, diferente do que acontece na CPU onde o sistema de EDOs é resolvido em um ponto de cada vez. Obviamente o nível de paralelismo será determinado pelo número de processadores na GPU em que a computação será realizada. A solução da equação

$$\frac{dV(x,t)}{dt} = \frac{-I_{total}(V,x,t)}{Cm}$$

utilizando a linguagem GLSL é dada por

```
float V_antigo = textureRect2D(texture_V_antigo, gl_FragCoord.xy).x;
float I_total_antigo = textureRect2D(texture_I_total_antigo, gl_FragCoord.xy).x;
gl_FragData[0].x = dt*(-I_total_antigo / Cm) + V_antigo;
```

que é apenas um trecho do código do shader. Podemos observar que não é necessário nenhum loop para a computação de todos os elementos do tecido já que essa mesma linha de execução é realizada em paralelo para todos os elementos. Neste exemplo apenas o valor de V é calculado. No entanto, para o modelo do ventrículo humano

outras dezesseis EDOs precisam ser calculadas e terem seus valores armazenados em texturas. As GPUs atuais não permitem a escrita aleatória na memória, por isso cada processamento paralelo na GPU possui uma posição de escrita fixa que não pode ser modificada. Entretanto, é possível escrever em mais de uma textura diferente em um mesmo processamento paralelo (*Multiple Render Targets*). Ainda assim, existe um limite para estes destinos de escrita. Em nossos testes esse limite era de 12 para as GPUs mais atuais e de 8 para uma outra GPU não tão recente. Podemos perceber que esse limite não é suficiente para escrevermos os valores de todas as 17 equações em uma única passagem. Para as GPUs com 12 *render targets* a resolução do sistema de EDOs precisa ser realizada pela passagem de dois shaders diferentes. Já para a GPU com 8 *render targets* três shaders são necessários. No trecho de código mostrado anteriormente, o índice 0 em *gl_FragData* informa em qual destino de renderização esse novo valor calculado será armazenado. É importante notar também que nos códigos GLSL mostrados no Capítulo 4, os valores eram armazenados em *gl_FragColor* que é o destino padrão de renderização, pois não era necessária a utilização de múltiplos destinos de renderização.

A Figura 5.7 ilustra a interação entre CPU e GPU para a solução das EDOs. O primeiro passo consiste em tranferir para a GPU as aproximações iniciais para as variáveis do sistema de EDOs. Apenas a variável V é transferida em todas as iterações do simulador cardíaco, as outras são transferidas apenas na primeira iteração. Em seguida a CPU invoca a computação do primeiro shader que contém o código para o cálculo de 9 das dezessete EDOs. O primeiro shader começa a ser computado pelo primeiro estágio da *pipeline* da GPU. Enquanto isso a CPU invoca a computação do segundo shader contendo as 8 EDOs restantes. O segundo

shader começa a ser computado assim que o primeiro shader sai do primeiro estágio. Antes da invocação do segundo shader a CPU associa outras texturas aos destinos de renderização para que o resultado da computação do segundo shader não sobrescreva os resultados da aplicação do primeiro shader.

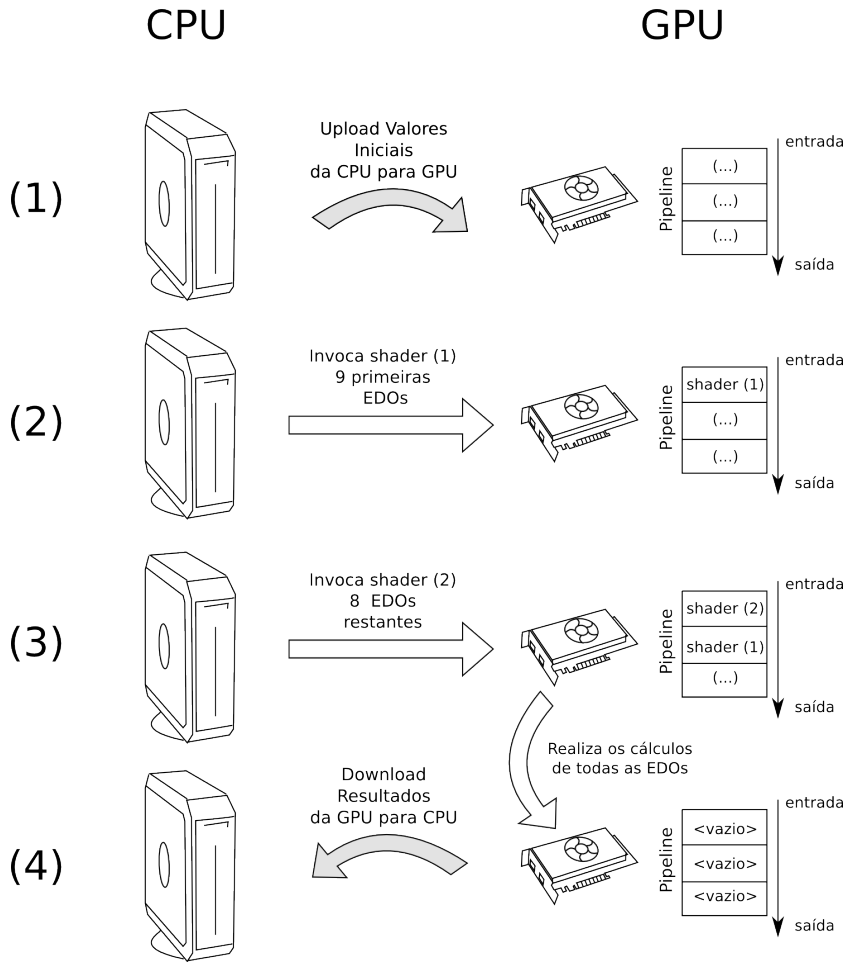

Figura 5.7: Interação CPU e GPU para o cálculo do sistema de EDOs. No passo (1) os dados contendo os valores iniciais são transferidos da CPU para a GPU, no passo (2) o shader para o cálculo das 9 primeiras EDOs é invocado. No passo (3) o shader para calcular as 8 EDOs restantes é invocado. No passo (4) os resultados são tranferidos da GPU para a CPU.

A resolução das EDOs em GPU envolve a alocação de memória para 17 texturas com dimensões $L \times A$ onde L e A são as dimensões do tecido simulado. Mais 17

texturas semelhantes precisam ser alocadas para que seja possível aplicar a técnica ping-pong. Essas texturas são mantidas na GPU até o final da simulação. Apenas a textura que guarda os valores para o potencial V é utilizada também pelos outros métodos numéricos.

5.2.2 Resolução das EDPs

A implementação dos métodos numéricos para a resolução das EDPs em GPU é um pouco mais complicada que a implementação da resolução das EDOs onde o código é simplesmente uma adaptação do código em C para o GLSL modificando apenas a leitura e escrita dos valores calculados. Alguns cuidados precisam ser tomados no cálculo da multiplicação matriz vetor, e o cálculo do produto interno de dois vetores é um pouco mais complicado por se tratar de uma operação não totalmente paralelizável. Nesta seção serão mostrados alguns detalhes da implementação das funções que compõem o método dos gradientes conjugados precondicionado e os precondicionadores Jacobi e multigrid.

O precondicionador Jacobi é utilizado no lugar do precondicionador ILU, que é o precondicionador atualmente mais apropriado para a resolução do sistema linear derivado da discretização da equação parabólica (SANTOS *et al.*, 2004). Esta escolha é feita pois o precondicionador Jacobi é facilmente paralelizável ao contrário do precondicionador ILU que envolve as operações de substituição e retro-substituição.

Gradientes Conjugados Precondicionado

O método dos gradientes conjugados precondicionado é implementado de acordo com o algoritmo da Seção 3.5. É necessário no entanto, detalhar um pouco sua implementação em GPU. As funções da multiplicação matriz vetor, produto interno e os precondicionadores Jacobi e multigrid serão detalhadas em seguida.

Para a implementação do método dos gradientes conjugados foram utilizadas as funções da biblioteca criada neste trabalho para a implementação GPGPU com OpenGL. Não mostraremos o código em si, apenas discutiremos alguns detalhes que não são óbvios para quem está apenas acostumado com o modelo de programação para CPUs.

Um dos requisitos para que uma implementação GPGPU seja eficiente é que a troca de dados entre CPU e GPU seja mínima. Por esta razão toda implementação é realizada de maneira a manter os dados o máximo possível na GPU. O loop do algoritmo é implementado em CPU que também é responsável por fazer as chamadas para a computação em GPU. A CPU realiza apenas as chamadas das funções da GPU para a manipulação das texturas de maneira a utilizar uma textura com o resultado de uma computação como entrada na próxima função. No entanto ao final de cada loop uma troca de dados entre GPU e CPU é necessária para a obtenção da norma do resíduo que é utilizada na condição de saída do loop na CPU. Porém, como o volume de dados é muito pequeno, isso não é crítico para o desempenho da implementação. A Figura 5.8 mostra de maneira simplificada como é a interação entre CPU e GPU para a realização do loop do método dos gradientes conjugados precondicionado.

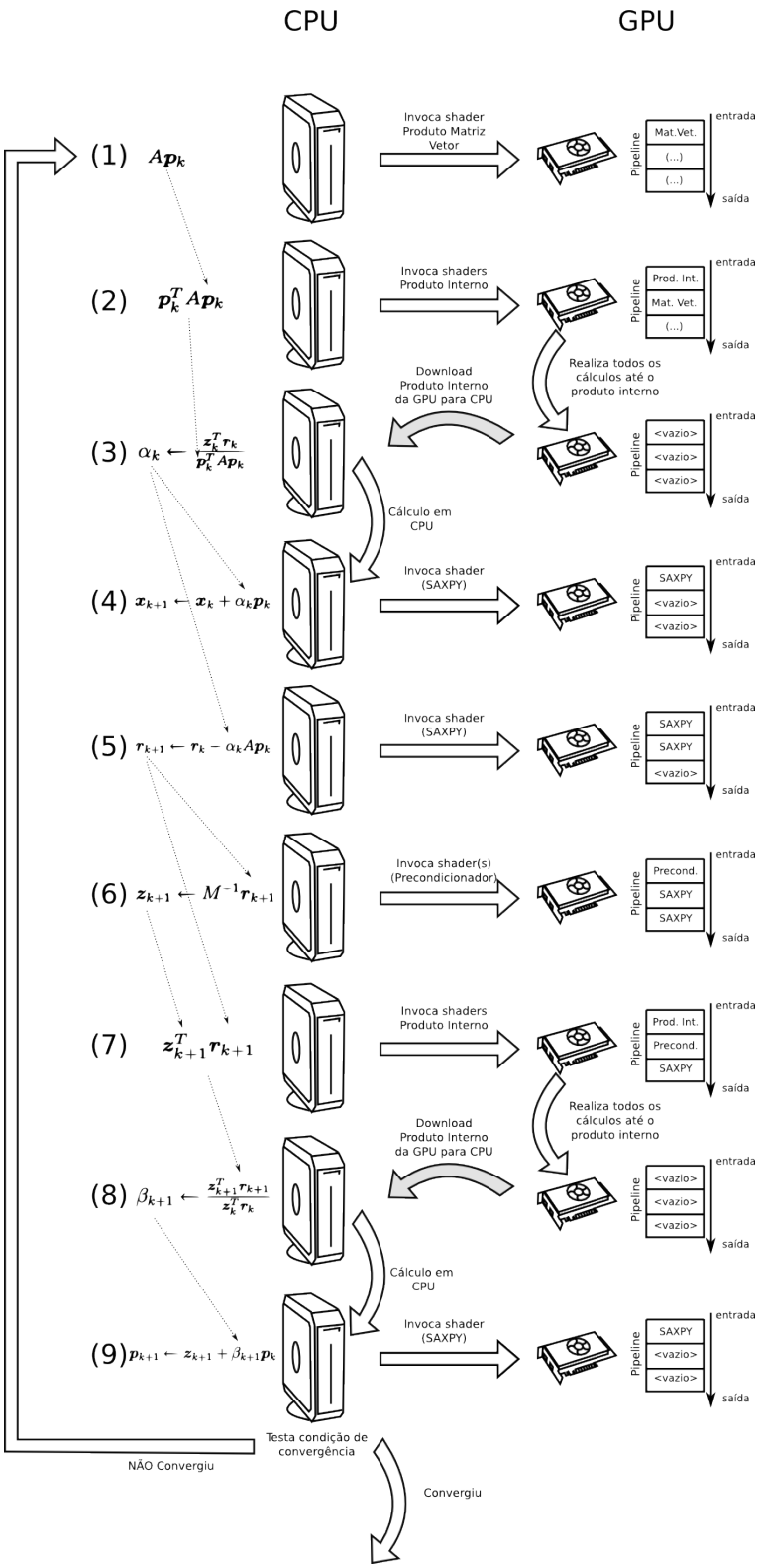

Figura 5.8: Esquema simplificado da interação entre CPU e GPU para o loop do método dos gradientes conjugados precondicionado.

Podemos notar na Figura 5.8 dois tipos de comportamento na interação entre CPU e GPU. O primeiro é que a CPU, para os passos (1), (4), (5) e (6), pode invocar a computação em GPU e seguir para a próxima instrução. Já nos passos (2), (3), (7) e (8), a CPU invoca a computação em um passo mas precisa aguardar a GPU terminar toda a computação no próximo passo, esvaziando a *pipeline* e, desta forma, prejudicando o desempenho.

Multiplicação matriz vetor

A multiplicação matriz vetor é realizada na GPU através de apenas um shader. No entanto alguns cuidados especiais precisam ser tomados. O primeiro está relacionado à utilização de apenas 5 diagonais ao invés das 9 diagonais dos sistemas em virtude da simetria das matrizes. Cada uma das 5 diagonais é armazenada em uma textura. Como a diagonal é estendida com zeros, as diagonais inferiores não necessitam de nenhum tipo de indexação diferenciada, já que o primeiro elemento da textura corresponde ao primeiro elemento da diagonal inferior. Cada elemento resultante da multiplicação matriz vetor é calculado em paralelo, ou seja, as linhas da matriz são multiplicadas paralelamente. Como podemos ver na Figura 5.9 o primeiro elemento da diagonal superior corresponde a um elemento da diagonal inferior com uma indexação diferenciada.

As diagonais, como dito anteriormente, são armazenadas em texturas. Estas texturas são alocadas na GPU antes do início da simulação e permanecem alocadas até o final. Devido ao prolongamento com zeros as texturas precisam ter suas dimensões alteradas. Porém esse aumento é realizado apenas na altura da textura para que a indexação das diagonais superiores seja possível uma vez que para os

elementos pertencentes à borda do tecido a indexação não segue a mesma regra dos pontos interiores.

A indexação das texturas é em 2D. A indexação para uma diagonal superior seria, por exemplo, $(i+1, j+1)$, onde (i, j) é a indexação para a diagonal inferior. No entanto, para $j = L$, onde L é a largura da textura, esta indexação não funcionaria pois para a diagonal superior o elemento $(i+1, L+1)$, que não existe, estaria sendo acessado. Por isso um teste precisa ser realizado para identificar os elementos das bordas e realizar uma indexação diferente, neste caso $(i+2, 1)$. Podemos ver na Figura 5.9 a indexação nas texturas para um exemplo simplificado de uma matriz com 3 diagonais. A textura 2 na figura é utilizada para a diagonal superior e inferior uma vez que a matriz representada é simétrica.

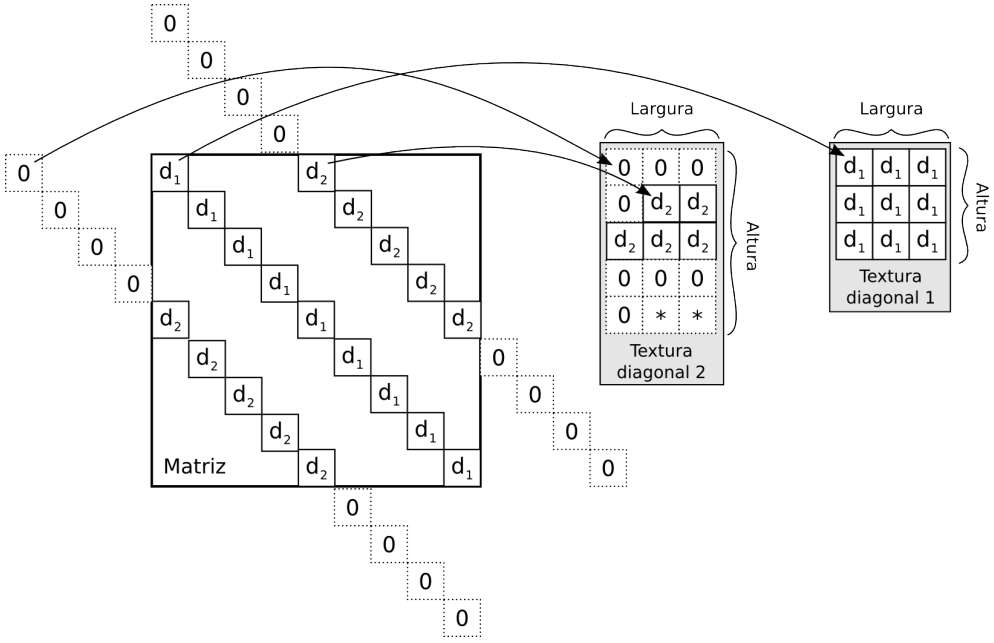

Figura 5.9: Mapeamento das diagonais em vetores para texturas.

O problema de indexação pode ser resolvido armazenando cada uma das 9 diagonais em uma textura diferente. Entretanto, o ganho em desempenho seria pequeno

se comparado ao custo do armazenamento. Por exemplo, para um tecido com dimensão 513×1025 o ganho em desempenho verificado foi de 10% apenas, contra 80% a mais de memória. Apresentamos a seguir um trecho do shader da multiplicação matriz vetor que mostra como a indexação das diagonais é realizada.

```
vec2 coord_tex_tex = glFragCoord.xy;
vec2 coord = coord_tex - (0.5, 0.5);

// valores das diagonais inferiores
diag_inf4 = texture2DRect(texture_diag4, coord_tex).x;
diag_inf3 = texture2DRect(texture_diag3, coord_tex).x;
diag_inf2 = texture2DRect(texture_diag2, coord_tex).x;
diag_inf1 = texture2DRect(texture_diag1, coord_tex).x;
diag      = texture2DRect(texture_diag0, coord_tex).x;

// se borda direita
if(coord.x == float(L)-1){
   diag_sup1 = texture2DRect(texture_diag1_0, coord_tex + vec2(-L,1)).x;
   diag_sup4.x = texture2DRect(texture_diag4_0, coord_tex + vec2(-L,2)).x;
}
else{
   diag_sup1.x = texture2DRect(texture_diag1_0, coord_tex + vec2(1,0)).x;
   diag_sup4.x = texture2DRect(texture_diag4_0, coord_tex + vec2(1,1)).x;
}
// se borda esquerda
if(coord.x == 0.0){
   diag_sup2.x = texture2DRect(texture_diag2_0, coord_tex + vec2(L,0)).x;
}
else{
   diag_sup2.x = texture2DRect(texture_diag2_0, coord_tex + vec2(-1,1)).x;
}

diag_sup3.x = texture2DRect(texture_diag3_0, coord_tex + vec2(0,1)).x;
```

onde $diag\_sup$ e dia_inf correspondem às diagonais inferiores e superiores respectivamente. A variável *coord* corresponde à coordenada do elemento sendo calculado e $coord\_tex$ as coordenadas da texturas. As coordenadas das texturas são mapeadas para coicidirem com centro do texel, por esta razão, para encontrarmos os índices de *coord*, precisamos subtrair $0,5$ de cada direção.

Devido às restrições de precisão foi necessário implementar uma emulação de precisão dupla no shader da multiplicação de matrizes. Isso foi necessário pois o erro da aproximação final da solução do sistema linear estava sendo condicionado pela precisão dos dados de entrada que continham as diagonais da matriz. Para a implementação em GPU é sempre necessário alocar os dados em CPU e depois transferí-los para uma textura na GPU. As diagonais da matriz em CPU são armazenadas em precisão dupla. Porém as texturas na GPU só possuem precisão simples. Logo para a emulação de precisão dupla em GPU as diagonais em precisão dupla em CPU são armazenados em duas texturas de precisão simples na GPU. Uma textura contém a parte de alta ordem do valor em precisão dupla e, a outra textura contém o valor de baixa ordem. Por exemplo, um elemento em precisão dupla (double) é decomposto em dois elementos de precisão simples (float) da seguinte forma:

```
double valor_original;
float alta_ordem;
float baixa_ordem;

valor_original = algum_valor_em_precisao_dupla;

alta_ordem  = (float) valor_original;
baixa_ordem = (float) (valor_original - alta_ordem);
```

Porém, a decomposição em precisão simples não possui a mesma precisão da precisão dupla. O armazenamento em precisão dupla utilizado, com 64 bits, é realizado utilizando 52 bits para a mantissa, 11 para o expoente e outro bit que representa o sinal do número, sendo que existe um bit escondido na mantissa totalizando 53 bits de mantissa. Para a precisão simples utilizada, com 32 bits, a mantissa possui (23+1) bits, onde 1 bit é escondido, 8 bits para o expoente e 1 bit para o sinal. Logo

a decomposição em dois valores de precisão simples terá uma precisão com 48 bits de mantissa $((23+1)+(23+1))$ ao invés dos 53 da precisão dupla.

Operações aritméticas utilizando a decomposição de precisão dupla em dois valores em precisão simples, requerem um tratamento cuidadoso de *overflows* e *underflows* entre as partes de baixa e alta ordem, utilizando apenas operações aritméticas em precisão simples. A adição de dois valores $c = a + b$ utilizando a emulação de precisão dupla é simples. As partes de maior ordem são adicionadas, em seguida as partes de menor ordem são adicionadas incluindo o erro da adição da maior ordem. Finalmente o *overflow* da parte de menor ordem é incluído na parte de maior ordem. O código na linguagem GLSL para a emulação de uma soma é

```
 vec2 double_add(vec2 a, vec2 b)
{
   float t1= a.x + b.x;
   float e = t1 - a.x;
   float t2 = ((b.x-e) + (a.x - (t1 -e))) + a.y + b.y;
   vec2 c;
   c.x = t1 + t2;
   c.y = t2 - (c.x - t1);

   return c;
}
```

onde *vec*2 é uma estrutura que contém dois valores em precisão simples e é nativo da linguagem GLSL. Neste código, a parte de maior e menor ordem são armazenadas em x e y de *vec*2, respectivamente. Como mostrado no código acima, são necessárias 11 operações nativas para emular a adição. Na primeira linha do código as partes contendo as altas ordens são somadas. Em seguida o erro da soma é colocado em e e o termo de baixa ordem é calculado baseado no erro encontrado e na soma dos elementos de baixa ordem. Depois o *overflow* da soma em baixa ordem é somado

à parte contendo a alta ordem da soma e a baixa ordem da soma é finalmente calculada.

A emulação da operação de multiplicação segue as mesmas idéias descritas para a emulação da operação de adição. Mais detalhes sobre emulação de precisão dupla podem ser encontrados em GÖDDEKE *et al.* (2007).

É importante notar que a emulação apresenta um impacto no desempenho, devido ao grande número de operações necessárias para realizar uma simples adição ou multiplicação. No entanto, sua aplicação é essencial para a obtenção de boas aproximações da solução dos sistemas lineares. A emulação de precisão dupla foi necessária na multiplicação matriz vetor pois fornece uma representação mais precisa da matriz do sistema linear. As outras operações envolvidas no método dos gradientes conjugados precondicionado, que não envolvem esta matriz, como produto interno e SAXPY, não necessitaram desta emulação de precisão dupla.

SAXPY

A implementação da função SAXPY é bem simples de ser realizada e segue basicamente a descrição dada no Apêndice A. A computação de cada elemento do vetor é realizada em paralelo na GPU, ou seja não é necessário nenhum loop iterando por todos elementos uma vez que todos os elementos são computados paralelamente. A única diferença desta implementação para a mostrada no Apêndice A é o programa do shader. A seguir é mostrado o código GLSL para a função SAXPY

```
#extension GL_ARB_texture_rectangle : enable

uniform sampler2DRect texture_y;
uniform sampler2DRect texture_x;
```

```
uniform float alpha;

void main(void){
   float y = texture2DRect(texture_y, gl_FragCoord.xy).x;
   float x = texture2DRect(texture_x, gl_FragCoord.xy).x;
   gl_FragColor.x =  x + alpha*y;
}
```

onde *alpha* é um escalar. No entanto este valor de *alpha* precisa ser informado no código em CPU. E isto é realizado da seguinte maneira

```
glUseProgram(programa_saxpy);
float alpha = 3.0;
GLuint alpha_l = glGetUniformLocation(programa_saxpy, "alpha");
glUniform1f(alpha_l, alpha);
```

O código CPU mostrado acima escolhe o programa contendo o shader SAXPY, atribui um valor qualquer para *alpha* na CPU, procura a variável *alpha* no shader e coloca seu endereço em $alpha\_l$ e finalmente passa este valor para a GPU.

Produto interno

O produto interno de dois vetores na GPU é a operação mais complexa de ser implementada. O primeiro passo para a realização do produto interno é a multiplicação elemento a elemento dos dois vetores. Essa multiplicação é muito semelhante a soma de dois vetores, a única diferença está no shader GLSL que, ao invés de realizar uma soma de dois vetores, realiza a multiplicação elemento a elemento. O resultado da aplicação do shader para essa multiplicação é armazenado em uma textura. O próximo passo é somar todos esses elementos. Esta soma utiliza a técnica de redução descrita na Seção 4.3. O processo reduz um vetor armazenado em

uma textura em um escalar. Para o caso de uma textura quadrada com dimensões 2^n a implementação é mais simples, bastando apenas ir reduzindo o domínio por 2. A Figura 5.10 mostra este processo para um vetor armazenado em uma textura quadrada 4×4.

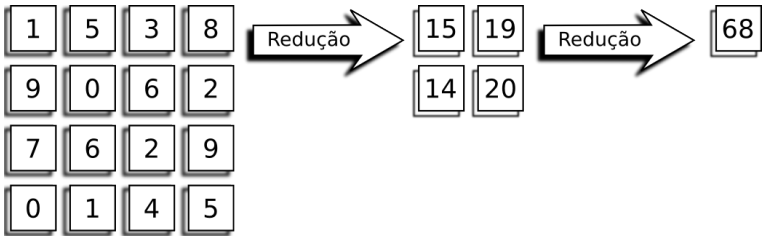

Figura 5.10: Redução de um vetor com 16 elementos armazenado em uma textura 4x4.

A redução para este caso particular com a textura quadrada com dimensão $2^n \times 2^n$ é realizada em n passos. Cada parte 2×2 da textura é somada e reduzida a 1×1 até que apenas um elemento sobre. O código em C para invocar a computação deste exemplo é dado a seguir

```
glPolygonMode(GL_FRONT,GL_FILL);
int dim = dimensao_textura;

for(int i = dim/2; i > 0; i = i/2){
   glBegin(GL_QUADS);
      glVertex2f(0, 0);
      glVertex2f(dim, 0);
      glVertex2f(dim, dim);
      glVertex2f(0, dim);
   glEnd();
}
```

O domínio de computação é reduzido a cada passagem do loop. Isso significa que um quadrilátero menor que o destino de renderização é desenhado pois a cada passo os resultados são guardados em uma porção menor da matriz. O código GLSL a

seguir é responsável por um passo da redução.

```
void main(void){
   vec2 topleft = (gl_FragCoord.xy-0.5)*2.0 + 0.5;

   float val1 = texture2DRect(texture_x, topleft).x;
   float val2 = texture2DRect(texture_x, topleft + vec2(1.0,0.0)).x;
   float val3 = texture2DRect(texture_x, topleft + vec2(0.0,1.0)).x;
   float val4 = texture2DRect(texture_x, topleft + vec2(1.0,1.0)).x;

   gl_FragColor.x = val1 + val2 + val3 + val4;
}
```

A variável $topleft$ recebe a coordenada na textura de entrada. Como queremos reduzir a entrada da computação em uma saída com a metade do tamanho em cada direção, cada elemento da textura de saída será a soma de quatro pontos adjacentes (2×2) da textura de entrada. Por esta razão, cada elemento da textura de saída com coordenada (i, j) utiliza a coordenada base na textura de entrada dada por $(2i, 2j)$ para acessar os elementos. O valor 0.5 é subtraído pois as coordenadas na textura são dadas no centro do texel, ou seja, um ponto em $(0, 0)$ tem coordenada na textura $(0.5, 0.5)$. Após a obtenção dessas coordenadas base, os 4 pontos que serão reduzidos em um ponto são obtidos com as coordenadas dos texels adjacentes. A cada passo da redução a técnica ping-pong é utilizada fazendo a última redução como entrada para o próximo passo de redução.

Entretanto esse processo pode ser mais complicado se as dimensões não forem potências de dois e ainda mais se não for quadrada. Um exemplo de um caso um pouco mais complicado seria um vetor armazenado em uma textura com dimensões 3×7. O primeiro problema neste exemplo é que não se pode ir reduzindo essas dimensões sem antes ajustar as dimensões para que sejam divisíveis por 2. Precisamos,

desta maneira, realizar um ajuste para que os valores a serem reduzidos estejam em uma porção que tenha dimensões que sejam potência de dois. Isso pode ser realizado somando os valores que estejam em alguma posição além da maior potência de 2 que seja menor que a dimensão original. Fazemos primeiro isso na largura e depois na altura e podemos proceder normalmente com a redução. Como as dimensões após esses ajustes não são necessariamente iguais, a redução é realizada normalmente até que a altura ou a largura possua tamanho igual a 1, após isso, apenas a parte que tem dimensão maior que 1 é reduzida de 2 em 2 elementos nesta última parte do cálculo. Desta forma, quatro shaders diferentes são aplicados. A Figura 5.11 mostra este processo.

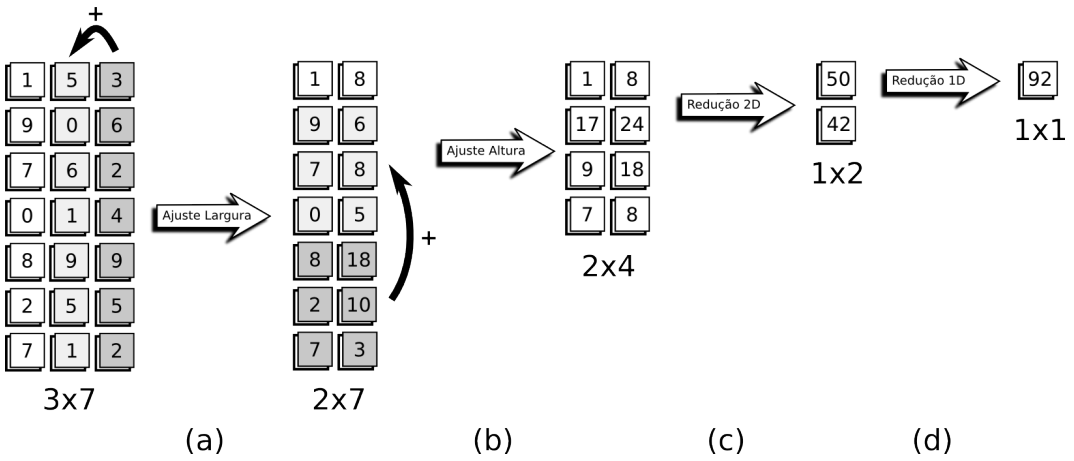

Figura 5.11: Redução para uma textura 3×7. Em (a) e (b) a largura e altura são ajustadas para a maior potência de 2 menor que as dimensões originais. Em (c) a redução normal em 2D é realizada e em (d) a redução apenas na altura é realizada (1D).

Apesar de os passos de (a) e (b) na Figura 5.11 parecerem apenas uma operação, dois shaders são implementados com o intuito de manter a simplicidade do código em GPU. O primeiro shader faz uma cópia dos valores na textura que não serão modificados (elementos em branco em (a) e (b)). O segundo realiza a soma dos

elementos que estão além da potência de 2 (elementos em cinza).

Precondicionador Jacobi

A implementação do precondicionador Jacobi em GPU é bem simples, sendo realizada por apenas um shader GLSL dado a seguir.

```
#extension GL_ARB_texture_rectangle : enable

uniform sampler2DRect texture_x;
uniform sampler2DRect texture_diag;
uniform float alpha;

void main(void){
   float r = texture2DRect(texture_x, gl_FragCoord.xy).x;
   float diag = texture2DRect(texture_diag, gl_FragCoord.xy).x;
   gl_FragColor.x = r/diag;
}
```

O precondicionador Jacobi geralmente não é muito utilizado por não reduzir significativamente o número de iterações do método. No entanto é muito fácil de ser implementado em GPU por ser totalmente paralelizável.

Precondicionador Multigrid

A implementação do método multigrid em GPU não possui nenhuma complicação como o caso da implementação do produto interno. Apenas as 3 operações básicas do multigrid serão mostradas. Todas as operações envolvidas no multigrid utilizando a relaxação por ω-Jacobi são facilmente implementadas. Esta foi a razão da escolha do método de ω-Jacobi que pode ser facilmente paralelizado. As outras operações são a interpolação e a restrição.

Implementação do método ω-Jacobi

A implementação do método de ω-Jacobi é simples e bem parecida com a implementação da multiplicação matriz vetor, necessitando dos mesmo cuidados para os elementos da matriz nas bordas laterais. Se forem necessárias mais iterações do método, a técnica de ping-pong é utilizada para que a última aproximação seja utilizada como entrada para a próxima iteração.

Interpolação

A interpolação em GPU segue o mesmo algoritmo descrito para a implementação em CPU. A única diferença é que o processo é realizado em paralelo pelos processadores da GPU. A implementação em CPU da interpolação pode ser encontrada na Seção 5.1.2.

Restrição

A implementação da restrição em GPU é semelhante à implementação da restrição em CPU. No entanto, cada ponto do grid grosseiro é calculado em paralelo. O algoritmo calcula todas as contribuições dos pontos do grid fino ao redor do ponto do grid grosseiro. Devido à possibilidade de se utilizar qualquer tamanho de tecido, o algoritmo precisa ser generalizado como mostrado para CPU, o que acarreta um número maior de operações para a identificação dos elementos que estão ao redor de cada ponto no grid grosseiro. Os casos que o algoritmo precisa tratar são os mesmos descritos para a CPU onde os elementos do grid fino estão exatamente entre os elementos do grid grosseiro, ou o caso onde esses elementos do grid fino podem

estar em qualquer posição entre os elementos do grid grosseiro. O trecho de código em GLSL apresentado abaixo é utilizado para o cálculo da restrição em GPU.

```
(...)
// calculos dos espacamentos entre
// pontos de um mesmo grid.
hxf = 1.0/(l-1);
hyf = 1.0/(a-1);
hxg = 1.0/(L-1);
hyg = 1.0/(A-1);

// teste de dimensão para identificação
// do numero de pontos finos entre os
// pontos do grid grosseiro
// se largura é par
if(l % 2 == 0) num_nos_finos_x = 4;
else           num_nos_finos_x = 3;
// se altura é par
if(a % 2 == 0) num_nos_finos_y = 4;
else           num_nos_finos_y = 3;

// coordenadas do ponto no grid grosseiro
I = glFragCoord.y - 0.5;
J = glFragCoord.x - 0.5;

G_acum = 0.0;
// se ponto interior do grid grosseiro
if(I != 0 & I != A-1 & J != 0 & J != L-1){
  // iterando sobre os pontos do grid fino
  // que contribuem para o cálculo
  i_ini = 2*I-1;
  for(int i=i_ini; i< (i_ini+num_nos finos_y); i++){
    yg = I*hyg;
    yf = i*hyf;
    // calculo do peso da contribuição em y
    u = (yg>yf)?(1.0-(yg-yf)/hyg):(1.0-(yf-yg)/hyg);
    j_ini = 2*J-1;
    for(int j=j_ini; j<(j_ini+num_nos finos_y); j++){
      xg = J*hxg;
      xf = q*hxf;
      // calculo do peso da contribuição em x
      t = (xg>xf)?(1.0-(xg-xf)/hxg):(1.0-(xf-xg)/hxg);

      xf = texture2DRect(texture_xf, vec2(i,j) + (0.5,0.5)).x;
      // computando a contribuicao do ponto fino i,j
        G_acum += t*u*xf[i][j];
      }
```

```
  }
  // escrevendo o resultado na textura
  glFragColor.x = G_acum;
}
// restrição se ponto na borda do tecido ...
(...)
```

Capítulo 6

Metodologia

Nesta seção será apresentada a metodologia utilizada para a realização dos testes comparativos entre CPU e GPU que serão mostrados no Capítulo 7.

6.1 Simulação

Por ser o coração um órgão muito complexo, uma simulação em toda a sua região seria muito custosa e inviável computacionalmente. Por esta razão, costuma-se utilizar em simulações cardíacas uma técnica conhecida como *Wedge*. Esta é uma técnica muito utilizada em laboratórios de fisiologia, consistindo em retirar um pedaço do tecido cardíaco e realizar um estudo sobre o pedaço de tecido retirado. A Figura 6.1 mostra como é realizada a preparação experimental para o uso da técnica *Wedge*. Nesta figura podemos ver os três principais tipos de células que compõem o tecido cardíaco do ventrículo humano. Ou seja as células do epicárdio, endocárdio e do tipo M.

O pedaço de tecido retirado é colocado em uma solução passiva e isotrópica,

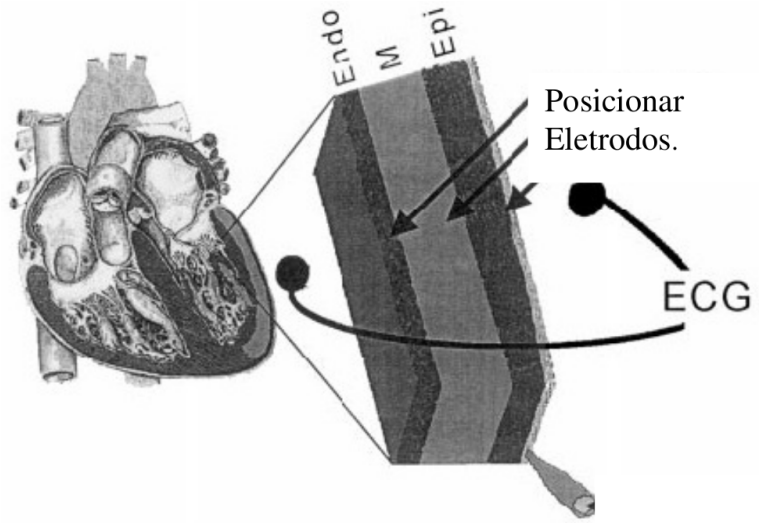

Figura 6.1: Esquema para a preparação do experimento *Wedge*. Adaptado de YAN *et al.* (1998).

chamada banho, que mantém o tecido vivo e permite que os experimentos sejam realizados.

As medidas elétricas obtidas em preparações experimentais tipo Wedge são muito semelhantes aos eletrocardiogramas obtidos na superfície do corpo humano. Em particular, o *Wedge* é capaz de simular algumas patologias cardíacas importantes (YAN *et al.*, 1998).

Nossas simulações do experimento *Wedge* são baseadas nas equações do bidomínio, que levam em consideração o domínio intracelular e extracelular do tecido cardíaco acoplado a três modelos de células de miócitos ventriculares: epicárdio, M e endocárdio. Estes modelos do miócito são baseados no modelo ventricular humano de TEN TUSSCHER *et al.* (2004) descrito na Seção 2.6. Os parâmetros utilizados são baseados naqueles descritos em SANTOS *et al.* (2004). Um tensor de condutividade ortotrópico 3D é utilizado para capturar a estrutura laminar das fibras do

coração: uma maior velocidade de condutividade ocorre ao longo das fibras, uma média velocidade de condutividade ocorre através das fibras em uma mesma folha e uma menor velocidade de condutividade ocorre através das folhas do tecido.

Os valores das condutividades do tecido cardíaco utilizados em SANTOS *et al.* (2004) foram uniformemente reescalados para coincidir com as velocidades de $70cm/s$ ápice-base e $45cm/s$ transmural. A condutividade do banho utilizada é de $20mS/cm$. A capacitância por unidade de área e a razão área para volume é de $2\mu F/cm^2$ e $2000/cm$, respectivamente. As interfaces entre tecido e banho são modeladas de acordo com o descrito em MUZIKANT e HENRIQUEZ (1998). Todas as outras bordas são eletricamente isoladas. A distribuição dos ângulos das fibras e folhas foi utilizado de maneira a reproduzir a orientação obtida do trabalho de YAN *et al.* (1998).

As simulações tiveram uma duração de 40 passos de tempo, sendo 20 passos realizados após a aplicação da corrente de estímulo em uma parte selecionada do endocárdio. A discretização espacial e temporal utilizada no método de elementos finitos foi de $40\mu m$ e $10\mu s$, respectivamente. Foram simulados 5 tecidos com tamanhos diferentes, todos com largura de 513 elementos e alturas que variam de 65, 129, 257, 513 e 1025 elementos. Estas dimensões dos tecidos foram escolhidas como sendo potências de 2 somadas de 1 devido a restrições na implementação **IMP-CLUSTER** que será detalhada adiante. A Figura 6.2 ilustra um tecido cardíaco com 513 por 257 elementos simulado.

A Figura 6.3 ilustra a propagação elétrica em um tecido cardíaco com dimensão 513×1025 para uma simulação com um total de 80 passos de tempo. Utilizando os

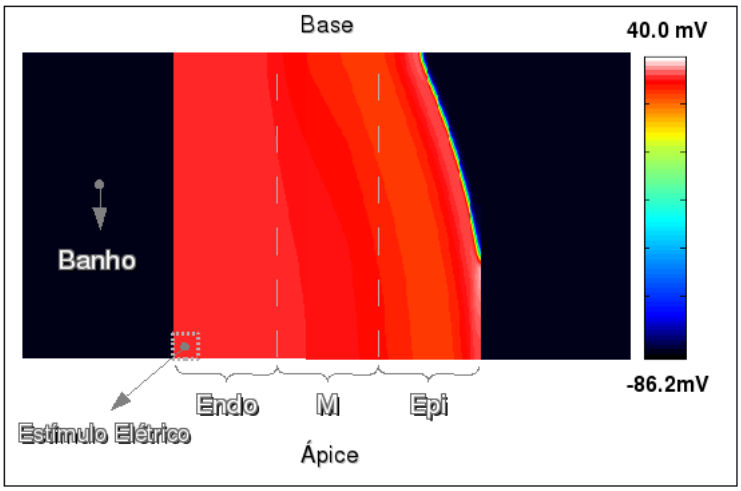

Figura 6.2: Tecido cardíaco discretizado em 513 por 257 elementos quadrados. O tecido se divide em banho, células do endocárdio, células M, e células do epicárdio. Um estímulo elétrico é aplicado na região do endocárdio indicada.

mesmo parâmetros descritos anteriormente.

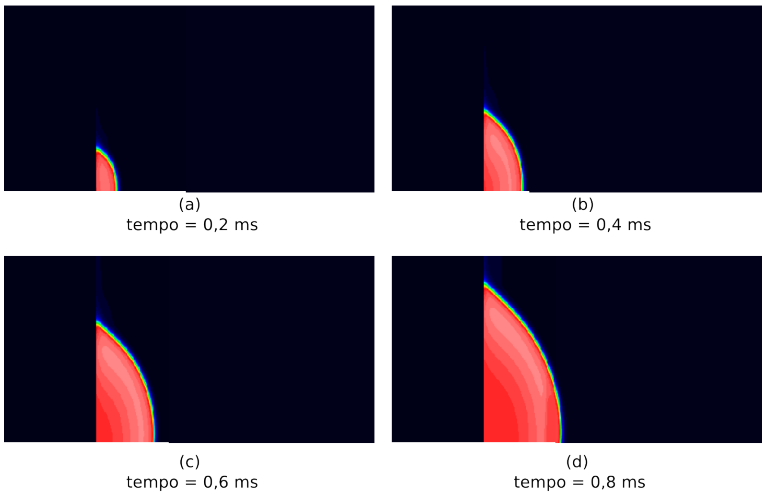

Figura 6.3: Simulação de $0,8ms$ da propagação elétrica em um tecido cardíaco com dimensão 513×1025.

Três implementações diferentes foram utilizadas para a comparação de desempe-

nho. A primeira implementação, **IMP-GPU**, é baseada em GPU e utiliza o método de Euler explícito para a solução das EDOs e o método dos gradientes conjugados precondicionado por Jacobi e multigrid para as EDPs parabólica e elíptica, respectivamente. A segunda implementação, **IMP-CPU**, é uma implementação serial em CPU dos mesmos métodos da implementação **IMP-GPU**. A terceira implementação, **IMP-CLUSTER**, é uma implementação paralela da solução das equações do bidomínio utilizando o método dos gradientes conjugados precondicionado por ILU e multigrid para as EDPs parabólica e elíptica, respectivamente. O critério de parada utilizado em todas as implementações é a tolerância absoluta de 10^{-6}.

A implementação **IMP-CLUSTER** utiliza apenas 2 grids e a resolução no grid mais grosseiro é realizada pelo método direto LU, implementado utilizando a biblioteca científica PETSc e biblioteca para passagem de mensagens MPI. Maiores detalhes sobre esta implementação podem ser encontrados em SANTOS *et al.* (2004).

Os erros no Capítulo 7 são calculados para a solução das equações parabólica e elíptica, considerando como solução exata aquela obtida pela implementação em CPU **IMP-CPU** com tolerância absoluta de 10^{-12}. A porcentagem de erro das implementações é dada pela Equação 6.1

$$ERRO_{rel(\%)} = 100 \frac{\sqrt{\sum_t \sum_{i,j} (\varphi_{i,j}^t - \phi_{i,j}^t)^2}}{\sqrt{\sum_t \sum_{i,j} (\varphi_{i,j}^t)^2}} \tag{6.1}$$

Onde i e j são os índices no espaço, t no tempo, φ o potencial tido como exato e ϕ o potencial calculado.

6.2 Hardware

Foram utilizados diversos ambientes computacionais para a realização das simulações. As configurações de hardware utilizadas estão descritas nas Tabelas 6.1, 6.2 que contém as CPUs e GPUs utilizadas nos testes respectivamente.

Tabela 6.1: Configuração das CPUs utilizadas.

Nome	Processador	Memória RAM
PD	Intel Pentium D 820 2.8GHz	2GB
AMD	Athlon 64 X2 4200+ 2.2GHz	1GB
XE	Intel Xeon 1.6GHz	4GB

Tabela 6.2: Configuração das GPUs utilizadas.

Nome	GPU	Num. Processadores	Memória RAM
7600GS	GeForce 7600GS	12	256MB
8600GT	GeForce 8600GT	32	256MB
8800GT	GeForce 8800GT	112	512MB

Os hardwares das GPUs e CPUs foram combinados de diferentes formas. No entanto, devido à algumas restrições de hardware algumas combinações não foram possíveis. Foram realizados testes combinando a CPU AMD com as GPUs 7600GS e 8800GT; a CPU PD foi combinada com todas as GPUs descritas; já a CPU XE não foi combinada com nenhuma GPU sendo utilizada apenas para a implementação em CPU.

Para a implementação paralela **IMP-Cluster** foram utilizadas duas CPUs XE conectadas por uma rede Ethernet de 1Gb/s. Cada CPU XE possui 4 processadores.

As paralelizações foram realizadas utilizando 4 e 8 processadores e são referenciadas no texto como XE4 e 2-XE4, respectivamente.

Todas as simulações foram realizadas três vezes em cada combinação de hardware descrita anteriormente. Foi realizada uma média dos tempos medidos para cada implementação. Este resultados serão apresentados no Capítulo 7. O desvio padrão observado foi desprezível visto que as execuções foram realizadas em modo exclusivo.

Capítulo 7

Resultados

Neste capítulo apresentaremos os resultados do desempenho e erros obtidos com as implementações em CPU e GPU.

7.1 Erros Numéricos

A implementação em GPU, **IMP-GPU** utiliza a emulação de precisão dupla na multiplicação matriz vetor para reduzir o erro da solução. A Tabela 7.1 mostra os erros encontrados. Através da Tabela 7.1 é possível perceber que para o potencial extracelular (φ_e) o erro é bem maior chegando a 15% para a implementação em precisão simples no tecido com dimensão 513×1025. Estes resultados justificam a utilização da implementação com emulação de precisão dupla que será utilizada no restante deste trabalho sendo referenciada no texto por **IMP-GPU**. Os maiores erros encontrados na implementação em GPU com precisão dupla são de aproximadamente $0,5\%$. Esta ordem de precisão mínima, na área de biologia é considerada pequena e as soluções exatas e aproximadas podem ser consideradas equivalentes.

Tabela 7.1: Comparação entre erros nas implementações em CPU, GPU com precisão simples e GPU precisão dupla, para φ e φ_e.

Dimensão \	CPU		GPU Precisão Simples		GPU Precisão Dupla	
Imp.	Erro φ (%)	Erro φ_e (%)	Erro φ (%)	Erro φ_e (%)	Erro φ (%)	Erro φ_e (%)
513x65	0,000001	0,000005	0,007673	1,221275	0,000267	0,087711
513x129	0,000001	0,000007	0,006574	1,717772	0,000233	0,122430
513x257	0,000001	0,000009	0,009063	3,226198	0,000340	0,220218
513x513	0,000001	0,000009	0,017424	6,707596	0,000691	0,236950
513x1025	0,000001	0,000009	0,040513	15,844730	0,001317	0,362497

A comparação entre as implementações com e sem a emulação de precisão dupla revelou que o desempenho entre as duas implementações é semelhante. Isto ocorreu devido a um aumento em torno de 4% do número de iterações necessárias para a solução da equação elíptica na implementação em precisão simples. Este aumento no número de iterações foi causado pelos erros numéricos da multiplicação matriz vetor na implementação em precisão simples.

7.2 Parâmetros do Precondicionador Multigrid

O precondicionador multigrid para as implementações **IMP-GPU** e **IMP-CPU** utilizou o número máximo de grids possível para cada dimensão de problema. Por exemplo, um tecido com dimensão 513×65 possui 7 grids, um tecido com dimensão 513×129 possui 8 grids e um tecido com dimensão 513×257 possui 9 grids. Em cada grid é aplicado um número diferente de iterações do método de relaxação. Este número é dado por $n(g+1)$ onde n é um inteiro constante e g é o número do grid, sendo $g = 0$ para o grid mais fino. A constante n foi escolhida empiricamente como sendo igual a 1. O método de relaxação utilizado é o método de ω-Jacobi com ω

escolhido igual a 0.8. Para o grid mais grosseiro utilizamos o método dos gradientes conjugados com tolerância absoluta de 10^{-15}. Os mesmos parâmetros foram utilizados em todas as dimensões de tecidos. Estes parâmetros foram encontrados empiricamente como pode ser visto nas Figuras 7.1 e 7.2. Na Figura 7.2 são realizadas combinações entre a constante n, do número de iterações (IT) do método ω-Jacobi e a tolerância absoluta do método dos gradiente conjugados no grid mais grosseiro (TOL).

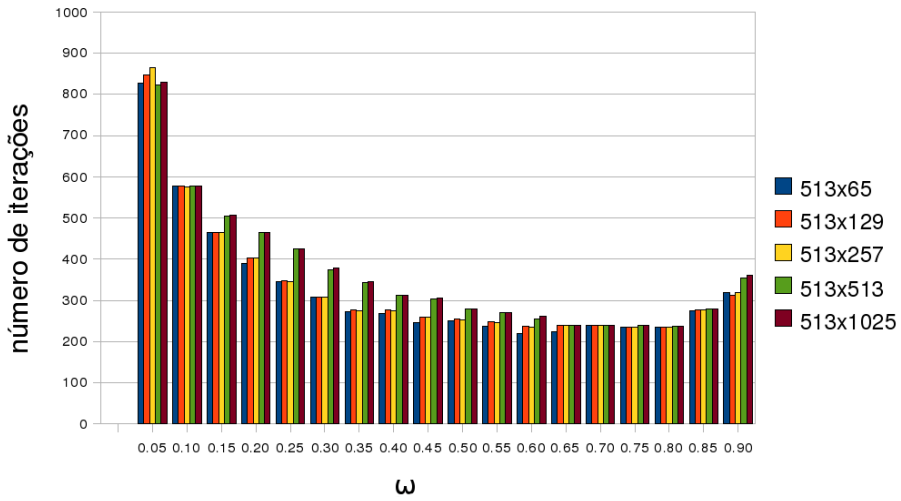

Figura 7.1: Escolha do ω mais apropriado para o método de ω-Jacobi. Escolha de $\omega = 0.8$.

7.3 IMP-GPU versus IMP-CPU

Nesta seção comparamos as implementações em CPU (**IMP-CPU**) e em GPU (**IMP-GPU**). As CPUs e GPUs utilizadas foram as descritas no Capítulo 6.

A Tabela 7.2 ilustra os desempenhos obtidos para o tecido de maior dimensão, 513 por 1025 elementos, para a resolução da Equação Parabólica. O número to-

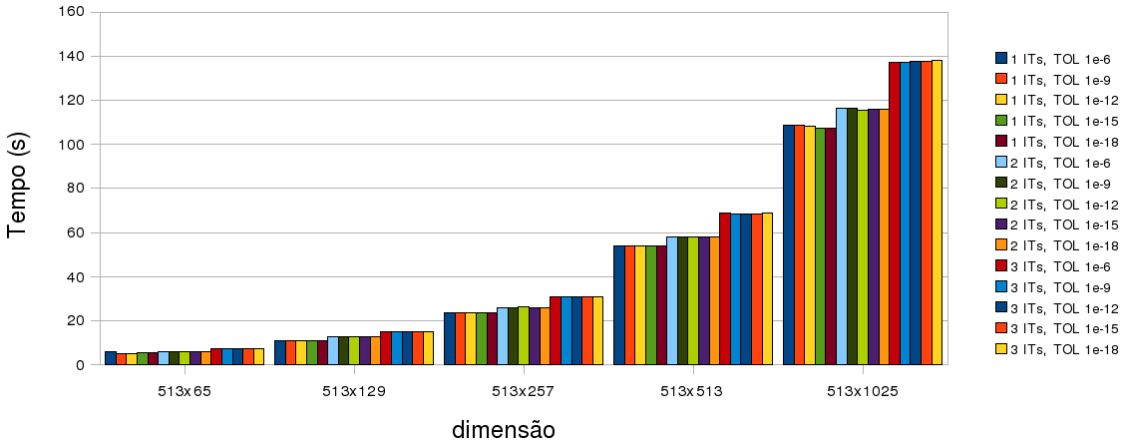

Figura 7.2: Escolha da combinação do número n de passos $n(g+1)$ do método de relaxação (ITs) com a tolerância absoluta utilizada no método dos gradientes conjugados no grid mais grosseiro (TOL).

tal de iterações realizado pelo método dos gradientes conjugados precondicionado (GCP) para a equação parabólica durante toda a simulação foi de 907 para ambas as implementações. O desempenho da implementação **IMP-GPU** do método GCP foi cerca de 5 vezes melhor do que o da **IMP-CPU** usando a GPU mais moderna 8800GT no processador AMD. No processador PD os ganhos de desempenho das GPUs 8800GT, 8600GT e 7600GS foram de $4,6$, $2,8$ e $1,7$, respectivamente.

Tabela 7.2: Solução da Equação Parabólica por PGC-Jacobi nas diferentes configurações de hardware.

Nome\Tempo	CPU (s)	GPU 7600GS (s)	GPU 8600GT (s)	GPU 8800GT (s)
PD	54,63	32,12	19,6	11,87
AMD	46,52	33,32	-	9,32
XE	38,13	-	-	-

As Figuras 7.3 e 7.4 mostram os tempos e o speedup das duas implementações

do método GCP precondicionado por Jacobi para a solução da Equação Parabólica utilizando a CPU AMD e a GPU 8800GT para as diferentes dimensões de tecidos simulados. Podemos observar que quanto maior o tecido maior a diferença entre os desempenhos das duas implementações.

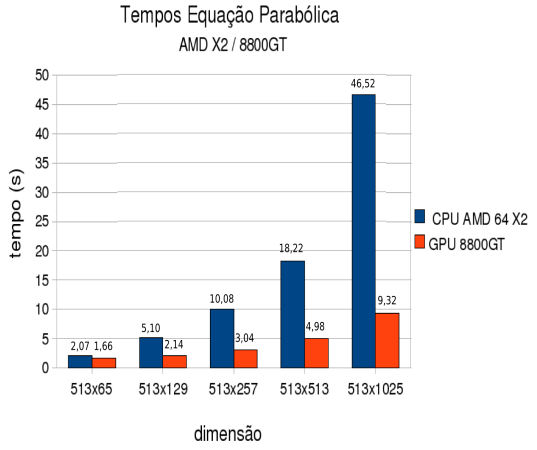

Figura 7.3: Comparação de desempenho da implementação da equação parabólica entre CPU e GPU.

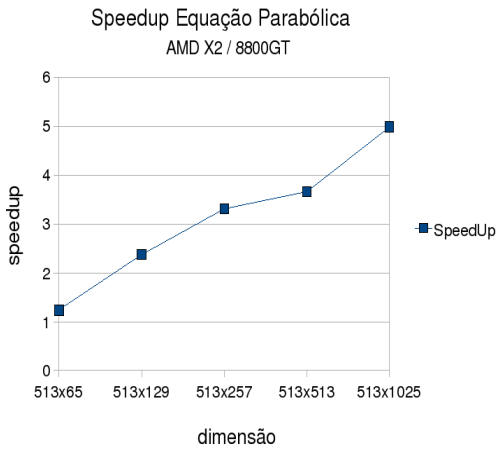

Figura 7.4: Speedup da implementação da equação parabólica em GPU sobre a CPU.

A Tabela 7.3 ilustra os desempenhos obtidos para o tecido de maior dimensão para a resolução da Equação Elíptica. O número de iterações do método CGP foi de 349 para a implementação em CPU e de 385 para a implementação em GPU. O desempenho da implementação **IMP-GPU** do GCP precondicionado por Multigrid foi cerca de $5,7$ vezes melhor do que o da **IMP-CPU** usando a GPU mais moderna 8800GT no processador AMD. No processador PD os ganhos de desempenho das GPUs 8800GT, 8600GT e 7600GS foram de $4,6$, $2,4$ e $1,3$, respectivamente.

As Figuras 7.5 e 7.6 mostram os tempos e o speedup das duas implementações do método GCP precondicionado por multigrid para a solução da Equação Elíptica utilizando a CPU AMD e a GPU 8800GT para os diferentes tecidos simulados.

Tabela 7.3: Solução da Equação Elíptica por PGC-Multigrid nas diferentes configurações de hardware.

Nome\Tempo	CPU (s)	GPU 7600GS (s)	GPU 8600GT (s)	GPU 8800GT (s)
PD	132,87	99,13	55,17	29,19
AMD	117,74	87,22	-	20,69
XE	115,62	-	-	-

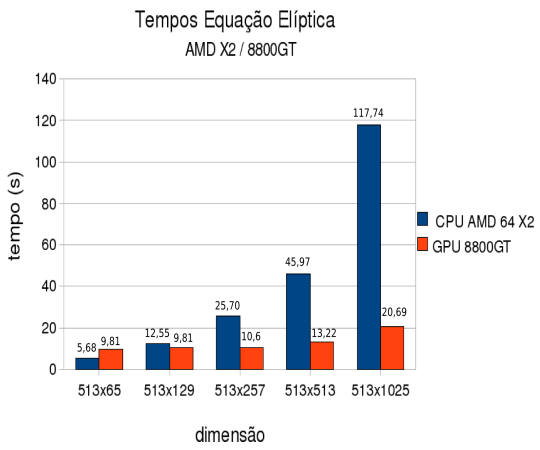

Figura 7.5: Comparação de desempenho da implementação da equação elíptica entre CPU e GPU.

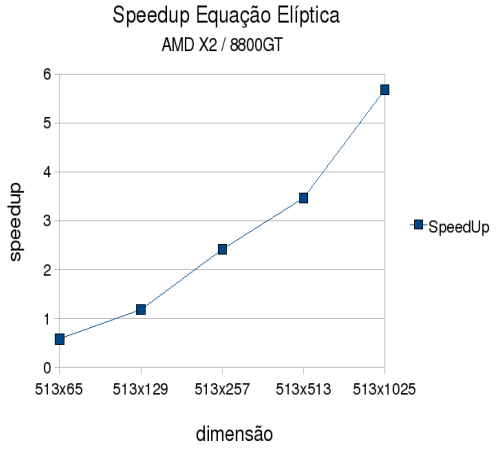

Figura 7.6: Speedup da implementação da equação elíptica em GPU sobre a CPU.

A Tabela 7.4 mostra o desempenho as implementações em CPU (**IMP-CPU**) e em GPU (**IMP-GPU**) para a solução dos sistemas de EDOs. O desempenho da implementação **IMP-GPU** da solução das EDOs foi cerca de $84,1$ vezes melhor do que o da **IMP-CPU** usando a GPU mais moderna 8800GT no processador AMD. No processador PD os ganhos de desempenho das GPUs 8800GT, 8600GT e 7600GS foram de $86,3$, $66,9$ e $30,7$, respectivamente.

As Figuras 7.7 e 7.8 mostram os tempos e o speedup das duas implementações

Tabela 7.4: EDOs nas diferentes configurações de hardware.

Nome\Tempo	CPU (s)	GPU 7600GS (s)	GPU 8600GT (s)	GPU 8800GT (s)
PD	107,05	3,49	1,6	1,24
AMD	63,95	3,21	-	0,76
XE	71,28	-	-	-

do método EDO-Euler usando a CPU AMD e a GPU 8800GT para os diferentes tecidos simulados. Novamente, quanto maior o tecido maior a diferença entre os desempenhos das duas implementações.

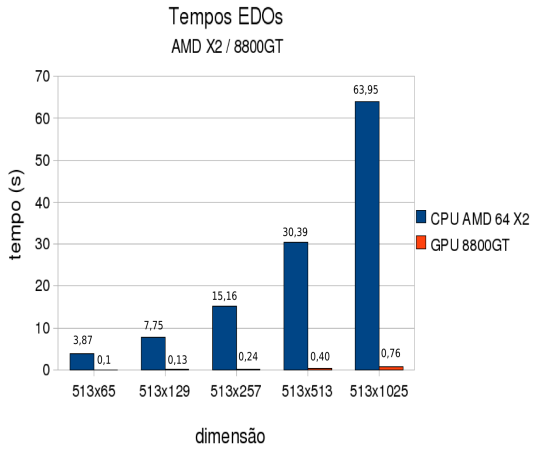

Figura 7.7: Comparação de desempenho da solução das EDOs entre CPU e GPU.

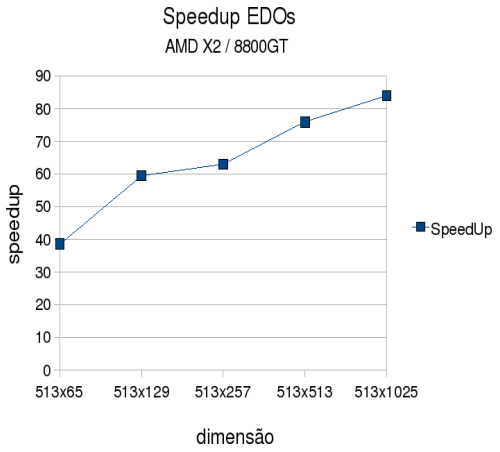

Figura 7.8: Speedup da implementação da solução das EDOs em GPU sobre a CPU.

A Tabela 7.5 mostra o desempenho das implementações em CPU (**IMP-CPU**) e em GPU (**IMP-GPU**) para a solução total das equações do bidomínio. Vale lembrar que estes tempos não são a soma dos tempos anteriores uma vez que existem alguns processos intermediários entre a solução da EDP parabólica, EDP eliptica e EDOs. Para a resolução das equações do bidomínio, o desempenho da implementação **IMP-GPU** no par AMD-8800GT foi cerca de 7 vezes melhor do que o da **IMP-CPU**

no processador mais rápido XE. No processador PD os ganhos de desempenho das GPUs 8800GT, 8600GT e 7600GS foram de 7, 3, 9 e 2, 2, respectivamente.

Tabela 7.5: Tempo Total nas diferentes configurações de hardware.

Nome\Tempo	CPU (s)	GPU 7600GS (s)	GPU 8600GT (s)	GPU 8800GT (s)
PD	299,91	135,51	77,66	43,04
AMD	228,63	124.73	-	32,30
XE	226,30	-	-	-

As Figuras 7.9 e 7.10 mostram os tempos e speedup das duas implementações **IMP-CPU** e **IMP-GPU** usando a CPU AMD e a GPU 8800GT para os diferentes tecidos simulados.

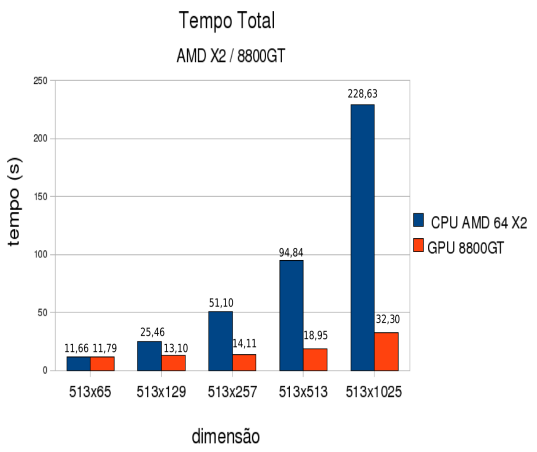

Figura 7.9: Comparação de desempenho da solução das equações do bidomínio entre CPU e GPU.

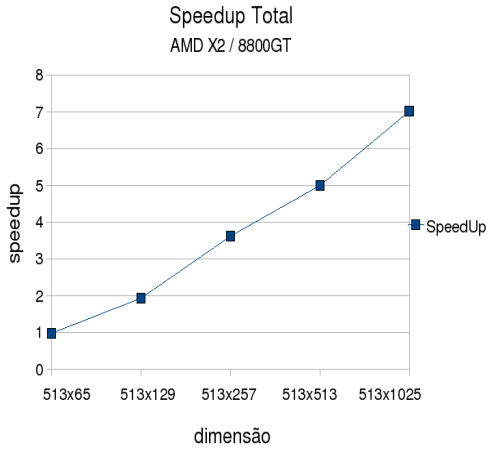

Figura 7.10: Speedup da implementação da solução das equações do bidomínio em GPU sobre a CPU.

7.3.1 Depuração de desempenho

Nas comparações anteriores, os tempos de execução para a solução das equações do bídominio levaram em consideração os tempos para a realização de algumas

operações intermediárias além dos métodos numéricos para a solução das equações. No entanto, agora iremos comparar apenas os métodos numéricos para a solução das equações. Os métodos numéricos utilizados para a resolução das equações do bidomínio são constituídos de algumas operações básicas. Por esta razão, compararemos o desempenho de cada uma destas operações separadamente, no intuito de encontrar as operações em GPU com desempenho abaixo das expectativas. Através desta análise será possível em trabalhos futuros aprimorar os algoritmos para melhorar ainda mais seu desempenho. Para a realização destes testes foram contados os tempos totais e quantas vezes foi executada cada uma das operações em uma simulação completa como as simulações anteriores. Os tempos para a execução de cada operação em CPU foram comparados com os tempos em GPU para um tecido com dimensão 513×1025. Para esta comparação de desempenho foi utilizada a CPU AMD com a GPU 8800GT. A Tabela 7.6 mostra o desempenho das funções que implementam os métodos numéricos, GCP-Jacobi, GCP-Multigrid e EDOs. O tempo de computação das funções intermediárias para a solução dos problemas elíptico e parabólico, assim como os tempos de inicialização estão representados no campo *Restante*. Isto é, GCP-Jacobi possui um tempo menor do que o tempo da EDP parabólica, como apresentado nas seções anteriores, pois não incluem outras operações como a montagem do lado direito do sistema linear. A mesma diferença existe entre EDP elíptica e GCP-Multigrid. O campo (%) corresponde a quantos por cento do tempo de execução total a operação ocupa.

Na Tabela 7.6, fica claro que o *speedup* obtido para solução das EDOs foi muito superior aos demais. Já o *speedup* obtido para o método GCP precondicionado por Multigrid foi o menor entre os métodos numéricos, diferente do que foi visto

Tabela 7.6: Comparação de desempenho entre os métodos numéricos em CPU e GPU em um tecido 513×1025.

Operação	CPU		GPU		Speedup
	Tempo (s)	(%)	Tempo (s)	(%)	
GCP-Jacobi	42,01	18,31	5,60	16,36	7,50
GCP-Multigrid	116,50	50,77	20,80	60,80	5,60
EDOs	63,95	27,87	0,76	2,22	84,14
Restante	6,99	3,05	7,05	20,62	-
Total	229,45	100,00	34,21	100,00	6,71

para a resolução da equação elíptica. Isto acontece pois o tempo para o cálculo do lado direito das equações parabólica e elíptica não é considerado. Esse cálculo é realizado na CPU e o cálculo do lado direito da equação parabólica é mais custoso que o da equação elíptica, explicando a melhora no desempenho da parte parabólica do problema.

Para uma melhor compreensão do desempenho dos métodos GCP, iremos analisar separadamente o desempenho das operações que compõem os métodos GCP-Jacobi e GCP-Multigrid. A Tabela 7.7 mostra o desempenho das operações que compõem o método PCG-Jacobi responsável pela solução da EDP parabólica. Pode-se perceber pela Tabela 7.7 que o desempenho da implementação em GPU é fortemente limitado pelo desempenho da operação de multiplicação matriz vetor e o pelo cálculo do produto interno. O desempenho do produto interno em GPU é limitado devido a dois fatores. O primeiro é o fato do cálculo da operação não poder ser trivialmente paralelizável, sendo realizada em reduções sucessivas. O outro fator está relacionado com o número de acessos à memória que é bem maior nesta implementação por reduções sucessivas e também a problemas na utilização eficiente da cache de textura, que é otimizada para acessos no padrão dado pela operação SAXPY. Algumas mo-

dificações podem ser aplicadas para melhorar o desempenho nesta operação e serão discutidas no Capítulo 8.

Tabela 7.7: Desempenho das operações que compõem o método GCP-Jacobi.

Operação	CPU			GPU			Speedup/
	Tempo (s)	(%)	Num. Cham.	Tempo (s)	(%)	Num. Cham	operação
Prec. Jacobi	4,49	10,69	947	0,22	3,93	947	20,41
SAXPY	9,62	22,90	2761	0,61	10,85	2761	15,84
M. Matriz Vetor	22,45	53,44	947	2,58	46,07	947	8,70
Produto Interno	5,31	12,64	1993	1,57	28,04	1993	3,38
Restante	0,14	0,33	-	0,62	11,12	-	-
Total	42,01	-	-	5,60	-	-	-

A Tabela 7.8 mostra o desempenho das operações que compõem o método PCG-Multigrid responsável pela solução da EDP elíptica.

Tabela 7.8: Desempenho das operações que compõem o método GCP-Multigrid.

Operação	CPU			GPU			Speedup/
	Tempo (s)	(%)	Num. Cham.	Tempo (s)	(%)	Num. Cham	operação
SAXPY	7,64	6,56	8089	1,21	5,83	8845	6,89
M. Matriz Vetor	20,80	17,86	3890	3,07	14,76	4250	7,40
Produto Interno	2,22	1,91	807	0,71	3,41	895	3,47
w-Jacobi	70,10	60,17	7391	9,80	47,12	8075	7,82
Interpolação	3,70	3,18	3501	2,20	10,58	3825	1,84
Restrição	10,36	8,89	3501	2,62	12,60	3825	4,32
Restante	1,68	1,44	-	1,19	3,47	-	-
Total	116,50	-	-	20,80	-	-	-

Podemos observar na Tabela 7.8 que as operações de restrição, interpolação e produto interno possuem o pior desempenho em GPU, e são aquelas em que o padrão de acesso à memória não é igual ao de uma operação do tipo SAXPY. Outro ponto que deve ser ressaltado é o desempenho inferior das operações SAXPY e produto matriz vetor do GCP-Multigrid em relação ao GCP-Jacobi. Isto se deve ao fato de

que o método multigrid realiza estas operações em grids menores onde o desempenho destas operações em GPU não é tão bom quanto em grids maiores.

7.4 IMP-Cluster versus IMP-GPU

As seções anteriores mostraram uma comparação de desempenho entre GPU e CPU utilizando o mesmo algoritmo nas duas plataformas. No entanto, sabe-se que para a resolução da Equação Parabólica e Elíptica outros precondicionadores produzem melhores resultados em desempenho. O precondicionador ILU(0) para a simulação sequencial e Block-ILU para a simulação paralela para a equação parabólica e multigrid com dois níveis e LU no grid mais grosseiro para a equação elíptica são considerados, atualmente, os melhores precondicionadores para o bidomínio (SANTOS *et al.*, 2004). Para a implementação desses métodos foi utilizada a biblioteca PETSc versão 2.3.2 com MPI (BALAY *et al.*, 2002). Nesta seção estendemos o conceito de *speedup* e comparamos duas implementações distintas, pois são computadas soluções numericamente equivalentes, isto é, com erro inferior a $0,05\%$, como apresentado na Seção 7.1. A Tabela 7.9 mostra o número de iterações necessárias para a resolução das EDPs parabólica e elíptica nas implementações descritas.

Como o precondicionador utilizado na implementação em GPU é diferente do utilizado em CPU, o número de iterações necessárias para a resolução das equações em GPU é maior que o necessário em CPU, o que é mostrado na Tabela 7.9.

Tabela 7.9: Número de iterações para as diferentes implementações em um tecido 513×1025.

Nome	Iterações Parabólico (s)		Iterações Elíptico	
	CPU	GPU 8800GT	CPU	GPU 8800GT
PD	183	907	280	385
AMD	183	907	280	385
XE1	183	-	280	-
XE4	280	-	280	-
2-XE4	300	-	280	-

A Tabela 7.10 mostra a comparação do desempenho entre as implementações destes precondicionadores em CPU com o Jacobi utilizado nas GPUs. É importante lembrar que os precondicionadores descritos nessa seção (ILU(0) e Block-ILU) não seriam uma boa escolha para implementação em GPU por não serem totalmente paralelizáveis. Na tabela, XE4 e 2-XE4 apresentam os resultados de **IMP-Cluster** em plataformas paralelas de 4 e 8 cores, respectivamente. **IMP-GPU** apresenta os tempos obtidos com a placa 8800GT.

Tabela 7.10: Comparação com o estado da arte em um tecido com dimensão 513×1025.

Nome	Parabólico (s)		EDO (s)		Elíptico (s)		Total (s)	
	CPU	GPU 8800GT	CPU	GPU 8800GT	CPU	GPU 8800GT	CPU	GPU 8800GT
PD	22,78	11,87	107,05	1,24	130,75	29,19	269,51	43,04
AMD	18,32	9,32	63,95	0,76	108,06	21,97	224,91	32,30
XE1	16,98	-	71,28	-	92,98	-	188,50	-
XE4	11,69	-	18,09	-	58,98	-	95,10	-
2-XE4	6,40	-	9,13	-	37,45	-	58,88	-

Através da Tabela 7.10 é possível notar que o melhor desempenho em GPU foi obtido com a configuração AMD-8800GT. O tempo total foi $5,84$ vezes menor do que o melhor tempo obtido por uma implementação em CPU (**IMP-Cluster**) com um único processador, configuração XE. O *speedup* obtido apenas para o problema parabólico, foi menor, em torno de 2. Para o problema elíptico o *speedup* foi de

3,28 em relação a configuração XE. Já para a resolução das EDOs, foi obtido um *speedup* de 84,14, em relação a configuração AMD. Também observamos que **IMP-GPU** (1 CPU + 1 GPU) oferece um desempenho melhor do que **IMP-Cluster** em 2 máquinas de 4 processadores, 2-XE4.

Capítulo 8

Discussão

Apresentamos no Capítulo 7 dois tipos de comparações de desempenho, uma comparação direta na Seção 7.3 entre a implementação em GPU, **IMP-GPU**, com uma implementação serial em CPU correspondente, **IMP-CPU**, e uma outra comparação na Seção 7.4 entre **IMP-GPU** e uma implementação em CPU com os melhores métodos conhecidos atualmente para resolução as equações do bidomínio em um pequeno cluster **IMP-Cluster**.

Para o caso **IMP-GPU** versus **IMP-CPU** obtemos um *speedup* em torno de $4,1$ da GPU 8800GT em relação à máquina XE para o problema parabólico. Para o problema elíptico o *speedup* obtido pela GPU 8800GT foi em torno de $5,6$ em relação à máquina XE. Para a resolução dos sistemas de EDOs o *speedup* obtido pela GPU foi bem maior, chegando a $93,8$ em relação a CPU XE. O *speedup* para todo o processo de solução das equações do bidomínio foi em torno de 7 obtido pela GPU 8800GT em relação à máquina XE. Também foi apresentado uma comparação mais detalhada entre as operações que compõem os métodos numéricos. Foi possível observar que as operações que obtiveram o pior desempenho em GPU foram as operações que

realizavam um acesso à memória não sequencial. A cache das texturas em GPUs são bem reduzidas e otimizadas para uma localidade em duas dimensões, obtendo um desempenho ótimo em acessos sequenciais como os realizados pela operações SAXPY. O acesso à memória das operações de restrição, interpolação e produto interno não é sequencial e prejudica o desempenho dessas operações. O produto interno sofre de outros dois problemas. O primeiro é o maior número de acessos à memória ocasionadas pelas reduções sucessivas. O segundo problema é que quando o grid é reduzido existe um tamanho em que não é mais vantajoso realizar o cálculo em GPU. O mesmo problema ocorre para as operações de SAXPY, multiplicação matriz vetor e ω-Jacobi para os grids mais grosseiros no método multigrid. Para o caso do produto interno, como o resultado precisa ser transferido para a CPU, uma maneira de melhorar seu desempenho seria não realizar as reduções na GPU até obter um único valor mas sim encontrar um número ótimo de reduções na GPU tal que seja mais eficiente transferir os dados restantes para a CPU que finalizaria a computação. Para o multigrid onde os dados não precisam ser trocados com a CPU, esta solução não seria apropriada, devido ao alto custo de transferência de dados entre CPU e GPU e a grande quantidade de vezes que isso seria necessário.

Para a comparação **IMP-GPU** versus **IMP-Cluster** em serial os *speedups* obtidos pela GPU 8800GT em relação à implementação serial **IMP-Cluster** foram de $1,82$, $3,28$, $93,8$, e $5,84$ para o problema parabólico, elíptico, sistemas de EDOs e o tempo total respectivamente. A comparação foi realizada em relação à máquina XE. Já comparando **IMP-GPU** com a implementação paralela **IMP-Cluster** utilizando duas máquinas Quad Core Xeon (2-XE4) totalizando 8 processadores a implementação na GPU 8800GT obteve *speedups* de $1,7$, $12,01$ e $1,82$ para o problema elíptico,

os sistemas de EDOS e o tempo total. Para o problema parabólico o hardware 2-XE4 obteve um *speedup* sobre a GPU 8800GT de $1,46$.

O que mais se destacou nas implementações foi o *speedup* na resolução dos sistemas de EDOs pela GPU em relação às CPUs. Isto ocorreu pois para cada passo de tempo da simulação um conjunto de sistemas de EDOs é resolvido, sendo um sistema de EDOs para cada elemento do tecido. Esta operação é totalmente paralelizável, o que explica o ótimo desempenho das GPUs comparados com os desempenhos das CPUs.

Já para o problema parabólico o que se destacou foi o *slowdown* da implementação em GPU em relação ao hardware com 8 processadores. Um dos motivos para isto foi a diferença entre os precondicionadores utilizados nas duas versões. A implementação em GPU utilizou o precondicionador Jacobi que faz com que a solução da equação parabólica necessite de aproximadamente 20 iterações do método dos gradientes conjugados precondicionado por cada passo de tempo. Já a implementação paralela em CPU utilizou o precondicionador Block-ILU(0), que é muito mais eficiente, onde o método dos gradientes conjugados precondicionado necessita em torno de 7 iterações apenas.

Para a solução do problema elíptico a **IMP-GPU** se mostrou mais eficiente do que a implementação **IMP-Cluster** mesmo com a utilização de um precondicionador mais eficiente pela implementação em CPU. A implementação em GPU obteve um *speedup* de $1,7$ em relação a implementação paralela em CPU com 8 processadores (2-XE4).

Mesmo utilizando métodos numéricos menos eficientes, a GPU se mostrou mais

eficiente que uma implementação paralela do simulador cardíaco em 2 máquinas com 4 cores cada uma, totalizando 8 processadores, sendo $1,82$ vezes mais rápida, utilizando apenas uma CPU AMD e uma placa de vídeo GeForce 8800GT. Este resultado é ainda mais significativo se compararmos o custo benefício das duas plataformas. O custo da plataforma AMD-8800GT é, atualmente, cerca de um sexto da plataforma 2-XE4.

8.1 Trabalhos Futuros

Pôde ser observado pelos dados referentes ao desempenho que algumas operações obtiveram um desempenho fraco na implementação em GPU. Principalmente as operações de restrição, interpolação e produto interno. Como citado anteriormente, o algoritmo de produto interno pode ser modificado para que a parte final da computação seja realizada na CPU. Entretanto, para cada configuração de GPU e CPU um balanceamento diferente precisa ser encontrado. Além disso, outros algoritmos para estas operações podem ser desenvolvidos na tentativa de se obter um padrão de acesso à memória que utilize melhor a cache das texturas. Outra abordagem para melhorar o desempenho do simulador cardíaco é encontrar precondicionadores ou métodos numéricos que, paralelizados em GPU, obtenham um desempenho melhor.

Neste trabalho foi mostrada uma abordagem para a programação de GPUs utilizando OpenGL e GLSL. No entanto, já está em andamento testes da utilização da linguagem CUDA que vem se tornando um padrão para a programação GPGPU. Realizamos alguns testes iniciais comparando o desempenho entre CPU, OpenGL e CUDA para a implementação do método de relaxação ω-Jacobi. A Tabela 8.1

mostra uma comparação de desempenho entre duas implementações em GPU, com uma implementação em CPU para 1000 iterações do método em um tecido com dimensão 513×1025.

Tabela 8.1: Speedup das implementações em GPU (CUDA e OpenGL) em relação a implementação em CPU em um tecido com dimensão 513×1025.

	8600GT		8800GT	
	Tempo (s)	Speedup	Tempo (s)	Speedup
CUDA	2,868	16,73	1,042	48,68
OpenGL	3,737	12,84	1,277	39,72

Podemos perceber que o desempenho obtido pela implementação em CUDA foi superior ao desempenho obtido pela implementação em OpenGL. Estes resultados preliminares indicam que a utilização da linguagem CUDA pode melhorar ainda mais o desempenho da implementação em GPU das equações do bidomínio, além de permitir que seja utilizada a precisão dupla nativa dos hardwares mais recentes. Além disso, a linguagem CUDA permite que programadores sem experiência em computação gráfica possam implementar seus problemas quase da mesma maneira que fariam na linguagem C o que torna os códigos mais legíveis. Uma outra vantagem é que a programação em CUDA oferece mais flexibilidade na utilização da GPU, permitindo que escritas aleatórias na memória possam ser realizadas. No entanto, ainda existe muita discussão sobre a superioridade do desempenho de implementações em OpenGL sobre implementações em CUDA. A biblioteca OpenGL é desenvolvida para obter o melhor desempenho em aplicações gráficas enquanto a linguagem CUDA mapeia melhor o hardware gráfico NVIDIA para a programação de propósito geral. Como trabalho futuro, propomos uma implementação do simu-

lador cardíaco em GPU utilizando a linguagem CUDA e uma comparação com a abordagem com OpenGL com as modificações propostas.

Um outro trabalho a ser realizado é a modificação do código em GPU para que possa ser paralelizado em um cluster de GPUs. O cluster de GPUs consiste em várias GPUs que podem estar em diferentes máquinas interconectadas por uma rede, ou até mesmo em uma única máquina multicore (FAN *et al.*, 2004).

Capítulo 9

Conclusões

Neste trabalho foi apresentada uma abordagem utilizando GPU para a solução das equações que modelam a atividade elétrica no coração. Foi mostrado que a modelagem cardíaca, pode ser dividida em duas partes, um modelo para as células cardíacas e um modelo para o tecido cardíaco. Esta modelagem gera um sistema de equações diferenciais não-lineares que, através de um operador *splitting*, pode ser resolvido por um esquema numérico de 3 passos envolvendo a solução de uma EDP parabólica, um sistema não-linear de EDOs e uma EDP elíptica. O sistema de EDOs é resolvido através do método de Euler explícito. Ambas EDPs são discretizadas no espaço utilizando o método dos elementos finitos com elementos quadrados e interpolação bilinear. Após esta discretização a solução das EDPs é dada pela solução de um sistema linear de equações. Os sistemas lineares gerados foram resolvidos pelo método dos gradientes conjugados precondicionado por Jacobi e Multigrid, para as EDPs parabólica e elíptica, respectivamente. Estes métodos numéricos foram implementados em CPU e também em GPU para serem comparados. Também foram implementados outros precondicionadores como o ILU(0) para uma implementação

serial em CPU e o block-ILU para uma implementação paralela em CPU.

Apesar das GPUs serem projetadas para propósitos gráficos, as GPUs atuais podem ser utilizadas para propósitos gerais, pois suas pipelines possuem processadores que são programáveis. Utilizamos nesse trabalho uma abordagem para a programação de GPUs utilizando a API gráfica OpenGL em conjunto com a linguagem de programação dos processadores gráficos, GLSL. Grande parte das operações envolvidas nos métodos numéricos para a solução dos sistemas lineares é facilmente paralelizável e, portanto foi trivial sua implementação em GPU. Porém existem operações onde a paralelização em GPU não é tão simples, necessitando de algoritmos mais complicados do que os implementados em CPU, como é o caso do produto interno. A implementação da solução dos sistemas de EDOs em GPU foi simples, sendo que cada sistema é resolvido para cada célula do tecido paralelamente. A abordagem utilizada possui o inconveniente de suportar apenas operações em ponto flutuante com precisão simples. No entanto, para o caso da equação elíptica, os erros devido à esta restrição são grandes, o que tornou necessária a implementação de uma emulação de precisão dupla para o caso da operação de multiplicação matriz vetor.

A implementação em GPU da solução das equações do bidomínio obteve bons resultados em termos de desempenho. A simulação total em GPU de um problema com dimensão 513×1025 em 40 passos de tempo, descrito no Capítulo 6, obteve um *speedup* de aproximadamente 7 comparado à uma implementação em CPU utilizando os mesmos métodos numéricos. A implementação em GPU também foi comparada com uma implementação em CPU utilizando os melhores métodos numéricos conhe-

cidos atualmente para a resolução das equações do bidomínio. A implementação em GPU obteve um *speedup* de $5,8$ neste caso. Utilizando uma implementação paralela dos melhores métodos numéricos para CPU, a GPU ainda assim obteve um *speedup* de $1,8$ comparado a um cluster com 8 processadores.

Estes resultados são ainda mais significativos se compararmos os custos das plataformas de computação. A plataforma AMD com a GPU GeForce 8800GT tem o custo de aproximadamente um sexto do custo do pequeno cluster de dois computadores 2-XE4. Além disso, o consumo elétrico e de espaço físico das plataformas também indicam uma grande vantagem das GPUs sobre as CPUs. Desta forma, concluímos que a tecnologia de GPUs é bastante promissora para aceleração de simulações cardíacas.

Apêndice A

Criando um programa GPGPU completo

Na Seção 4.2.2 foi mostrado um exemplo da soma de vetores em GPU, no entanto, muitos detalhes da implementação foram omitidos. Iremos detalhar a implementação desta soma de vetores utilizando a GPU.

A idéia básica é que em computação gráfica temos texturas que são utilizadas como dados de entrada para a renderização. Os hardwares gráficos atuais também permitem que o resultado da renderização seja escrito em uma textura e isso será importante mais adiante. Desta maneira, as texturas serão utilizadas para guardar os vetores de entrada que iremos somar, bem como o vetor contendo o resultado da soma.

Geralmente, em computação gráfica, as texturas possuem três canais de cores *Red Green Blue* (RGB) e um canal de transparência *Alpha* para cada *texel* ou elemento de textura onde cada canal possui 8 bits para guardar seu valor. No entanto, não queremos guardar os valores de nossos vetores em 8 bits porque isso limitaria muito

a precisão. Para isso existe um conjunto de extensões do OpenGL que fornecem suporte à ponto flutuante de 32 bits e também nos permitem realizar renderização fora da tela em uma textura.

A.1 Inicializando o OpenGL

O primeiro passo para a programação em GPU é realizar algumas inicializações do OpenGL. Iremos utilizar a biblioteca *OpenGL Utility Toolkit* (GLUT) para criar um contexto OpenGL tornando o hardware gráfico acessível. A biblioteca GLUT também faz o código independente do sistema de janelas tornando o programa mais portável. É também necessária a criação de uma janela gráfica mesmo que não seja nosso objetivo desenhar na tela.

Muitas das funções que precisamos para trabalhar com ponto flutuante na GPU não são parte do OpenGL, no entanto, extensões OpenGL fornecem um mecanismo para que essas funções sejam utilizadas. A obtenção de ponteiros para as funções que as extensões definem é complicada, para isso utilizamos a biblioteca *OpenGL Extension Wrangler Library* (GLEW) que fornece tudo que precisamos de uma maneira simples.

O código final para as inicializações é dado à seguir:

```
glutInit(&argc, argv);
GLuint glutWindowHandle = glutCreateWindow("Nome qualquer da janela");
glewInit();
```

A.2 Configurando a renderização fora da tela

Em computação gráfica existem dois tipos básicos de projeções que transformam os objetos em um espaço 3D para o espaço da tela em 2D, a projeção ortogonal, e a projeção perspectiva. Na projeção ortogonal os objetos são projetados diretamente no espaço 2D sem nenhuma diferença de tamanho caso estejam mais distantes da câmera. Já na projeção perspectiva os objetos mais distantes da câmera são projetados menores no espaço 2D, essa projeção é mais realista pois se assemelha com nossa forma de visão, onde os objetos em maior distância são vistos menores pelo observador. A Figura A.1 mostra um cubo projetado com projeção ortogonal em (a) e com projeção perspectiva em (b).

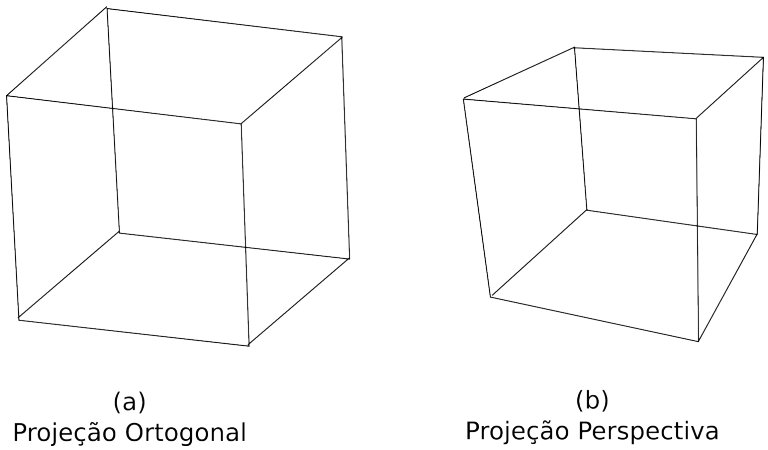

Figura A.1: Um cubo projetado com os dois tipos de projeções. Em (a) a projeção ortogonal e em (b) a projeção perspectiva.

Apesar da projeção perspectiva parecer mais realista, é na projeção ortogonal que estamos interessados pois ela não causa nenhuma deformação e, desta forma, não irá influenciar no cálculo da soma dos vetores. Para configurarmos a visualização da maneira correta precisamos utilizar a projeção ortogonal e, para isso, podemos utilizar a função *gluOrtho2d* da biblioteca *OpenGL Utility Library* (GLU) que é uma

biblioteca de comandos auxiliares ao OpenGL (SHREINER *et al.*, 2005). Abaixo apresentamos o protótipo dessa função.

```
void gluOrtho2D( GLdouble left,    GLdouble right,
                 GLdouble bottom,  GLdouble top )
```

Além da transformação de projeção existe a transformação da janela de visualização. Essas duas transformações determinam como a cena será mapeada na tela. A transformação de visualização define o formato da área de tela em que a cena será mapeada. Para definirmos esse formato usamos a seguinte função OpenGL:

```
void glViewport( GLint   x,     GLint   y,
                 GLsizei width, GLsizei height )
```

O trecho de código completo para a configuração da visualização é o seguinte:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, largura, 0.0, altura);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0, 0, largura, altura);
```

A função *glMatrixMode* é utilizada para dizer em qual matriz de transformação será aplicada a transformação. A função *glLoadIdentity* coloca a matriz identidade na matriz especificada por *glMatrixMode* pois a transformação que será aplicada é construída em forma de multiplicação de matrizes e precisamos, dessa maneira, descartar as transformações realizadas anteriormente. Os parâmetros das funções *gluOrtho2D* e *glViewport* são escolhidos para que a projeção seja realizada a partir de $(0, 0)$ até $(largura, altura)$. A transformação de projeção é configurada para ser

2D pois não precisamos de profundidade para somar os vetores. Desta forma, agora já preparamos a renderização corretamente para colocar o resultado da soma dos vetores. Com esta configuração poderemos ter vetores com a dimensão *largura* × *altura*.

O próximo passo é a criação de um *framebuffer* para que possamos renderizar em uma textura ao invés de renderizar na tela. Podemos fazer isso da seguinte maneira:

```
GLuint fb;
glGenFramebuffersEXT(1, &fb);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
```

A função *glGenFramebuffersEXT* recebe dois parâmetros, o primeiro informa o número de framebuffers a serem gerados e o segundo é um ponteiro para armazenar os framebuffers. A segunda função *glBindFramebufferEXT* torna o framebuffer criado o destino da renderização, permitindo uma renderização *offscreen*, ou seja, fora da tela em uma textura como queremos.

A.3 Criando texturas para guardar os vetores

Agora precisamos criar as texturas que guardarão os vetores tanto de entrada como de saída. Fazemos isso com a sequência de funções:

```
GLuint textura1;
glGenTextures(1, &textura1);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, textura1);

glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MIN_FILTER,
                GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MAG_FILTER,
                GL_NEAREST);
```

```
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_S,
                GL_CLAMP);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_T,
                GL_CLAMP);

glTexImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, GL_FLOAT_R32_NV,
             largura, altura, 0, GL_LUMINANCE, GL_FLOAT, 0);
```

A função *glGenTextures* cria uma textura chamada *textura1* e a função e *glBindTexture* diz que os parâmetros que serão modificados em seguida por *glTexParameteri* afetarão apenas a textura criada. Os dois primeiros parâmetros da textura modificados por *glTexParameteri* com *GL_NEAREST* diz que o texel com o centro mais próximo de cada pixel será o texel utilizado e não será feita uma média. Isto é necessário para que o valor não seja erroneamente interpolado o que poderia acontecer se tivéssemos utilizado o parâmetro *GL_LINEAR*. Os dois últimos parâmetros da textura modificados por *glTexParameteri* dizem que quando as coordenadas das texturas são especificadas fora dos seus limites, como no caso de um acesso de um *array* fora dos limites em CPU, o acesso à textura irá retornar o valor da borda mais próximo da textura usando *GL_CLAMP*. A última função, *glTexImage2D*, aloca a memória para guardar a textura 2D na GPU de acordo com o tipo de armazenamento especificado pelos parâmetros dados que dizem basicamente que cada elemento da textura será tratado como apenas um ponto flutuante de 32 bits.

Devemos repetir o procedimento anterior para todas as texturas criadas inclusive a textura que irá receber o valor da computação. Porém essa textura precisa ser anexada ao framebuffer para que receba o resultado. Para fazermos isso devemos proceder da seguinte maneira:

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          GL_COLOR_ATTACHMENT0_EXT,
                          GL_TEXTURE_RECTANGLE_ARB,
                          textura1, 0);
glDrawBuffer (GL_COLOR_ATTACHMENT0_EXT);
```

A função *glBindFramebufferEXT* apenas certifica que o framebuffer atual é o framebuffer *fb* que criamos anteriormente. Já a função *glFramebufferTexture2DEXT* anexa a *textura1* ao framebuffer. E *glDrawBuffer* diz que a escrita será realizada no buffer que a textura foi anexada na função *glFramebufferTexture2DEXT*.

A.4 Criando o programa para a GPU

Agora está tudo pronto para iniciar a computação, mas antes precisamos criar o programa que será executado pelo processador de fragmentos e calculará a soma dos dois vetores (texturas) e guardará o resultado na textura anexada ao framebuffer. Esse programa é conhecido como *fragment shader* e é mostrado na Figura 4.8 e repetido a seguir:

```
#extension GL_ARB_texture_rectangle : enable
uniform sampler2DRect texture_y;
uniform sampler2DRect texture_x;

void main(void){
   float y = texture2DRect(texture_y, gl_FragCoord.xy).x;
   float x = texture2DRect(texture_x, gl_FragCoord.xy).x;
   gl_FragColor.x =  x + y;
}
```

Os parâmetros *uniform sampler2DRect* são as texturas de entrada contendo os vetores que desejamos somar. A função *texture2DRect* lê o valor da textura no

ponto dado por *gl_FragCoord*. Esse programa é executado em paralelo para todos os elementos da textura e o resultado da soma é colocado na textura anexada ao framebuffer.

Precisamos no entanto carregar o código e compilar para que seja executado na GPU. Todo esse processo é feito em tempo de execução. Para isso precisamos colocar o código do shader em um *array* de texto (ou *string*). Isto pode ser feito diretamente no código-fonte C colocando o texto contendo o código-fonte do shader em um *array* estático, ou uma função que faz a leitura de um arquivo contendo o código-fonte do shader e o guarda em um *array* dinâmico. No código que apresentamos a seguir a função *carrega_shader* tem esse objetivo e precisa ser implementada pelo programador.

```
GLuint programa_soma_vetores = glCreateProgram();
GLuint frag_shader_soma_vetores = glCreateShader(GL_FRAGMENT_SHADER_ARB);

const char *shader_fonte = carrega_shader("nome_do_arquivo_fonte");
glShaderSource(frag_shader_soma_vetores, 1, &shader_fonte, NULL);

glCompileShader(frag_shader_soma_vetores);
glAttachShader(programa_soma_vetores, frag_shader_soma_vetores);
glLinkProgram(programa_soma_vetores);
```

O trecho de código acima está dividido em três partes, na primeira parte são criados um programa objeto vazio e um shader objeto vazio. Um programa objeto é um programa no qual um shader pode ser anexado. Na segunda parte o arquivo contendo o código-fonte do shader é lido e depois o código-fonte é associado ao shader objeto criado anteriormente. Na terceira parte o shader é compilado e anexado ao programa objeto que é finalmente ligado criando um executável que será executado no processador de fragmentos.

A.5 Copiando vetores na CPU para texturas na GPU

Grande parte computação em GPU pode ser realizada sem a interferência da CPU, no entanto, precisamos passar valores para as texturas ao menos para dar inicio à computação. Se desejarmos transferir os valores contidos em um *array* na CPU podemos fazê-lo da seguinte forma:

```
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, textura2);
glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, 0, 0,
                largura, altura, GL_LUMINANCE,
                GL_FLOAT, vetor1_entrada_cpu);
```

A função *glBindTexture* serve para dizer qual a textura será afetada pela função *glTexSubImage2D* que copia os valores da memória da CPU para a memória da GPU. É importante lembrar que o tamanho do vetor deve que ser igual a $largura \times altura$.

Agora precisamos associar as texturas com os dados de entrada ao *shader*. A técnica de multi texturização permite que várias texturas sejam aplicadas a um mesmo polígono e precisamos disso para podermos utilizar as duas texturas como entrada para a soma dos dois vetores. Existem as unidades de textura que são utilizadas para esse proposito e precisamos dizer em qual unidade de textura nossa textura será associada. Então procedemos da seguinte maneira :

```
glUseProgram(programa_soma_vetores);

GLuint shader_var0 = glGetUniformLocation(programa_shader,
                                          "texture_x");
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, textura2);
glUniform1i(shader_var0, 0);
```

A função *glUseProgram* escolhe o programa a ser executado pelo processador de vértices e de fragmentos. Neste caso apenas o programa para o processador de fragmentos foi modificado para realizar a soma dos vetores. Isto é necessário pois é possível se ter mais de um programa compilado na GPU e desta maneira precisamos escolher qual iremos executar. Já a função *glGetUniformLocation* retorna a localização da variável *texture_x* no programa escolhido por *glUseProgram*. Depois tornamos a unidade de textura 0 ativa com *glActiveTexture* e fazemos com que a *textura2* seja textura atual com *glBindTexture*. Finalmente a textura atual é utilizada como a textura de número 0 para a multi-texturização e associada no shader com a variável *texture_x* através da localização obtida em *glGetUniformLocation*. Se fôssemos realizar a mesma operação para a variável *texture_y* com uma textura chamada *textura3* faríamos o seguinte:

```
GLuint shader_var1 = glGetUniformLocation(programa_soma_vetores,
                                          "texture_y");
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, textura3);
glUniform1i(shader_var1, 1);
```

A.6 Realizando a computação

O próximo passo é a realização da computação da soma dos vetores guardados nas texturas. Para isso temos que proceder como se fôssemos desenhar um quadrilátero que cobrisse exatamente toda a região da nossa janela de visualização. Para isso basta desenharmos o quadrilátero com as extremidades em $(0, 0)$ e $(largura, altura)$ usando o seguinte código:

```
glPolygonMode(GL_FRONT,GL_FILL);
glBegin(GL_QUADS);
    glVertex2f(0, 0);
    glVertex2f(largura, 0);
    glVertex2f(largura, altura);
    glVertex2f(0, altura);
glEnd();
```

A função *glPolygonMode* possui dois parâmetros *GL_FRONT* e *GL_FILL*. O primeiro diz que apenas a parte da frente do polígono será renderizada evitando cálculos desnecessários para desenhar a superfície de trás do polígono que nem mesmo é visível por esse ângulo. O sentido da inserção dos vértices define qual será a parte da frente do polígono, neste caso o sentido é o anti-horário. O segundo parâmetro diz que o polígono deve ser preenchido ou todo colorido e não apenas as bordas desenhadas. As próximas funções definem o tipo de polígono a ser desenhado, nesse caso um quadrilátero, e a posição dos vértices deste quadrilátero.

Logo após a execução deste trecho de código os vértices são enviados para a *pipeline* onde receberá as transformações necessárias e no processador de fragmentos cada pixel receberá a soma dos elementos das duas texturas de entrada, de acordo com o shader implementado, e o resultado será armazenado na textura anexada ao framebuffer.

O ponto mais importante em toda configuração realizada anteriormente é que cada elemento de textura deve ser mapeado exatamente para um pixel. Para isso também é necessário que o quadrilátero tenha os vértices nas posições corretas para que nenhuma interpolação seja realizada com os dados das texturas.

A.7 Copiando resultados para a CPU

Depois de realizados os cálculos, queremos copiar o resultado de volta para CPU. O seguinte trecho de código e responsável por esta tarefa:

```
glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
glReadPixels(0 , 0, largura, altura, GL_LUMINANCE,
             GL_FLOAT, vetor_saida_cpu);
```

Primeiramente escolhemos o buffer de leitura com *glReadBuffer*. A textura de saída foi associada à esse buffer anteriormente. Finalmente lemos os dados deste buffer contendo a soma dos vetores para um *array* na CPU.

Apêndice B

Biblioteca RGP

Neste apêndice apresentamos as funções da biblioteca RGP criada neste trabalho para facilitar a implementação dos métodos numéricos em GPU. Foram implementadas estruturas de dados para guardar informações sobre os vetores criados em GPU. Esses vetores são armazenados na GPU em forma de texturas. Nesta biblioteca foram implementados dois tipos básicos de vetores, um tipo de vetor de somente leitura (Rgp_Vector_Read) e um tipo de vetor de leitura e escrita (Rgp_Vector_PingPong). A Tabela B.1 mostra e descreve as funções da biblioteca.

Tabela B.1: Listagem e descrição das funções da biblioteca RGP.

Função	Descrição
`char *rgp_text_file_read(char *filename)`	Função para retornar uma string (char $*$) com o conteúdo do arquivo de nome *filename*. Utilizada para carregar arquivos-fonte dos shaders.
`void rgp_init(int argc, char **argv)`	Função de chamada obrigatória no início de qualquer programa que utilize alguma outra função desta biblioteca. É responsável por configurar e receber parâmetros do OpenGL. Os parâmetros argc e argv são os recebidos na linha de comando pelo programa main.
`void rgp_finalize()`	Função para finalizar o OpenGL

`void rgp_create_frame_buffer(GLuint *fb,`
`GLuint tex_size,`
`int fixed_widthsize)`
	Cria um frame buffer fb. É necessário para escrever nas texturas, ou seja, com o frame buffer é possível fazer uma textura passar de read-only para write-only. O parâmetro *tex_size* especifica o tamanho em 1D, ou seja, largura vezes altura. Caso o parâmetro *fixed_width_size* seja igual a zero a função tentar criar um frame buffer quadrado.

Função	Descrição
`void rgp_delete_frame_buffer(GLuint *fb)`	Função para liberar o espaço do frame buffer *fb* quando este não for mais utilizado.
`void rgp_create_shaders(GLuint *program_object, GLuint *fragment_shader_object, GLuint *vertex_shader_object)`	Cria o programa objeto e os shaders objetos. Caso apenas um dos shaders seja necessário, o outro deve ser passado como *NULL*
`void rgp_shader_compiler_error_msgs( GLuint *shader_object)`	Imprime na tela os erros de compilação do shader caso existam
`void rgp_load_shader(GLuint *shader_object, GLuint *program_object, char *filename)`	Carrega o shader de seu arquivo, compila e faz a *linkagem* ao programa objeto.
`void rgp_compute_reduction2( GLuint xi, GLuint yi, GLuint tsize_w, GLuint tsize_h)`	Invoca a computação em GPU de um retângulo com dimensão (t_size_w, t_size_h) a partir das coordenadas (xi,yi) do destino de renderização.
`void rgp_texture_to_write( Rgp_Vector_PingPong *texture, GLuint *fb)`	Especifica uma textura (*texture*) como destino de renderização do frame buffer *fb*.
`void rgp_textures_to_write8( Rgp_Vector_PingPong *texture0, Rgp_Vector_PingPong *texture1, Rgp_Vector_PingPong *texture2, Rgp_Vector_PingPong *texture3, Rgp_Vector_PingPong *texture4, Rgp_Vector_PingPong *texture5, Rgp_Vector_PingPong *texture6, Rgp_Vector_PingPong *texture7, GLuint *fb)`	Especifica oito texturas (*texture*$*$) como destinos de renderização do frame buffer *fb*.

Função	Descrição

`void rgp_swap_PingPong(`
`Rgp_Vector_PingPong *texture,`
`GLuint *fb,`
`int isUsedInShader)`
	Realiza a operação de ping-pong, onde a textura de escrita se torna de leitura e vice-versa. Caso a textura ainda não tenha sido associada a nenhum shader é necessário ter *isUsedInShader* = *RGP_NOT_USED_IN_SHADER*, caso contrário, o valor é default. Ao fazer a chamada, a textura write do vetor ping-pong se torna automaticamente a textura de escrita do frame buffer *fb*.

`void rgp_create_Vector_Read(`
`Rgp_Vector_Read *texture,`
`GLuint vsize,`
`GLuint fixed_widthsize)`
	Cria um vetor de apenas leitura na GPU alocando o espaço para o armazenamento da textura. O parâmetro *vsize* especifica o tamanho em 1D da textura sendo alocada. A largura da textura é dada por *fixed_widthsize*, caso o valor deste parâmetro seja zero a função tenta criar uma textura quadrada.

`void rgp_create_Vector_PingPong(`
`Rgp_Vector_PingPong *texture,`
`GLuint vsize,`
`GLuint *fb,`
`int fixed_widthsize)`
	Cria um vetor ping pong onde é possível realizar a escrita e leitura na GPU alocando o espaço para o armazenamento de duas texturas. O parâmetro *vsize* especifica o tamanho em 1D da textura sendo alocada. A largura da textura é dada por *fixed_widthsize*, caso o valor deste parâmetro seja zero a função tenta criar uma textura quadrada.

Função	Descrição

`void rgp_transfer_to_Vector(`
`Rgp_Vector_Read *texture,`
`float *data)`
	Transfere um vetor (*data*) da memoria RAM da CPU para um vetor somente leitura (*texture*) da GPU.

`void rgp_transfer_to_Vector(`
`Rgp_Vector_PingPong *texture,`
`float *data)`
	Transfere um vetor (*data*) da memoria RAM da CPU para um vetor ping-pong (*texture*) da GPU.

`void rgp_transfer_Vector_to_shader(`
`Rgp_Vector_Read *texture,`
`const GLchar *variable_name,`
`Rgp_Program *program_object)`
	Associa um vetor somente leitura da GPU (*texture*) com uma textura utilizada do shader especificado no programa *program_object* com o nome *variable_name*.

`void rgp_transfer_Vector_to_shader(`
`Rgp_Vector_PingPong *texture,`
`const GLchar *variable_name,`
`Rgp_Program *program_object)`
	Associa um vetor ping-pong da GPU (*texture*) com uma textura utilizada do shader especificado no programa *program_object* com o nome *variable_name*.

`void rgp_transfer_FrameBuffer_to_Vector(`
`float *data,`
`Rgp_Vector_PingPong *texture,`
`int sizew, int sizeh)`
	Transfere os dados de um vetor ping-pong na GPU (*texture*) para a o vetor em CPU (*data*). São transferidos os dados da coordenada (0,0) da textura até a coordenada (*sizew*, *sizeh*).

`void rgp_delete_Vector(` `Rgp_Vector_PingPong *texture)`	Libera o espaço ocupado na GPU pelo vetor ping-pong (*texture*).

Função	Descrição
`void rgp_delete_Vector(` `Rgp_Vector_Read *texture)`	Libera o espaço ocupado na GPU pelo vetor somente leitura (*texture*).

Referências Bibliográficas

AMORIM, R. M. e W., S. R. 2008, Solução das equações do monodomínio em unidades de processamento gráfico, In: *XI Encontro de Modelagem Computacional*, pp. 1–10, Volta Redonda, RJ.

BALAY, S., BUSCHELMAN, K., GROPP, W., KAUSHIK, D., KNEPLEY, M., MCINNES, L., SMITH, B. e ZHANG, H. 2002, *PETSc user manual*, Rel. Téc. ANL-95/11 - Revision 2.3.2, Argonne National Laboratory.

BARRETT, R., BERRY, M., CHAN, T., DEMMEL, J., DONATO, J., DONGARRA, J., EIJKHOUT, V., POZO, R., ROMINE, C. e DER, H. 1994, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

BRIGGS, W. L., HENSON, V. E. e MCCORMICK, S. F. 2000, *A multigrid tutorial (2nd ed.)*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, ISBN 0-89871-462-1.

BROWN, S. 1996, "Fpga architectural research: a survey", *IEEE, Design & Test of Computers*, v. 13, n. 4, pp. 9–15.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M. e HANRAHAN, P. 2004, Brook for gpus: stream computing on graphics

hardware, In: *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pp. 777–786, New York, NY, USA, ACM Press.

CRANK, J. e NICOLSON, P. 1947, "A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type", *Proceedings of the Cambridge Philosophical Society*, v. 43, pp. 50–67.

FAN, Z., QIU, F., KAUFMAN, A. e YOAKUM-STOVER, S. 2004, Gpu cluster for high performance computing, In: *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, p. 47, Washington, DC, USA, IEEE Computer Society.

FOX, G. e GANNON, D. 2001, "Computational grids", *Computing in Science and Engineering*, v. 3, n. 4, pp. 74–77.

GOCKENBACH, M. 2007, "Understanding and implementing the finite element method", *Sci. Program.*, v. 15, n. 2, pp. 117–119.

GÖDDEKE, D., STRZODKA, R. e TUREK, S. 2007, "Performance and accuracy of hardware-oriented native emulated and mixed-precision solvers in FEM simulations", *International Journal of Parallel, Emergent and Distributed Systems*, v. 22, n. 4, pp. 221–256.

HESTENES, M. R. e STIEFEL, E. 1952, "Methods of conjugate gradients for solving linear systems", *Journal of Research of the National Bureau of Standards*, v. 49, pp. 409–436.

HODGKIN, A. L. e HUXLEY, A. F. 1952, "A quantitative description of membrane

current and its application to conduction and excitation in nerve.", *J Physiol*, v. 117, n. 4, pp. 500–544.

KEENER, J. e SNEYD, J. 2001, *Mathematical Physiology*, Springer, ISBN 0387983813.

LEFEBVRE, S., HORNUS, S. e NEYRET, F. 2005, *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pp. 451–587, Addison Wesley.

MPI, F. P. I. 1994, *MPI: A Message-Passing Interface Standard*, Rel. Téc. UT-CS-94-230, University of Tennessee, Knoxville, TN, USA.

MUZIKANT, A. e HENRIQUEZ, C. 1998, "Validation of three-dimensional conduction models using experimental mapping: Are we getting closer?", *Prog Biophys Mol Biol*, pp. 205–223.

NVIDIA 2008, *CUDA Compute Unified Device Architecture*, NVIDIA, 2º edic., Programming Guide.

OLIVEIRA, R. S. 2008, *Ajuste Automático de Modelos Celulares Apoiado por Algoritmos Genéticos*, Dissert. de Mestrado, Programa de Modelagem Computacional, Universidade Federal de Juiz de Fora.

OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRÜGER, J., LEFOHN, A. E. e PURCELL, T. J. 2007, "A survey of general-purpose computation on graphics hardware", *Computer Graphics Forum*, v. 26, n. 1, pp. 80–113.

PRESS, W. H., FLANNERY, B. P., TEUKOLSKY, S. A. e VETTERLING, W. T.

1992, *Numerical Recipes in C : The Art of Scientific Computing*, Cambridge University Press, ISBN 0521431085.

PUWAL, S. e ROTH, B. J. 2007, "Forward euler stability of the bidomain model of cardiac tissue.", *IEEE Transactions on Biomedical Engineering*, v. 54, n. 5, pp. 951–953.

ROCHA, B. M. 2008, *Um modelo de acoplamento eletromecânico do tecido cardíaco através do método dos elementos finitos*, Dissert. de Mestrado, Programa de Modelagem Computacional, Universidade Federal de Juiz de Fora.

ROST, R. J. 2005, *OpenGL(R) Shading Language (2nd Edition)*, Addison-Wesley Professional, ISBN 0321334892.

SACHSE, F. B. 2004, *Computational Cardiology: Modeling of Anatomy, Electrophysiology, and Mechanics*, (Lecture Notes in Computer Science 2966), Springer, ISBN 3540219072.

SANTOS, R. W. 2002, *Modelagem da Eletrofisiologia Cardíaca*, Tese de Doutorado, Instituto de Matemática, Universidade Federal do Rio de Janeiro.

SANTOS, R. W., PLANK, G., BAUER, S. e VIGMOND, E. J. 2004, "Preconditioning techniques for the bidomain equations", *Lect Notes Comput Sci Eng 2004*, pp. 571–580.

SCHULTE, R. F., SACHSE, F. B., WERNER, C. D. e DÖSSEL, O. 2000, "Rule based assignment of myocardial sheet orientation", *Biomedizinische Technik*, v. 45, n. 2, pp. 97–102.

SHEWCHUK, J. R. 1994, *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*, Rel. Téc., Pittsburgh, PA, USA.

SHREINER, D., WOO, M., NEIDER, J. e DAVIS, T. 2005, *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*, Addison-Wesley Professional, ISBN 0321335732.

STRIKWERDA, J. C. 1989, *Finite Difference Schemes and Partial Differential Equations*, Mathematics Series, Wadsworth & Brooks/Cole, Pacific Grove, California.

TEN TUSSCHER, K., NOBLE, D., NOBLE, P. e PANFILOV, A. 2004, "A model for human ventricular tissue.", *Am J Physiol Heart Circ Physiol*, v. 286, n. 4, .

VIGMOND, E. J., AGUEL, F. e TRAYANOVA, N. A. 2002, "Computational techniques for solving the bidomain equations in three dimensions", *IEEE Transactions on Biomedical Engineering*, v. 49, n. 11, pp. 1260–1269.

WESSELING, P. 1992, *An Introduction to Multigrid Methods*, John Wiley & Sons, Chichester, UK, ISBN 978-1-930217-08-9, Reprinted by www.MGNet.org.

YAN, G. X., SHIMIZU, W. e ANTZELEVITCH, C. 1998, "Characteristics and distribution of m cells in arterially perfused canine left ventricular wedge preparations", *American Heart Association, Circulation*, v. 98, pp. 1921–1927.

YEO, C., BUYYA, R., POURREZA, H., ESKICIOGLU, R., GRAHAM, P. e SOMMERS, F. 2006, Cluster Computing: High-Performance, High-Availability, and High-Throughput Processing on a Network of Computers, In: A. Y. Zomaya,

ed., *Handbook of Nature-Inspired and Innovative Computing: Integrating Classical Models with Emerging Technologies*, cap. 16, pp. 521–551, Springer, New York, NY, USA, ISBN 0-387-40532-1.