Universidade Federal de Juiz de Fora
Instituto de Ciencias Exatas
Bacharelado em Bacharelado em Sistemas de Informação

Impacto de otimizações no desempenho de um simulador cardíaco paralelo em um ambiente multi-GPU

Gabriel Lomba Aguiar Costa

JUIZ DE FORA
DEZEMBRO, 2017

Impacto de otimizações no desempenho de um simulador cardíaco paralelo em um ambiente multi-GPU

Gabriel Lomba Aguiar Costa

Universidade Federal de Juiz de Fora

Instituto de Ciencias Exatas

Departamento de Ciência da Computação

Bacharelado em Bacharelado em Sistemas de Informação

Orientador: Marcelo Lobosco

JUIZ DE FORA

DEZEMBRO, 2017

Impacto de otimizações no desempenho de um simulador cardíaco paralelo em um ambiente multi-GPU

Gabriel Lomba Aguiar Costa

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIENCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM BACHARELADO EM SISTEMAS DE INFORMAÇÃO.

Aprovada por:

Marcelo Lobosco
D.Sc., Engenharia de Sistemas e Computação (UFRJ)

Rodrigo Weber dos Santos
Doutor

Bernardo Martins Rocha
Doutor

JUIZ DE FORA
01 DE DEZEMBRO, 2017

Aos meus pais, por sempre me apoiarem, principalmente na minha educação

Resumo

A simulação de processos biofísicos é um fator importante no estudo de doenças e eventual desenvolvimento de tratamentos. Naturalmente, devido à complexidade destes fenômenos, os modelos matemáticas gerados são compostos de muitas equações, tornando as simulações custosas. Neste trabalho, um simulador cardíaco que requer a resolução de sistemas lineares de EDPs e não-lineares de EDOs é analisado e modificado a fim de diminuir seu tempo de execução, aplicando otimizações conhecidas na literatura. As otimizações buscam melhorar os acessos à memória da GPU e são compostas de: redução do número de variáveis alocadas, remoção de acessos desnecessários e utilização da memória de constantes da GPU. Estas mudanças são aplicadas e seus resultados são comparados à versão original, em um mesmo ambiente sob os mesmos parâmetros. Pelos resultados obtidos, percebeu-se que refatorações a fim de otimizar os acessos à memória podem impactar no desempenho significativamente. Neste trabalho, as otimizações geraram um ganho de desempenho de, aproximadamente, 3.5%.

Palavras-chave: Simulador cardíaco, ambientes multi-GPU, Otimização de acesso a memória, Computação de Alto Desempenho

Abstract

The simulation of biophysical processes is an important factor in the study of diseases and eventual development of treatments. Naturally, due to the complexity of these phenomena, the mathematical models generated are composed by many equations, making the simulations computationally expensive. In this work, a cardiac simulator, which requires the resolution of linear PDE systems as well as non-linear ODE systems, is analyzed and modified in order to reduce its execution time, applying for this purpose well known optimizations found in the literature. The optimizations try to reduce the number of memory access performed by the GPU and consist of: reducing the number of variables allocated in memory, removing unnecessary accesses, and using the GPU constant memory. These changes are applied and their results are compared to the original version, in the same environment and using the same set of parameters. The results obtained have shown that refactorings to optimize memory accesses can significantly impact performance. In this work, the optimizations contributed to performance gains up to 3.5%, approximately.

Keywords: Cardiac simulator, Multi-GPU environments, Memory accesses optimizations, High-performance computing

Agradecimentos

A todos os meus parentes e amigos, pelo encorajamento e apoio.

Ao professor Marcelo Lobosco pela ótima orientação e paciência, sem a qual este trabalho não se realizaria.

Aos professores do Departamento de Ciência da Computação pelos seus ensinamentos e aos funcionários do curso, que durante esses anos, contribuíram de algum modo para o nosso enriquecimento pessoal e profissional.

Conteúdo

Lista de Tabelas

1 Introdução

1.1 Motivação

Segundo a *American Heart Association* (AHA), doenças cardiovasculares são a principal causa global de mortes, exceto na África sub-Saariana (Benjamin et al, 2017). A AHA estima que 92,1 milhões de adultos estadunidenses têm ao menos um tipo de doença cardiovascular, o que representa mais que 1/3 da população do país. Portanto, entender os fenômenos associados a estas doenças é crucial e um dos primeiros passos que levam à descoberta de novos tratamentos é a melhor compreensão de suas causas.

Neste cenário, modelos computacionais se tornaram ferramentas valiosas, já que podem ser usados para testar novos medicamentos e tratamentos, além de ajudarem no desenvolvimento de novos aparelhos médicos e de novas técnicas de diagnóstico não invasivas para diversas doenças cardíacas (Morgan et al, 2009; Panfilov et al, 2000; Biktashev et al, 1996; Dos Santos et al, 2006, 2004; Campos et al, 2013).

Como se pode imaginar, usar ferramentas matemáticas e computacionais para descrever a função e estrutura do coração não é uma tarefa trivial. O fenômeno de propagação elétrica no coração envolve um conjunto de processos biofísicos complexos e não lineares. Estes fenômenos geram modelos computacionais complexos, que são descritos por sistemas não lineares de equações diferenciais parciais (EDPs) acoplados a um sistema não linear de equações diferenciais ordinárias (EDOs), resultando em um problema com milhões de variáveis e centenas de parâmetros.

As simulações em grande escala, como as resultantes da discretização de um coração inteiro, representam um grande desafio computacional. Arquiteturas paralelas podem ser usadas para reduzir o tempo de execução desses modelos complexos. Em trabalhos anteriores (Barros et al, 2013; Cordeiro et al, 2017), foi apresentada uma solução para este problema com base em plataformas Multi-GPU (*clusters* equipados com unidades de processamento de gráficos) que permitem simulações rápidas de modelos de tecido microscópico. A solução baseou-se na fusão de duas técnicas de alto desempenho dife-

rentes já investigadas anteriormente para modelagem cardíaca: computação em *clusters* baseada em comunicações com passagem de mensagens (MPI) (Dos Santos et al, 2004; Plank et al, 2007; Dos Santos et al, 2004; Xavier et al, 2009); e GPGPU (computação de propósito geral em unidades de processamento gráfico) (Rocha et al, 2011, 2010; Amorim et al, 2010, 2009; Oliveira et al, 2012; Amorim et al, 2012). Também foi apresentada uma técnica para reunir dados e execuções em núcleos de GPU em um ambiente Multi-GPU que não possui suporte à tecnologia *Hyper-Q* (Cordeiro et al, 2017).

Este trabalho foca no emprego de técnicas de computação de alto desempenho (área da Ciência da Computação preocupada com o ganho de desempenho na execução de aplicações que requerem uma quantidade significativa de recursos em sua execução) na execução de uma aplicação que calcula a atividade elétrica do tecido cardíaco, apresentada em (Cordeiro et al, 2017). Uma série de otimizações já conhecidas serão implantadas no simulador, viabilizando a comparação entre as vantagens (ou desvantagens) de cada otimização.

1.2 Objetivos gerais e específicos

O objetivo geral do trabalho é estudar o impacto das otimizações propostas no desempenho da aplicação estudada. Estas otimizações também podem vir a ser aplicadas em outros sistemas executados em plataformas multi-CPU e multi-GPU.

1.3 Método

Para fins de análise do impacto das otimizações, o seguinte método será utilizado. Primeiro, o simulador em sua versão original será executado e os tempos de execução serão coletados. Em seguida, as otimizações serão aplicadas no código do simulador, que será novamente executado para coleta dos tempos de execução de modo que, posteriormente, sejam realizadas as comparações.

2 Fundamentação Teórica

Neste capítulo, serão apresentados as arquiteturas utilizadas, tais quais os conceitos de programação paralela em CPU e GPU. Estes conceitos serão importantes para o entendimento do ambiente no qual a aplicação será executada e para a análise do impacto das otimizações implementadas neste trabalho.

2.1 *Clusters* de Computadores

Um *cluster* é um sistema composto por uma série de computadores independentes interconectados por uma rede. No âmbito da execução de algoritmo massivamente paralelos, os *clusters* são ferramentas poderosas e escaláveis.

A arquitetura de memória utilizada em um *cluster* é a de memória distribuída. Ela permite que os computadores, denominados nós, funcionem de modo independente, possuindo sua própria memória, disco e cópia do Sistema Operacional.

Visto que a implementação do paralelismo de dados nesta arquitetura requer repasse de dados e eventuais comunicações e/ou sincronizações, a latência da rede exerce um papel importante no desempenho da aplicação, devido às comunicações frequentes entre nós.

Nos últimos anos, os *clusters* tornaram-se sistemas híbridos, visto que cada nó é, geralmente, um sistema de memória compartilhada composto por processadores com um ou mais núcleos de processamento (Pacheco et al, 2011). Também observa-se uma tendência de agregar, a cada nó, unidades especializadas no processamento de grandes quantidades de dados, os chamados aceleradores. GPUs (*General Processing Units*) e FPGAs (*Field-programmable Gate Arrays*) são exemplos de aceleradores.

Como visto na figura 2.1, quando a computação é realizada em um *cluster*, normalmente os dados são oriundos de um computador mestre ou raiz, que os distribui para os demais nós, chamados de escravos. Estes nós computam os resultados parciais e, ao final, os repassam para o computador mestre para que possam ser unidos em um resultado

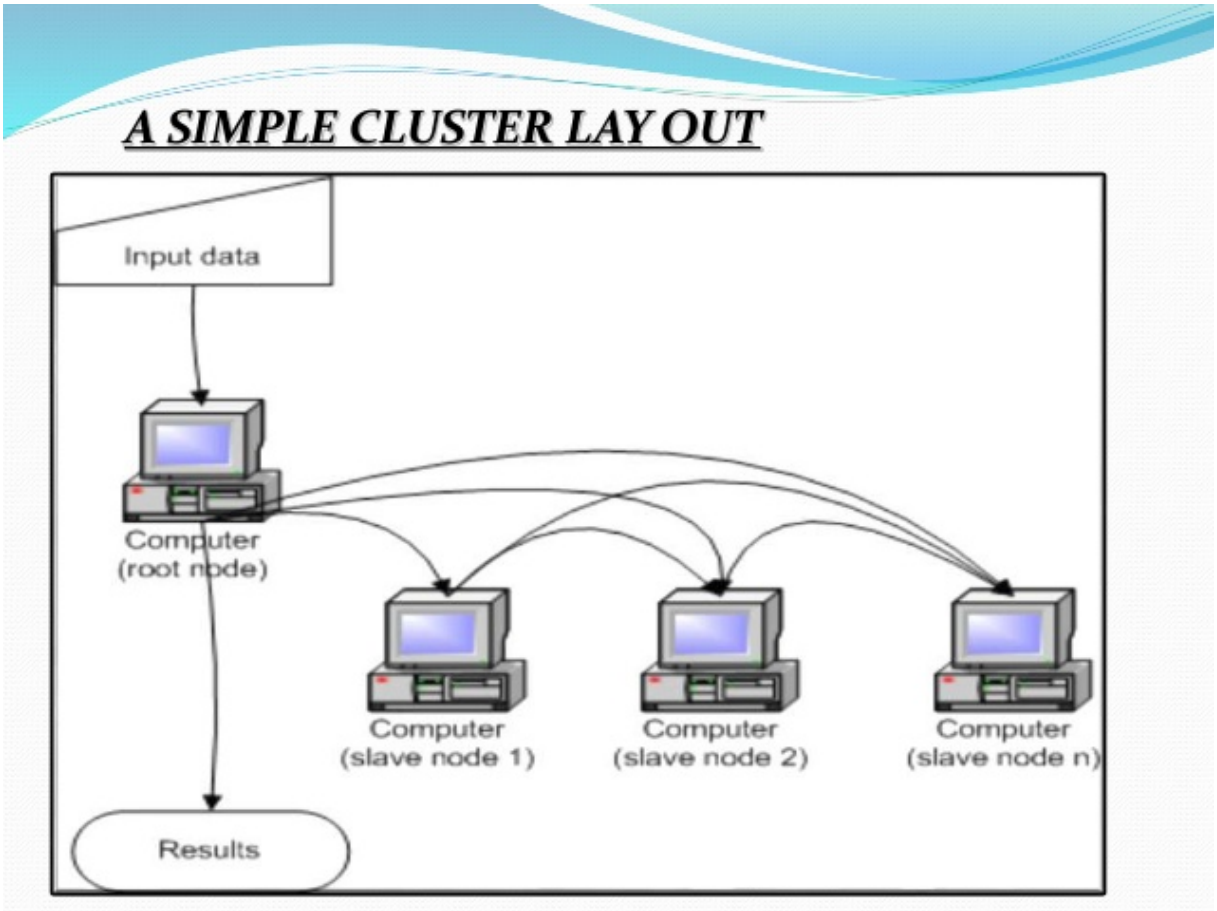

Figura 2.1: Um exemplo simples de uma computação executada em um *cluster*

final.

2.2 Programação Paralela em CPU

Uma das formas de se programar aplicações paralelas para tirar proveito da existência de múltiplas CPUs consiste na criação de múltiplos processos ou múltiplas *threads* para executar simultaneamente o mesmo programa. Cada um desses recebe parte dos dados para computar, assim reduzindo o tempo de execução do programa. Na literatura são descritas distintas ferramentas para desenvolver aplicações paralelas, cada qual apropriada para uma das diversas arquiteturas paralelas existentes. As utilizadas neste trabalho serão o MPI e o CUDA, que serão explicados na próxima seção.

2.2.1 MPI

Uma aplicação que utiliza múltiplos processos geralmente precisa que eles se comuniquem de alguma forma para juntar os resultados parciais em um final, para propósito de sincronização, entre outros. A fim de se padronizar esta troca de mensagens entre processos, um grupo formado por acadêmicos e industriais desenvolveu o padrão MPI (*message-passing interface*).

O MPI não é uma linguagem de programação em si. É um conjunto de rotinas de uma biblioteca implementada pelos fabricantes, possibilitando otimização e portabilidade entre diferentes arquiteturas (Wilkinson et al, 2005). Hoje, é possível utilizá-lo em diversas linguagens, como C, C++ e FORTRAN (Barros et al, 2013).

Há múltiplas implementações da especificação MPI. Uma das mais conhecidas na literatura é o MPICH (Gropp et al, 1996). Além de prover a portabilidade prometida pela especificação, seus autores argumentam que o MPICH mantém alto desempenho em diferentes arquiteturas: foi mostrado que o MPICH foi capaz de oferecer desempenho similar aos obtidos pelas bibliotecas específicas dos fabricantes em arquiteturas MPPs (do inglês, *massively parallel processing*), SMP (do inglês, *symmetric multiprocessing*) e em *clusters*(Gropp et al, 1996).

Como em qualquer protocolo de comunicação, há de haver ao menos uma função de envio e de recebimento de mensagens. No MPI, estas funções são *MPI_Send* e *MPI_Recv*, respectivamente. Além disso, o MPI possibilita a comunicação entre um determinado grupo de processadores ao utilizar o conceito de comunicador, sendo o comunicador global chamado de *MPI_COMM_WORLD*. Na especificação, ainda há métodos visando a otimização de comunicações intensas, como as que simultaneamente trocam muitos dados entre muitos processadores, como ocorre nas funções *MPI_Gather* e *MPI_Scatter*.

A aplicação estudada neste trabalho, quando usada com múltiplos processadores, utiliza MPI para comunicação dos valores parciais e finais das EDPs calculadas.

2.3 Programação paralela em GPU

GPUs são placas que realizam o processamento gráfico de um computador. Devido à quantidade maior de unidades de processamento, as chamadas unidades lógico-aritméticas (do inglês ALUs - *arithmetic-logic units*), as GPUs geralmente possuem um poder de processamento superior ao das CPUs. Este fato possibilita uma maior paralelização na execução, que, dependendo das características da aplicação, pode resultar em um ganho significativo no desempenho.

2.3.1 CUDA

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

CUDA é uma extensão da linguagem C, que permite ao programador definir qual pedaço do código será executado na CPU (chamado em CUDA de *host*) e qual será executado na GPU (chamado de *device*). Em CUDA geralmente o *host* dispara tarefas (chamadas de *kernel*) que utilizarão várias unidades de processamento da GPU em paralelo. Cada unidade de processamento da GPU executará instruções de um código para um determinado conjunto de dados. Para executar um *kernel*, o programador deve definir, além de seus parâmetros, quantas *threads* CUDA serão criadas para sua execução (Cook et al, 2012). Por exemplo, a instrução **kernel_function<<<nBlocks,nThreads>>>(args);** chama a função **kernel_function**, criando **nBlocks** blocos, chamados de *grids*, onde cada bloco possui **nThreads** *threads*. Assim a função será executada em paralelo por um total de **nBlocks** $\times$ **nThreads** *threads*. O parâmetro **args** se refere aos argumentos ordinários da função **kernel_function**.

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

dependendo das características do *hardware* e quantidade de dados transmitidos.

A hierarquia de memória utilizada pelas GPUs com CUDA é apresentada na figura 2.3.1. Ela consta de diferentes tipos de memória que serão explicados abaixo e podem ser usados para otimizações dependendo dos tipos de acesso.

Há a memória global, que possui a maior latência e comporta mais dados. Ela pode ser acessada por qualquer *thread* do *grid* da GPU. É nela que os vetores passados pela CPU ficam alocados e são acessados.

Já a memória local representada na figura é privativa de cada *thread*. No entanto, ela está contida na memória global e, consequentemente, também possui alta latência e não é ideal para otimizações de acesso de memória.

A memória de constantes é uma memória que somente permite leitura internamente na GPU e pode ser acessada por todas as *threads*. Ela é comumente usada quando as execuções envolvem um grande conjunto de valores constantes sendo acessados simultaneamente por múltiplas *threads*, caso onde ela apresenta uma melhor latência do que a memória global por implementar um mecanismo semelhante a uma operação de *broadcast*.

A memória compartilhada da GPU, como o próprio nome diz, é usada para compartilhar dados entre *threads* de um mesmo bloco. Ela tem uma latência menor quando comparada às memórias citadas acima mas, por outro lado, tem tamanho limitado (na casa de 48KB por bloco para GPUs com *compute capability* 2.0) e muitas das vezes insuficiente para ser usada largamente.

Cada *thread* no *grid* possui um conjunto de registradores, que são alocados na pilha de execução do algoritmo. Estes registradores oferecem a menor latência possível para algoritmos executados na GPU e devem ser usados ao máximo a fim de otimizar o tempo de acesso. No entanto, os registradores são ainda mais limitados: há apenas 64 registradores em GPUs mais novas (com *compute capability 6.0*).

2.4 Trabalhos relacionados

Em um trabalho anterior, Barros et al (2013) apresentaram a implementação e melhoria do desempenho de um simulador cardíaco através do uso de placas gráficas para realizar parte da computação.

Na implementação descrita no trabalho, as CPUs foram responsáveis pelo cálculo das EDPs do modelo computacional, usando a biblioteca PETSC, e as GPUs foram as responsáveis por resolver o sistema de EDOs. Desta maneira, o autor obteve *speedups* de 10 vezes na solução da EDP e de 450 vezes na soluções dos sistemas não lineares de EDOs, ao aproveitar o paralelismo nas execuções de *kernels*.

Segundo o trabalho, mesmo um computador moderno equipado com 64 núcleos de processamento não supera uma única GPU, devido às suas múltiplas ALUs, como apresentado na seção 2.3. O poder computacional da GPU, quando atrelado ao fato de que 90% do tempo de execução mensurado foi gasto resolvendo EDOs, motivam a implementação e otimização de GPUs na solução do modelo microscópico paralelo apresentado.

No trabalho, a implementação baseada em *cluster* utilizando somente CPUs foi bem sucedida, obtendo *speedups* quase lineares (61 com 64 núcleos). Porém, mesmo após esta melhora, o tempo total de execução continuou elevado. Por outro lado, a implementação que utiliza múltiplas CPUs e GPUs obteve um sucesso ainda maior, diminuindo o tempo de execução de 6 dias (usando um núcleo de processamento) para 21 minutos (utilizando a plataforma multi-GPU de 8 nós). Ao comparar ambas as abordagens, foi observado que a implementação multi-GPU foi quase 7 vezes mais rápida que a implementação baseada em *clusters*, quando executadas em 8 computadores.

Dando continuidade ao trabalho apresentado anteriormente, Cordeiro et al (2017) otimizaram ainda mais o simulador cardíaco usando agrupamento de dados para minimizar o custo incorrido às GPUs nas múltiplas chamadas de *kernels* pelos diversos processos CPUs criados para a execução paralela do simulador.

Seu trabalho mostrou que, de fato, quando há mais processos executando em CPUs do que o número de GPUs disponíveis na máquina (o que ocorre comumente), as *kernels* invocadas a partir das CPUs podem impor uma sobrecarga às placas gráficas, prejudicando o tempo de execução do simulador. Arquiteturas atuais já possuem tecnologia capaz de evitar tal sobrecarga, chamada de *Hyper-Q*, porém estas arquiteturas podem não ser muito acessíveis a instituições e pesquisadores, além do que pode existir um parque composto de GPUs antigas que deve ser de alguma forma melhor aproveitado.

Ainda em seu trabalho, o autor apresenta uma versão modificada do simulador

que evita tal sobrecarga. Esta versão, como já dito, divide os processos em CPU em n grupos, onde n é o número de GPUs disponíveis na máquina, e nomeia um processo em cada grupo para agrupar os dados dos demais processos do grupo, resolver as EDOs na GPU e retornar os valores computados para os demais processos.

A versão modificada foi testada em 3 arquiteturas diferentes e foi capaz de reduzir o tempo total de execução em até 25%. Inclusive, a arquitetura *Hyper-Q* foi usada em uma destas arquiteturas testadas e os resultados entre a versão original e a modificada foram os mesmos, indicando que a implementação do agrupamento de dados não requer um tempo significativo de computação.

Esta versão modificada será usada neste trabalho como base para a aplicação das otimizações apresentadas no próximo capítulo.

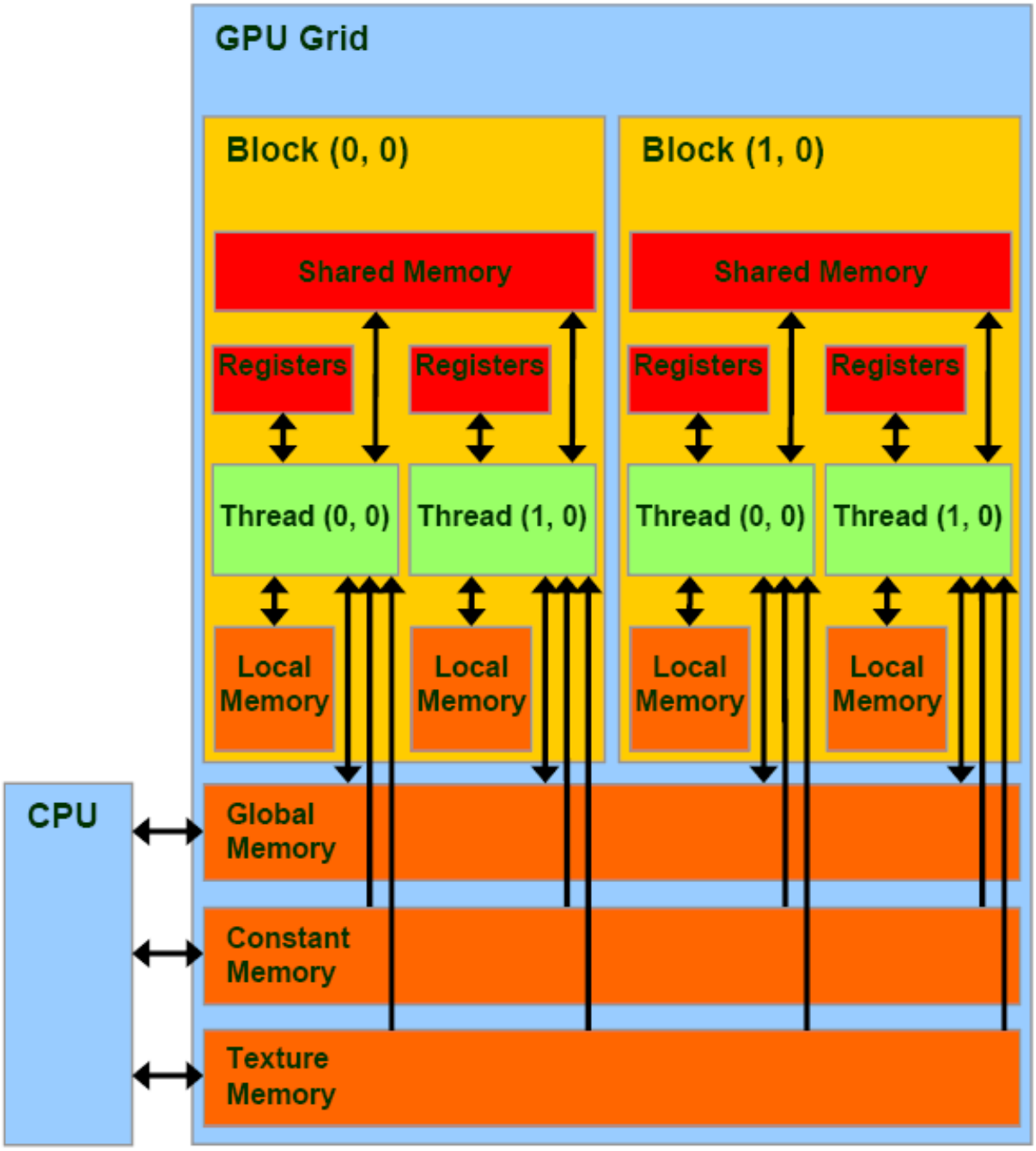

Figura 2.2: Arquitetura de memória utilizada pelas *kernels* do CUDA

3 Otimizações

Neste capítulo, serão apresentadas as otimizações implementadas no simulador cardíaco com o objetivo de reduzir o seu tempo total de computação. Especificamente, são propostas modificações em parte do código que resolve as EDOs do modelo e que, conforme descrito ao longo do Capítulo 1, é executada pela GPU.

3.1 Redução do Número de Variáveis

A versão original do simulador cardíaco declara inúmeras variáveis no código do *kernel* responsáveis pela solução das EDOs, conforme apresentado no Algoritmo 1. Basicamente, uma variável é declarada e utilizada em apenas um passo intermediário da computação das EDOs. Esta implementação estourava o limite de registradores disponíveis por *thread*, forçando-a a acessar sua memória local durante a realização dos cálculos.

A fim de diminuir o uso de memória e a latência nos acessos à memória da GPU na execução da *kernel*, foi declarado apenas um reduzido conjunto de variáveis auxiliares que são reutilizadas durante todo o processamento das EDOs, conforme ilustrado no Algoritmo 2. Esta diminuição foi feita no intuito de que todas as variáveis auxiliares pudessem residir nos registradores das GPUs, evitando o uso da memória global que, como apresentado anteriormente, tem uma maior latência de acesso.

Como podemos ver no fragmento de código apresentado no Algoritmo 1, o código original utilizava uma variável para cada constante, cada cálculo intermediário e cada cálculo final, demandando um total de 154 variáveis locais. A modificação proposta nesta seção (Algoritmo 2) diminui este número para 32 variáveis auxiliares, buscando reorganizar as operações de modo a reler o mínimo de informação possível na memória. Ao se reduzir o total de variáveis utilizadas, reduz-se o uso da técnica de *register spilling*, empregada automaticamente pelo compilador quando o número de registradores disponíveis é inferior ao número de variáveis utilizadas no programa, e que consiste da alocação em registradores apenas das variáveis que são acessadas em um determinado momento, man-

Algoritmo 1: Trecho de código original, antes da redução no número de variáveis.

```
// antes da refatoracao
// SOLVE RHS
inline __device__ void RHS_gpu (Real *sv_, Real *rDY_, Real
   Istim, Real time, Real _stim_start, Real _stim_dur, int
   threadID_, size_t pitch_, Real dt) {

        Real V_old_ = *((Real*)((char*)sv_ + pitch_ * 0) +
           threadID_);
        Real Cai_old_ = *((Real*)((char*)sv_ + pitch_ * 1) +
           threadID_);
        Real Cass_old_ = *((Real*)((char*)sv_ + pitch_ * 2) +
           threadID_);
        Real CaJSR_old_ = *((Real*)((char*)sv_ + pitch_ * 3) +
           threadID_);
        Real CaNSR_old_ = *((Real*)((char*)sv_ + pitch_ * 4) +
           threadID_);
        // declarar todas as 41 variaveis de estado...

        Real calc_i_stim = ((time>=_stim_start) && (time<=
           _stim_start+_stim_dur)) ? Istim: 0.0;
        Real calc_Bi = pow((1.0+((CMDN_tot*Km_CMDN)/pow((Km_CMDN
           +Cai_old_),2.0))),(-1.0));
        Real calc_Bss = pow((1.0+((CMDN_tot*Km_CMDN)/pow((
           Km_CMDN+Cass_old_),2.0))),(-1.0));
        Real calc_BJSR = pow((1.0+((CSQN_tot*Km_CSQN)/pow((
           Km_CSQN+CaJSR_old_),2.0))),(-1.0));
        Real calc_J_rel = (v1*(P_O1_old_+P_O2_old_)*(CaJSR_old_-
           Cass_old_)*P_RyR_old_);
        // calcular valores intermediarios (72 calculos)...

        Real d_dt_V = (-(calc_i_CaL+calc_i_pCa+calc_i_NaCa+
           calc_i_Cab+calc_i_Na+calc_i_Nab+calc_i_NaK+
           calc_i_Kto_f+calc_i_Kto_s+calc_i_K1+calc_i_Ks+
           calc_i_Kur+calc_i_Kss+calc_i_Kr+calc_i_ClCa+
           calc_i_stim));
        Real d_dt_Cai = (calc_Bi*((calc_J_leak+calc_J_xfer)-(
           calc_J_up+calc_J_trpn+((((calc_i_Cab+calc_i_pCa)
           -(2.0*calc_i_NaCa))*Acap*Cm)/(2.0*Vmyo*F)))));
        Real d_dt_Cass = (calc_Bss*(((calc_J_rel*VJSR)/Vss)-(((
           calc_J_xfer*Vmyo)/Vss)+((calc_i_CaL*Acap*Cm)/(2.0*Vss
           *F)))));
        Real d_dt_CaJSR = (calc_BJSR*(calc_J_tr-calc_J_rel));
        Real d_dt_CaNSR = ((((calc_J_up-calc_J_leak)*Vmyo)/VNSR)
           -((calc_J_tr*VJSR)/VNSR));
        // calcular EDOs e retornar os 41 novos valores
}
```

Algoritmo 2: Trecho do código modificado para reduzir o número de variáveis.

```
// apos refatoracao
// SOLVE RHS
inline __device__ void RHS_gpu (Real *sv_, Real *rDY_, Real
   Istim, Real time, Real _stim_start, Real _stim_dur, int
   threadID_, size_t pitch_, Real dt) {

    Real aux0, aux1, aux2, aux3, aux4, aux5, aux6, aux7, aux8,
       aux9, aux10, aux11, aux12, aux13, aux14, aux15, aux16;
    Real aux17, aux18, aux19, aux20, aux21, aux22, aux23, aux24,
        aux25, aux26, aux27, aux28, aux29, aux30, aux31;

    aux0 = *((Real *) ((char *) sv_) + threadID_);
    aux1 = *((Real *) ((char *) sv_ + pitch_ * 11) + threadID_);
    aux2 = g_CaL * aux1 * (aux0 - E_CaL);
    aux3 = *((Real *) ((char *) sv_ + pitch_) + threadID_);
    aux4 = (i_pCa_max * pow(aux3, 2.0)) / (pow(Km_pCa, 2.0) +
       pow(aux3, 2.0));
    aux5 = *((Real *) ((char *) sv_ + pitch_ * 18) + threadID_);
    // auxiliares sao usados para constantes e calculos
       intermediarios. Novos valores vao direto no vetor
       auxiliar
}
```

tendo as demais na memória global, o que aumenta a latência de acesso quando um novo acesso às variáveis em memória faz-se necessário.

Vale ressaltar que 32 variáveis foram instanciadas separadamente pois vetores são automaticamente alocados na memória global, o que aumenta a latência de acesso e, portanto, foge do objetivo de otimização aqui proposto.

3.2 Eliminação do Vetor Auxiliar

No código original da aplicação, um vetor auxiliar de 41 posições é usado para guardar os valores computados na resolução das EDOs por cada *thread* na GPU. Cada posição do vetor corresponde a uma variável do sistema de EDOs. Como vetores são alocados por padrão na memória privativa global (memória local) de cada *thread*, esta implementação gerava 82 acessos desnecessários à memória quando da atualização do vetor global (41 de leituras no vetor auxiliar e 41 de escritas no vetor global).

Como vimos na seção 2.3, este tipo de acesso, dentro da arquitetura, não é a princípio a melhor escolha em termos de desempenho. Portanto, espera-se que diminuindo sua ocorrência acarrete na diminuição do tempo gasto no acesso à memória.

A fim de reduzir estes acessos, a implementação foi modificada de forma a não utilizar um vetor auxiliar, escrevendo os valores diretamente no vetor final, contribuindo adicionalmente para uma redução do espaço de memória.

Conforme pode-se observar no Algoritmo 3, o emprego do vetor auxiliar implica em um custo adicional em termos de memória e de execução. Em relação aos custos de memória, cada *thread* possui uma cópia do vetor guardado na memória global. Os acessos a memória para ler, de cada uma das 41 posições do vetor, dois valores da memória, somá-los e armazenar o resultado na memória, quando executados milhares de vezes, podem tornar os custos de execução significativos.

Algoritmo 3: Trecho do código original, antes da eliminação do vetor auxiliar.

```
// antes da refatoracao
// The Forward Euler Method
inline __device__ void solve_Forward_Euler_gpu(Real *sv, Real dt
    , Real stim, Real t, Real _stim_start, Real _stim_dur, int
    threadID, size_t pitch) {

    // declara e inicializa o vetor
        Real rDY[NUMBER_EQUATIONS_CELL_MODEL];
        for(int i = 0; i < NUMBER_EQUATIONS_CELL_MODEL; i++) rDY
            [i] = 0;

    // passa o vetor
        RHS_gpu(sv, rDY, stim, t, _stim_start, _stim_dur,
            threadID, pitch, dt);

    // itera e guarda os resultados na memoria
        for(int i = 0; i < NUMBER_EQUATIONS_CELL_MODEL; i++)
                *((Real*)((char*)sv + pitch * i) + threadID) =
                    dt*rDY[i] + *((Real*)((char*)sv + pitch * i)
                    + threadID);
}

// SOLVE RHS
inline __device__ void RHS_gpu(Real *sv_, Real *rDY_, Real Istim
    , Real time, Real _stim_start, Real _stim_dur, int threadID_,
     size_t pitch_, Real dt) {

        Real d_dt_V = (-(calc_i_CaL+calc_i_pCa+calc_i_NaCa+
            calc_i_Cab+calc_i_Na+calc_i_Nab+calc_i_NaK+
            calc_i_Kto_f+calc_i_Kto_s+calc_i_K1+calc_i_Ks+
            calc_i_Kur+calc_i_Kss+calc_i_Kr+calc_i_ClCa+
            calc_i_stim));
        Real d_dt_Cai = (calc_Bi*((calc_J_leak+calc_J_xfer)-(
            calc_J_up+calc_J_trpn+((((calc_i_Cab+calc_i_pCa)
            -(2.0*calc_i_NaCa))*Acap*Cm)/(2.0*Vmyo*F)))));
        Real d_dt_Cass = (calc_Bss*(((calc_J_rel*VJSR)/Vss)-(((
            calc_J_xfer*Vmyo)/Vss)+((calc_i_CaL*Acap*Cm)/(2.0*Vss
            *F)))));
        Real d_dt_CaJSR = (calc_BJSR*(calc_J_tr-calc_J_rel));
        Real d_dt_CaNSR = ((((calc_J_up-calc_J_leak)*Vmyo)/VNSR)
            -((calc_J_tr*VJSR)/VNSR));
        // calcular EDOs e retornar os 41 novos valores...

        rDY_[0] = d_dt_V;
        rDY_[1] = d_dt_Cai;
        rDY_[2] = d_dt_Cass;
        rDY_[3] = d_dt_CaJSR;
        rDY_[4] = d_dt_CaNSR;
        rDY_[5] = d_dt_P_RyR;
        // guardar os valores calculados no vetor...
}
```

Algoritmo 4: Trecho do código modificado, após a eliminação do vetor auxiliar.

```
// apos refatoracao
// The Forward Euler Method
inline __device__ void solve_Forward_Euler_gpu(Real *sv, Real dt
    , Real stim, Real t, Real _stim_start, Real _stim_dur, int
    threadID, size_t pitch) {

        RHS_gpu(sv, stim, t, _stim_start, _stim_dur, threadID,
            pitch, dt);
}


// SOLVE RHS
inline __device__ void RHS_gpu(Real *sv_, Real Istim, Real time,
     Real _stim_start, Real _stim_dur, int threadID_, size_t
    pitch_, Real dt) {

        Real d_dt_V = (-(calc_i_CaL+calc_i_pCa+calc_i_NaCa+
            calc_i_Cab+calc_i_Na+calc_i_Nab+calc_i_NaK+
            calc_i_Kto_f+calc_i_Kto_s+calc_i_K1+calc_i_Ks+
            calc_i_Kur+calc_i_Kss+calc_i_Kr+calc_i_ClCa+
            calc_i_stim));
        Real d_dt_Cai = (calc_Bi*((calc_J_leak+calc_J_xfer)-(
            calc_J_up+calc_J_trpn+((((calc_i_Cab+calc_i_pCa)
            -(2.0*calc_i_NaCa))*Acap*Cm)/(2.0*Vmyo*F)))));
        Real d_dt_Cass = (calc_Bss*(((calc_J_rel*VJSR)/Vss)-(((
            calc_J_xfer*Vmyo)/Vss)+((calc_i_CaL*Acap*Cm)/(2.0*Vss
            *F)))));
        Real d_dt_CaJSR = (calc_BJSR*(calc_J_tr-calc_J_rel));
        Real d_dt_CaNSR = ((((calc_J_up-calc_J_leak)*Vmyo)/VNSR)
            -((calc_J_tr*VJSR)/VNSR));
        // calcular EDOs e retornar os 41 novos valores...

    *((Real*)((char*)sv_) + threadID_) = V_old_ + dt * d_dt_V;
    *((Real*)((char*)sv_ + pitch_) + threadID_) = Cai_old_ + dt
        * d_dt_Cai;
    *((Real*)((char*)sv_ + pitch_ * 2) + threadID_) = Cass_old_
        + dt * d_dt_Cass;
    *((Real*)((char*)sv_ + pitch_ * 3) + threadID_) = CaJSR_old_
        + dt * d_dt_CaJSR;
    *((Real*)((char*)sv_ + pitch_ * 4) + threadID_) = CaNSR_old_
        + dt * d_dt_CaNSR;
    *((Real*)((char*)sv_ + pitch_ * 5) + threadID_) = P_RyR_old_
        + dt * d_dt_P_RyR;
    // guarda os valores na memoria diretamente
```

3.3 Declaração de Valores Constantes na Memória de Constantes

Na implementação original da aplicação, muitas constantes eram usadas no cálculo das equações do modelo. Essas constantes eram declaradas no *kernel* com o uso do modificador **__constant__** de CUDA. No entanto, esta forma de declaração faz com que a variável, apesar de constante, seja alocada na memória local da *thread* que, como dito anteriormente, reside na memória global e, portanto, tem uma latência maior.

Uma otimização utilizada neste trabalho foi usar a memória de constantes da GPU para a alocação de tais constantes. Quando valores constantes são lidos simultaneamente a partir da memória de constantes por múltiplas *threads*, o seu tempo de acesso tende a ser menor.

O Algoritmo 5 mostra que a versão original usava valores definidos com o modificador **__constant__**, mas não são alocados na memória de constantes pois ainda é necessário realizar a chamada à função $cudaMemcpyToSymbol()$.

Já a versão modificada, apresentada no Algoritmo 6, faz a declaração correta. Foi criada uma função especifica para realizar a alocação na memória de constantes ($initCudaConstants()$), que deve ser invocada uma vez no começo da execução da aplicação.

Desta forma, como cada *thread* tende a acessar as 66 constantes dentro dos 72 cálculos intermediários, usar a memória de constantes, que é somente leitura (do inglês, *read only*), tende a melhorar a latência pois não há necessidade de por os acessos em fila no barramento, o que normalmente aconteceria na memória global.

Algoritmo 5: Código original da aplicação, antes da declaração dos valores constantes na memória de constantes.

```
// antes da refatoracao

__constant__ Real Acap = 1.534e-4;
__constant__ Real Cm = 1;
__constant__ Real Vmyo = 25.84e-6;
__constant__ Real F = 96.5;
__constant__ Real VJSR = 0.12e-6;
// declarar as 66 constantes utilizadas

// SOLVE RHS
inline __device__ void RHS_gpu(Real *sv_, Real *rDY_, Real Istim
    , Real time, Real _stim_start, Real _stim_dur, int threadID_,
     size_t pitch_, Real dt) {

        Real calc_i_stim = ((time>=_stim_start) && (time<=
            _stim_start+_stim_dur)) ? Istim: 0.0;
        Real calc_Bi = pow((1.0+((CMDN_tot*Km_CMDN)/pow((Km_CMDN
            +Cai_old_),2.0))),(-1.0));
        Real calc_Bss = pow((1.0+((CMDN_tot*Km_CMDN)/pow((
            Km_CMDN+Cass_old_),2.0))),(-1.0));
        Real calc_BJSR = pow((1.0+((CSQN_tot*Km_CSQN)/pow((
            Km_CSQN+CaJSR_old_),2.0))),(-1.0));
        Real calc_J_rel = (v1*(P_O1_old_+P_O2_old_)*(CaJSR_old_-
            Cass_old_)*P_RyR_old_);
        // constantes sao acessadas nos calculos intermediarios
            ...
}
```

Algoritmo 6: Código da aplicação modificado, após a declaração dos valores constantes na memória de constantes.

```
// apos refatoracao

__constant__ Real constants[66];

void initCudaConstants(){
    // Parameters
    Real localConstants[66];
    localConstants[0] = 1.534e-4;      // Acap
    localConstants[1] = 1;      // Cm
    localConstants[2] = 25.84e-6;      // Vmyo
    localConstants[3] = 96.5;      // F
    localConstants[4] = 0.12e-6;      // VJSR
    // preencher o vetor localConstants com os valores das
       constantes...

    // copiar o vetor para a memoria de constantes da GPU
    cudaMemcpyToSymbol(constants, localConstants, sizeof(Real) *
        66);
}

// SOLVE RHS
inline __device__ void RHS_gpu(Real *sv_, Real *rDY_, Real Istim
   , Real time, Real _stim_start, Real _stim_dur, int threadID_,
    size_t pitch_, Real dt) {

        Real calc_i_stim = ((time>=_stim_start)&&(time<=
           _stim_start+_stim_dur)) ? Istim: 0.0;
        Real calc_Bi = pow((1.0+((constants[7]*constants[8])/pow
           ((constants[8]+Cai_old_),2.0))),(-1.0));
        Real calc_Bss = pow((1.0+((constants[7]*constants[8])/
           pow((constants[8]+Cass_old_),2.0))),(-1.0));
        Real calc_BJSR = pow((1.0+((constants[9]*constants[10])/
           pow((constants[10]+CaJSR_old_),2.0))),(-1.0));
        Real calc_J_rel = (constants[11]*(P_O1_old_+P_O2_old_)*(
           CaJSR_old_-Cass_old_)*P_RyR_old_);
    // vetor constants eh usado em todos os calculos
       intermediarios
}
```

4 Experimentos

Neste capítulo, serão apresentados os ambiente computacional usado nos experimentos, bem como os resultados obtidos.

4.1 Ambiente computacional

O simulador em sua versão original, bem como todas as suas variantes obtidas pela incorporação das otimizações descritas no capítulo anterior, foram executadas em duas máquinas, ambas executando o sistema operacional Linux 2.6.32 e equipadas com dois processadores *quad-core* Intel E5620, com 12 GB de memória. Uma das máquinas possui duas GPUs Tesla M2050, com capacidade CUDA 2.0 e 448 *cores* CUDA. A outra possui duas GPUs Tesla C1060, com capacidade CUDA 1.3 e 240 *cores* CUDA. Foram ainda utilizados a versão 3.3 *patch* 7 do PETSC, driver CUDA versão 6.5 e MPICH versão 1.4.1. Os códigos em C foram compilados com o compilador g++, versão 4.7.2, e os códigos CUDA C foram compilados pelo nvcc, versão 6.5.12, com as *flags arch=compute_13* e *code=compute_13*.

As distintas versões do simulador foram submetidas para execução nas máquinas através de um sistema de fila, e suas execuções ocorreram em modo exclusivo.

Os mesmos parâmetros foram utilizados para executar todas as versões do simulador em uma malha de 0.5cm × 0.5cm. Tais quais os parâmetros usados no trabalho original (Barros et al, 2013), os parâmetros definidos na execução desta aplicação são: $\sigma_m = 0.0$, $\sigma_c = 0.4\mu S/\mu m$, $G_p = 0.5\mu S$, $G_i = 0.33\mu S$, $G_c = 0.062\mu S$, $\beta = 0.14cm^{-1}$, $C_m = 1.0\mu F/cm^2$, $\Delta t_p = 0.01ms$, $\Delta t_o = 0.0001ms$. A quantidade de passos de tempo foi definida como 10.

Versão	EDP	EDO	Total
Original	1599.62	1170.57	2776.17
Variáveis	1588.81	1218.95 (**+3,5%**)	2829.43
Vetor Auxiliar	1604.37	1133.84 (**-4%**)	2749.01
Constantes	1582.69	1170.9	2764.49
Todas	1579.91	1141.78 (**-2,5%**)	2732.12

Tabela 4.1: Tempos médios de execução, onde EDP é a média dos tempos de resolução das EDPs, feitas pela CPU; EDO é a média dos tempos de resolução das EDOs, realizadas pela GPU, e Total é a média dos tempos totais de execução.

4.2 Resultados

Os resultados obtidos mostraram que, de fato, há melhora ao utilizar algumas das otimizações descritas no capítulo anterior. As médias aritméticas dos tempos coletados se encontram na tabela 4.2. Todas as versões utilizaram 16 CPUs e 4 GPUs em sua execução, foram executadas, ao menos, 5 vezes e todos os desvios padrões ficaram abaixo de 1%.

Como pode ser observado na tabela, a versão que retira o vetor auxiliar apresentou uma melhora de, aproximadamente, 3.5% no tempo de resolução das EDOs, computação esta feita nas GPUs. Esta melhora serve como indicativo do quanto o acesso à memória dentro da GPU consome recursos computacionais pois, como explicado na seção 3.2, foram removidos 82 acessos à memória global por *thread* CUDA.

Já a versão que utiliza a memória de constantes não apresentou mudanças no tempo de execução das EDOs. Uma possível explicação para que o tempo tenha se mantido constante é que a aplicação aqui apresentada pode não apresentar um padrão de acesso às constantes intenso o suficiente para que o uso da memória de constantes gere algum ganho de desempenho, pois, como visto na seção 2.3.1, a memória de constantes tem latência similar à memória global.

Por outro lado, a versão que reduz o número de variáveis apresentou uma piora nos tempos de execução de, aproximadamente, 4%. Esperava-se uma melhoria pois a redução do número de variáveis foi considerável (154 para 32). No entanto, ela apresentou o pior resultado, mesmo confirmando-se que o número de operações de *register spilling* foi reduzido. A confirmação foi obtida pela compilação de ambos os códigos (versão original e versão com número de variáveis reduzidas) com a *flag* --ptxas-options=-v".

Por fim, a versão que usa todas as otimizações simultaneamente também apresen-

tou melhora nos tempos de execução das EDOs, de aproximadamente, 2.5%. Observa-se que o uso conjunto das otimizações mitigou a perda de desempenho decorrente do uso de um número reduzido de variáveis.

5 Conclusão

Simuladores cardíacos são ferramentas que podem ser utilizadas tanto no estudo de doenças quanto no desenvolvimento de novos tratamentos para as mesmas. No entanto, como a complexidade dos processos biofísicos envolvendo o coração é refletida nos modelos matemáticos utilizados nos simuladores, resolver estes serialmente se torna uma tarefa computacionalmente custosa: como o simulador discretiza o tecido cardíaco em milhões de pedaços microscópicos e calcula várias equações para cada um destes, milhões de equações deverão resolvidas a cada passo de tempo. Este processo pode se beneficiar de ferramentas de computação paralela, o que já foi mostrado em trabalhos anteriores (Barros et al, 2013; Cordeiro et al, 2017).

O objetivo deste trabalho foi avaliar o impacto de distintas otimizações no desempenho da versão paralela deste simulador cardíaco, implementada em um ambiente multi-GPU (mais especificamente, em um *cluster* de computadores), utilizando em sua codificação MPI, CUDA e PETSC.

As otimizações aqui avaliadas foram: redução do número de variáveis para que coubessem nos registradores de uma *thread*, evitando acessos caros de memória; remoção de um vetor auxiliar que realizava acessos desnecessários à memória global da GPU; e uso da memória de constantes da GPU para guardar as constantes utilizadas nos cálculos executados pela mesma.

O trabalho mostrou que modificações no código visando diminuir tanto a latência quanto o numero de acessos à memória podem levar à resultados satisfatórios, como foi o caso da otimização que removeu o vetor auxiliar, diminuindo a quantidade de acessos à memória global e melhorando em 3.5% o desempenho da aplicação.

Apesar de não terem sido verificados ganhos de desempenho em todas as versões do simulador que aplicam as otimizações descritas, este trabalho cumpre os objetivos propostos pois as otimizações foram aplicadas no código do simulador e seus resultados foram comparados sob as mesmas configurações de hardware e sob os mesmos parâmetros durante suas execuções.

Para trabalhos futuros, pode ser feita uma análise mais detalhada, com o uso de ferramentas de *profile*, dos motivos que levaram: a) a piora no desempenho quando foi aplicada a otimização que reduz o número de variáveis, b) a manutenção do tempo de execução ao utilizar a otimização que emprega a memória de constantes da GPU, e c) a melhoraria no desempenho na versão que aplica todas as modificações. Além disso, também seria interessante adaptar o código para utilizar as novas funcionalidades da última versão do CUDA (atualmente, o CUDA 9), estudar outras maneiras de melhorar o desempenho dentro da aplicação, como estudar o uso de outros tipos de memória ainda não utilizados (*shared memory* e *texture memory*, por exemplo).

Bibliografia

Amorim, R. M.; Haase, G.; Liebmann, M. ; dos Santos, R. W. **Comparing cuda and opengl implementations for a jacobi iteration**. In: High Performance Computing Simulation, 2009. International Conference on, p. 22 –32, 2009.

Amorim, R. M.; Rocha, B. M.; Campos, F. O. ; dos Santos, R. W. **Automatic code generation for solvers of cardiac cellular membrane dynamics in gpus**. In: Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE, p. 2666–2669, 2010.

Amorim, R. M.; dos Santos, R. W. Solving the cardiac bidomain equations using graphics processing units. **Journal of Computational Science**, 2012.

Barros, B. G. d.; others. Simulações computacionais de arritmias cardíacas em ambientes de computação de alto desempenho do tipo multi-gpu. 2013.

Benjamin, E. J.; Blaha, M. J.; Chiuve, S. E.; Cushman, M.; Das, S. R.; Deo, R.; de Ferranti, S. D.; Floyd, J.; Fornage, M.; Gillespie, C. ; others. Heart disease and stroke statistics—2017 update: a report from the american heart association. **Circulation**, v.135, n.10, p. e146–e603, 2017.

Biktashev, V.; Holden, A. Re-entrant activity and its control in a model of mammalian ventricular tissue. **Proc. of the Royal Soc. of London. Series B: Biological Sciences**, v.263, n.1375, p. 1373–1382, 1996.

Campos, F. O.; Wiener, T.; Prassl, A. J.; Weber dos Santos, R.; Sanchez-Quintana, D.; Ahammer, H.; Plank, G. ; Hofer, E. Electroanatomical characterization of atrial microfibrosis in a histologically detailed computer model. **Biomedical Engineering, IEEE Transactions on**, v.60, n.8, p. 2339–2349, 2013.

Cook, S. **CUDA programming: a developer's guide to parallel computing with GPUs**. Newnes, 2012.

Cordeiro, R. P.; others. Agrupando dados e kernels de um simulador cardíaco em um ambiente multi-gpu. 2017.

Dos Santos, R. W.; Kosch, O.; Steinhoff, U.; Bauer, S.; Trahms, L. ; Koch, H. Mcg to ecg source differences: measurements and a two-dimensional computer model study. **Journal of electrocardiology**, v.37, p. 123–127, 2004.

dos Santos, R. W.; Plank, G.; Bauer, S. ; Vigmond, E. J. Parallel multigrid preconditioner for the cardiac bidomain model. **IEEE Trans Biomed Eng**, v.51, n.11, p. 1960–8, 2004.

dos Santos, R. W.; Plank, G.; Bauer, S. ; Vigmond, E. J. Preconditioning techniques for the bidomain equations. **Lecture Notes In Computational Science And Engineering**, v.40, p. 571–580, 2004.

dos Santos, R. W.; Campos, F.; Neumann, L.; A., N. ; Giles, W. ATX-II effects on the apparent location of m cells in a computational human left ventricular wedge. **Journal of Cardiovascular Electrophysiology**, v.17, p. S86–S95, 2006.

Gropp, W.; Lusk, E.; Doss, N. ; Skjellum, A. A high-performance, portable implementation of the mpi message passing interface standard. **Parallel computing**, v.22, n.6, p. 789–828, 1996.

Morgan, S.; Plank, G.; Biktasheva, I. ; Biktashev, V. Low energy defibrillation in human cardiac tissue: a simulation study. **Biophysical j.**, v.96, n.4, p. 1364–1373, 2009.

Oliveira, R. S.; Rocha, B. M.; Amorim, R. M.; Campos, F.; Meira, W.; Toledo, E. ; dos Santos, R. W. **Comparing cuda, opencl and opengl implementations of the cardiac monodomain equations**. In: Parallel Processing and Applied Mathematics, volume 7204 de **Lecture Notes in Computer Science**, p. 111–120. Springer Berlin, Heidelberg, 2012.

Pacheco, P. **An introduction to parallel programming**. Elsevier, 2011.

Panfilov, A.; Müller, S.; Zykov, V. ; Keener, J. Elimination of spiral waves in cardiac tissue by multiple electrical shocks. **Physical Review E**, v.61, n.4, p. 4644, 2000.

Plank, G.; Liebmann, M.; dos Santos, R. W.; Vigmond, E. J. ; Haase, G. Algebraic Multigrid Preconditioner for the Cardiac Bidomain Model. **IEEE Trans Biomed Eng**, v.54, p. 585–96, 2007.

Rocha, B. M.; Campos, F. O.; Plank, G.; dos Santos, R. W.; Liebmann, M. ; Haase, G. **Simulations of the electrical activity in the heart with graphic processing units**. In: Parallel Processing and Applied Mathematics, volume 6067 de **Lecture Notes in Computer Science**, p. 439–448. Springer Berlin, Heidelberg, 2010.

Rocha, B. M.; Campos, F. O.; Amorim, R. M.; Plank, G.; Santos, R. W. d.; Liebmann, M. ; Haase, G. Accelerating cardiac excitation spread simulations using graphics processing units. **Concurrency and Computation: Practice and Experience**, v.23, n.7, p. 708–720, 2011.

Wilkinson, B.; Allen, M. Parallel computers. **Programming: Techniques and Applications Using Networked Workstations and Parallel Computers**, p. 3–42, 2005.

Xavier, C.; Oliveira, R.; Vieira, V. F.; dos Santos, R. W. ; Meira, W. Multi-level parallelism for the cardiac bidomain equations. **International Journal of Parallel Programming**, v.37, p. 572–592, 2009.