UNIVERSIDADE DE CAXIAS DO SUL
CENTRO DE COMPUTAÇÃO E TECNOLOGIA DA INFORMAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MAURÍCIO GRISA

GPU Computing: Implementação do método do Gradiente Conjugado utilizando CUDA

Prof. André Luis Martinotto
Orientador

Caxias do Sul, dezembro de 2010

"Insanity: doing the same thing over and over again and expecting different results."
— Albert Einstein

AGRADECIMENTOS

Nesta caminhada de quase sete anos para a obtenção do diploma no curso de Bacharelado em Ciência da Computação, primeiramente gostaria de agradecer a minha família pela educação que tive e pela oportunidade de frequentar uma universidade. Pai, mãe, irmão: sou extremamente grato e amo muito vocês.

Quero agradecer aos meus amigos, especialmente àqueles que convivi na universidade, fazendo trabalhos e estudando para as provas. Sentirei falta deste período que, apesar de fatigante, foi muito proveitoso e divertido. Agradeço também aos professores que foram capazes de transmitir o conhecimento para mim, especialmente àqueles que me ajudaram na confecção deste trabalho. Aqui registro os meus sinceros agradecimentos.

Enfim, agradeço a todos com quem tive a oportunidade de conviver durante a minha vida e que, de alguma forma, foram importantes na minha evolução como homem e como profissional.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

2D	Duas dimensões
3D	Três dimensões
API	Interface de programação de aplicações
CPU	Unidade central de processamento
CSC	Colunas esparsas comprimidas
CSR	Linhas esparsas comprimidas
CUDA	*Compute Unified Device Architecture*
DRAM	*Dynamic random access memory*
EDP	Equações Diferenciais Parciais
G80	*GeForce 8880 serie*
GC	Gradiente Conjugado
GDDR	*Graphics double data rate*
GFLOP	Bilhões de operações de ponto-flutuante por segundo
GMRES	Método do Resíduo Mínimo Generalizado.
GPGPU	*General-Purpose computation on Graphics Processing Units*
GPU	Unidade de processamento gráfica
IEEE	*Institute of Electrical and Eletronics Engineers*
MDF	Métodos de Diferenças Finitas
MEF	Métodos de Elementos Finitos
PTX	*Parallel Thread eXecution*
SDP	Simétrica e Definida Positiva
SFU	*Special function unit*

SIMT	*Single-instruction, multiple-thread*
SM	*Streaming multiprocessor*
SP	*Streaming processor*
TB	*Threads Block*
TPC	*Texture/Processor cluster*
ULA	Unidade Lógica e Aritmética

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O uso do paralelismo tem aumentado consideravelmente nas aplicações *desktop*. Devido a limitações físicas, os processadores têm sido projetados com múltiplos núcleos de processamento, tornando assim o uso da programação paralela essencial. Recentemente, as unidades de processamento gráficas (GPUs) emergiram como um poderoso dispositivo de computação paralela, atingindo níveis de processamento que ultrapassam os processadores das unidades centrais de processamento (CPUs). O presente trabalho consiste na implementação do método do Gradiente Conjugado utilizando *GPU Computing*, mais especificadamente através da utilização do modelo de programação CUDA. Nessa implementação, a GPU foi utilizada para executar o método do Gradiente Conjugado e, deste modo, foi avaliado o tempo de processamento necessário para a obtenção da solução do sistemas lineares em comparação com uma implementação sequencial. Para a realização dos testes, foram utilizadas algumas matrizes disponíveis de repositório *online Matrix Market*.

Palavras-chave: Arquitetura Tesla, *GPU computing*, CUDA, método do gradiente conjugado.

GPU Computing: Implementation of method of Conjugate Gradient using CUDA

ABSTRACT

The use of paralelism has considerably increased in *desktop* applications. Due to physics limitations, processors have been designed with several processing cores, thus making the use the parallel computing essential. Recently, the graphics processing units (GPUs) have emerged as powerful parallel computing devices, achieving level of processing that exceeds central processing units (CPUs). This work consists in the implementation of method of Conjugate Gradient using GPU Computing, more specifically by the use of CUDA programming model. In this implementation, the GPU was used in order to compute the method of Conjugate Gradient and, thus, was evaluated the processing time to the obtainment of the solution of the linear systems in comparison to a sequential implementation. For the realization of tests, it was used some avaliable matrices from the online repository *Matrix Market*.

Keywords: Tesla architecture, GPU computing, CUDA, method of conjugate gradient.

1 INTRODUÇÃO

Os microprocessadores baseados em uma única unidade central de processamento (CPU - *Central Processing Unit*) tiveram uma grande evolução nas últimas duas décadas. Esses microprocessadores possibilitam a execução de bilhões de operações de ponto-flutuante por segundo (GFLOPS - *Giga floating-point operations per second*) em computadores *desktops* e centenas de GFLOPS em *clusters* de computadores. Entretanto, a taxa de crescimento do poder computacional destes processadores têm diminuído desde 2003. Questões como a dissipação de calor e o consumo de energia dos processadores têm limitado o aumento da frequência de *clock* e, desta forma, a taxa de aumento na velocidade de execução das aplicações sequenciais também tem diminuído. Para contornar essas limitações físicas, os fabricantes modificaram a arquitetura dos modelos de microprocessadores, tornando-os processadores *multicore*. Nestes modelos, múltiplas unidades de processamento, referidas como núcleos, são projetadas em cada *chip*. Assim, perante as limitações para o aumento de *clock*, a idéia consiste em aumentar o número de núcleos e, consequentemente, elevar o poder total de processamento dos microprocessadores (KIRK; HWU, 2010).

Os processadores com vários núcleos provocaram um grande impacto na comunidade de desenvolvimento de *software*. A maioria das aplicações é desenvolvida como programas sequenciais, ou seja, tais aplicações estão preparadas para serem executadas em apenas um dos núcleos do processador. Desta forma, os programas sequenciais não utilizam massivamente os recursos disponíveis e, assim, o desempenho do programa tende a não ser o esperado. Para tirar proveito das arquiteturas *multicore*, as aplicações devem ser construídas como programas paralelos, de forma que múltiplos fluxos de execução (chamados de *threads*) sejam distribuídos em diferentes núcleos e cooperem entre si para realizar uma determinada tarefa (KIRK; HWU, 2010).

O uso do paralelismo para o processamento de aplicações não é novidade na área de computação, embora o seu uso seja recente em computadores do tipo *desktop*. Por décadas, a computação paralela tem sido um meio para fornecer processamento de alto desempenho (KIRK; HWU, 2010). A programação paralela consiste na

distribuição de fluxos de execução em diferentes processadores, consequentemente, diminuindo o tempo necessário para a solução de uma tarefa. Por ser utilizada geralmente em computadores caros e de grande porte, a computação paralela ficou restrita a um pequeno conjunto de aplicações. Porém, já que a maioria dos processadores tem sido projetada com mais de um núcleo, o número de aplicações desenvolvidas como programas paralelos tem aumentado e continuará aumentando consideravelmente.

Recentemente, o uso de unidades de processamento gráficas (GPUs - *Graphics Processing Units*) emergiu como uma poderosa plataforma de computação paralela (LINDHOLM et al., 2008). As GPUs são processadores com múltiplos núcleos que, inicialmente desenvolvidos apenas para processamento gráfico, também têm sido utilizadas para processamento não-gráfico. Isto se deve, principalmente, a sua alta capacidade de processamento paralelo e intensidade aritmética (razão entre as operações aritméticas e as operações de memória). Os multiprocessadores da GPU têm evoluído rapidamente desde a década passada e atingiram níveis de processamento que ultrapassam os processadores da CPU. Os últimos modelos de GPU atingem um taxa de GFLOPS cerca de dez vezes maior do que os modelos das últimas CPUs, sendo que tanto o consumo de energia quanto o preço de ambas são similares (BOYD, 2008). O uso de processamento não-gráfico em GPUs é conhecido como *GPU Computing* (GREEN et al., 2008).

Inicialmente, o desenvolvimento de aplicações para GPUs não era trivial. Até 2006, os programadores precisavam utilizar funções de interfaces de programação de aplicações (APIs - *Application programming interface*) gráficas para programar os núcleos dos processadores. Mesmo em ambientes de programação de alto nível, a codificação ainda era limitada pelo uso dessas APIs, sendo necessário mapear o programa em termos do *pipeline* gráfico tradicional que a GPU utiliza para o processamento gráfico das aplicações (HARRIS, 2005). Esta técnica tornou-se conhecida como *General-Purpose Computation on Graphics Processing Units* (GPGPU). Apesar das dificuldades, os pesquisadores obtiveram excelentes resultados em suas aplicações em termos de desempenho (KIRK; HWU, 2010).

Com o intuito de facilitar o desenvolvimento de aplicações de *GPU Computing*, foram criados ambientes de programação que abstraem o *pipeline* gráfico do usuário. Além disso, foram realizadas mudanças no *hardware* da GPU (KIRK; HWU, 2010). No final de 2006, a NVIDIA lançou a arquitetura Tesla, que unificava os processadores programáveis da GPU. Entre os ambientes para desenvolvimento de aplicações para *GPU Computing*, destaca-se a CUDA (*Compute Unified Device Architeture*), que é uma plataforma de software fornecida pela empresa NVIDIA (NVIDIA, 2010a). Essa plataforma inclui, entre outros, ferramentas de desenvolvimento em C/C++ e um conjunto de bibliotecas que simplificam a codificação.

Muitas das aplicações de *GPU Computing* envolvem problemas de simulação numérica (BOYD, 2008). A simulação numérica é um campo essencial da computação científica, pois essa permite a realização de experimentos virtuais (através de computadores) em circunstâncias onde o experimento real é inviável (CAVALHEIRO; PASIN, 2003). Muitas dessas simulações podem ser modeladas por meio de sistemas de Equações Diferenciais Parciais (EDPs) e, em diversos casos, como não é possível encontrar uma solução analítica para essas EDPs, métodos numéricos são utilizados para calcular uma aproximação desta solução. Na discretização de uma EDP, o domínio é dividido em um conjunto finito de pontos que, interligados, formam uma malha. A EDP é então aproximada em cada um dos pontos da malha, resultando um sistema de equações lineares que é geralmente de grande porte e esparso.

Para a resolução de sistemas de equações lineares nestas condições, frequentemente são utilizados métodos iterativos (CAVALHEIRO; PASIN, 2003). Uma vez que os métodos iterativos são formados basicamente por operações de álgebra linear envolvendo matrizes e vetores, tais métodos podem ser executados eficientemente na arquitetura massivamente paralela da GPU. Dentre os métodos iterativos, um que possui destaque é o método do Gradiente Conjugado (GC), sendo o método numérico mais utilizado quando o sistema é esparso, simétrico e definidos-positivo (SDP) (CANAL, 2000).

Dentro deste contexto, nesse trabalho será desenvolvida uma implementação do método GC utilizando CUDA. Nesta implementação, os diversos núcleos da GPU serão utilizados para a execução do GC em paralelo e, deste modo, tentando reduzir o tempo de processamento na comparação com o tempo necessário para obter-se a solução em uma implementação sequencial.

1.1 Objetivos

O objetivo desse trabalho consiste na implementação em *GPU Computing* do método do Gradiente Conjugado através do modelo de programação CUDA. Esse método será utilizado para a resolução de sistemas lineares esparsos, de grande porte, simétricos e definidos-positivo (SDP). A implementação em CUDA será comparada com uma implementação sequencial do método, ou seja, utilizando unicamente os recursos da CPU para processamento. Dessa forma, poder-se-á avaliar e comparar as duas implementações e relatar o ganho de desempenho, em termos de redução de tempo de processamento, da implementação que será executada paralelamente nos núcleos da GPU.

1.2 Estrutura do Trabalho

O Capítulo 2, primeiramente, descreve o funcionamento do *pipeline* gráfico da GPU e o histórico de evolução de seus processadores. Nesse capítulo, são detalhadas a arquitetura das GPUs e a diferença da mesma em relação à arquitetura das CPUs tradicionais. A arquitetura de GPUs Tesla é o foco principal deste capítulo, uma vez que está presente nas GPUs da NVIDIA, no qual um modelo com esta arquitetura será utilizado no desenvolvimento desse trabalho.

O Capítulo 3 tem como objetivo a descrição do modelo de programação da NVIDIA CUDA, sendo descritos alguns conceitos da CUDA e seu funcionamento. Ademais, o modelo de programação será detalhado em questões fundamentais como chamadas de funções do tipo *kernel*, transferências de dados entre a CPU e a GPU, organização das *threads* e declaração de variáveis.

O Capítulo 4 tem o objetivo de descrever como surgem os sistemas de equações lineares em problemas de simulação numérica, descrevendo o mapeamento de problemas físicos através de equações diferenciais parciais (EDPs), a discretização do mesmo que gera um sistema de equações lineares esparso e de grande porte e a resolução destes sistemas por meio de métodos iterativos. O método do gradiente conjugado (GC), que será o método iterativo utilizado nesse trabalho, também será apresentado nesse capítulo.

O Capítulo 5 descreve como será realizada a implementações do método do GC. Será detalhado o formato de armazenamento de matrizes, o fluxo de execução do programa, as implementações das principais operações de álgebra linear, utilização de memória constante e recursos das GPUs.

O Capítulo 6 tem o objetivo de informar os resultados obtidos na avaliação de desempenho realizada. Será informada a origem das matrizes de teste. Os recursos de *hardware* também serão informados, especialmente os detalhes que tangem o modelo de GPU utilizado. Finalmente, serão descritos os resultados dos testes, comparando a implementação sequencial com a implementação em CUDA.

O Capítulo 7 tem o objetivo de relatar as conclusões relacionadas ao trabalho. Os resultados obtidos serão avaliados, bem como as tendências de programação utilizando *GPU Computing*. Além disso, serão informadas algumas idéias de trabalhos futuros na área.

2 UNIDADE DE PROCESSAMENTO GRÁFICO

As unidades de processamento gráficas tornaram-se uma parte integral dos computadores atuais. Devido à alta demanda do mercado de jogos 3D, as GPUs evoluiram de um pipeline gráfico de funções-fixas para um conjunto de multiprocessadores programáveis e flexíveis com um poder computacional muito superior ao das tradicionais CPUs (LINDHOLM et al., 2008). Recentemente, as GPUs também têm sido utilizadas para processamento não-gráfico, ou seja, estas têm sido utilizadas em aplicações de propósito geral como, por exemplo, em processamento de alto-desempenho em áreas como engenharia, química, economia, etc. (BOYD, 2008).

Nesse capítulo, é abordada a arquitetura das GPUs. A Seção 2.1 detalha as principais diferenças entre a CPU e a GPU no que diz respeito ao processamento de dados. A seguir, na Seção 2.2, é descrito o funcionamento convencional do *pipeline* gráfico das GPUs, de forma a oferecer uma visão sobre a evolução das GPUs contida na Seção 2.3. Finalmente, na Seção 2.4, é descrito o funcionamento da arquitetura Tesla, presente nas GPUs atuais e no qual um modelo será utilizado no desenvolvimento desse trabalho.

2.1 Diferenças entre unidades de processamento centrais e gráficas

Em 2009, o poder computacional para operações de ponto flutuante de uma GPU era cerca de dez vezes superiores ao de uma CPU tradicional. Já a sua largura de banda era cerca de três vezes superior (KIRK; HWU, 2010). Para compreender os motivos desta diferença de desempenho, é necessário entender os propósitos de cada um desses processadores.

A CPU é otimizada para desempenho de código sequencial, sendo que nessa, geralmente, utilizam-se de uma lógica de controle mais sofisticada, fazendo com que a CPU seja projetada com mecanismos como previsão de desvios e execução fora de ordem. Desta forma, permite-se que um único fluxo de instruções seja executado em paralelo enquanto a aparência de uma execução sequencial é mantida. Além disso,

a CPU também é otimizada de forma a reduzir a latência de acesso aos dados e às instruções na memória principal. De fato, grande parte dos recursos da CPU são utilizadas para o gerenciamento de vários níveis de memória *cache*, que possuem o objetivo de minimizar o tempo necessário para que os dados requisitados sejam de fato utilizados pelo processador. Uma vez que a CPU tem como alvo programas de propósito geral, poucos de seus recursos são utilizados para processamento de operações de ponto flutuante (KIRK; HWU, 2010).

A GPU é adequada para a solução de problemas onde o processamento dos dados pode ser realizado massivamente em paralelo, ou seja, o mesmo programa é executado simultaneamente em muitos elementos com alta intensidade aritmética (NVIDIA, 2010a). Por causa disto, as GPUs dedicam a maior parte de seus recursos ao processamento de dados ao invés de *cache* de dados e controle de fluxo (KIRK; HWU, 2010).

Na Figura 2.1, pode-se observar a maneira com que os transistores do *hardware* são empregados nos *chips* da CPU e GPU. Na CPU, grande parte da área do *chip* é dedicada à memória *cache* e lógica de controle. Na GPU, diferentemente da CPU, a maior parte dos recursos do *chip* é projetada para processamento massivamente paralelo de operações de ponto-flutuante. Assim, sua capacidade de processamento é aumentada consideravelmente (NVIDIA, 2010a).

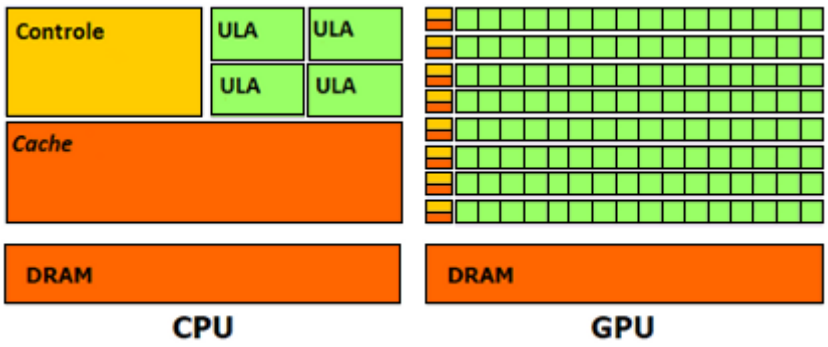


Figura 2.1: Uso dos transistores para processamento na CPU e na GPU.

Na GPU, grande parte dos dados é utilizada uma única vez, sendo que estes dados precisam ser transferidos da memória principal da GPU para o uso de seus multiprocessadores. Desta forma, o sistema de memória da GPU é projetado para maximizar a taxa de transferência de dados (*throughput*), ou seja, a GPU é projetada de forma a aumentar a largura de banda, ao invés de minimizar a latência para o acesso aos dados. Esta grande largura de banda é obtida através do uso de amplos barramentos e memórias *dynamic random access memory* (DRAM) do tipo *graphics double data rate* (GDDR) (FATAHALIAN; HOUSTON, 2008).

É importante destacar que nem todas as aplicações terão um desempenho melhor na GPU em relação à CPU. As GPUs são projetadas como *engines* de processamento

numérico, e não executam bem os programas sequenciais que as CPUs normalmente executam. Assim, deve-se utilizar a placa GPU como um co-processador para auxiliar a CPU na execução dos programas, de forma que sejam atribuídas as partes sequenciais à CPU e as partes com processamento paralelo aritmético à GPU (KIRK; HWU, 2010).

2.2 Funcionamento do *pipeline* gráfico

Embora a *GPU Computing* não trate especificamente de computação gráfica, é importante conhecer a forma de funcionamento do *pipeline* gráfico de uma GPU. Desta forma, é mais fácil compreender as razões que levaram a GPU a se tornar uma plataforma para a execução de programas de aplicações não-gráficas (KIRK; HWU, 2010).

O objetivo dos sistemas gráficos é gerar imagens que representam visões de uma cena virtual. A visão desta cena é descrita pela localização de uma câmera virtual e é definida pela geometria, orientação e propriedades materiais da superfície dos objetos, bem como das características das fontes de luz. As APIs gráficas como OpenGL (OPENGL, 2010) e DirectX (Microsoft, 2010) representam este processo como um *pipeline* que executa uma série de operações sobre um conjunto de vértices enviados pela CPU, sendo que cada vértice possui alguns atributos, tais como posição, cor e vetor normal (FATAHALIAN; HOUSTON, 2008).

O *vertex shader* é um programa que executa um conjunto de operações para cada um dos vértices de entrada, com o intuíto de projetar cada vértice, baseado em sua posição relativa à camera virtual, em um espaço de tela 2D. Destes vértices, é montado um conjunto de triângulos, que representam os objetos no espaço 2D. Assim, quanto maior a quantidade de triângulos, melhor a qualidade com que o objeto será representado (FATAHALIAN; HOUSTON, 2008). Visto que cada cena possui milhares de vértices e cada um deles pode ser tratado independentemente, os vértices podem ser processados paralelamente (GREEN et al., 2008).

A seguir, ocorre o processo de rasterização, que consiste em determinar quais espaços da tela são cobertos por cada triângulo. Esse processo resulta na geração de fragmentos para cada espaço de tela coberto. Um fragmento pode ser considerado um "*proto-pixel*", que contem todas as informações necessárias para gerar um *pixel* na imagem final (profundidade, localização no *frame buffer*, etc.). A partir da posição da câmera virtual, os fragmentos que são ocultos por outros fragmentos são descartados (GREEN et al., 2008).

Já o *pixel shader* opera sobre a saída gerada pelo processo de rasterização. O *pixel shader* é um programa que consiste em um conjunto de operações que são executadas sobre cada fragmento, antes que estes sejam plotados na tela. Utilizando

as informações de cor dos vértices e, possivelmente, buscando dados adicionais na memória global em forma de texturas (imagens que são mapeadas sobre as superfícies dos objetos), cada fragmento é processado para obter-se a cor final do *pixel*. Assim como na etapa de processamento de vértices, os fragmentos são independentes e podem ser processados em paralelo. Esta etapa é a que tipicamente demanda maior processamento dentro da estrutura do *pipeline* gráfico (GREEN et al., 2008).

Uma vez que os programas de *shader* necessitam ser aplicados em milhares de vértices e *pixels* independentemente, as GPUs evoluíram para um conjunto de multiprocessadores massivamente paralelos. Além disso, dependendo do balanceamento da carga de trabalho da aplicação, apesar dos *pixels* serem dependentes dos vértices, ambos podem ser executados paralelamente. Esta característica resultou no aumento da programabilidade dos multiprocessadores da GPU (KIRK; HWU, 2010). Mais detalhes sobre a evolução das GPUs serão vistos na Seção 2.3.

2.3 Evolução dos processadores das GPUs

Em 1999, a *NVIDIA GeForce 256* foi lançada no mercado, sendo a primeira GPU a ser comercializada. Esta possuia um *pipeline* gráfico de funções fixas compostos de dois tipos de processadores: processadores de vértices e processadores de *pixels* (LINDHOLM et al., 2008). Ambos os processadores eram configuráveis, mas não eram programáveis. Esta falta de flexibilidade tornou-se um fator limitante para o desenvolvimento das aplicações, que eram cada vez mais sofisticadas e possuiam características cada mais vez distintas (KIRK; HWU, 2010).

Em 2001, a *NVIDIA GeForce 3* introduziu o primeiro processador de vértices programável. Visto que o processador de vértices tradicionalmente suporta um processamento mais complexo que o processador de *pixels*, eles foram os primeiros processadores da GPU a se tornarem programáveis. Já em 2003, a *ATI Radeon 9700* introduziu o primeiro processador de *pixels* programável (LINDHOLM et al., 2008).

Os processadores de vértices e *pixels* evoluíram em taxas diferentes. Os processadores de vértices foram projetados para operações aritméticas de alta precisão e baixa latência, enquanto os processadores de *pixels* foram otimizados para filtros de textura de baixa precisão e alta latência. Uma vez que as GPUs tipicamente processam mais *pixels* do que vértices, o número de processadores de *pixels* era tradicionalmente três vezes superior ao número de processadores de vértices. Contudo, a carga de trabalho nem sempre era balanceada desta maneira, levando à ineficiência de processamento em algumas situações. Em 2005, a Microsoft introduziu no *chip ATI Xenos* do *XBox 360*, a primeira GPU unificada, de forma que as operações sobre os vértices e os *pixels* fossem executadas no mesmo tipo de processador, permitindo que o balanceamento da carga de trabalho fosse realizado pela aplicação

(LINDHOLM et al., 2008).

Em novembro de 2006, a NVIDIA introduziu a arquitetura Tesla em suas GPUs *GeForce 8-series*. O primeiro objetivo da arquitetura Tesla era a execução de operações sobre vértices e *pixels* no mesmo processador. Esta unificação permitiu o balanceamento dinâmico de cargas de trabalho variáveis de vértices e *pixels*, além de introduzir novos estágios de *shaders*, tais como o *Geometry Shader* no DirectX 10 (Microsoft, 2010). Esta unificação dos processadores também disponibilizou uma arquitetura de processamento paralelo de alto desempenho para aplicações de *GPU Computing* (LINDHOLM et al., 2008).

2.4 Arquitetura Tesla

A arquitetura Tesla é uma arquitetura unificada para gráficos e processamento que fornece multiprocessadores programáveis tanto para a execução de aplicações gráficas quanto para programas de *GPU Computing* (processamento não-gráfico) (LINDHOLM et al., 2008). Uma vez que esse trabalho consiste no desenvolvimento de uma aplicação para *GPU Computing*, os detalhes referentes ao *pipeline* gráfico dessa arquitetura não serão abordados na descrição dessa arquitetura.

A arquitetura Tesla é organizada em um *array* de processadores escalares, que possui mecanismos para o gerenciamento e controle de milhares de *threads*. Esse *array* contem unidades de processamento independentes chamadas de *Texture/Processor Clusters* (TPCs), que são formados por dois *Streaming Multiprocessors* (SMs), e que são interconectados a uma memória DRAM, referenciada como memória global. O número de TPCs em uma GPU varia em relação à geração da mesma, sendo que cada SM é composto de um determinado número de *Streaming Processors* (SPs) (LINDHOLM et al., 2008).

Nessa seção, a arquitetura Tesla será demonstrada a partir da estrutura de uma *NVIDIA GeForce 8800* (G80). A G80 foi a primeira GPU de alto desempenho da NVIDIA que foi projetada com base na arquitura Tesla, sendo que os modelos de GPU baseados em arquitetura Tesla lançados posteriormente possuem a mesma estrutura. Basicamente, o que diferencia os modelos são a velocidade de acesso aos dados, o número de SMs na GPU e a quantidade de SPs em cada SM (KIRK; HWU, 2010). A Figura 2.2 ilustra a arquitetura Tesla em uma NVIDIA G80 para *GPU Computing*. Basicamente, essa possui oito TCPs, com dois SMs cada, e que compartilham de uma mesma unidade de textura. Cada SM possui uma memória compartilhada e oito SPs, totalizando 128 SPs em uma NVIDIA G80.

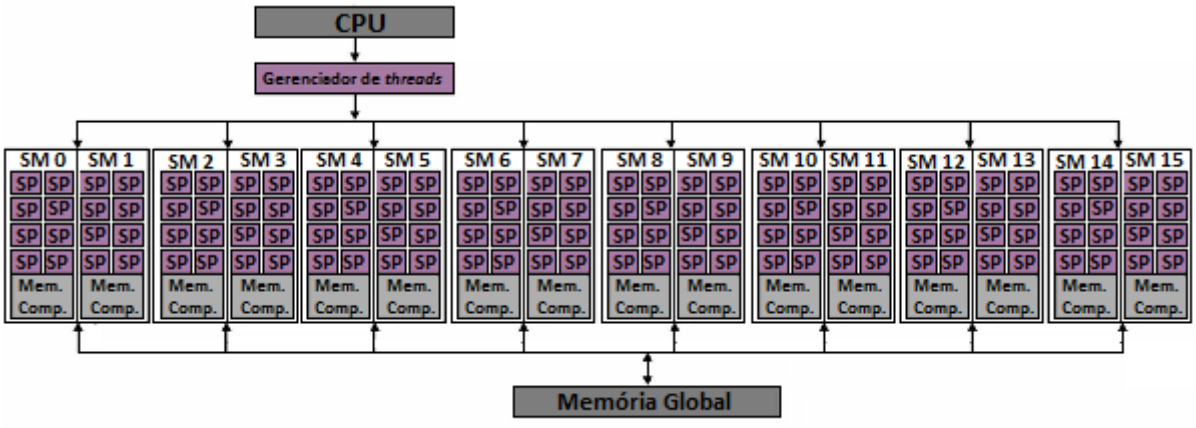


Figura 2.2: Modelo simplificado da arquitetura Tesla em uma *NVIDIA GeForce 8800*.

2.4.1 *Streaming Multiprocessors*

O *Streaming Multiprocessor* (SM) é um multiprocessador unificado que executa programas em paralelo (LINDHOLM et al., 2008). Em uma NVIDIA G80, por exemplo, ele consiste basicamente de oito *Streaming Processors* (SPs), duas *Special Function Units* (SFUs), um *cache* de instruções, um *cache* para memória constante, uma unidade *multithreaded* de busca e despacho de instruções e uma memória compartilhada *on-chip* de 16 *KBytes*.

Os SPs são os núcleos de processamento do SM e são responsáveis pelas execuções das *threads*, sendo que as *threads* de um mesmo SP compartilham lógica de controle e *cache* de instruções. A quantidade de SPs em um SM é variável e depende do modelo da GPU. Assim, quanto maior a quantidade de SPs, maior será a quantidade de *threads* que o SM conseguirá gerenciar (LINDHOLM et al., 2008). Na terminologia de *GPU Computing*, os SPs também são referenciados como *CUDA cores*.

As SFUs são utilizadas para operações de ponto-flutuante complexas e funções transcendentais, tais como seno e cosseno (LINDHOLM et al., 2008). Segundo (WALDSCHMIDT, 2009), "uma função transcendental é uma função a qual não satisfaz uma equação polinomial cujos coeficientes são eles próprios polinomiais. Mais tecnicamente, uma função de uma variável é transcendental se ela é algebricamente independente desta variável".

Geralmente, as aplicações em GPUs instanciam muitas *threads* que processam grandes conjuntos de dados. Para executar eficientemente programas concorrentes, o SM foi desenvolvido com suporte *multithreading*, ou seja, ele é capaz de executar milhares de *threads* concorrentes com baixo *overhead* de escalonamento (LINDHOLM et al., 2008). Em uma NVIDIA G80, cada um dos SMs é capaz de gerenciar até 768 *threads*, sendo 96 *threads* por SP. Deste modo, utilizando-se os dezesseis SMs é possível executar até 12.228 *threads*.

2.4.2 *Single-Instruction, Multiple-Thread*

Para gerenciar e executar milhares de *threads*, os SMs utilizam uma arquitetura de processador chamada de *Single-Instruction, Multiple-Thread* (SIMT). As unidades de instrução *multithreaded* SIMT criam, gerenciam e executam as *threads* em grupos de 32 *threads* chamados *warps*. Em uma NVIDIA G80, cada SM gerencia até 24 *warps*, totalizando 768 *threads* por SM (LINDHOLM et al., 2008).

As *threads* que compõem um *warp* são do mesmo tipo e iniciam juntas no mesmo endereço de instrução, mas estas são livres para desviarem do caminho de execução original e executarem independentemente. A Figura 2.3 ilustra o funcionamento de um escalonador de *warps*. A cada ciclo, o escalonador de *warps* seleciona um *warp* que está pronto para ser executado e emite a próxima instrução para as *threads* do *warp*. Esta instrução é então executada simultaneamente por todas as *threads* ativas do *warp*. As *threads* que estão bloqueadas devido à divergência do caminho de execução dentro do *warp* não executarão essa instrução (LINDHOLM et al., 2008).

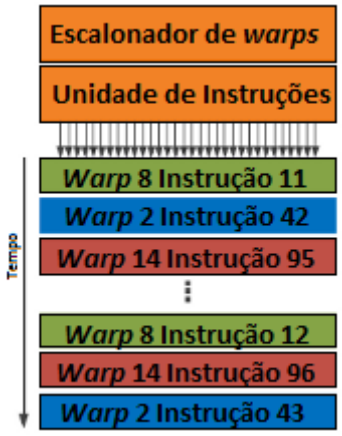


Figura 2.3: Funcionamento do escalonador de *warps*.

Os SMs mapeiam as *threads* dos *warps* para os seus núcleos SP, e cada *thread* executa independentemente com seu próprio endereço de instrução e registrador de estado. Se as *threads* de um *warp* divergirem do caminho de execução original, o *warp* executa cada novo caminho de execução sequencialmente, bloqueando as *threads* que não pertencem ao caminho. Assim, o processador obtém eficiência e desempenho máximo quando todas as *threads* de um *warp* possuem o mesmo caminho de execução (LINDHOLM et al., 2008).

O motivo pelo qual a quantidade de *warps* gerados é maior que a quantidade de SPs deve-se à forma na qual os SMs executam as operações de alta latência como, por exemplo, operações de acesso à memória global. Quando uma instrução executada pelas *threads* de um *warp* encontra-se a espera pelos operandos, este *warp* não é selecionado para a execução. Assim, um outro *warp* que não se encontra a espera

por um operando é selecionado. Com a criação de diversos *warps*, o escalonador de *warps* possivelmente encontrará um *warp* pronto para executar a cada ciclo, diminuindo o tempo ocioso do SM (KIRK; HWU, 2010).

2.4.3 *Thread Blocks*

Os modelos de programação paralela requerem que as *threads* sincronizem-se e cooperem entre si. Para gerenciar um grande número de *threads*, a arquitetura de processamento Tesla introduziu os *cooperative thread arrays*, que são conhecidos como blocos de *threads* (TBs - *threads blocks*) na terminologia CUDA.

Um TB é um conjunto de *threads* que executam o mesmo código (LINDHOLM et al., 2008). Para cada uma das *threads* de um TB, é atribuido um identificador único, referente à posição da *thread* dentro do TB. As *threads* de um TB sincronizam-se através de instruções do tipo barreira e podem compartilhar dados tanto na memória global quanto na memória compartilhada. Nas GPUs atuais, um TB pode conter até 512 *threads*. Este limite deve-se ao fato que todas as *threads* de um TB devem ser executadas no mesmo SM e portanto devem compartilhar os recursos do mesmo (registradores, memória, etc.) (NVIDIA, 2010a).

Os TBs que executam um mesmo programa são organizados em uma estrutura de *grid*, sendo que dentro do *grid*, atribui-se um identificador único para cada TB criado. Cada um dos TBs é executado independentemente dos demais. A Figura 2.4 ilustra o particionamento das *threads* dentro de um conjunto de TBs. Neste exemplo, o *grid* está estruturado como uma matriz de TBs com dimensões 2x3. Já os TBs estão estruturados como uma matriz de *threads* com dimensões 3x4, totalizando doze *threads*. Uma vez que seis TBs foram gerados, este *grid* é formado por 72 *threads*.

A distribuição dos TBs entre os SMs é realizada pela unidade distribuidora de trabalho. Esta unidade balanceia dinamicamente a carga de trabalho da GPU, distribuindo um fluxo de TBs para os SMs com recursos disponíveis. Cada SM pode executar até oito TBs concorrentemente, sendo que, se não existirem recursos suficientes para a execução de um TB, o mesmo é mantido em uma lista de espera até que os recursos necessários sejam liberados (LINDHOLM et al., 2008).

Os TBs são executados como *warps* de 32 *threads*. Os TBs são particionados em *warps* baseados nos identificadores das *threads*, que estão em ordem crescente e consecutiva. Em um *warp* n, por exemplo, a primeira *thread* possuirá o identificador $32 \times n$ e a última *thread* possuirá o identificador $32 \times (n + 1) - 1$. Para um TB cujo o tamanho não é múltiplo de 32, o último *warp* será preenchido com *threads* adicionais para completar as 32 *threads* necessárias (KIRK; HWU, 2010).

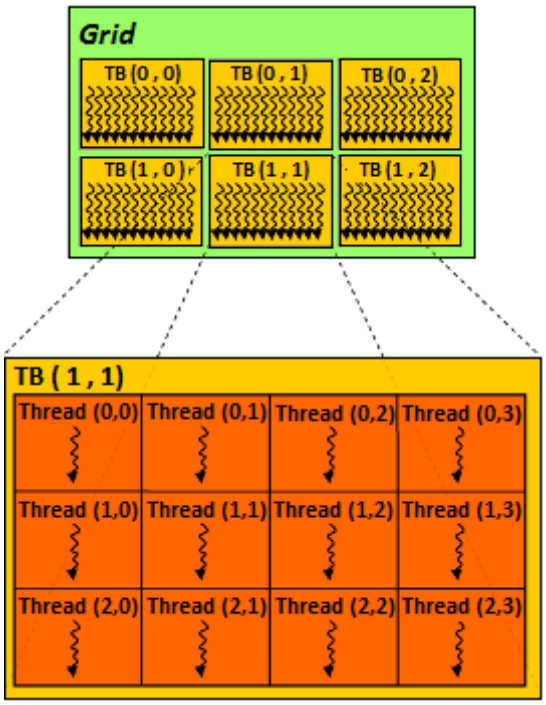


Figura 2.4: Exemplo do particionamento das *threads* dos Threads Blocks em um *grid*.

2.4.4 Escalabilidade

A capacidade de paralelismo das GPUs pode variar substancialmente, dependendo para qual segmento de mercado a GPU foi projetada. Enquanto uma GPU pode possuir um único SM com oito SP escalares, uma GPU mais poderosa poderá conter vários SMs, totalizando centenas de núcleos SP. Para que um programa não seja dependente de um modelo específico de GPU, a arquitetura de processamentos da GPU escalona, de forma transparente, os TBs de um programa entre os SMs e SPs disponíveis no *chip* gráfico.

Esse escalonamento é realizado a partir da decomposição do problema em TBs independentes, como descrito na Seção 2.4.3, sendo que a unidade distribuidora de trabalho da GPU distribui os TBs entre os SMs disponíveis. Uma vez que os TBs não se comunicam dentro de um mesmo *grid*, o mesmo resultado será obtido se os TBs forem executados concorrentemente em vários SPs, sequencialmente em apenas um SM ou paralelizados parcialmente em alguns SPs. A Figura 2.5 ilustra o escalonamento dos TBs em duas GPUs com um número diferentes de SMs (LINDHOLM et al., 2008).

2.4.5 Hierarquia de Memória

A Figura 2.6 ilustra a hierarquia de memória em uma GPU com arquitetura Tesla. A memória da placa gráfica que compõe a GPU pode ser dividida em cinco tipos principais:

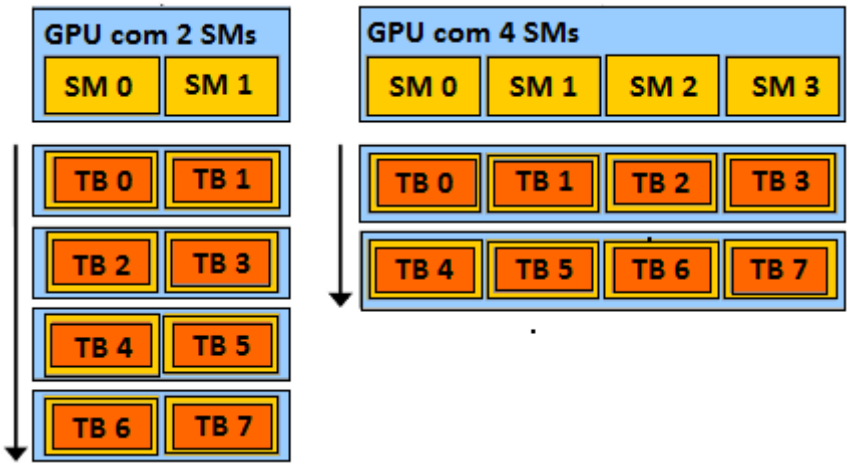


Figura 2.5: Distribuição de Threads Blocks em GPUs com números diferentes de SM.

- Registradores: estão localizados nos SPs e são o tipo de memória com acesso mais rápido da GPU. Os registradores são geralmente utilizados para armazenar as variáveis locais e os dados temporários das *threads*. Embora seu uso seja bastante eficiente, a quantidade de registradores é bastante limitada. Cada SP possui 1024 registradores de 32 bits, que são compartilhados pelas *threads* dos TBs.

- Memória compartilhada: é uma memória localizada em cada um dos SM e que apresenta um acesso de baixa latência. Cada SM possui 16K de memória compartilhada, que é distribuída entre os TBs. Cada TB aloca uma área da memória compartilhada de maneira privada, de forma que somente as *threads* de um mesmo TB possam acessá-la. Essa área de memória possibilita que as *threads* de um mesmo TB compartilhem dados entre si.

- Memória global: implementada na memória DRAM da placa gráfica, essa memória foi projetada para possuir alta latência e grande largura de banda, sendo utilizada para a transferência de dados entre a CPU e a GPU. Embora o acesso aos dados seja lento, sua capacidade de armazenamento é alta. Outra característica importante é que ela pode ser acessada por todas as *threads* de todos os programas, possibilitando o compartilhamento de resultados intermediários. É importante ressaltar que não existe nenhum nível de memória *cache* para o acesso à memória global.

- Memória constante: embora também esteja localizada na memória global, a memória constante possui características que a diferencia da memória global da GPU. A principal diferença consiste que as *threads* podem acessar a memória constante com baixa latência (cada SM possui um *cache* para armazenar os

dados da memória constante), contudo, as *threads* podem acessá-la apenas em modo de leitura. A principal desvantagem da memória constante é que esta possui uma capacidade de armazenamento limitada a 64 *KBytes*.

- Memória de textura: utilizada geralmente para processamento gráfico, embora também possa ser utilizada para *GPU Computing*. Seus dados também são implementados na memória global, mas foram projetados dois níveis de *cache*, um *on-chip* (L1) e outro *off-chip* (L2), bem como uma unidade específica para o gerenciamento das texturas armazenadas. No caso da memória de textura, as *threads* possuem acesso aos dados somente no modo de leitura.

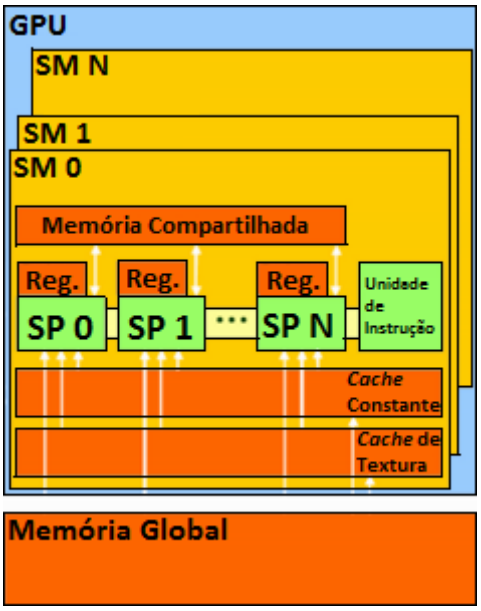


Figura 2.6: Hierarquia de memória da arquitetura Tesla.

2.4.6 Operações de Ponto Flutuante

Uma questão importante na seleção de um processador para a execução de aplicações em computação numérica é o suporte ao padrão da notação de ponto flutuante do *Institute of Electrical and Eletronics Engineers* (IEEE). Esse padrão torna possível a obtenção de resultados semelhantes entre processadores de diferentes fabricantes. Na NVIDIA G80, a representação de ponto flutuante de precisão simples possue divergências em relação à norma IEEE-754, que padroniza a notação de ponto flutuante. Dessa forma, é provável que existam diferenças entre operações realizadas na GPU e na CPU. Além disso, na G80, não há suporte para operações de ponto flutuante de precisão dupla, ou seja, aplicações que necessitam desse suporte não são adequadas para serem executadas nessa GPU. Entretanto, as GPUs mais recentes têm evoluído nessa questão (KIRK; HWU, 2010).

Outra característica importante diz respeito à instrução multiplica-soma (MAD - *Multiply-add*). Uma sequência de operações que é frequentemente utilizada em aplicações de computação gráfica consiste em multiplicar dois números, somando o produto resultante com um terceiro número (por exemplo, $D = A \times B + C$). Assim, as GPUs possuem suporte a uma instrução em que essas duas operações sejam executadas em um único ciclo de *clock*. Contudo, os resultados intermediários da multiplicação são truncados, resultando em perda de precisão (NVIDIA, 2010a).

3 NVIDIA CUDA

A CUDA é uma arquitetura de processamento paralelo de propósito geral desenvolvida pela NVIDIA, introduzida no mercado em novembro de 2006. Através de um um novo modelo de programação paralela e um conjunto de instruções (PTX - *Parallel Thread eXecution*) compatíveis com a arquitetura Tesla, a CUDA consiste em uma plataforma de software que possui ferramentas de desenvolvimento, bibliotecas de funções e um mecanismo de abstração de *hardware* que facilita o desenvolvimento de aplicações de *GPU Computing* (NVIDIA, 2010a).

O ambiente CUDA foi projetado com o objetivo de tornar a programação de GPUs mais acessível, diminuindo a curva de aprendizagem do seu modelo de programação (NVIDIA, 2010a). Os desenvolvedores podem utilizar linguagens de alto nível com extensões para a escrita de código, tais como C, C++, Fortran, OpenCL e *DirectCompute*. Nesse trabalho, será utilizada a linguagem C.

Esse capítulo consiste em apresentar o modelo de programação da CUDA. Na Seção 3.1, são descritos a estrutura e o fluxo de execução de um programa CUDA. A Seção 3.2 mostra as principais funções de transferência de dados e alocação de memória. Na Seção 3.3, é detalhada a invocação de funções do tipo *kernel*, bem como a organização, a identificação e a sincronização das *threads* geradas. A Seção 3.4 possui informações sobre a declaração e os tipos de variáveis. A Seção 3.5 apresenta um exemplo de código em CUDA e sua comparação com uma implementação sequencial.

3.1 Estrutura de um programa CUDA

Uma das principais características da CUDA consiste na abstração da arquitetura da placa gráfica. A CUDA fornece três abstrações principais ao programador: hierarquia de *threads*, memória compartilhada e barreiras de sincronização. Tais abstrações são mostradas ao programador como um pequeno conjunto de funções estendidas à linguagem C (NVIDIA, 2010a).

A estrutura de um programa CUDA pode ser dividida em duas partes: o código

que será executado na CPU, referenciada como *host*, e aquele que será executado na GPU. A programação para ambas as partes é realizada no mesmo arquivo fonte, sendo que no momento da compilação, os códigos são separados pelo *NVIDIA C Compiler* (NVCC). Depois de separados, cada código é compilado independentemente, gerando arquivos com instruções próprias para cada uma das arquiteturas (KIRK; HWU, 2010).

As funções que são processadas na GPU são chamadas de *kernel*. Quando um *kernel* é chamado pelo *host*, a execução do programa é transferida para a GPU. Cada chamada de *kernel* corresponde a criação de um *grid*, onde todas as *threads* criadas são controladas e executadas automaticamente em um conjunto de TBs. Quando todas as *threads* finalizam suas execuções, o *grid* é concluído e o programa volta a ser executado pelo *host* até que uma nova chamada de *kernel* seja realizada (KIRK; HWU, 2010).

A Figura 3.1 mostra o fluxo de execução de um programa CUDA. Inicialmente, a execução sequencial está sendo realizada pelo *host*. Quando um *kernel* é chamado pelo *host*, o fluxo de execução é transferido para a GPU, onde centenas de *threads* são criadas e executadas. Terminada a execução de todas as *threads*, o fluxo de execução volta a ser realizado sequencialmente pelo *host*.

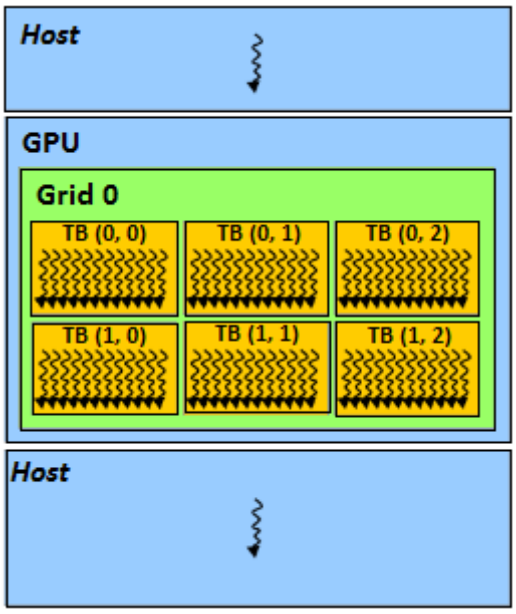


Figura 3.1: Fluxo de execução de um programa CUDA.

No início da execução do programa, os dados estão na memória principal do *host* e, para que a GPU execute instruções sobre esses dados, é necessário que os mesmos sejam transferidos para a memória da GPU. Da mesma forma, ao final do processamento, os dados devem ser transferidos novamente para a memória principal (KIRK; HWU, 2010).

3.2 Transferência de dados

Em CUDA, o *host* e a GPU possuem e trabalham sobre diferentes dispositivos de memória. Isto ocorre em função da arquitetura da GPU possuir sua própria memória DRAM. Para que uma função do tipo *kernel* seja executada, é necessário primeiramente que uma área de memória seja alocada na GPU para que, depois, os dados sejam transferidos pelo *host*. Similarmente, após a execução do *kernel*, os dados resultantes devem ser transferidos para o *host* e a memória deve ser desalocada (KIRK; HWU, 2010).

Na API fornecida pela plataforma de desenvolvimento CUDA, são encontradas funções de alocação de memória e transferência de dados entre as duas arquiteturas. As principais funções desse gênero são:

- *cudaError_t cudaMalloc (void **devPtr, size_t size)*: é a função utilizada para a alocação linear de uma área de memória na GPU. Esta função retorna pelo parâmetro *devPtr* o endereço da área de memória alocada.

- *cudaError_t cudaMemcpy (void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)*: essa função executa a cópia de uma quantidade de *bytes* da memória de origem para a memória de destino. O argumento *kind* define a origem e o destino para a cópia de uma área de memória e pode ser uma das seguintes constantes: *cudaMemcpyHostToHost* (de uma área do *host* para outra área do *host*), *cudaMemcpyHostToDevice* (de uma área do *host* para uma área da GPU), *cudaMemcpyDeviceToHost* (de uma área da GPU para uma área do *host*) ou *cudaMemcpyDeviceToDevice* (de uma área do GPU para uma outra área da GPU).

- *cudaError_t cudaFree (void *devPtr)*: essa função libera o espaço de memória cujo o endereço está armazenado em *devPtr*.

O código da Figura 3.2 mostra a alocação de memória e transferência de dados em um programa CUDA. Nela, os vetores M e Md correspondem aos dados que serão processados, enquanto os vetores P e Pd armazerarão os dados resultantes. Nas linhas 5 e 6, é realizada a alocação dos vetores M e P na memória do *host*. A alocação desses vetores é realizada através da função *malloc()*. Já nas linhas 7 e 8, é feita a alocação dos vetores Md e Pd na memória na GPU. A alocação destes vetores é realizada através da função *cudaMalloc()*. Na linha 10, após M ter sido inicializado, realiza-se a cópia dos dados de M para Md, ou seja, da memória do *host* para a memória da GPU. Na linha 12, após a GPU processar os dados iniciais e armazená-los em Pd, os dados resultantes são copiados para o vetor P, que está localizado na memória do *host*. Finalmente, nas linhas 14 e 15, libera-se

a memória alocada tanto da GPU quanto do *host* através das funções *cudaFree()* e *free()*, respectivamente.

```
01. int main() {
02.     float *M, *Md,*P, *Pd;
03.     int tam = ... ;
04.     ...
05.     M = (float*) malloc (tam);
06.     P = (float*) malloc (tam);
07.     cudaMalloc((void**) &Md, tam);
08.     cudaMalloc((void**) &Pd, tam);
09.     ...
10.     cudaMemcpy (Md, M, tam, cudaMemcpyHostToDevice);
11.     ...
12.     cudaMemcpy (P, Pd, tam, cudaMemcpyDeviceToHost);
13.     ...
14.     cudaFree(Md); cudaFree(Pd);
15.     free(Md); free(Pd);
16. }
```

Figura 3.2: Exemplo de código em CUDA para a transferência de dados.

A memória na GPU também pode ser alocada como *arrays* de duas ou três dimensões, através de funções como *cudaMalloc3D()*, *cudaMallocArray()*, *cudaMemcpy2D()*, *cudaMemcpy3D()* e *cudaFreeArray()*. Contudo, estas funções não serão utilizadas nesse trabalho. Maiores informações podem ser encontradas no *NVIDIA CUDA: Reference Manual* (NVIDIA, 2010b).

3.3 *Kernel*

Como mencionado na Seção 3.1, o *kernel* especifica o código que será executado na GPU pelas *threads*, sendo que todas as *threads* executam o mesmo código, ou seja, o mesmo *kernel*.

3.3.1 Funções de *Kernel*

Em CUDA, os *kernels* são declarados como funções padrões da linguagem C. Entretanto, para a diferenciação dos *kernels* das funções tradicionais, são utilizadas três palavras-chaves que antecedem a declaração das funções, que são:

- *__global__*: indica que a função declarada é um *kernel* CUDA. Esta função pode ser chamada apenas pelo *host*.
- *__device__*: indica que a função declarada é uma função dentro do *kernel*. Desta forma, essas funções somente poderão ser chamadas por funções declaradas

como *__global__* ou *__device__*. É conveniente destacar que, utilizando-se esse tipo de função, não é possível realizar chamadas recursivas.

- *__host__*: indica que a função declarada será executada no *host*, isto é, é uma função tradicional da linguagem C. Esta é a opção *default* da CUDA, ou seja, caso nenhuma das três palavras-chaves seja informada, o compilador tratará a função como uma função a ser executada no *host*.

3.3.2 Organização das *Threads*

Quando um *kernel* é chamado pelo *host*, ele é executado como um *grid* de *threads*, sendo que este *grid* é organizado como um *array* de uma, duas ou três dimensões de blocos de *threads* (TBs). Dentro do *grid*, todos os TBs são igualmente organizados e possuem o mesmo número de *threads* (KIRK; HWU, 2010). Em um *grid*, cada TB possui um identificador único de três posições (x,y,z) que são setadas automaticamente pelo *hardware multithreaded* da GPU.

Os TBs são organizados como um *array* de uma, duas ou três dimensões de até 512 *threads*, sendo que em cada TB, as *threads* possuem um identificador único de três posições (x,y,z), que também é setado automaticamente pelo *hardware multithreaded* da GPU. É importante ressaltar que o identificador de uma *thread* é único apenas dentro de um TB. O *grid* gerado pela execução de um *kernel* frequentemente possuirá mais de um TB, portanto mais de uma *thread* receberá o mesmo identificador dentro de um *grid* (KIRK; HWU, 2010).

```
01. __global__ void kernel1() {
02.     ...
03. }
04. 05. int main() {
06.     ...
07.     dim3 dimGrid(2,3,1);
08.     dim3 dimBlock(3,5,1);
09.     kernel1<<<dimGrid,dimBlock>>>();
10.     ...
11. }
```

Figura 3.3: Exemplo de código para a chamada de um *kernel*.

No momento em que um *kernel* é executado pelo *host*, pode-se configurar dinamicamente a dimensão do *grid* e de seus TBs. É possível especificar a quantidade de TBs dentro do *grid* e a quantidade de *threads* dentro de cada TB, por meio de dois parâmetros que não seguem o padrão convencional da linguagem C. Estes parâmetros são inseridos entre a chamada do *kernel* e seus parâmetros tradicionais com a seguinte sintaxe: *kernel<<<dimGrid, dimBlock>>>(args)*. O *dimGrid* e

o *dimBlock* são variáveis que podem ser do tipo *int* ou *dim3*. O tipo *int* é utilizado para declarar estruturas de apenas uma dimensão. Já o tipo *dim3*, que é um padrão de *struct* fornecida pela CUDA, é utilizado para estruturas de duas ou três dimensões. Assim que uma variável *dim3* é declarada, ela recebe o tamanho das suas dimensões. As dimensões não utilizadas são setadas automaticamente com o valor 1.

O exemplo de código da Figura 3.3 demonstra a chamada da função *kernel1()*. Neste exemplo, as variáveis *dimGrim* (linha 7) e *dimBlock* (linha 8) são do tipo *dim3* e foram declaradas como estruturas de duas dimensões. Na linha 9, essas dimensões são setadas na chamada da função *kernel1*. A organização das *threads* criadas durante a execução da função *kernel1* é ilustrada pela Figura 3.4. O *grid* gerado é estruturado como uma matriz de TB com dimensões 2x3. Já os TB estão estruturados como uma matriz de *threads* com dimensões 3x5, totalizando quinze *threads*. Assim, uma vez que seis TBs foram gerados, esse *grid* é formado por noventa *threads*.

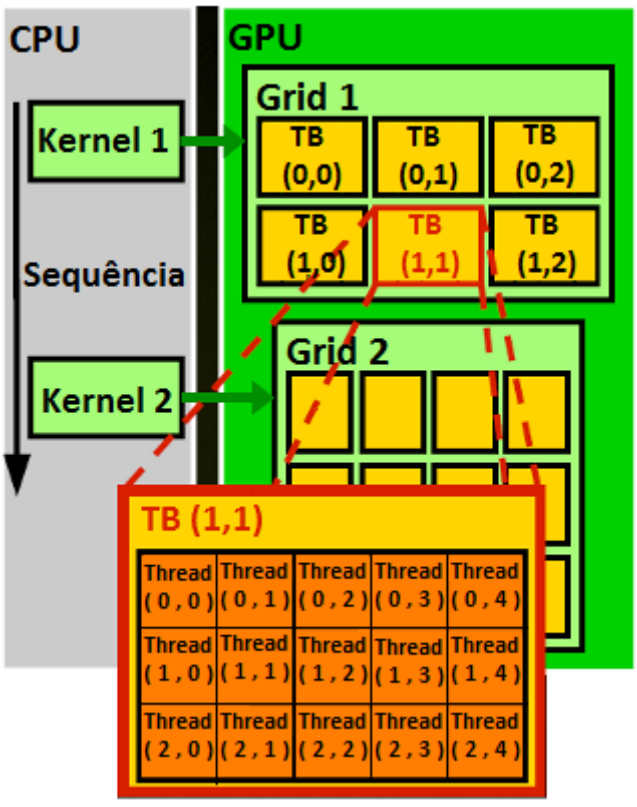


Figura 3.4: Ilustração da execução de um *kernel*.

3.3.3 Identificação das *Threads*

Uma vez que todas as threads executam o mesmo código de seu kernel de origem, existe um mecanismo para distinguí-las e direcioná-las para trabalhar com elementos específicos. A CUDA fornece quatro variáveis pré-definidas que permitem identificar

as *threads* e os elementos que serão acessados pela mesma: *threadIdx* (índice identificador da *thread* dentro de um TB), *blockIdx* (índice identificador do TB dentro de um *grid*), *blockDim* (dimensões do TB) e *gridDim* (dimensões do *grid*). Assim, através da combinação destas variáveis, pode-se fazer com que cada *thread* compute uma parte dos dados independentemente (KIRK; HWU, 2010).

Por exemplo, supondo que haja uma matriz de dimensões 6x15 e deseja-se realizar uma determinada operação sobre cada um de seus elementos. O *grid* gerado na Figura 3.4 é adequado para realizar esta tarefa. Baseado nesse *grid*, o código da Figura 3.5 pode direcionar cada uma das 90 *threads* geradas para processar um único elemento da matriz. Nas linhas 2 e 3, as variáveis i e j correspondem ao índice (linha e coluna, respectivamente) do elemento da matriz que será processado. A Figura 3.6 demonstra o direcionamento das *threads* sobre cada um dos elementos. Uma vez que todos os TB se comportam da mesma forma, somente as *threads* do TB(1,1) serão ilustradas. O TB(1,1) possui quinze *threads* que estão no intervalo 60-74.

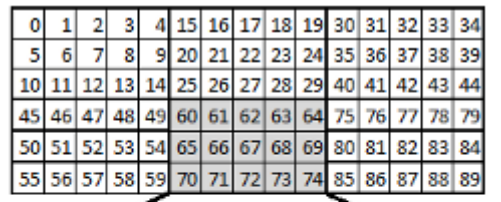
```
01. __global__ void kernel() {
02.     int i = blockDim.x * blockIdx.x + threadIdx.x;
03.     int j = blockDim.y * blockIdx.y + threadIdx.y;
04.     ...
05.     // Realizar a operação sobre o elemento
06.     ...
07. }
```

Figura 3.5: Exemplo de código em CUDA para direcionamento de *threads*.

3.3.4 Sincronização de *Threads*

As *threads* de um mesmo TB podem cooperar entre si através da memória compartilhada e sincronizar suas execuções com funções do tipo barreira. Quando um *kernel* utiliza a função *__syncthreads*, as *threads* só continuarão suas execuções no momento em que todas as *threads* do TB chamarem esta função. A Figura 3.7 exemplifica o uso da função *__syncthreads*. Com o uso de barreiras, garante-se que todas as *threads* do TB tenham concluído uma tarefa antes de executarem as próximas instruções do *kernel*.

É importante ressaltar que não é possível que *threads* de diferentes TB sincronizem-se e cooperem entre si de uma maneira segura (KIRK; HWU, 2010). Isto deve-se ao fato que cada TB é executado independentemente em qualquer ordem. Esta característica está diretamente ligada à escalabilidade das GPUs, descrita na Seção 2.4.4.



# Thread	blockDim		blockIdx		threadIdx		Elemento	
	x	y	x	y	x	y	i	j
60	3	5	1	1	0	0	3	5
61	3	5	1	1	0	1	3	6
62	3	5	1	1	0	2	3	7
63	3	5	1	1	0	3	3	8
64	3	5	1	1	0	4	3	9
65	3	5	1	1	1	0	4	5
66	3	5	1	1	1	1	4	6
67	3	5	1	1	1	2	4	7
68	3	5	1	1	1	3	4	8
69	3	5	1	1	1	4	4	9
70	3	5	1	1	2	0	5	5
71	3	5	1	1	2	1	5	6
72	3	5	1	1	2	2	5	7
73	3	5	1	1	2	3	5	8
74	3	5	1	1	2	4	5	9

Figura 3.6: Exemplo de direcionamento de *threads* em um conjunto de dados.

3.3.5 Funções Atômicas

Uma função atômica é uma operação que é executada por uma *thread* sem a interferência de outras *threads*. Em CUDA, uma função atômica executa uma operação de leitura ou escrita em endereços de memória localizados na memória global e na memória compartilhada, sem permitir que uma outra *thread* acesse um endereço de memória até que a operação esteja completa (NVIDIA, 2010a).

Uma limitação importante consiste que, em CUDA, as funções atômicas não possuem suporte para operações de ponto-flutuante. Desta forma, não serão utilizadas nesse trabalho. Para operações com inteiros, existem algumas funções, tais como *atomicAdd()*, *atomicSub()* e *atomicInc()*. Maiores informações sobre as funções atômicas existentes podem ser encontradas em *NVIDIA CUDA: Reference Manual* (NVIDIA, 2010a).

```
01. __global__ void kernel() {
02.     ...
03.     __syncthreads();
04.     ...
05. }
```

Figura 3.7: Exemplo de utilização da função *__syncthreads()*.

3.4 Declaração de Variáveis

A CUDA permite que as variáveis do programa sejam declaradas nos diferentes tipos de memória disponíveis na placa gráfica. A escolha do tipo de memória permite à aplicação determinar o escopo, o tempo de vida e a velocidade de acesso para as variáveis declaradas (KIRK; HWU, 2010). As variáveis podem ser de quatro tipos: variáveis locais, variáveis compartilhadas, variáveis globais e variáveis constantes.

3.4.1 Variáveis Locais

As variáveis locais são declaradas dentro das funções de *kernel*, sendo que dentro do *kernel*, as variáveis locais são o tipo *default*, ou seja, não é necessária a utilização de nenhuma palavra-chave para a identificação das mesmas.

As variáveis locais podem ser divididas em dois tipos: variáveis escalares e variáveis de *array*. De acordo com o tipo, as variáveis locais são tratadas de maneira diferente no momento da alocação de memória. As variáveis escalares são alocadas em registradores da GPU, de modo que possam ser acessadas com baixa latência. Já as variáveis de *array* são armazenadas na memória global, apresentando uma latência maior.

As variáveis locais de uma *thread* não podem ser acessadas pelas demais *threads* do *kernel*. Quando uma variável é declarada dentro de uma função de *kernel*, são geradas cópias privadas para todas as *threads*. No momento que uma *thread* é finalizada, todas suas variáveis locais são desalocadas.

3.4.2 Variáveis Compartilhadas

A memória compartilhada é implementada nos SMs e possibilita que as variáveis sejam acessadas com baixa latência e de uma maneira massivamente paralela pelas *threads* de um TB. As variáveis compartilhadas devem ser declaradas dentro de funções executadas pelo *kernel*, sendo que a declaração de variáveis é precedida pela palavra-chave *__shared__*.

O escopo das variáveis compartilhadas é um TB, isto é, todas as *threads* do TB podem acessar a variável compartilhada. Uma cópia privada é gerada para cada TB durante a execução do *kernel*. O tempo de vida desse tipo de variável está relacionado à execução do *kernel*, de modo que a memória compartilhada é desalocada quando todas as *threads* do TB terminarem as suas execuções.

As variáveis compartilhadas são um meio eficiente para que as *threads* de um mesmo TB cooperem entre si e compartilhem os resultados intermediários de sua execução. Além disso, os programadores CUDA utilizam a memória compartilhada para armazenar dados da memória global que são usados durante a execução do *kernel* com o intuito de diminuir a latência de acesso aos dados.

3.4.3 Variáveis Globais

As variáveis localizadas na memória global devem ser declaradas fora dos funções executadas em um *kernel*. Essas variáveis são alocadas pelo *host*, mas também podem ser lidas e escritas durante a execução de um *kernel*. A declaração das variáveis globais é precedida pela palavra-chave *__device__*.

O escopo das variáveis globais é geral, isto é, estas variáveis podem ser acessadas pelas *threads* de todos os *grids*. O tempo de vida de uma variável global consiste até o término da execução da aplicação. As variáveis globais são frequentemente utilizadas para o compartilhamento de informações de uma invocação *kernel* para outra invocação de *kernel*. Por serem alocadas em memória *off-chip*, o acesso às variáveis demanda alta latência.

3.4.4 Variáveis Constantes

As variáveis constantes são armazenadas na memória global da GPU. Assim, a forma de alocação, o escopo e o tempo de vida destas variáveis são os mesmos das variáveis globais. A declaração das variáveis constantes é precedida pela palavra-chave *__constant__*.

De forma a diminuir o tempo de acesso às variáveis constantes, uma *cache* foi projetada na GPU, evitando assim a necessidade de acessos à memória global. A limitação imposta pela utilização dessa *cache* consiste que, durante a execução de um *kernel*, as variáveis constantes são acessadas apenas em modo de leitura. Ademais, o total de memória constante disponível é limitado a 64 *Kbytes* em uma NVIDIA G80.

As variáveis constantes são utilizadas para reduzir o acesso à memória global. Os dados que são lidos frequentemente dentro do *kernel*, mas que não são modificados pelos mesmos, são adequados para serem armazenados na memória constante (KIRK; HWU, 2010).

3.5 Programa exemplo

Os exemplos de códigos apresentados nessa seção foram retirados de (BUCK et al., 2008). Nele, deseja-se computar o resultado de $y = ax + y$, sendo que a é um escalar e x e y são vetores compostos de N números de ponto-flutuante. Essa é uma operação muito frequente em problemas compostos de operações de álgebra linear.

O código da Figura 3.8 consiste de uma função que computa serialmente a operação $y = ax + y$. O código serial consiste de um *loop* onde, a cada iteração, um elemento do vetor y é computado individualmente. Uma vez que cada iteração é independente das demais, as mesmas podem ser executadas em paralelo.

```
01. void saxpy (int N, float a, float *x, float *y) {
02.     int i = 0;
03.     for (i = 0; i <=N; i++)
04.         y[i] = a*x[i] + y[i];
05. }
```

Figura 3.8: Exemplo de código serial para computar a operação $y = ax + y$.

O código da Figura 3.9 consiste de um programa em CUDA que possui uma função *kernel* que executa a operação $y = ax + y$. Na chamada do *kernel*, é gerado um *grid* com a quantidade necessária de TBs para computar os N elementos de y, sendo que cada TB possui 256 *threads*. Como se está trabalhando com vetores de uma dimensão, não é necessário trabalhar com *grids* e TBs com mais de uma dimensão.

```
01. __global__ void saxpy (int N, float a, float *x, float *y) {
02.     int i = blockIdx.x*blockDim*x + threadIdx.x;
03.     if (i < N)
04.         y[i] = a*x[i] + y[i];
05. }
06.
07. int main() {
08.     ...
09.     int nblocks = (N + 255) / 256;
10.     saxpy<<<nblocks, 256>>>(N, 2.0, x, y);
11.     ...
12. }
```

Figura 3.9: Exemplo de código em CUDA para computar a operação $y = ax + y$.

As versões serial e paralela da função *saxpy()* são similares. Na versão em CUDA, o *loop* da versão serial é eliminado, pois cada iteração do *loop* torna-se uma *thread*, sendo que cada *thread* é direcionada a um elemento através da combinação dos identificadores da *thread* e do TB. Neste caso, um único elemento de saída é atribuído para cada *thread*, evitando-se a necessidade de qualquer sincronização entre as *threads*.

O comando *if* da linha 3 é utilizado para evitar acessos fora do vetor y. Isto acontece quando a quantidade de elementos a serem computados não é divisível pelo número de *threads* de cada TB (BUCK et al., 2008).

4 SISTEMAS DE EQUAÇÕES LINEARES

A simulação numérica é uma ferramenta essencial em uma grande variedade de aplicações, pois essa permite a realização de experimentos virtuais (através de computadores) em circunstâncias onde o experimento real é inviável. Muitas dessas simulações podem ser modeladas por meio de sistemas de Equações Diferenciais Parciais (EDPs). Em diversos casos, como não é possível encontrar uma solução analítica para essas EDPs, são utilizados métodos numéricos para calcular uma aproximação desta solução. Na discretização de uma EDP, o domínio é dividido em um conjunto finito de pontos que, interligados, formam uma malha. A EDP é então aproximada em cada um dos pontos da malha, resultando um sistema de equações lineares (CAVALHEIRO; PASIN, 2003). O método do Gradiente Conjugado é um método iterativo para a solução de sistemas de equações lineares, geralmente aqueles provenientes da discretização destas equações diferenciais parciais (CUNHA, 2000).

Nesse capítulo, é apresentado o contexto dos sistemas de equações lineares. Na Seção 4.1, é descrito como os problemas físicos são mapeados em termos de EDPs. Na Seção 4.2, são apresentados o conceito e os tipos de matrizes esparsas, que são resultantes da discretização da maioria dos problemas mapeados como sistemas de EDPs. A Seção 4.3 é dedicada aos sistemas de equações lineares e aos tipos de métodos existentes para a resolução desses sistemas. Finalmente, na Seção 4.4, é apresentada uma descrição dos principais conceitos e do funcionamento do método do Gradiente Conjugado, cujo o algoritmo será implementado nesse trabalho.

4.1 Sistemas de Equações Diferenciais Parciais

Equações diferenciais parciais são equações cujas incógnitas são funções de duas ou mais variáveis. Vários fenômenos físicos são descritos matematicamente através de EDPs. Isto ocorre porque muitas entidades físicas (tais como pressão e temperatura) são funções com mais de uma variável e as taxas de variação dessas entidades são representadas por suas derivadas parciais (CUNHA, 2000).

Tais fenômenos tratam-se de problemas de condições iniciais e de condições de

contorno, definidos em domínios contínuos. Uma vez que a solução analítica de problemas de EDP é muitas vezes inviável, métodos numéricos são utilizados para calcular uma solução aproximada das equações em um domínio discreto, ou seja, representado por um conjunto de pontos (CAVALHEIRO; PASIN, 2003).

O tratamento computacional de um problema modelado por uma EDP passa pela discretização do domínio físico, que é realizada a partir da definição de uma malha. Uma malha é fundamentalmente representada por uma lista de N pontos e uma lista que expressa a interconexão entre estes pontos. A solução da EDP é então aproximada em cada um dos N pontos da malha, resultando em um sistema de N equações. Geralmente, a solução de cada ponto depende apenas da solução em pontos vizinhos na malha, fazendo com que os sistemas de equações resultantes da discretização sejam caracterizados por uma matriz esparsa (CAVALHEIRO; PASIN, 2003).

Segundo a topologia de interconexão, as malhas podem ser classificadas como estruturadas ou não estruturadas (CAVALHEIRO; PASIN, 2003). As Figuras 4.1 e 4.2 ilustram, respectivamente, um exemplo de uma malha estruturada e de uma malha não-estruturada. Nas malhas estruturadas, o esquema de interconexão é tal que, basta conhecer o índice do ponto, para obter-se sua posição na malha. Já nas malhas não-estruturadas, o esquema de interconexão de pontos pode ser de qualquer tipo. As malhas não-estruturada são mais genéricas, permitindo assim o recobrimento de domínios com geometrias mais complexas.

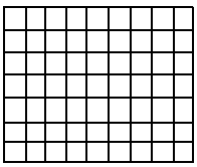


Figura 4.1: Exemplo de uma malha estruturada.

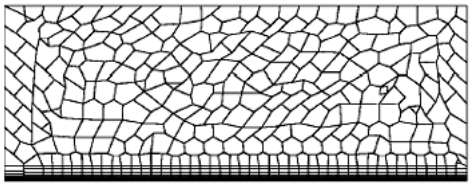


Figura 4.2: Exemplo de uma malha não estruturada.

As características das malhas e os métodos de discretização estão ligados ao problema físico a ser resolvido e às suas características específicas. Consequentemente, a natureza do problema e também a qualidade numérica condicionam a escolha do tipo de malha. Além disso, quanto maior a densidade de pontos na malha gerada, maior será a precisão da solução da EDP, bem como o tempo necessário para processá-lo (CAVALHEIRO; PASIN, 2003). Entre os métodos de discretização mais populares, estão os métodos de diferenças finitas (MDF) e os métodos de elementos finitos (MEF). Uma vez que os métodos de discretização não são o foco do trabalho, os mesmos não serão abordados. Maiores informações sobre métodos de discretização de EDPs podem ser encontradas em (DOUGLAS; HAASE; LANGER, 2003).

4.2 Matrizes Esparsas

Uma fonte substancial de sistemas de equações lineares é a solução de EDPs por meio dos métodos de discretização, como MDF ou MEF. Em geral, os sistemas que resultam da discretização das EDPs são de grande porte e esparsos (CAVALHEIRO; PASIN, 2003).

Uma matriz é denominada esparsa quando a maioria dos seus elementos são nulos (zeros). De uma forma geral, existem dois tipos de matrizes esparsas: estruturadas e desestruturadas (SAAD, 2000). Uma matriz estruturada é aquela cujos os elementos não-nulos formam um padrão regular, sendo que estes estão posicionados frequentemente nas diagonais próximas à diagonal principal (matriz banda). Já uma matriz desestruturada possui seus elementos não-nulos de uma forma irregular. As Figuras 4.3 e 4.4 mostram, respectivamente, um exemplo de uma matriz estruturada e de uma matriz desestruturada. Os pontos pretos correspondem aos elementos não-nulos das matrizes.

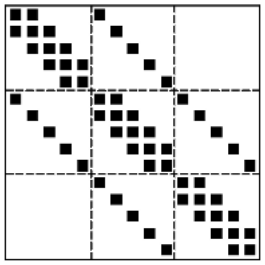


Figura 4.3: Exemplo de uma matriz estruturada.

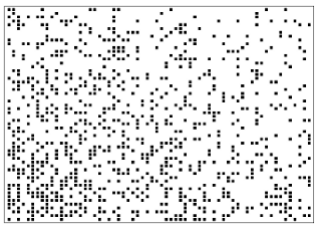


Figura 4.4: Exemplo de uma matriz desestruturada.

A esparsabilidade de uma matriz proporciona otimizações tanto no armazenamento quanto nas operações de álgebra linear, uma vez que serão considerados apenas os elementos não nulos da matriz. Existem diversos formatos para otimizar o armazenamento. Esses se baseiam somente no armazenamento dos elementos não-nulos e, assim, diminuindo a quantidade de memória necessária para o armazenamento da matriz. Pode-se citar, por exemplo, os formatos de armazenamento por linhas ou colunas esparsas comprimidas, linha esparsa modificada, formato coordenado, formato diagonal, representação por listas, etc. Para cada um deles, as operações de álgebra linear adquirem particularidades. Na escolha da estrutura de armazenamento, deve-se levar em consideração primeiramente o tipo da matriz, bem como o modelo de arquitetura computacional que será utilizado e a maneira em que o algoritmo será afetado pela utilização dessa estrutura (CANAL, 2000). O formato utilizado nesse trabalho é o por linhas esparsas comprimidas (CSR) e será descrito detalhadamente no Capítulo 5. Maiores informações sobre formatos de armazenamento de matrizes esparsas podem ser encontradas em (SAAD, 2000).

4.3 Resolução de Sistemas de Equações Lineares

Os sistemas de equações lineares podem ser representados matricialmente na forma $Ax = b$, onde A é a matriz dos coeficientes, x é o vetor de incógnitas e b é o vetor de termos independentes (CAVALHEIRO; PASIN, 2003). A resolução destes sistemas pode ser efetuada através de diferentes métodos. Tipicamente, estes métodos estão agrupados em duas categorias: métodos diretos e métodos iterativos (CANAL, 2000).

4.3.1 Métodos Diretos

Os métodos diretos são aqueles que conduzem à solução exata (salvo erros de arredondamento) do sistema de equações lineares após um número finito de passos. Basicamente, os métodos diretos trabalham em duas etapas: primeiro reorganiza-se a matriz dos coeficientes (geralmente triangularizando-a) e depois faz-se a substituição das variáveis (CANAL, 2000). Entre os métodos diretos mais conhecidos, estão os métodos de Eliminação de Gauss e Gauss-Jordan.

A principal vantagem dos métodos diretos está na sua precisão e estabilidade numérica. Entretanto, a complexidade destes métodos em nível computacional restringe sua utilização em sistemas de grande porte, cujo as dimensões são maiores que 5000 (CAVALHEIRO; PASIN, 2003). Além disso, são realizadas operações elementares entre linhas e colunas que tendem a destruir a estrutura esparsa das matrizes. Por conseguinte, as matrizes tornam-se mais densas do que a matriz original. Devido a este fato, torna-se difícil encontrar um formato de armazenamento que otimize a ocupação de espaço (CANAL, 2000).

Além disso, tais métodos são difíceis de serem paralelizados (CANAL, 2000). Desta forma, a utilização dos métodos diretos não é aconselhável no contexto desse trabalho, que tem como base uma implementação paralela que é executada na GPU. Maiores informações sobre métodos diretos podem ser encontradas em (CUNHA, 2000).

4.3.2 Métodos Iterativos

Diferentemente dos métodos diretos, os métodos iterativos calculam gradualmente a solução até que um critério de parada seja alcançado. Partindo-se de uma solução inicial arbitrária, a cada iteração, a solução do sistema é aproximada da solução final de acordo com as regras do método. Os critérios de condição de parada podem estar relacionados a um número máximo de iterações, à quantidade correta de algarismos significativos da solução aproximada e ao limite de tolerância do erro. Geralmente, o limite de tolerância do erro é o critério mais utilizado (CANAL, 2000).

Diferentemente dos métodos diretos, os métodos indiretos não destroem a espar-

sidade da matriz. Por esta característica, sua utilização é mais adequada para a resolução de sistemas esparsos de grande porte (CANAL, 2000). Uma outra vantagem dos métodos iterativos é que estes envolvem operações simples como produtos matriz-vetor, produtos vetoriais e produtos escalares. Tais operações podem ser facilmente paralelizadas em um ambiente computacional, possibilitando a redução de tempo necessário para encontrar-se a solução do sistema linear (CAVALHEIRO; PASIN, 2003).

Pode-se dividir os métodos iterativos em duas categorias: métodos estacionários e métodos não-estacionários (CANAL, 2000). Nos métodos estacionários, cada iteração não envolve os resultados da iteração anterior e esses métodos manipulam as variáveis do sistema linear através de operações elementares entre linhas e colunas da matriz. Alguns exemplos de métodos estacionários são: Jacobi, Gauss-Seidel e SOR. Já os métodos não-estacionários, por outro lado, trabalham sob a ótica da minimização da função quadrática ou por projeção, manipulando vetores e matrizes inteiros e incluindo hereditariedade em suas iterações. Entre exemplo de métodos não-estacionários, destaca-se o método do Gradiente Conjugado (GC) e o método do Resíduo Mínimo Generalizado (GMRES).Esse trabalho consiste na implementação do método do Gradiente Conjugado. Maiores informações sobre métodos iterativos para solução de sistemas de equações lineares podem ser encontradas em (CUNHA, 2000), (GOLUB; Van Loan, 1991) e (SAAD, 2000).

4.4 Método do Gradiente Conjugado

O método do Gradiente Conjugado (GC) é um método iterativo e não-estacionário, sendo considerado o método iterativo mais eficiente para a resolução de sistemas lineares de grande porte e esparsos, onde a matriz é simétrica definida-positiva (SDP). O GC enquadra-se na classe de métodos do subespaço de Krylov, uma vez que se baseia na minimização da função quadrática e não na intersecção de hiperplanos como os métodos iterativos clássicos (CANAL, 2000).

O objetivo desse trabalho consiste na implementação do método do GC utilizando *GPU Computing*. Para isso, apenas o entendimento do algoritmo desse método faz-se necessário. A demonstração e a comprovação do modelo matemático não são o foco do trabalho, portanto essa seção limita-se a uma descrição dos principais conceitos e do funcionamento do método do GC. Uma abordagem mais detalhada sobre o método do GC pode ser encontrada em (SHEWCHUK, 1994).

Quando as equações matriciais de um sistema $Ax = b$ são abertas, verifica-se que *F(x)* é uma função quadrática nas componentes do vetor x. O gráfico de *F(x)* é uma parabolóide e esta função possui um único mínimo. O mínimo da função é atingido no vetor x que anula o gradiente da função (CUNHA, 2000). O GC parte

do princípio de que o gradiente, que é um campo vetorial, aponta sempre na direção de crescimento máximo da função quadrática. A Figura 4.5 ilustra o gráfico de uma função quadrática. Geometricamente, o gradiente é zero na base da parabolóide (mínimo) e consiste na solução do sistema linear (CANAL, 2000).

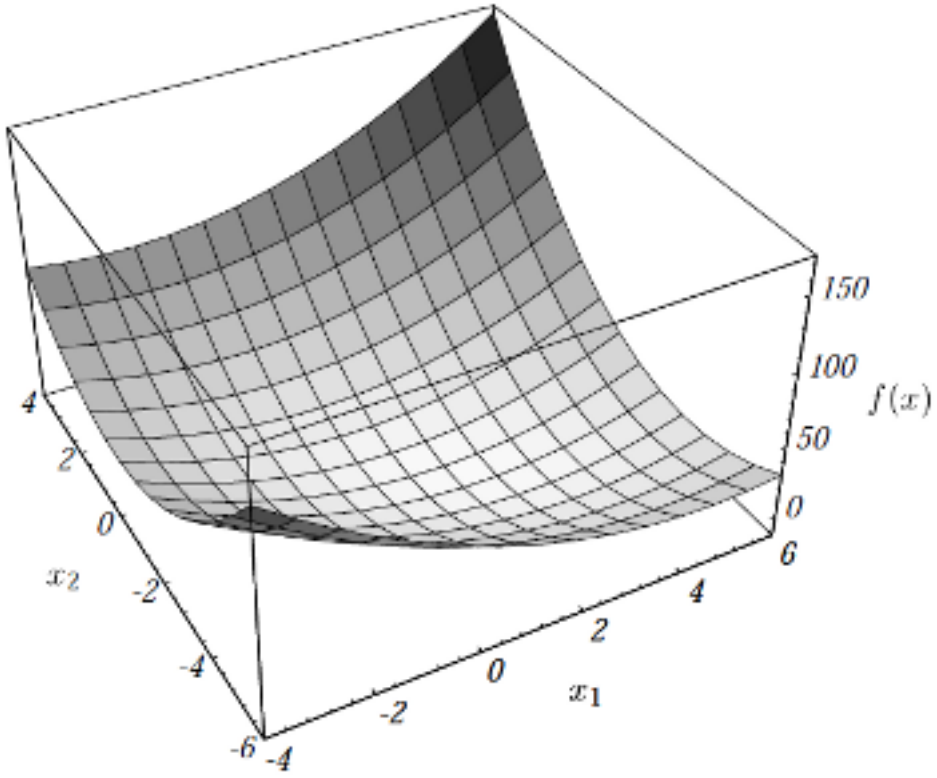


Figura 4.5: Exemplo de função quadrática.

Uma vez que o vetor gradiente aponta para a direção de crescimente da função, é natural que, no passo de busca do ponto mínimo da função, siga-se na direção contrária do gradiente, sendo que a cada passo, calcula-se um resíduo que indica o quão distante se está da solução do sistema de equações. O resíduo é usado para construir as direções de pesquisa, sendo que a cada nova aproximação herda-se informações dos resíduos das iterações anteriores (CUNHA, 2000).

Simplificadamente, a idéia do método consiste em, dada uma solução inicial arbitrária, caminhar na direção oposta a do gradiente, de forma que uma direção já pesquisada não seja repetida até encontrar o mínimo estrito e global (CUNHA, 2000). Esta minimização ocorre sobre certos espaços de vetores, gerados a partir dos resíduos das iterações anteriores, chamados de espaços de Krylov (subespaços de pesquisa) (CANAL, 2000).

Na primeira aproximação do método do GC, quando não se tem uma boa estimativa para x, faz-se a aproximação inicial com $x = 0$. Geralmente há dois critérios de parada para o método: quando um número máximo de iterações for atingido

ou por meio da norma do resíduo (quando a norma do resíduo for menor que uma determinada fração da norma do resíduo inicial) (CANAL, 2000).

A Figura 4.6 apresenta o algoritmo do método do GC. Este método é formado basicamente de operações entre vetores e matrizes, como, por exemplo, soma e produto escalar de vetores e multiplicação matriz-vetor. Desta forma, é adequado para ser executado em um ambiente paralelo, mais especificamente na GPU (CAVALHEIRO; PASIN, 2003; BOYD, 2008).

Defina i_{max} e ε
$i = 0$
$r = b - Ax$
$d = -r$
$\delta_{novo} = r^T r$
$\delta_0 = \delta_{novo}$
Enquanto $i < i_{max}$ e $\delta_{novo} > \varepsilon^2 \delta_0$
    $q = Ad$
    $\alpha = \delta_{novo} / d^T q$
    $x = x + \alpha d$
    Se i é divisível por 50
        $r = b - Ax$
    Senão
        $r = r - \alpha q$
    $\delta_{velho} = \delta_{novo}$
    $\delta_{novo} = r^T r$
    $\beta = \delta_{novo} / \delta_{velho}$
    $d = r + \beta d$
    $i = i + 1$

Figura 4.6: Algoritmo do método do Gradiente Conjugado.

O método do GC geralmente converge em no máximo N iterações, sendo N a quantidade de linhas e colunas da matriz. Entretanto, dependendo da matriz, o sistema pode convergir em um número de iterações maior que N ou pode não convergir. Nestes casos, a convergência pode ser acelerada com o uso de um pré-condicionador (CANAL, 2000). O método do GC com pré-condicionador é chamado Gradiente Conjugado Pré-Condicionado. Embora o número de passos para a convergência das matrizes utilizadas nesse trabalho seja superior a N, nenhum pré-condicionador será utilizado durante a implementação. Maiores informações sobre o uso de pré-condicionadores podem ser encontradas em (CANAL, 2000), (CUNHA, 2000) e (SHEWCHUK, 1994).

5 MÉTODO DO GRADIENTE CONJUGADO COM CUDA

Neste capítulo, é abordada a implementação do método do GC utilizando CUDA. Na Seção 5.1, é descrito o formato utilizado para o armazenamento das matrizes. Após, na Seção 5.2, é apresentada a estrutura da implementação desenvolvida. Na Seção 5.3, são detalhadas as implementações das principais operações de álgebra linear que compõem o método do GC. Na Seção 5.4, é descrita a utilização da memória constante da GPU. Por fim, na Seção 5.5, é explicado o particionamento dos recursos na GPU no momento de sua execução.

5.1 Formato para armazenamento de matrizes esparsas

Nesse trabalho, é utilizado o formato de linhas esparsas comprimidas (CSR - *Compressed Sparse Row*). O formato CSR é provavelmente o mais popular para o armazenamento de matrizes esparsas irregulares (SAAD, 2000). Para uma melhor compreensão desse formato, é válido iniciar a descrição por um formato mais simples.

O modelo mais simples de armazenamento para matrizes esparsas é conhecido como formato de coordenadas. A Figura 5.1 ilustra como uma matriz esparsa de dimensões 5x5 é armazenada neste formato. A estrutura de dados consiste em três vetores: um vetor (AA) que contém o valor dos elementos não-nulos em qualquer ordem e dois vetores que possuem os índices da linha (JR) e da coluna (JC) dos elementos não-nulos da matriz esparsa. Consequentemente, o tamanho de cada vetor é a quantidade de elementos não-nulos.

Na matriz da Figura 5.1, os elementos são listados em uma ordem aleatória. Se os elementos forem listados por linhas, o vetor JC passa a conter informações reduntantes e pode ser substituídos por um vetor que identifica o início de cada linha dos vetores AA e JA. Assim, a matriz apresentada na Figura 5.1 pode ser armazenada no formato CSR, como ilustra a Figura 5.2. A nova estrutura de dados possui três vetores com as seguintes funções:

- Vetor AA: contém os valores armazenados linha a linha, de 0 a N. O tamanho

$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

AA	12.	9.	7.	5.	1.	2.	11.	3.	6.	4.	8.	10.
JR	5	3	3	2	1	1	4	2	3	2	3	4
JC	5	5	3	4	1	4	4	1	1	2	4	3

Figura 5.1: Exemplo de matriz esparsa armazenada no formato de coordenadas.

de AA é a quantidade de elementos não-nulos.

- Vetor JA: equivalente ao vetor JC, contém os índices das colunas dos elementos armazenados no vetor AA. O tamanho de JA é a quantidade de elementos não-nulos.

- Vetor IA: contém os apontadores para o início de cada linha da matriz nos vetores AA e JA. Assim, o conteúdo de *IA(i)* é a posição nos vetores AA e JA onde a i-nézima linha começa. O tamanho de IA é $n + 1$, onde a posição *IA(n + 1)* contem *IA(1)* o número de elementos não nulos, ou seja, o endereço em AA e JA do início de uma linha fictícia número $n + 1$.

AA	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
JA	1	4	1	2	4	1	3	4	5	3	4	5
IA	1	3	6	10	12	13						

Figura 5.2: Exemplo de matriz armazenada com o formato CSR.

Existem variações do formato CSR, sendo que a mais óbvia, conhecida como formato de colunas esparsas comprimidas (CSC - *Compressed Sparse Column*), consiste em armazenar as colunas ao invés das linhas. Maiores informações podem ser encontradas em (SAAD, 2000).

As matrizes utilizadas para os testes de avaliação de desempenho estão armazenadas em arquivos texto, sendo que as matrizes estão representadas no formato de coordenadas. Cada linha do arquivo possui um elemento não-nulo, com sua respectiva posição na matriz (em termos de colunas e linhas). Nesses arquivos, uma vez

que essas matrizes são simétricas, são armazenados apenas os elementos que estão abaixo da diagonal principal. Antes da execução do método do GC, as matrizes são convertidas para o formato CSR. Maiores informações sobre as matrizes utilizadas serão dadas na Seção 6.1.

5.2 Estrutura da implementação

Essa seção tem por objetivo explicar a estrutura do método do GC desenvolvido em CUDA. Esse foi implementado baseado no algoritmo sequencial ilustrado na Figura 4.6. A implementação desenvolvida em CUDA está presente no anexo desse trabalho.

Antes do ínicio das iterações, é necessária uma etapa de preparação da GPU. Primeiramente, é alocada memória na placa gráfica para o armazenamento da matriz e dos vetores utilizados. A seguir, os dados da matriz são copiados para a memória da placa e, por fim, os vetores são inicializados por meio de uma função de *kernel*.

A próxima etapa consiste na realização das iterações do método do GC e, por conseguinte, a solução do sistema seja obtida. O controle das iterações é realizado pela CPU. Cabe à CPU chamar as funções de *kernel*, que são responsáveis por todas as operações do algoritmo. A cada iteração, deve-se testar se a solução do sistema foi atingida em relação à tolerância de erro definida. Para tanto, essa informação deve ser copiada da memória global da GPU para a memória principal, de forma que a CPU faça o controle das iterações.

Quando a precisão da solução for atingida, o controle de iterações é encerrado. Outro critério que parada que pode ser utilizado é estabelecer um número máximo de iterações. Após a execução do método do GC, os dados correspondentes a solução do sistema são copiados para a memória principal. Por fim, o espaço de memória utilizado pela GPU é desalocado.

5.3 Paralelização das Operações de Álgebra Linear

O algoritmo sequencial do método do GC é composto basicamente de operações de álgebra linear. Assim, a paralelização do método pode ser realizada a partir da paralelização de suas operações básicas (CANAL, 2000). Nessa seção, serão detalhadas as implementações dessas operações em CUDA. Mais especificamente, serão detalhadas as operações de soma e a subtração de vetores, a multiplicação de escalar por vetor, a multiplicação matrix-vetor e o produto escalar.

5.3.1 Soma e Subtração de Vetores

Para a realização da soma ou subtração de dois vetores em CUDA, podem ser geradas *N threads*, sendo que *N* é o tamanho dos vetores. Essas *threads* são agrupadas em diferentes TB. Uma vez que cada operação sobre um elemento é independente das demais, cada *thread* é direcionada para um elemento do vetor. Assim, um único elemento é atribuído para cada *thread*, evitando-se a necessidade de sincronização entre as *threads* (BUCK et al., 2008). O código da Figura 5.3 ilustra a função de *kernel* em CUDA que realiza a soma de dois vetores e armazena o resultado em um novo vetor.

```
01. __global__ void somaVetores (int N, float *x, float *y, float *z) {
02.     int i = blockIdx.x * blockDim.x + threadIdx.x;
03.     if (i < N)
04.         z[i] = x[i] + y[i];
05. }
```

Figura 5.3: Exemplo de um *kernel* em CUDA para a soma de dois vetores.

Na linha 2, têm-se a definição da posição do vetor que cada *thread* irá calcular. Já na linha 3, há uma estrutura condicional de forma a assegurar que o índice calculado esteja dentro dos limites do vetor. Isso se faz necessário uma vez que a a quantidade de *threads* criadas durante a execução do *kernel* pode não ser idêntica ao tamanho dos vetores. A linha 4, por fim, realiza a soma dos vetores x e y, e cujo o resultado é armazenado no vetor z.

5.3.2 Multiplicação de Escalar por Vetor

Para a realização da multiplicação de um escalar por um vetor em CUDA, é utilizada uma estrutura identica à operação de soma e subtração de vetores descrita na Sub-seção 5.3.1. Na Figura 5.4, tem-se o código de uma função em CUDA que implementa a multiplicação de um escalar a por um vetor y , e cujo o resultado é armazenado no vetor z.

```
01. __global__ void multVetEsc (int N, float a, float *y, float *z) {
02.     int i = blockIdx.x * blockDim.x + threadIdx.x;
03.     if (i < N)
04.         z[i] = a * y[i];
05. }
```

Figura 5.4: Exemplo de um *kernel* em CUDA para a multiplicação de um escalar por um vetor.

5.3.3 Multiplicação Matriz-Vetor

A multiplicação matriz-vetor é a operação com maior custo computacional no método do GC. Contudo, uma vez que o produto escalar entre uma linha da matriz e o vetor pode ser computado independentemente, essa operação é facilmente paralelizável (CANAL, 2000). O código da Figura 5.5 ilustra uma função em CUDA que implementa a operação da multiplicação matriz-vetor. O *kernel* calcula a operação $y = Ax$, sendo que A é uma matriz passada por parâmetro no formato CSR (AA, JA, IA), conforme descrito na Seção 5.1.

```
01. __global__ void multMatrizVetor (float *aa, float *ja, float *ia,
02.                                  float *x, float *y, int N) {
03.     int i, j, fim;
04.     float pe;
05.     i = blockIdx.x * blockDim.x + threadIdx.x;
06.     if (i < N) {
07.         pe = 0.0;
08.         fim = ia[ i + 1];
09.         for (j = ia[i]; j < fim; j++)
10.             pe += aa[j] * x[ja[j]];
11.         y[i] = pe;
12.     }
13. }
```

Figura 5.5: Exemplo de um *kernel* em CUDA para a multiplicação matriz-vetor.

Na implementação desenvolvida, cada *thread* é direcionada para o cálculo do produto escalar de uma linha da matriz A com o vetor x. Para a realização desta operação, foram geradas N *threads*, sendo N o número de linhas da matriz. Uma vez que cada elemento do vetor y é calculado é independentemente, não é necessária nenhuma sincronização entre as *threads*.

A posição do vetor a ser calculada por cada *thread* é definida na linha 5. Na linha 6, há uma estrutura condicional de forma a assegurar que o índice calculado esteja dentro dos limites do vetor. Nas linhas 9 e 10, têm-se o laço de repetição responsável por efetuar o produto escalar de uma linha da matriz pelo vetor x, gerando assim um elemeto do vetor y.

5.3.4 Produto escalar

No produto escalar de dois vetores em CUDA, existe a necessidade de comunicação entre as *threads* geradas. A operação de produto escalar torna-se mais complexa em CUDA, uma vez que não existe mecanismo de sincronização das *threads* de diferentes TBs. Além disso, não existem operações atômicas para cálculos envolvendo ponto flutuante (BUCK et al., 2008).

Para a implementação dessa operação, tornou-se necessária a utilização de um algoritmo de redução paralela (KIRK; HWU, 2010; BUCK et al., 2008). Um algoritmo de redução extrai um único valor de um conjunto de elementos (um vetor, por exemplo), que pode ser a soma, o maior valor, o menor valor, etc. entre todos os valores. No caso do produto escalar, utiliza-se uma redução de soma. Em um processo sequencial, a redução pode ser facilmente obtida percorrendo o vetor e acumulando os elementos em uma variável de soma. O algoritmo sequencial termina quando todos os elementos forem visitados (KIRK; HWU, 2010). Em CUDA, cada TB soma uma parte do vetor de entrada, gerando-se somas parciais e, por fim, é efetuada uma operação de redução sobre as somas parciais (BUCK et al., 2008). O exemplo de código da Figura 5.6 ilustra o funcionamento da operação de produto escalar em CUDA. Para o desenvolvimento dessa função, foi necessária a implementação de dois *kernels*, sendo que o segundo pode ser chamado mais de uma vez.

O *kernel prodEscalar_1()* (linha 1) é chamado apenas uma vez. Inicialmente, os vetores *x1* e *x2* estão na memória global. Depois de definida a posição do vetor que cada *thread* irá calcular (linha 3), cada *thread* carrega um elemento dos vetores de entrada e realiza a multiplicação dos mesmos (linha 6). O valor resultante é armazenado no vetor *xt*, que está armazenado na memória compartilhada. Nesta memória, as *threads* de um mesmo TB podem compartilhar os valores parciais calculados, evitando o acesso aos dados diretamente na memória global (BUCK et al., 2008). Na linha 9, a função *__syncthreads()* funciona como uma barreira que assegura que todos as *threads* de um TB tenham armazenado o resultado da multiplicação antes de continuar a execução.

O próximo passo consiste em realizar a operação de redução. Cada iteração do laço de repetição (linha 10) implementa uma etapa da redução. A cada etapa, os elementos do vetor *xt* serão substituídos por somas parciais geradas até que a redução seja finalizada. A estrutura condicional da linha 11 controla quais são as *threads* do TB que possuem os resultados das somas parciais (KIRK; HWU, 2010). Na linha 13, uma segunda função de barreira *__syncthreads()* assegura que todas as somas parciais das iterações anteriores tenham sido geradas antes que as *threads* continuem sua execução.

Depois da última etapa da redução, a soma parcial de cada TB estará na posição 0 do vetor *xt*. Assim, deve-se combinar as somas parciais de todos os TBs no *grid* de modo a obter-se o produto escalar dos vetores. Uma alternativa atrativa seria utilizar uma função atômica de soma. Entretanto, a falta de operações atômicas que suportam operações de ponto flutuante na arquitetura Tesla inviabilizam essa alternativa. Desse modo, cada TB deve escrever sua soma parcial em um segundo vetor localizado na memória global e, então, executar um *kernel* de redução, repetindo o processo até que a sequência de somas seja reduzida a um único valor.

```
01. __global__ void prodEscalar_1(float *x1, float *x2, float *aux, int N) {
02.     int tid = threadIdx.x, s;
03.     int i = blockIdx.x * blockDim.x + tid;
04.     __shared__ float xt[NTHREADS], yt[NTHREADS];
05.     if (i < N)
06.         xt[tid] = x1[i] * x2[i];
07.     else
08.         xt[tid] = 0.0;
09.     __syncthreads();
10.     for (s = blockDim.x / 2; s > 0; s = s / 2) {
11.         if (tid < s)
12.             xt[tid] += xt[tid + s];
13.         __syncthreads();
14.     }
15.     if (tid == 0)
16.         aux[blockIdx.x] = xt[tid];
17. }
18.
19. __global__ void prodEscalar_2(float *x, float *res, int N) {
20.     int tid = threadIdx.x, s;
21.     int i = blockIdx.x * blockDim.x + tid;
22.     __shared__ float xtt[NTHREADS], ytt[NTHREADS];
23.     if (i < N)
24.         xtt[tid] = x[i];
25.     else
26.         xtt[tid] = 0.0;
27.     __syncthreads();
28.     for (s = blockDim.x / 2; s > 0; s = s / 2) {
29.         if (tid < s)
30.             xtt[tid] += xtt[tid+s];
31.         __syncthreads();
32.     }
33.     if (tid == 0)
34.         x[blockIdx.x] = xtt[tid];
35. }
36.
37. int main() {
38.     ...
39.     nroBlocos = (N + (NTHREADS - 1)) / NTHREADS;
40.     prodEscalar_1<<<nroBlocos, NTHREADS>>>(x1, x2, aux, N);
41.     while (nroBlocos > 1) {
42.         nroEle = nroBlocos;
43.         nroBlocos = (nroBlocos + (NTHREADS - 1)) / NTHREADS;
44.         prodEscalar_2<<<nroBlocos,NTHREADS>>>(aux, nroEle);
45.     }
46.     ...
47. }
```

Figura 5.6: Exemplo de um *kernel* em CUDA para a operação de produto escalar.

Isso leva para necessidade de vetores temporários adicionais e chamadas de *kernels* repetitivas (BUCK et al., 2008). A função *prodEscalar_2()* (linha 19) é chamado para realizar a operação de redução por soma. Sua estrutura é identica à função *prodEscalar_1()*, com a diferença que não é necessário multiplicar os elementos de dois vetores antes de iniciar a redução. No método *main*, o laço de repetição da linha 41 controla a chamada sucessivas do *kernel prodEscalar_2()*, sendo que o número de TBs e o tamanho do vetor temporário são tratados nesse processo.

5.4 Utilização da Memória Constante

Em CUDA, umas das técnicas para melhorar o desempenho das aplicações consiste em minimizar o custo de acesso aos dados com baixa latência. Perdas de desempenho podem ocorrer devido ao fato da memória global ter alta latência de acesso aos dados. Para tanto, uma estratégia de desenvolvimento é evitar o acesso à memória global, fazendo-se uso da memória constante sempre que possível (KIRK; HWU, 2010).

No método do GC, a matriz A é a única estrutura cujo os elementos não são modificados durante a execução. Desta forma, somente a matriz pode ser armazenada na memória constante. Uma vez que a implementação armazena as matrizes no formato CSR, serão armazenados os vetores que formam essa estrutura (AA, IA e JA). É importante salientar que a memória constante possui um limite de armazenamento de 64K. Dessa forma, é necessário controlar a quantidade de dados armazenados, de forma de a execução seja realizada corretamente.

Em uma matriz no formato CSR, a quantidade de dados armazenados está diretamente ligada ao seu número de elementos não nulos. O vetor IA armazena dados inteiros e seu tamanho corresponde à dimensão da matriz. O vetor JA também armazena dados inteiros e seu tamanho corresponde ao número de elementos não nulos da matriz. O vetor AA armazenas dados de ponto flutuante e seu tamanho também corresponde ao número de elementos não nulos da matriz. Supondo uma matriz SDP cujo as dimensões possuam o tamanho de 957 e cujo a quantidade de elementos não nulos seja de 4137. Os dois vetores de dados inteiros serão armazenados como *int* de quatro *bytes*, enquanto o vetor de ponto flutuante será armazenado como um *float* de quatro *bytes*. O vetor IA ocupará aproximadamente 3,73KB na memória constante. Os vetores JA e AA ocuparão, 16,10KB cada um. O total necessário neste caso para armazenar toda a matriz na memória constante é de 35,93KB, abaixo do limite de 64KB.

É possível que, dependendo da matriz a ser calculada, a mesma não possa ser armazenada completamente na memória constante. Nesse caso, ela deve ser armazenada na memória global. Supondo uma matriz SDP cujo as dimensões possuam

o tamanho de 1437 e cujo a quantidade de elementos não nulos seja de 34241. Os dois vetores de dados inteiros também serão armazenados como *int* de quatro *bytes*, enquanto o vetor de ponto flutuante também será armazenado como um *float* de quatro *bytes*. Os vetores IA, JA e AA ocuparão, respectivamente, 5,24KB, 133,24KB e 133,24KB. O total necessário neste caso para armazenar toda a matriz na memória constante é de 271,72KB, ultrapassando do limite de 64KB. Nesse caso, apenas o vetor IA poderá ser armazenado na memória constante, enquanto os outros dois vetores terão de ser armazenados na memória global.

5.5 Particionamento dos recursos da GPU

Os recursos de execução em um SM inclui registradores, TBs e *threads*. Estes recursos são particionados e atribuídos dinamicamente para as *threads* durante a execução das mesmas. Na programação com CUDA, isso está relacionado ao número de *threads* geradas em cada TB no momento da chamada de um *kernel* (KIRK; HWU, 2010).

O número padrão de *threads* por TB é de 256. Um exemplo comparando a execução de TBs com tamanhos de 64, 256 e 512 *threads* em uma G80 será utilizado para entender-se o porquê desse número. Com apenas 64 *threads* por TB, é necessário $768/64 = 12$ TBs para ocupar completamente um SM. Contudo, uma vez que cada SM está limitado em 8 TBs, será possível a execução de somente $64 \times 8 = 512$ *threads* e uma parte dos recursos não será utilizada. Com 256 threads, serão gerados $768/256 = 3$ TBs, que estão dentro do limite de oito TBs por SM. Esta é uma boa configuração porque o SM será capaz de executar 768 *threads*. Por fim, com 512 *threads* será gerado apenas $768/512 = 1$ TB. Como as *threads* de um TB devem ser executadas em um único SM, essa configuração também resultará na sub-utilização de recursos (KIRK; HWU, 2010).

Com base no exemplo anterior, os *kernels* desenvolvidos na implementação do método do GC geram TB de 256 *threads*. Em algumas aplicações mais críticas, entretanto, o programador precisa realizar testes com cada uma das alternativa e optar pela configuração de melhor desempenho para cada *kernel*, o que pode ser um trabalho intensivo e um processo tedioso (KIRK; HWU, 2010).

6 AVALIAÇÃO DE DESEMPENHO

Nesse capítulo, são apresentados os resultados obtidos com a implementação do método do GC em CUDA em relação à implementação sequencial equivalente. Na Seção 6.1, são descritas as matrizes utilizadas para os testes na implementação. Já na Seção 6.2, serão apresentadas os recursos de *hardware* e a GPU utilizada para a realização dos testes. Por fim, a Seção 6.3 apresenta uma análise de desempenho com base nos testes realizados.

6.1 Matrizes para teste

Para os testes da implementação desenvolvida, serão utilizadas algumas matrizes obtidas no *Matrix Market* (NIST, 2010). O *Matrix Market* é um depósito virtual de dados para o uso em estudos de algoritmos de álgebra linear. Neste depósito, estão disponíveis quase quinhentas matrizes esparsas oriundas de uma variedade de aplicações científicas e de engenharia. Nesse trabalho, serão utilizadas apenas matrizes do tipo SDP. Isto se deve ao fato de método do GC ser próprio para a solução de sistemas onde a matriz de coeficientes possui essa característica.

Para a realização dos testes de desempenho, serão utilizadas sete matrizes. Para a seleção destas matrizes, considerou-se o número de equações do sistema. A Tabela 6.1 contêm as informações das matrizes de teste. Nela, encontra-se o nome da matriz no *Matrix Market*, o número de equações (corresponde às dimensões da matriz), a quantidade de elementos não-nulos e o tipo de estrutura esparsa (descrito no Capítulo 4). Para a realização dos testes, foram utilizadas as seguintes matrizes: BCSSTK11, BCSSTK13, BCSSTK14, BCSSTK15, BCSSTK17, BCSSTK25 e NOS2. Maiores informações sobre cada uma dessas matrizes podem ser encontradas em (NIST, 2010).

6.2 Recursos de *hardware*

Para a avaliação de desempenho, foi utilizada um computador com processador *Intel(R) Core(TM) 2*. Nele, cada núcleo possui um *clock* de 2.2 GHz, 64 KBytes de

Tabela 6.1: Informações das matrizes de teste.

Nome	Dimensão	Qtde não-nulos	Estrutura
BCSSTK11	1473	34241	Desestruturada
BCSSTK13	2003	83883	Desestruturada
BCSSTK14	1806	63454	Banda
BCSSTK15	3948	117816	Banda
BCSSTK17	10974	428650	Banda
BCSSTM25	15439	15439	Diagonal
NOS2	4137	4137	Diagonal

cache L1 e 2048 KBytes de *cache* L2. A memória RAM é do tipo DDR2 e possui 1 GByte. A GPU utilizada é uma *NVIDIA GeForce GTS 250*, que possui memória DDR3 de 1GB e *clock* de 1836 MHz. O sistema operacional utilizado é o Ubuntu 10.4 com *Kernel* versão 2.6.32-24. A versão do CUDA *Toolkit* utilizado é a 3.2 (NVIDIA, 2010b).

A NVIDIA classifica suas GPUs em termos da capacidade computacional das mesmas (NVIDIA, 2010a). A capacidade computacional é definida por um número no formato *X,x*, onde X e x corresponde ao número de revisão principal e ao número de revisão secundário, respectivamente. As GPUs com o mesmo número de revisão principal possuem o mesmo tipo de arquitetura, sendo 1 para a arquitetura Tesla e 2 para a arquitetura Fermi (NVIDIA, 2010a). Já o número de revisão secundário correspondem a uma melhora incremental na arquitetura dos núcleos, como suporte a novos recursos (tais como suporte a ponto flutuante de precisão dupla). Na arquitetura Tesla, esse número varia de 1,0 a 1,3. A NVIDIA GeForce GTS250, que será utilizada para a avaliação de desempenho, está agrupada nas GPUs de capacidade computacional 1,1, possuindo 16 dezesseis SMs constituidos de oito SPs, totalizando 128 SPs. A Figura 6.1 mostra as principais informações a respeito da GPU utilizada.

6.3 Avaliação de Desempenho

Para realizar a avaliação de desempenho, foi desenvolvido um programa sequencial do método do GC. Esta será comparada com uma versão equivalente desenvolvida utilizando CUDA. O desempenho das duas versões será comparado em termos de tempo de execução e número de iterações. Para o cálculo da média e de desvio padrão, cada sistema foi executado cem vezes. Os respectivos códigos podem ser encontrados nos anexos desse trabalho.

Os sistemas de equações lineares podem ser representados matricialmente na forma $Ax = b$, onde A corresponde às matrizes apresentadas na Seção 5.1. Um resumo das características das matrizes utilizadas pode ser encontrado na Tabela

Número de revisão principal: 1
Número de revisão secundário: 1
Quantidade total de memória global: 1 *GByte*
Número de SMs: 16
Número de SPs por SM: 8
Quantidade total de memória constante: 64 *KBytes*
Quantidade total de memória compartilhada por TB: 16 *KBytes*
Número total de registradores disponíveis por TB: 8192
Tamanho do *warp*: 32
Número máximo de *threads* por TB: 512
Tamanho máximo de cada dimensão de um TB: 512 x 512 x 64
Tamanho máximo de cada dimensão de um *grid*: 65535 x 65535 x 1
Taxa de *clock*: 1.84 GHz

Figura 6.1: Informações da GPU *NVIDIA GeForce GTS250*.

6.1. Para execução do método do GC, todos os elementos do vetor b foram setados com o valor 1. A precisão da solução foi atingida com uma tolerância de erro definida como 10^{-6}.

Os dados foram computados utilizando variáveis de ponto flutuante de precisão simples (*float*). A razão para esta escolha consiste que a GPU GeForce GTS250 não possui suporte para precisão dupla para ponto flutuante (*double*). Isso pode ser considerado um fator limitante para esse trabalho, uma vez que algumas matrizes disponíveis no Matrix Market necessitam ser computadas com precisão dupla.

A Tabela 6.2 contém os resultados do programa sequencial e a Tabela 6.3 contém os resultados do programa implementado com CUDA. Nelas, estão listadas a quantidade de iterações necessárias para a solução do sistema, o tempo médio de execução em segundos e o respectivo desvio padrão. Nota-se que o número de iterações necessário para a obteção das soluções é superior ao tamanho das matrizes, sendo provável que o uso de um pré-condicionador diminuisse o número de iterações.

Tabela 6.2: Estatísticas da execução da implementação sequencial.

Nome	Nro iterações	Tempo médio (seg)	Desvio padrão
BCSSTK11	32367	5,0	0,29
BCSSTK13	491123	175,1	1,03
BCSSTK14	22393	6,1	0,38
BCSSTK15	32800	17,4	0,81
BCSSTK17	40231	78,4	5,99
BCSSTM25	119285	31,6	0,49
NOS2	98434	4,1	0,85

Não foi possível obter a solução do sistema composto pela matriz NOS2 utili-

zando o processamento da GPU. Isto aconteceu, provavelmente, devido às limitações na representação dos resultados em ponto flutuante existentes na GPU utilizada. Observou-se que, a cada iteração, durante a operação de multiplicação matriz-vetor, os valores resultante divergiam cada vez mais daquele obtido na implementação sequencial.

Tabela 6.3: Estatísticas da execução da implementação em CUDA.

Nome	Nro iterações	Tempo médio (seg)	Desvio padrão
BCSSTK11	36652	6,5	0,51
BCSSTK13	599151	169,4	0,50
BCSSTK14	23152	5,6	0,41
BCSSTK15	34188	11,8	0,56
BCSSTK17	46741	44,8	0,43
BCSSTM25	169838	33,6	0,50
NOS2	-	-	-

Através dos resultados obtidos na Tabela 6.2 e na Tabela 6.3, é possível realizar uma análise dos mesmos. A Tabela 6.4 contêm um comparativo entre as duas implementações. Nela, estão listados o percentual de aumento do número de iterações e do percentual de redução do tempo médio necessário para a obtenção da solução do sistema.

Tabela 6.4: Estatísticas da comparação das implementações.

Nome	(%) Aumento iterações	(%) Redução tempo
BCSSTK11	13,24	-30,00
BCSSTK13	22,00	3,26
BCSSTK14	3,39	8,20
BCSSTK15	4,23	32,18
BCSSTK17	16,18	42,86
BCSSTM25	42,38	-6,33
NOS2	-	-

Na implementação com CUDA, o aumento do número de iterações ocorreu em todos os sistemas. Isso se deve também à representação dos valores em ponto flutuante. Assim, foram necessárias mais iterações para a solução atingir a tolerância de erro definida.

Nota-se que a resolução do sistema utilizando a GPU só é eficiente em sistemas de grande porte. Caso contrário, o ganho de desempenho é mínimo e, em alguns casos, é possível ocorrer perda de desempenho. Os sistemas compostos de matrizes com estrutura diagonal também não atingiram bons resultados nessa implementação. Os melhores resultados ocorreram com matrizes do tipo banda e de grande porte, cujo as

reduções do tempo médio de processamento nas matrizes BCSSTK15 e BCSSTK17 foram superiores a 30% e 40%, respectivamente.

Para avaliar apenas o poder de processamento da GPU, foi realizada um outro tipo de avaliação de desempenho. Este consiste em verificar o poder computacional da arquitetura, sem a necessidade em obter-se a solução do sistemas lineares na tolerância de erro definida. O critério de parada da execução do método do GC em CUDA consiste no número de iterações necessárias para a obtenção das soluções nas respectivas implementações sequenciais. A Tabela 6.5 contém os resultados obtidos nessa avaliação. Além das informações contidas na Tabela 6.2, ela contém o percentual de ganho de desempenho dessa execução.

Tabela 6.5: Estatísticas da avaliação do poder computacional da GPU.

Nome	Tempo médio (seg)	Desvio padrão	(%) Redução tempo
BCSSTK11	5,7	0,47	-14,00
BCSSTK13	139,3	1,30	20,45
BCSSTK14	5,2	0,49	14,75
BCSSTK15	11,6	0,49	33,33
BCSSTK17	38,6	0,50	50,77
BCSSTM25	23,5	0,51	25,63
NOS2	-	-	-

Com base nos resultados obtidos, foi possível observar ganhos de desempenho nas matrizes. A redução do tempo de processamento na matriz BCSSTK17, que é grande porte, foi superior a 50%. O tempo de execução no processamento da matriz BCSSTM25, que possui estrutura diagonal e não obteve bons resultados na obtenção da solução do método do GC, foi reduzido em mais de 25%. No entanto, ainda não foi possível obter ganho de desempenho no processamento da matriz BCSSTK11, que é a menor do conjunto de matrizes utilizado.

7 CONCLUSÃO

O objetivo desse trabalho consistiu na implementação do método do Gradiente Conjugado utilizando *GPU Computing*. Para tanto, na parte de revisão bibiográfica, foi apresentado uma descrição da arquitetura NVIDIA Tesla, um estudo sobre a programação em CUDA e o contexto da simulação numérica na resolução de sistemas de equações lineares, mais especificadamente sobre o método do Gradiente Conjugado.

Com base na avaliação de desempenho realizada, foi possível demonstrar o poder computacional das GPUs. Na maioria dos casos, houve ganho de desempenho com a implementação em CUDA. No melhor caso, a matriz BCSSTK17 obteve uma redução do tempo de processamento superior a 50%. Deste modo, no geral os resultados obtidos foram considerados positivos. Entretanto, em determinados casos não houve ganho de desempenho ou o ganho obtido foi quase irrelevante. Além disso, no caso da matriz NOS2, não foi possível obter a solução do sistema linear.

As matrizes utilizadas para os testes consistiram na principal limitação desse trabalho, devido à GPU utilizada. A computação e o armazenamento dos valores de seus elementos seriam mais adequadas utilizando ponto flutuante de precisão dupla, contudo essa GPU não possui suporte a esse tipo de dado. Infelizmente, essa situação foi descoberta em um estágio avançado do trabalho, impossilitando a obtenção de outras matrizes para a realização dos testes.

Existem alguns pontos que poderiam ser melhorarados nesse trabalho. O uso de uma GPU com suporte a ponto flutuante de precisão dupla tornaria a avaliação de desempenho mais precisa, sendo possível comparar os resultados de forma mais abrangente. Isso não foi possível devido à indisponibilidade do uso desse tipo de GPU. Outro aspecto que melhoraria o trabalho seria a implementação de um pré-condicionador no método do GC, que provavelmente reduziria o número de iterações para a obtenção das soluções dos sistemas.

Em relação à programação em *GPU Computing*, embora recente, têm sido utilizada por um número considerável de aplicações na comunidade científica e por algumas empresas. E a tendência é que sejam cada vez mais utilizadas. Isso se deve à popularização das GPUs. Em 2009, mais de 200 milhões de GPUs compatíveis

com CUDA já haviam sido comercializadas (KIRK; HWU, 2010). Essa grande quantidade tornou as GPUs economicamente atrativas para o devenvolvimento de aplicações.

Uma vez que o principal foco das GPUs ainda são o mercado de jogos e aplicações 3D, ainda existem diversas limitações de *hardware* e *software* para o desenvolvimento de aplicações em *GPU Computing*, embora tenha ocorrido um grande avanço a partir de 2007 com o lançamento da arquitetura Tesla e da plataforma CUDA. Para melhorar o modelo e o desempenho das aplicações, a NVIDIA projetou um novo modelo de arquitetura, Fermi, que traz novos recursos e elimina algumas das limitações da arquitetura Tesla. Entre as novas funcionalidades, pode-se destacar avanços na representação e desempenho de ponto flutuante, melhoras na hierarquia de memória *cache* e memória compartilhada, além de funções atômicas com suporte a ponto flutuante. Maiores informações sobre a arquitetura Fermi podem ser encontradas em (NVIDIA, 2010a).

O modelo de programação CUDA possui uma baixa curva de aprendizado para os desenvolvedores, embora esteja diretamento relacionada à arquitetura da GPU. Uma limitação da CUDA é que a mesma não é heterogênea, sendo que seu uso é restrito às GPUs na NVIDIA. Entretanto, o padrão OpenCL (KHRONOS GROUP, 2010) foi desenvolvido e está em processo de maturação. Esse consiste no primeiro padrão aberto para a programação paralela de sistemas heterogêneos, sendo possível a sua utilização em diversos dispositivos de diferentes fabricantes. Como uma alternativa à tecnologia CUDA, destaca-se também o modelo de programação *ATI Stream* (AMD, 2010), da AMD.

7.1 Trabalhos Futuros

Esse trabalho cria algumas opções para trabalhos futuros. Entre elas, pode-se citar:

- Utilização de múltiplas GPUs para processamento, ou seja, o agrupamento destas em um único computador de forma a serem utilizadas de maneira integrada.
- Utilização de *GPU Computing* em *clusters* de computadores. Um exemplo seria a utilização de CUDA em conjunto com o padrão *Message Passing Interface* (MPI).
- Utilização de outras plataformas de desenvolvimento para *GPU Computing*. Como exemplos, pode-se citar as plataformas OpenCL e *ATI Stream*.

REFERÊNCIAS

AMD. **Ati stream technology**. Disponível em: <http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/ Pages/stream-technology.aspx>.
Acesso em: dezembro 2010.

BOYD, C. Data-Parallel Computing. **ACM Queue**, v.6, n.2, p.30–39, março 2008.

BUCK, I.; GARLAND, M.; NICKOLLS, J.; SKADRON, K. Scalable Parallel Programming with CUDA. **ACM Queue**, v.6, n.2, p.40–53, março 2008.

CANAL, A. P. **Paralelização de Métodos de Resolução de Sistemas Lineares Esparsos com o DECK em um Cluster de PCs**. 2000. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul, Instituto de Informática, Porto Alegre, Brasil.

CAVALHEIRO, G. G. H.; PASIN, M. Aplicações de Alto Desempenho. **Anais da Escola Regional de Alto Desempenho**, p.135–168, 2003.

CUNHA, M. **Métodos Numéricos**. 2.ed. Campinas, Brasil: Unicamp, 2000. 289p.

DOUGLAS, C. C.; HAASE, G.; LANGER, U. **A tutorial on elliptic pde solvers and their parallelization**. [S.l.]: SIAM, 2003.

FATAHALIAN, K.; HOUSTON, M. GPUs: A Closer Look. **ACM Queue**, v.6, n.2, p.18–29, março 2008.

GOLUB, G. H.; Van Loan, C. F. **Matrix Computation**. 2.ed. Londres: The John Hopkins University Press, Baltimore, 1991.

GREEN, S.; HOUSTON, M.; LUEBKE, D.; OWENS, J. D.; PHILLIPS, J. C.; STONE, J. E. GPU Computing. **Proceedings of the IEEE**, v.96, n.5, p.879–899, maio 2008.

HARRIS, M. Mapping computational concepts to GPUs. **ACM SIGGRAPH 2005 Courses**, Los Angeles, California, USA, agosto 2005.

KHRONOS GROUP. **Opencl overview**. Disponível em: <http://www.khronos.org/opencl/>. Acesso em: dezembro 2010.

KIRK, D. B.; HWU, W. W. **Programming Massively Parallel Processors**: A Hands-on Approach. 1.ed. Burlington, USA: Elsevier, 2010. 258p.

LINDHOLM, E.; NICKOLLS, J.; OBERMAN, S.; MONTRYM, J. NVIDIA Tesla: A Unified Graphics and Computing Architecture. **IEEE Micro**, v.28, n.2, p.39–55, março 2008.

Microsoft. **Microsoft Directx 11**. Disponível em: <www.microsoft.com/windows/directx/>. Acesso em: dezembro 2010.

NIST. **MATRIZ MARKET**. Disponível em: <http://math.nist.gov/MatrixMarket/>. Acesso em: dezembro 2010.

NVIDIA. **NVIDIA CUDA**: Programming Guide. Disponível em: <http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf>. Acesso em: dezembro 2010.

NVIDIA. **NVIDIA's Next Generation CUDA Compute Architecture**: Fermi. v.1, n.1. Disponível em: <http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>. Acesso em: dezembro 2010.

NVIDIA. **NVIDIA CUDA**: Reference Manual. Disponível em: <http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CudaReferenceManual.pdf.pdf>. Acesso em: dezembro 2010.

NVIDIA. **CUDA Downloads**. Disponível em: <http://developer.nvidia.com/object/cuda_3_0_downloads.html/>. Acesso em: dezembro 2010.

OPENGL. **OpenGL**. Disponível em: <http://www.opengl.org/>. Acesso em: dezembro 2010.

SAAD, Y. **Iterative Methods for Sparse Linear Systems**. 2.ed. Boston, USA: Pws, 2000.

SHEWCHUK, J. R. **An Introduction to the Conjugate Gradient Method Without the Agonizing Pain**. Pittsburgh: Carnegie Mellon Univerity, 1994.

WALDSCHMIDT, M. Auxiliary Functions in Transcendental Number Theory. **The Ramanujan Journal**, v.20, n.3, p.341–373, outubro 2009.

8 ANEXOS

8.1 Implementação do método do Gradiente Conjugado sequencial e utilizando CUDA

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#include <cutil_inline.h>
#include <cuda.h>

#define NTHREADS 256
#define NMAXITER 999999

typedef float real;

void swapI(int* a, int* b) {
        int tmp;
        tmp = *a;
        *a = *b;
        *b = tmp;
}

void swapF(float* a, float* b) {
        float tmp;
        tmp = *a;
        *a = *b;
        *b = tmp;
}

void quickSort(int *pos, int *ra, int *ja, float *aa, int left, int right) {
        int i, j;
        if (right > left) {
                i = left;
                for (j = left + 1; j <= right; ++j) {
                        if (pos[j] < pos[left]) {
                                ++i;
                                swapI(&pos[i], &pos[j]);
                                swapI(&ra[i], &ra[j]);
                                swapI(&ja[i], &ja[j]);
                                swapF(&aa[i], &aa[j]);
                        }
                }
                swapI(&pos[left], &pos[i]);
                swapI(&ra[left], &ra[i]);
                swapI(&ja[left], &ja[i]);
                swapF(&aa[left], &aa[i]);
                quickSort(pos, ra, ja, aa, left, i - 1);
                quickSort(pos, ra, ja, aa, i + 1, right);
        }
}


// Setar vetor b para vetor x ser todo 1
void setarVetorB (float *aa, int *ja, int *ia, float *b, int nroLin, int nroCol) {
        int i,j;
        for (i = 0; i < nroLin; i++) {
                b[i] = 0.0;
                for (j = ia[i]; j < ia[i+1]; j++)
//                      b[i] += aa[j];
                b[i] = 1.0;
        }
}

void escreverResultados(FILE *arq, char *nome, float *a, int n) {
        int i;
        for (i = 0; i < n; i++)
                fprintf (arq,"%f ",a[i]);
        fprintf(arq,"\n");
}
```

```
void escreverVetor(char *nome, float *a, int n) {
	int i;
	printf ("%s: ",nome);
	for (i = 0; i < n; i++)
		printf ("%f ",a[i]);
	printf("\n");
}

__global__ void parte1(float *b, float *x, float *r, float *w, float *z, int N) {
	int i = blockIdx.x*blockDim.x + threadIdx.x;
	if (i < N) {
		x[i] = 0.0; // X = 0
		r[i] = b[i]; // r =  b - Ax (Ax = 0)
		w[i] = -r[i]; // w = -r
		z[i] = 0.0; // z = 0 (para z = A*w)
	}
}

__global__ void parte2(float *aa, int *ja, int *ia, float *w, float *z, float *r,
		float *aux1, float *aux2, int N) {
	int tid = threadIdx.x, s, j, fim;
	int i =blockIdx.x*blockDim.x + tid;
	float pe;
	__shared__ float xt[NTHREADS], yt[NTHREADS];

	if (i < N) {
		pe = 0.0;
		fim = ia[i+1];
		for (j = ia[i]; j < fim; j++)
			pe += aa[j] * w[ja[j]];
		z[i] = pe;
		xt[tid] = r[i]*w[i];
		yt[tid] = w[i]*pe;
	}
	else
		xt[tid] = yt[tid] = 0.0;
	__syncthreads();
	for (s = blockDim.x/2; s > 0; s = s/2) {
		if (tid < s) {
			xt[tid] += xt[tid+s];
			yt[tid] += yt[tid+s];
		}
		__syncthreads();
	}
	if (tid == 0) {
		aux1[blockIdx.x] = xt[tid];
		aux2[blockIdx.x] = yt[tid];
	}

}

__global__ void parte3(float *x, float *y, int N, float *res) {
	int tid = threadIdx.x, s;
	int i = blockIdx.x*blockDim.x + tid;
	__shared__ float xtt[NTHREADS], ytt[NTHREADS];
	if (i < N) {
		xtt[tid] = x[i];
		ytt[tid] = y[i];
	}
	else
		xtt[tid] = ytt[tid] = 0.0;
	__syncthreads();
	for (s = blockDim.x/2; s > 0; s = s/2) {
		if (tid < s) {
			xtt[tid] += xtt[tid+s];
			ytt[tid] += ytt[tid+s];
		}
		__syncthreads();
	}
	if (tid == 0) {
		res[0] = xtt[tid]/ytt[tid];
		x[blockIdx.x] = xtt[tid];
		y[blockIdx.x] = ytt[tid];
	}
}

__global__ void parte4(float *x, float *w, float *r, float *z, float *normaR, float *alpha_d, int N) {
	int tid = threadIdx.x, s;
	int i = blockIdx.x*blockDim.x + tid;
	float res, alpha = alpha_d[0];
	__shared__ float nr[NTHREADS];
	if (i < N) {
		x[i] = x[i] + w[i]*alpha; // x = x + alpha*w
		res = r[i] - z[i]*alpha; // r = r - alpha*z
		r[i] = res;
		nr[tid] = res*res;
	}
	else {
		nr[tid] = 0.0;
	}
	__syncthreads();
	for (s = blockDim.x/2; s > 0; s = s/2) {
		if (tid < s)
			nr[tid] += nr[tid+s];
		__syncthreads();
	}
	if (tid == 0)
		normaR[blockIdx.x] = nr[tid];
}
```

```
__global__ void parte5(float *x, int N) {
        int tid = threadIdx.x, s;
        int i = blockIdx.x*blockDim.x + tid;
        __shared__ float xtt[NTHREADS];
        xtt[tid] = (i < N) ? x[i] : 0.0;
        __syncthreads();
        for (s = blockDim.x/2; s > 0; s = s/2) {
                if (tid < s)
                        xtt[tid] += xtt[tid+s];
                __syncthreads();
        }
        if (tid == 0) {
                x[blockIdx.x] = xtt[tid];
        }
}

__global__ void parte6(float *x1, float *x2, float *y1, float *y2, float *aux1, float *aux2, int N) {
        int tid = threadIdx.x, s;
        int i = blockIdx.x*blockDim.x + tid;
        __shared__ float xt[NTHREADS], yt[NTHREADS];
        if (i<N) {
                xt[tid] = x1[i]*x2[i];
                yt[tid] = y1[i]*y2[i];
        }
        else
                xt[tid] = yt[tid] = 0.0;
        __syncthreads();
        for (s = blockDim.x/2; s > 0; s = s/2) {
                if (tid < s) {
                        xt[tid] += xt[tid+s];
                        yt[tid] += yt[tid+s];
                }
                __syncthreads();
        }
        if (tid == 0) {
                aux1[blockIdx.x] = xt[tid];
                aux2[blockIdx.x] = yt[tid];
        }
}

__global__ void parte7(float *w, float *r, float *beta_d, int N) {
        int i = blockIdx.x*blockDim.x + threadIdx.x;
        if (i < N) {
                w[i] = -r[i] + w[i]*beta_d[0]; // w = -r + w*beta
        }
}

//                float dif = 10e-08;
                float dif = 1.0;

// Variaveis da matriz CSR em CUDA
__constant__ int *ia_d;
int *ja_d;
float *aa_d;

void gradienteConjugado(float *aa, int *ja, int *ia, float *b, float *x, float E,
                int *nro_iter, int N, int nroNaoNulos, int nromaxiter) {

        // Declaracao de variaveis
        float *b_d, *x_d, *r_d, *w_d, *z_d, *aux1_d, *aux2_d, *normaR_d; // variaveis CUDA
        float *alpha_d, *beta_d;
        float normaR, alpha = 0.0, beta = 0.0;
        int i, nroBlocos, nroEle;

        // Alocar memoria para vetores na GPU
        cudaMalloc ( (void**) &aa_d, nroNaoNulos*sizeof(float));
        cudaMalloc ( (void**) &ja_d, nroNaoNulos*sizeof(int));
        cudaMalloc ( (void**) &ia_d, (N+1)*sizeof(int));
        cudaMalloc ( (void**) &b_d, N*sizeof(float));
        cudaMalloc ( (void**) &x_d, N*sizeof(float));
        cudaMalloc ( (void**) &r_d, N*sizeof(float));
        cudaMalloc ( (void**) &w_d, N*sizeof(float));
        cudaMalloc ( (void**) &z_d, N*sizeof(float));
        cudaMalloc ( (void**) &aux1_d, N*sizeof(float));
        cudaMalloc ( (void**) &aux2_d, N*sizeof(float));
        cudaMalloc ( (void**) &normaR_d, N*sizeof(float));
        cudaMalloc ( (void**) &alpha_d, sizeof(float));
        cudaMalloc ( (void**) &beta_d, sizeof(float));

        // Tranferência de dados da CPU para GPU
        cudaMemcpy(aa_d, aa, nroNaoNulos*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(ja_d, ja, nroNaoNulos*sizeof(int), cudaMemcpyrHostToDevice);
        cudaMemcpy(ia_d, ia, (N+1)*sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(b_d, b, N*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(x_d, x, N*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(alpha_d, &alpha, sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(beta_d, &beta, sizeof(float), cudaMemcpyHostToDevice);

        // Parte 1
        parte1<<<(N+255)/256,256>>>(b_d, x_d, r_d, w_d, z_d, N);

        // Laco de controle de numero de iteracoes
        for (i = 0; i <= nromaxiter; i++) {

                nroBlocos = (N+(NTHREADS-1))/NTHREADS;
                parte2<<<nroBlocos, NTHREADS>>>(aa_d, ja_d, ia_d, w_d, z_d, r_d, aux1_d, aux2_d, N);
                while (nroBlocos > 1) {
                        nroEle = nroBlocos;
```

```
                        nroBlocos = (nroBlocos+(NTHREADS-1))/NTHREADS;
                        parte3<<<nroBlocos, NTHREADS>>>(aux1_d, aux2_d, nroEle, alpha_d);
                }

                nroBlocos = (N+(NTHREADS-1))/NTHREADS;
                parte4<<<nroBlocos, NTHREADS>>>(x_d, w_d, r_d, z_d, normaR_d, alpha_d, N);
                while (nroBlocos > 1) {
                        nroEle = nroBlocos;
                        nroBlocos = (nroBlocos+(NTHREADS-1))/NTHREADS;
                        parte5<<<nroBlocos, NTHREADS>>>(normaR_d, nroEle);
                }
                cudaMemcpy(&normaR, normaR_d, sizeof(float), cudaMemcpyDeviceToHost);

                // Para se atingir tolerancia minima de erro
                if (sqrt(normaR) < E)
                        break;

                nroBlocos = (N+(NTHREADS-1))/NTHREADS;
                parte6<<<nroBlocos, NTHREADS>>>(r_d, z_d, w_d, z_d, aux1_d, aux2_d, N);
                while (nroBlocos > 1) {
                        nroEle = nroBlocos;
                        nroBlocos = (nroBlocos+(NTHREADS-1))/NTHREADS;
                        parte3<<<nroBlocos, NTHREADS>>>(aux1_d, aux2_d, nroEle, beta_d);
                }

                nroBlocos = (N+(NTHREADS-1))/NTHREADS;
                parte7<<<nroBlocos, NTHREADS>>>(w_d, r_d, beta_d, N);

        }
        // Copiar resultados para a memória principal
        cudaMemcpy(x, x_d, N*sizeof(float), cudaMemcpyDeviceToHost);
        // Gravar numero de iteracoes para retorno
        *nro_iter = i;

        // Liberar memoria GPU
        cudaFree (aa_d);
        cudaFree (ja_d);
        cudaFree (ia_d);
        cudaFree (b_d);
        cudaFree (x_d);
        cudaFree (r_d);
        cudaFree (w_d);
        cudaFree (z_d);
        cudaFree (aux1_d);
        cudaFree (aux2_d);
        cudaFree (normaR_d);

}

void gradienteConjugadoSeq(float *aa, int *ja, int *ia, float *b, float *x, float E,
                        int *nro_iter, int N, int nroNaoNulos) {
        // Declaracao de variaveis
        float *r, *w, *z;
        float normaR, alpha = 0.0, beta = 0.0, aux1, aux2;
        int i, j, k;
        // Alocar memoria para vetores
        r = (float*) malloc (N*sizeof(float));
        w = (float*) malloc (N*sizeof(float));
        z = (float*) malloc (N*sizeof(float));

        for (j = 0; j < N; j++) {
                x[j] = 0.0; // X = 0
                r[j] = b[j]; // r =  b - Ax (Ax = 0)
                w[j] = -r[j]; // w = -r
        }

        // Laco de controle de numero de iteracoes
        for (i = 0; i <= NMAXITER; i++) {

                aux1 = aux2 = 0.0;
                for (j = 0; j < N; j++) {
                        // z = A*w
                        z[j] = 0.0;
                        for (k = ia[j]; k < ia[j+1]; k++) {
                                z[j] += w[ja[k]]*aa[k];
                        }
                        // alpha = (r*w)/(w*z)
                        aux1 = aux1 + r[j]*w[j];
                        aux2 = aux2 + w[j]*z[j];
                }
                alpha = aux1/aux2;

                // x = x + alpha*w
                for (j = 0; j < N; j++)
                        x[j] = x[j] + w[j]*alpha;

                normaR = 0.0;
                for (j = 0; j < N; j++) {
                        r[j] = r[j] - z[j]*alpha; // r = r - alpha*z
                        normaR = normaR + r[j]*r[j]; // Acumular para calcular a norma de R
                }

                // Para se atingir tolerancia minima de erro
                if (sqrt(normaR) < E)
                        break;

                // beta = (r*z)/(w*z)
                aux1 = aux2 = 0.0;
                for (j = 0; j < N; j++) {
```

```
                        aux1 = aux1 + r[j]*z[j];
                        aux2 = aux2 + w[j]*z[j];
                }
                beta = aux1/aux2;

                // w = -r + w*beta
                for (j = 0; j < N; j++) {
                        w[j] = -r[j] + w[j]*beta;
                }
        }

        // Gravar numero de iteracoes para retorno
        *nro_iter = i;
        // Liberar memoria
        free(r);
        free(w);
        free(z);
}

int main(int argc, char** argv)
{
        // Declaração da matriz A (aa, ja, ia) e dos vetores b e x
        float *aa, *b, *x;
        int *ja, *ia;
        // Dados da matriz
        int nroLin, nroCol, nroNaoNulos;
        // variaveis auxiliares
        int *pos, *ra;
        int i, nro_iter, j, k, contArq, qtdeExec;
        time_t tempoIni, tempoFim;
        char where;
        float aux;
        char linha[100];
        // abrir arquivo de resultados
        FILE *arqResultados, *arqEstatisticas, *arquivo;
        arqEstatisticas = fopen ("./Resultados/estatisticas.txt","w");
        arqResultados = fopen ("./Resultados/resultados.txt","w");
        // Para todos os arquivos enviados por parametro
        qtdeExec = atoi(argv[1]);
        where = argv[2][0];
        for (contArq = 3; contArq < argc; contArq++) {

                arquivo = fopen (argv[contArq],"r");
                printf ("%s\t",argv[contArq]); fflush(stdout);

                // Ler cabecalho
                do
                        fgets (linha,sizeof(linha),arquivo);
                while (linha[0] == '%');

                sscanf (linha,"%d %d %d", &nroLin, &nroCol, &nroNaoNulos);
                nroNaoNulos *= 2;
                // Alocar memoria para os vetores aa, ia, jc
                pos = (int*) malloc ((nroNaoNulos)*sizeof(int));
                ra = (int*) malloc ((nroNaoNulos)*sizeof(int));
                aa = (float*) malloc ((nroNaoNulos)*sizeof(float));
                ja = (int*) malloc ((nroNaoNulos)*sizeof(int));
                ia = (int*) malloc ((nroLin+1)*sizeof(int));
                // Ler valores da matriz
                k = 0;

                while (fgets (linha,100,arquivo) != NULL) {
                        sscanf (linha,"%d %d %e", &j, &i, &aux);
                        i--; j--;
                        pos[k] = i*(nroLin) + j; // Posicao linear
                        ra[k] = i; // Linha
                        ja[k] = j; // Coluna
                        aa[k] = aux * dif; // Valor
                        k++;
                        if (i != j) { // Caso nao for diagonal principal
                                pos[k] = j*(nroLin) + i; // Posicao lineaar
                                ra[k] = j; // Linha
                                ja[k] = i; // Coluna
                                aa[k] = aux * dif; // Valor
                                k++;
                        }
                }
                nroNaoNulos = k; // Ajustar numero de elementos nao-nulos
                printf ("%d\t%d\t%d\n", nroLin, nroCol, nroNaoNulos); fflush(stdout);
                // Fechar arquivo
                fclose (arquivo);

                // Ordenar vetores por sua posicao linear
                quickSort(pos,ra,ja,aa,0,nroNaoNulos-1);

                // Gerar vetor IA
                j = 0;
                for (i = 0; i < nroLin; i++) {
                        ia[i] = j;

                        while (ra[j] == i) {

                                j++;
                        }
                }
                ia[i] = nroNaoNulos;

                // Liberar memoria dos vetores auxiliares
                free(pos);
```

```
            free(ra);

            // Alocar memoria para os vetores b e x
            b = (float*) malloc (nroCol*sizeof(float));
            x = (float*) malloc (nroCol*sizeof(float));
            // Setar vetor b para o resultado da resolucao do sistema seja x=1
            setarVetorB(aa,ja,ia,b,nroLin,nroCol);

            if (where != 'G') {
                    puts ("CPU:");
                    for (i = 0; i <qtdeExec; i++) {
                            // Chamar gradiente conjugado
                            tempoIni = time(NULL);
                            gradienteConjugadoSeq(aa,ja,ia,b,x, 10e-6, &nro_iter,nroCol,nroNaoNulos);
                            tempoFim = time(NULL);
                            // Escrever resultados
                            printf ("H:\t%d\t%.2lf\n",nro_iter,difftime (tempoFim,tempoIni));
                            escreverResultados(arqResultados,"H: x",x,nroCol);
                            fprintf (arqEstatisticas,"H: %s\t%d\t%d\t%d\t%d\t%.2lf\n",argv[contArq], nroLin, nroCol,
                                    nroNaoNulos,nro_iter,difftime (tempoFim,tempoIni));
                    }
            }

            int ntoIterCPU = nro_iter;
            if (where != 'C') {
                    puts ("GPU:");
                    for (i = 0; i <qtdeExec; i++) {
                            // Chamar gradiente conjugado
                            tempoIni = time(NULL);
                            gradienteConjugado(aa,ja,ia,b,x, 10e-6, &nro_iter,nroCol,nroNaoNulos, ntoIterCPU);
                            tempoFim = time(NULL);
                            // Escrever resultados
                            printf ("D: \t%d\t%.2lf\n",nro_iter,difftime (tempoFim,tempoIni));
                            escreverResultados(arqResultados,"D: x",x,nroCol);
                            fprintf (arqEstatisticas,"D: %s\t%d\t%d\t%d\t%d\t%.2lf\n",argv[contArq], nroLin, nroCol,
                                    nroNaoNulos,nro_iter,difftime (tempoFim,tempoIni));
                    }
            }
            fprintf (arqEstatisticas,"\n");

            // Liberar memória
            free(aa);
            free(ja);
            free(ia);
            free(b);
            free(x);

    }
    // Fechar arquivo de resultados
    fclose(arqResultados);
    fclose(arqEstatisticas);
    // Terminar programa
    return 0;

}
```