UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

PARALELIZAÇÃO DA BIBLIOTECA DE OPERAÇÕES SOBRE MATRIZES ESPARSAS CSPARSE UTILIZANDO CUDA

Abraão Guimarães Flores

JUIZ DE FORA
DEZEMBRO, 2010

PARALELIZAÇÃO DA BIBLIOTECA DE OPERAÇÕES SOBRE MATRIZES ESPARSAS CSPARSE UTILIZANDO CUDA

ABRAÃO GUIMARÃES FLORES

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharel em Ciência da Computação

Orientador: Marcelo Lobosco

JUIZ DE FORA
DEZEMBRO, 2010

PARALELIZAÇÃO DA BIBLIOTECA DE OPERAÇÕES
SOBRE MATRIZES ESPARSAS CSPARSE UTILIZANDO CUDA

Abraão Guimarães Flores

MONOGRAFIA SUBMETIDADA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

_____
Marcelo Lobosco, D.Sc
(Presidente)

_____
Maicon Ribeiro Correa, D.Sc

_____
Rodrigo Weber dos Santos, D.Sc

JUIZ DE FORA, MG – BRASIL
Dezembro, 2010

Dedico este trabalho a minha família, a minha namorada que sempre esteve do meu lado e a todas as pessoas que me acompanharam nesses anos de faculdade.

AGRADECIMENTOS

Agradeço primeiramente a Deus, fonte de tudo. Aos meus pais Walter Flores e Maria do Rosário pelo amor e dedicação. Ao meu irmão Jonatan pelo incentivo. A minha namorada Claudiana pelo carinho, amor e companheirismo. A todos meus amigos. Agradeço ao meu orientador Marcelo Lobosco pela paciência e por sempre me ajudar quando eu precisei. Agradeço ainda a todos os professores que repartiram conosco os seus conhecimentos. Obrigado a todos.

"O único modo de escapar da corrupção causada pelo sucesso é continuar trabalhando."

Albert Einstein

SUMÁRIO

LISTA DE FIGURAS

LISTA DE SIGLAS E SÍMBOLOS

ANSI – American National Standards Institute.
API – Application Programming Interface.
CGMA – Compute to Global Memory Access.
CISE – Department of Computer & Information Science & Engineering.
CPU – Central Processing Unit.
CUDA – Compute Unified Device Architecture.
DRAM – Dynamic Random Access Memory.
GDDR – Graphics Double Data Rate.
GFLOPS – Giga Floating Point Operations Per Second.
GPU – Graphics Processing Unit.
NVCC – NVIDIA C Compiler.
SM – Streaming Multiprocessors.
SP – Streaming Processors.

RESUMO

Este projeto apresenta um trabalho de pesquisa na área de programação paralela, que abordará estudos sobre uma nova tecnologia para computação de alto desempenho, uma biblioteca de operações sobre matrizes esparsas, além da criação de versões paralelas de alguns módulos desta biblioteca. A tecnologia para computação paralela escolhida neste projeto foi desenvolvida pela NVIDIA e chama-se CUDA (*Compute Unified Device Architecture*). A biblioteca de operações sobre matrizes esparsas chama-se CSparse e foi desenvolvida pelo CISE (*Department of Computer and Information Science and Engineering*) da Universidade da Flórida. Esta biblioteca possui as estruturas de dados e os mais variados algoritmos que realizam diversas operações com matrizes esparsas. O objetivo principal deste projeto é a criação de versões paralelas de alguns módulos da biblioteca CSparse, realizando sempre uma comparação entre o tempo de execução da versão original que está na forma sequencial e o tempo da versão paralela, desenvolvida neste projeto. Para a realização do trabalho, foram estudadas bibliografias sobre os temas citados acima, buscando sempre desenvolver estratégias para a realização do objetivo proposto. O que motivou a escolha deste tema foi o fato das operações com grandes matrizes ter uma computação muito custosa, mas ao mesmo tempo passível de paralelização devido a certo grau de independência entre as iterações dos algoritmos. Este trabalho mostra ainda quais foram as dificuldades encontradas durante a sua realização, além de mostrar também os pontos importantes que os futuros desenvolvedores de módulos paralelos da biblioteca em questão deverão ter para que uma maior aceleração seja alcançada.

Palavras-chave: Programação Paralela, CUDA, CSparse, Matrizes Esparsas.

1 Introdução

Em análise numérica, uma matriz esparsa é uma matriz onde a grande maioria dos elementos são zeros.

Exemplo:

$$\begin{bmatrix} 0 & 0 & 7 & 0 & 0 & 11 & 0 \\ 8 & 0 & 0 & 0 & 0 & 0 & 3 \\ 0 & -1 & 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -3 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Estes sistemas esparsos são úteis não só na análise combinatória, como também em outras áreas de aplicação. Grandes matrizes esparsas, muitas vezes aparecem em ciência ou engenharia, na resolução de equações diferenciais parciais.

1.1 Motivação

Quando o armazenamento e manipulação de matrizes esparsas são realizados em um computador, a maioria das vezes é necessário usar algoritmos especializados e estruturas de dados especiais que se aproveitem ao máximo da estrutura esparsa da matriz. Operações com matrizes na sua forma densa normalmente são dotadas de algoritmos pesados e, na maioria das vezes, demoram certo tempo para darem resultados. Sendo assim, se forem aplicados a matrizes esparsas, especialmente de grande porte, irão consumir grandes quantidades de memória e recursos desnecessários.

Dados esparsos são facilmente comprimidos por natureza. Essa compressão na maioria das vezes resulta em menos uso da memória e de outros recursos computacionais.

Como a utilização de algoritmos de matrizes densas em matrizes esparsas gera gastos desnecessários de recursos computacionais, surgiu a necessidade de criar estruturas de dados e algoritmos próprios para armazenar, manipular e realizar operações com esta classe especial de matrizes.

De acordo com DAVIS (2006), a biblioteca CSparse tem os algoritmos fundamentais para as mais diversas operações com matrizes esparsas. Dentre estas operações podemos destacar multiplicação de duas matrizes, multiplicação de matriz por vetor, transposição, além da solução direta de sistemas lineares esparsos.

Como operações de matrizes são, na maioria das vezes, passiveis de paralelização, motivou-se então a criação de uma versão paralela de alguns módulos da CSparse, buscando sempre comparar o tempo de execução da versão originalmente

sequencial com a versão paralela criada neste projeto, sempre verificando se foi alcançado alguma aceleração dos resultados gerados pelos módulos trabalhados.

1.2 Hipótese

É possível criar uma versão paralela de módulos da biblioteca CSparse, tentando alcançar uma melhora no tempo de execução de seus algoritmos. Para isso, serão utilizadas novas arquiteturas e tecnologias, que estão presentes hoje no mundo acadêmico e científico. Com essas tecnologias, será explorando o paralelismo existente nos dados esparsos e também nas operações realizadas com matrizes.

1.3 Objetivos

Este projeto tem como objetivo paralelizar alguns módulos da biblioteca CSparse com o uso do CUDA (*Compute Unified Device Architecture*), que explora todo o poder de processamento existente na arquitetura paralela das GPUs (Unidades de Processamento Gráfico). O CUDA é uma tecnologia para computação paralela, criada pela NVIDIA, que possibilita aumentos significativos no desempenho computacional das aplicações.

O objetivo não é somente paralelizar esta biblioteca. É também deixar esse processamento paralelo o mais transparente possível aos usuários da biblioteca, de forma que não traga nenhum gasto com adaptação de código, ou até mesmo treinamento do usuário que for utilizar essa nova versão da CSparse. É claro que algumas adaptações terão que ser realizadas no ambiente computacional do usuário. Será necessário, por exemplo, que os usuários tenham GPUs em suas máquinas para que a versão paralela seja executada.

1.4 Justificativa

As operações presentes na biblioteca CSparse, tanto as mais simples como multiplicação de vetor densos por uma matriz esparsa, quanto as mais complexas como soluções de sistemas lineares esparsos, têm aplicações nas mais diversas áreas da ciência como Física e Engenharia.

Cabe ressaltar também que paralelizar algoritmos comumente utilizados na sua forma sequencial hoje é uma tendência do meio científico da Ciência da Computação.

1.5 Metodologia

A seguir, descreve-se a metodologia que foi utilizada no desenvolvimento deste projeto.

1.5.1 Definição

Foi realizado um estudo das bibliografias sobre a utilização do CUDA e também sobre a biblioteca CSparse, desenvolvendo estratégias para que fosse realizada a implementação da versão paralela da biblioteca em questão.

1.5.2 Desenvolvimento

Foram desenvolvidas versões de alguns módulos da biblioteca CSparse, para que possam ser testados e aplicados em matrizes das mais variadas dimensões.

1.5.3 Testes da Implementação

Os códigos foram testados em diversas matrizes, buscando sempre a comparação do tempo de execução entre a versão original da CSparse que está na forma sequencial e a versão paralela desenvolvida neste projeto.

1.5.4 Escrita e Finalização

A monografia foi montada criando primeiramente a estrutura dos capítulos para que posteriormente fossem escritos. Foram também apresentados os resultados obtidos com a implementação dos módulos paralelos, além de todas as estratégias utilizadas na implementação do projeto proposto.

1.6 Divisão dos Capítulos

O primeiro capítulo traz a introdução do projeto, mostrando sua motivação, hipótese, objetivos, justificativa e a metodologia utilizada para a criação deste projeto.

O segundo capítulo descreve o CUDA, suas características e arquitetura. Fala ainda sobre algumas funções principais do núcleo, além mostrar também a organização dos fluxos de computação.

O terceiro capítulo mostra a biblioteca de operações sobre matrizes esparsas CSparse. O foco principal desta seção é a estrutura de dados utilizada para o armazenamento interno de matrizes esparsas e uma visão geral sobre alguns módulos presentes na biblioteca em questão.

O quarto capítulo apresenta a criação de versões paralelas de alguns módulos da biblioteca CSparse. Neste capítulo são abordados a paralelização dos módulos da transposição de matrizes esparsas, além da multiplicação de uma matriz esparsa por um vetor denso. O capítulo traz ainda os resultados obtidos com a paralelização dos módulos, as dificuldades encontradas durante o desenvolvimento do projeto, além também de alguns pontos importantes que os futuros desenvolvedores de módulos

paralelos da CSparse devem ter, para que uma maior aceleração dos módulos paralelos seja alcançada.

O quinto capítulo traz as considerações finais, fazendo uma conclusão de todo o trabalho realizado.

2 CUDA

2.1 Introdução

Microprocessadores com base em uma única CPU (Unidade Central de Processamento) apresentaram, durante mais de duas décadas, rápidas melhorias de desempenho e redução de custos. Estes microprocessadores trouxeram dezenas de bilhões de operações de ponto flutuante por segundo (GFLOPS) para computadores pessoais e quatrilhões de operações de ponto flutuante por segundo (PETAFLOPS) para agregados de computadores (clusters). Esta grande melhoria de desempenho permitiu que novas classes de aplicações pudessem ser desenvolvidas, mais funcionalidades fossem adicionadas às aplicações já existentes, bem como melhores interfaces com o usuário fossem criadas. Os usuários, por sua vez, demandam ainda mais melhorias, criando assim um ciclo positivo para a indústria de computadores.

Segundo Sutter (2005 apud Kirk & Hwu, 2010), com o passar do tempo, a maioria dos desenvolvedores de software têm contado com os avanços em hardware para aumentar a velocidade de suas aplicações, onde o mesmo software simplesmente executa mais rápido à medida que cada nova geração de processadores é apresentada. Esta melhoria, no entanto, diminuiu desde 2003, devido ao consumo de energia e dissipação de calor, que são questões que têm limitado o aumento da frequência do *clock* e o nível de atividades produtivas que podem ser executadas em cada ciclo de *clock* dentro de uma única CPU.

Praticamente todos os fornecedores de microprocessadores têm mudado para modelos onde múltiplas unidades de processamento, denominadas núcleos do processador, são usadas em cada *chip* para aumentar o poder de processamento. Essa opção tem exercido um enorme impacto sobre a comunidade de desenvolvedores de software.

Tradicionalmente, a maioria das aplicações de software é escrita como programas sequenciais, conforme descrito por Neumann (1945 apud Kirk & Hwu, 2010), em seu relatório seminal. Historicamente, os usuários de computador se acostumaram com a expectativa de que estes programas são executados mais rapidamente com cada nova geração de microprocessadores. Essa expectativa atualmente não é mais estritamente válida. Um programa, quando desenvolvido na forma sequencial, funcionará somente em um dos núcleos do processador, o que não deixará a execução mais rápida nas novas gerações de processadores. Sem a melhoria do desempenho, os desenvolvedores de aplicativos não serão mais capazes de introduzir novas

funcionalidades em seu software a medida com que microprocessadores novos são introduzidos. Isso geraria uma redução nas oportunidades de crescimento da indústria de computadores.

Dessa maneira, os aplicativos que continuarão a utilizar as melhorias de desempenho dos novos *chips* serão os paralelos, no qual vários fluxos irão trabalhar em conjunto para executarem aplicação mais rapidamente.

A prática da programação paralela não é nova. A comunidade de computação de alto desempenho vem desenvolvendo programas paralelos durante décadas. Inicialmente, estes programas eram executados em supercomputadores que possuíam um preço muito elevado. Sendo assim, poucas aplicações podiam justificar a compra destes computadores, o que limitava a prática da programação paralela a um pequeno número de desenvolvedores.

Entretanto, com a popularização observada nos últimos anos dos microprocessadores paralelos, nota-se um grande crescimento no número de aplicativos paralelos desenvolvidos. Dessa forma, existe hoje uma grande necessidade dos desenvolvedores se adaptarem à forma de programação paralela, para que eles consigam utilizar os avanços de desempenho providos pela nova geração de *chips*.

2.2 Máquinas *Multicores* e *Many-cores*.

Segundo Kirk & Hwu (2010), desde 2003 a indústria de processadores se dividiu em duas trajetórias principais para o desenvolvimento de seus *chips*: as chamadas máquinas *multicores* e *many-cores*.

A trajetória *multicore* mantém a velocidade de execução de programas sequenciais, aumentando o número de núcleos disponíveis para o processamento. Esta trajetória começou com processadores de dois núcleos, praticamente dobrando o número de núcleos a cada nova geração de processadores.

Em contraste, a trajetória *many-core* se concentra mais na vazão da execução de aplicações paralelas. O *many-core* começou como um grande número de núcleos de processamento extremamente mais simples que os núcleos *multicore*. Da mesma maneira que nas máquinas *multicore*, o número de núcleos vem dobrando a cada nova geração.

Processadores *many-cores*, em especial as GPUs (unidades de processamento gráfico), vêm desde 2003 liderando a corrida pela busca desempenho. Este fenômeno é ilustrado na figura 2.1. Enquanto o desempenho das GPUs vem crescendo a passos largos, o desempenho dos microprocessadores de uso geral vem aumentando de forma muito mais modesta. A partir de 2008 observa-se que a relação entre as GPUs e as

CPUs é de cerca de 10 para 1, quando comparada a vazão de pico para processamento de ponto flutuante.

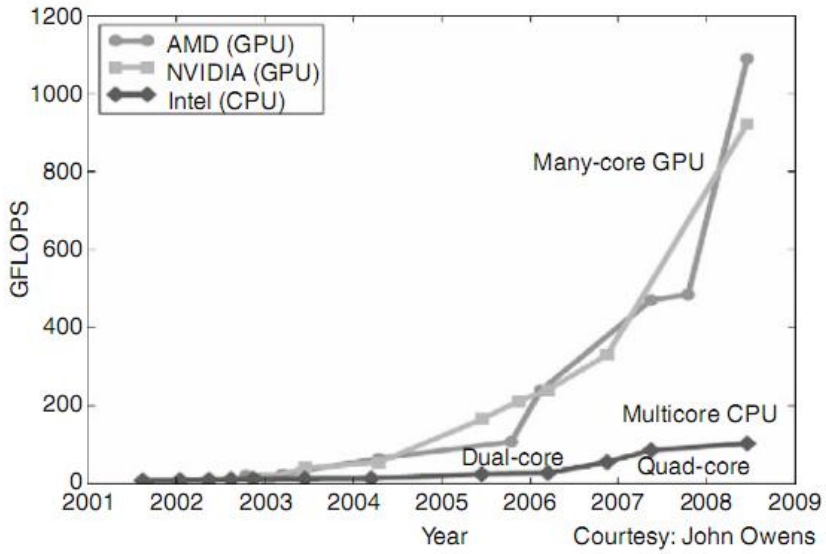

Figura 2.1. Ampliando a diferença de desempenho entre CPUs e GPUs. Figura extraída de Kirk & Hwu, 2010, p.3

De acordo com Kirk & Hwu (2010), esta grande diferença de desempenho já motivou muitos desenvolvedores de aplicações a mover partes computacionalmente pesadas de seus programas para as GPUs. Não surpreendentemente, estas mesmas partes computacionalmente pesadas são também o principal alvo da programação paralela.

Essa grande diferença de desempenho entre as GPUs e CPUs *multicore* está nas diferenças das filosofias de projeto entre os dois tipos de processadores, como ilustrado na figura 2.2. O projeto de uma CPU é otimizado para desempenho de código sequencial. Ele faz uso de lógica de controle sofisticada, permitindo que as instruções de um único fluxo possam executar em paralelo ou mesmo fora de sua ordem original, mas não perdendo a característica de uma execução sequencial.

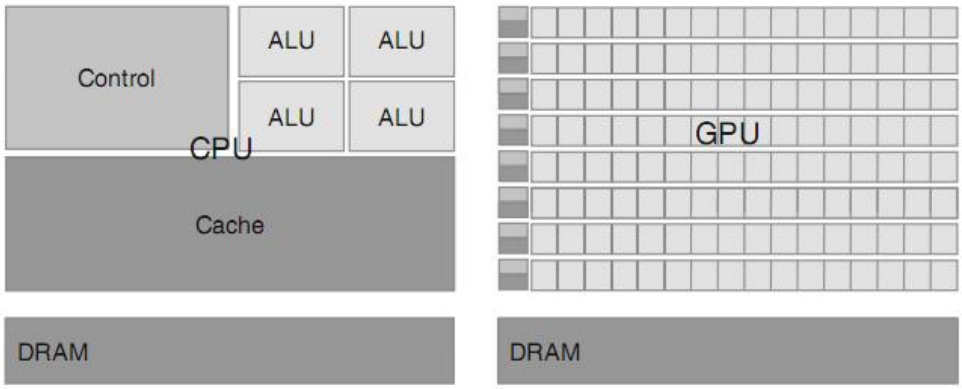

Figura 2.2. As diferenças na filosofia de design entre CPUs e GPUs. Figura extraída de Kirk & Hwu, 2010, p.4

O crescimento de desempenho observado nas GPUs é por muitos atribuída às demandas da indústria dos videogames. Esta indústria exerce uma enorme pressão econômica para um aumento da capacidade de executar um grande número de cálculos de ponto flutuante por quadro de vídeo em jogos avançados.

2.3 CUDA

As GPUs, portanto, são criadas com o intuito de executar mais rapidamente os cálculos mais intensos de uma aplicação. Desta maneira, elas não terão bons resultados onde as CPUs já apresenta um bom desempenho. Assim, existe a tendência de que a maioria das aplicações irá utilizar tanto as CPUs quanto as GPUs. A parte sequencial do código utilizará a CPU e as operações numericamente intensas, em especial operações executadas sobre diferentes conjuntos de dados, serão realizadas na GPU.

É com esse pensamento que o modelo de programação CUDA (*Compute Unified Device Architecture*), introduzido pela NVIDIA em 2007, foi projetado. Nele, as aplicações serão executadas em parte na CPU e parte na GPU.

É importante também notar que o desempenho não é o único fator de decisão quando os desenvolvedores buscam processadores para executar suas aplicações. A popularidade que certo processador tem é uma característica muito importante. Os desenvolvedores procuram criar suas aplicações para os processadores com maior presença no mercado, buscando assim um grande número de clientes. Este é um grande trunfo das GPUs. Devido à sua popularidade no mercado de microcomputadores, trazida pelos jogos, centenas de milhões de GPUs foram vendidas.

Portanto, com a difusão das GPUs, temos então a computação paralela sendo, pela primeira vez, explorada amplamente pelos desenvolvedores, com um produto comum no mercado.

Segundo Kirk & Hwu (2010), o surgimento do CUDA em 2007 difundiu a programação paralela. A NVIDIA realizou certas mudanças no *hardware*, dedicando efetivamente uma área do *chip* das placas de vídeo para facilitar programação paralela. Dessa maneira, o CUDA não representa apenas uma mudança no software.

As camadas de software existente no CUDA foram implementadas de tal forma que os desenvolvedores possam utilizar as ferramentas de programação existentes no C/C++. Isso facilita bastante a tarefa do programador, pois a grande difusão destas linguagens tornou-as bem familiares para a maior parte dos desenvolvedores.

2.4 Arquitetura das GPU's Modernas

A figura 2.3 mostra a arquitetura típica de uma GPU para desenvolver aplicações utilizando CUDA. Segundo Kirk & Hwu (2010), a arquitetura é organizada em um vetor de multiprocessadores (SMs). Na Figura 2.3, dois SMs formam um bloco de trabalho. No entanto, o número de SMs em um bloco pode variar de geração para de geração de GPUs CUDA. Além disso, cada SM na Figura 2.3 tem um certo número de processadores (SPs) que compartilham a lógica de controle e a memória *cache* de instruções. Cada GPU, atualmente, possui até 4 gigabytes de DRAM (Memória Dinâmica de Acesso Randômico) do tipo GDDR (Dupla Taxa de Dados Gráficos). Essa memória recebe o nome de memória global, que também está presente na figura 2.3.

As memórias GDDR diferem das memórias RAM de um computador comum. Elas são desenvolvidas especificamente para utilização em placas gráficas, podendo funcionar com um *clock* mais alto que as memórias do tipo RAM.

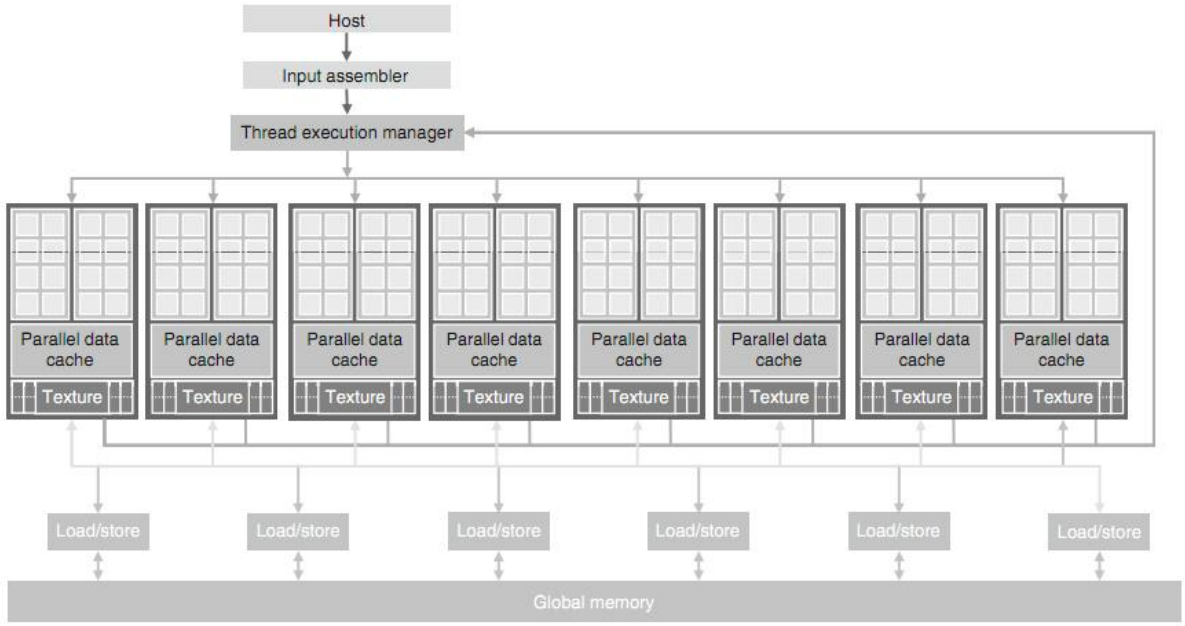

Figura 2.3. Arquitetura de uma GPU. Figura extraída de Kirk & Hwu, 2010, p.9

2.5 Estrutura do CUDA

Conforme dito anteriormente, um programa CUDA é dividido em partes, onde algumas executam na CPU e outras na GPU. As partes que apresentam pouco ou nenhum paralelismo de dados são implementadas para executar na CPU. Já as partes que apresentam grande quantidade de paralelismo de dados são implementadas para a GPU.

Segundo Stratton (2008 apud Kirk & Hwu, 2010), o NVCC (Compilador NVIDIA C) separa os dois tipos de códigos durante o processo de compilação. O código para o hospedeiro é padronizado em ANSI (Instituto Americano de Padrões Nacionais) C. Este é compilado com o compilador padrão C do hospedeiro e é executado como um processo comum em sua CPU.

Já o código para o dispositivo CUDA é escrito usando ANSI C estendido, com palavras-chave para a implementação de funções paralelas, chamadas núcleos. O código do dispositivo é compilado pelo NVCC e executado em um dispositivo GPU.

Segundo Kirk & Hwu (2010), as funções do núcleo geram um grande número de fluxos, que exploram o paralelismo de dados. Um fato interessante é que os fluxos do CUDA têm uma computação muito mais leve do que os fluxos da CPU.

Programadores CUDA podem assumir que os fluxos do CUDA gastam poucos ciclos de *clock* para serem gerados, devido ao suporte eficiente de hardware. Isso está em contraste com os fluxos da CPU, que normalmente exigem milhares de ciclos de *clock* para serem criados.

A execução de um programa de CUDA é mostrada na figura 2.4. A execução começa com o código sequencial executando na CPU do hospedeiro. Quando um núcleo paralelo é invocado, a execução é movida para o dispositivo GPU. Neste dispositivo, um grande número de fluxos é gerado, buscando explorar ao máximo o paralelismo de dados abundante. De acordo com Kirk & Hwu (2010), todos os fluxos que foram gerados por um núcleo durante uma chamada são organizados dentro de uma grade. A figura 2.4 mostra a execução de duas grades. Quando todos os fluxos de um núcleo completam a sua execução, a grade que os contém também se encerra, continuando a execução do código sequencial dentro da CPU do hospedeiro, até que outro núcleo seja invocado.

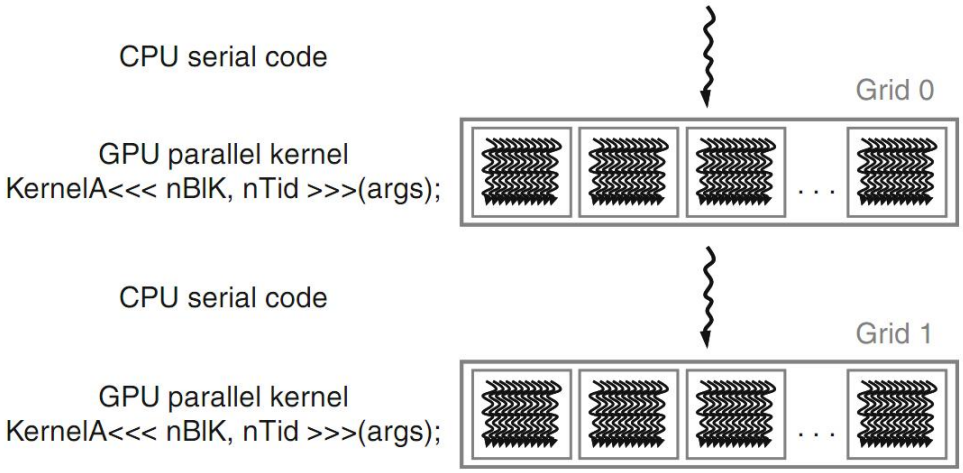

Figura 2.4. Execução de um programa CUDA. Figura extraída de Kirk & Hwu, 2010, p.42

2.6 Memória da GPU e Transferência de Dados

De acordo com Kirk & Hwu (2010), o hospedeiro e o dispositivo GPU têm espaços de memória separados e isso é refletido no CUDA. Isso acontece porque as GPUs são geralmente placas separadas que vêm com sua própria DRAM.

Para executar um núcleo, o programador precisa alocar um certo bloco de memória no dispositivo GPU, além de transferir os dados que são necessários para a execução, de dentro da memória do hospedeiro para a memória alocada no dispositivo. Da mesma forma, após finalizar os cálculos dentro do núcleo, o programador precisa transferir dados que estão na memória do dispositivo de volta para a memória do hospedeiro, além de liberar a memória que foi alocada dentro do dispositivo, pois esta não será mais necessária neste momento. A linguagem do CUDA fornece a API (Interface de Programação de Aplicações) necessária para realizar essas atividades, facilitando o desenvolvimento do programador.

A Figura 2.5 mostra uma visão geral do modelo de memória do dispositivo CUDA, dando uma ideia de como é realizada a transferência de dados, além de mostrar também sua hierarquia. Na parte inferior da figura, vemos a memória global e a memória constante. Estas são as memórias usadas para que o código do hospedeiro troque dados com o dispositivo GPU. É importante perceber que a memória do hospedeiro existe, mas não é mostrada pela figura 2.5.

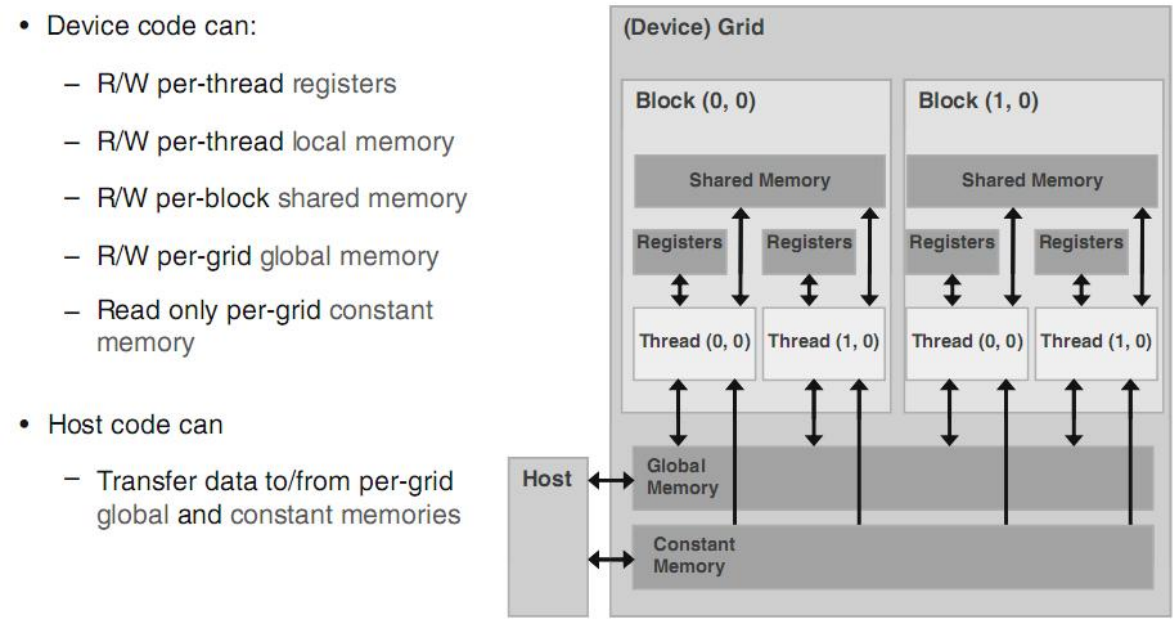

Figura 2.5. Visão geral do modelo de memória do dispositivo CUDA. Figura extraída de Kirk & Hwu, 2010, p.47

O modelo de memória CUDA contém certas funções que ajudam no gerenciamento dessas memórias pelos os programadores CUDA. A figura 2.6 mostra as funções que realizam a alocação e desalocação da memória global do dispositivo.

De acordo com Kirk & Hwu (2010), a função *cudaMalloc()* pode ser invocada a partir do código do hospedeiro, alocando uma parte da memória global. Pode ser notada facilmente a semelhança entre *cudaMalloc()* e a função *malloc* que é padrão da biblioteca C. Essa semelhança existe com o intuito de aproveitar o conhecimento que o programador já tem das funções padrões do C, para o entendimento da semântica das funções do CUDA.

Portanto, o CUDA é a linguagem C com certas extensões. Ele utiliza a função *malloc()*, que é padrão da biblioteca da linguagem C, para gerenciar a memória do hospedeiro e adiciona a função *cudaMalloc()* como uma extensão. A ideia é procurar manter a interface do CUDA bem próxima da biblioteca da linguagem C, buscando minimizar o tempo que um programador C precisaria para aprender o uso das funções do CUDA.

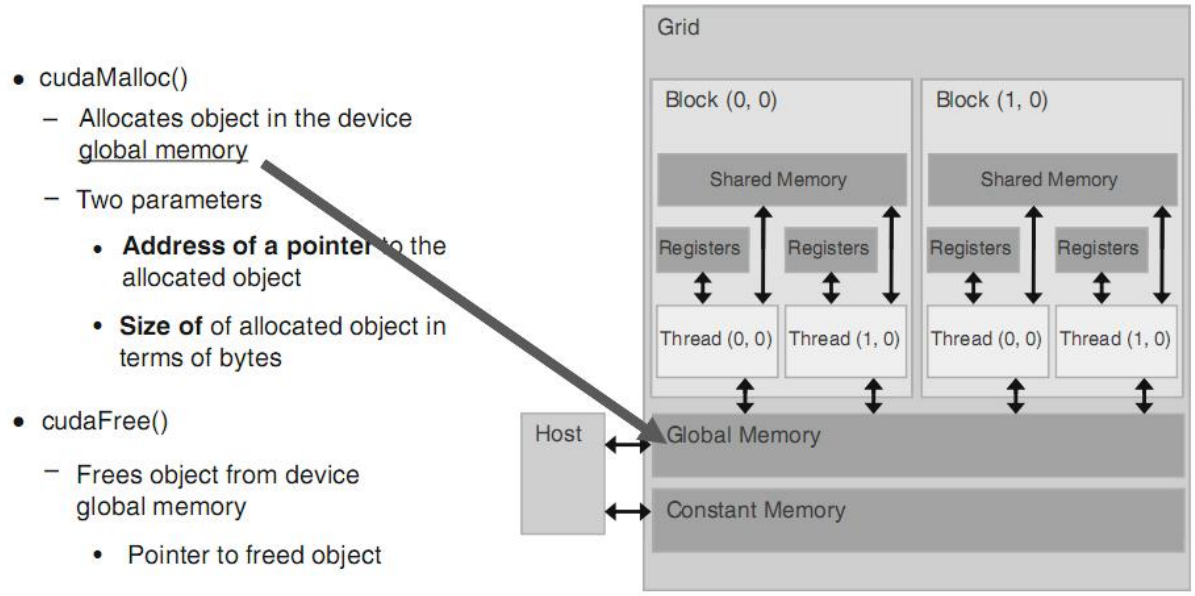

Figura 2.6. Funções da API CUDA para o gerenciamento da memória global do dispositivo. Figura extraída de Kirk & Hwu, 2010, p.47

O primeiro parâmetro da função *cudaMalloc()* é o endereço de um ponteiro que terá o bloco alocado após a chamada desta função. O tipo deste ponteiro deve ser convertido para *void***, pois a função espera um tipo genérico. Isto porque a função de alocação de memória é uma função genérica. Isso não poderia ser diferente, pois a alocação de memória não pode ser restrita a um determinado tipo de dados. O segundo parâmetro desta função dá a dimensão do bloco que será alocado, em termos de bytes. O uso deste parâmetro é coerente com o único parâmetro da função *malloc()*, padrão da linguagem C. Ele é quem define o tamanho da alocação que será realizada.

Após toda computação realizada no núcleo, a função *cudafree()* deve ser invocada para liberar a memória que foi alocada anteriormente pela função *cudaMalloc()*.

Uma vez que o programa alocou memória global do dispositivo, ele pode solicitar que os dados sejam transferidos do hospedeiro para o dispositivo. Isso é feito invocando a função *cudaMemcpy()*. Esta função realiza a transferência de dados entre os diversos tipos de memórias. A figura 2.7 mostra esta função para a transferência de dados.

Segundo Kirk & Hwu (2010), a função *cudaMemcpy()* tem quatro parâmetros. O primeiro parâmetro é um ponteiro para o local de destino dos dados que serão copiados. O segundo parâmetro é um ponteiro para bloco dos dados que serão copiados. O terceiro parâmetro especifica o número de bytes que será copiado. Já o quarto parâmetro indica a origem e o destino dos dados transferidos entre memórias: do hospedeiro para o hospedeiro, do hospedeiro para o dispositivo GPU, do dispositivo para o hospedeiro, além do dispositivo para o dispositivo. Cabe ainda ressaltar que a função *cudaMemcpy()*

não pode ser usada para copiar dados entre GPUs em diferentes sistemas compostos por várias GPUs.

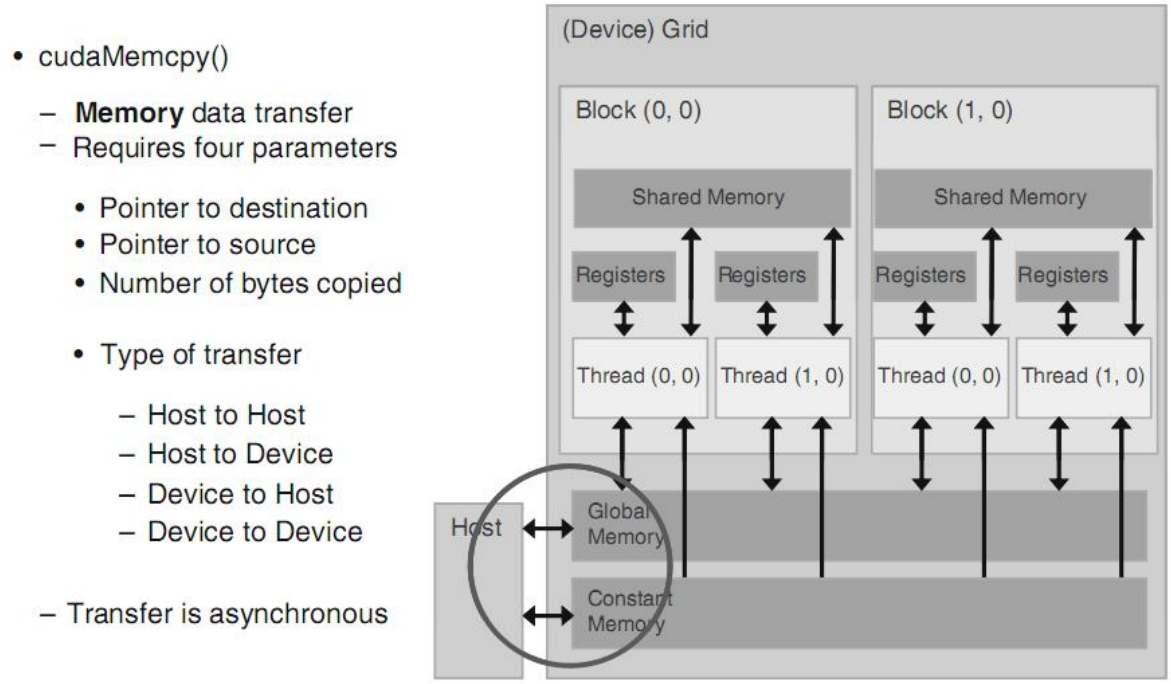

Figura 2.7. Funções da API CUDA para transferência de dados entre as memórias. Figura extraída de Kirk & Hwu, 2010, p.49

2.7 Funções do Núcleo

Em CUDA, uma função do tipo núcleo especifica o código a ser executado por todos os fluxos. De acordo com Atallah (1998 apud Kirk & Hwu, 2010), o CUDA é um exemplo do famoso estilo de programação paralela SPMD (único programa, múltiplos dados), pois todos os fluxos executam o mesmo código de programação. Cabe ainda dizer que este é um estilo popular de programação para sistemas de computação massivamente paralela.

A Figura 2.8 mostra uma função núcleo para multiplicação de matrizes. A sintaxe é ANSI C, com algumas extensões notáveis. Primeiro, há a palavra-chave específica do CUDA "__global__", na frente da declaração de *MatrixMulKernel()*. Esta palavra indica que a função é do tipo núcleo e que pode ser chamada por um hospedeiro, gerando uma grade de fluxos dentro do dispositivo CUDA.

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  // 2D Thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // Pvalue stores the Pd element that is computed by the thread
  float Pvalue = 0;

  for (int k = 0; k < Width; ++k)
  {
     float Mdelement = Md[ty * Width + k];
     float Ndelement = Nd[k * Width + tx];
     Pvalue += Mdelement * Ndelement;
  }

  // Write the matrix to device memory each thread writes one element
  Pd[ty * Width + tx] = Pvalue;
}
```

Figura 2.8. Núcleo para multiplicação de matrizes. Figura extraída de Kirk & Hwu, 2010, p.52

De uma maneira geral, o CUDA adiciona nas declarações de funções em C três palavras-chave. O significado de cada uma destas palavras é mostrado na figura 2.9. Segundo Kirk & Hwu (2010), o termo "__global__" indica que a função está sendo declarada como uma função núcleo do CUDA. Uma função núcleo do tipo "__global__" será executada no dispositivo GPU e só pode ser chamada a partir do hospedeiro, gerando assim uma grade de fluxos.

A palavra-chave "__device__" indica que a função pertence ao dispositivo GPU. Este tipo de função é executado na GPU e só pode ser invocada por uma função núcleo ou então por outra função do dispositivo. Já a palavra-chave "__host__" indica que esta é uma função CUDA para o hospedeiro. Este tipo é simplesmente uma função C tradicional. Este tipo de função executa na máquina do hospedeiro e só pode ser invocado por outra função dentro desta mesma máquina.

Por padrão, toda função em um programa CUDA que não possuir nenhuma das palavras-chave mostradas acima, será considerada uma função do hospedeiro. Sendo assim, o programador não precisará alterar suas funções originais que estão na linguagem C.

Segundo Nvidia Corporation (2006), o qualificador "__host__" também pode ser usado em combinação com o qualificador "__device__", no caso em que a função é compilada tanto para o hospedeiro quanto para o dispositivo.

	Executed on the:	Only callable from the:
__device__float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

Figura 2.9. Extensões do CUDA para declarações de funções em C. Figura extraída de Kirk & Hwu, 2010, p.52

As funções do tipo “__device__” e “__global__” não suportam recursão, não podem ter declarações de variáveis estáticas no seu corpo e não podem ter um número variável de argumentos. Os qualificadores “__global__” e “__host__” não podem ser usados juntos. Funções do tipo “__global__” têm que ter seu tipo de retorno como *void*.

Já em relação às variáveis, o qualificador “__device__” declara variáveis no dispositivo CUDA. No máximo, mais um dos outros qualificadores pode ser usado em conjunto com “__device__” para expandir o espaço de memória que a variável pertence. Se nenhum dos outros estiverem presentes, a variável reside do espaço de memória global. Variáveis deste tipo têm o mesmo tempo de vida de toda a aplicação e é acessível pelo hospedeiro e também por todos os fluxos da grade.

O qualificador “__constant__” é opcionalmente usado em conjunto com o qualificador “__device__”, colocando a variável no espaço de memória constante.

O qualificador “__shared__”, é também usado opcionalmente em conjunto com o qualificador “__device__”, declarando variáveis que residem no espaço de memória compartilhada de um bloco de fluxos. Variáveis compartilhadas têm também o mesmo tempo de vida de toda aplicação e só são acessíveis por todos os fluxos de um mesmo bloco.

Cabe ainda dizer que todos os qualificadores citados acima não podem pertencer a *struct* e *union*, que são estruturas de programação da linguagem C. Outra restrição seria o fato de variáveis do tipo “__device__”, “__shared__” e “__constant__” não poderem ser definidas como externas, usando a palavra-chave *extern*, pertencente também à linguagem C.

Outras extensões do ANSI C que podem ser notadas na figura 2.8, são as palavras-chave *threadIdx.x* e *threadIdx.y* que, segundo Kirk & Hwu (2010), se referem aos índices de um fluxo. Deve-se destacar que todos os fluxos executam o mesmo código do núcleo. Sendo assim é preciso utilizar um mecanismo do CUDA que permita distinguir cada um dos fluxos. Essas palavras-chave predefinidas do CUDA fornecem a

identificação das coordenadas para um fluxo. Dessa forma, diferentes fluxos verão valores diferentes nas suas variáveis *threadIdx.x* e *threadIdx.y*.

Quando um núcleo é chamado, este é executado em uma grade de fluxos paralelos. Na Figura 2.10, a chamada do núcleo 1 cria a Grade 1. Cada grade criada pelo CUDA normalmente é composta por milhares de leves fluxos.

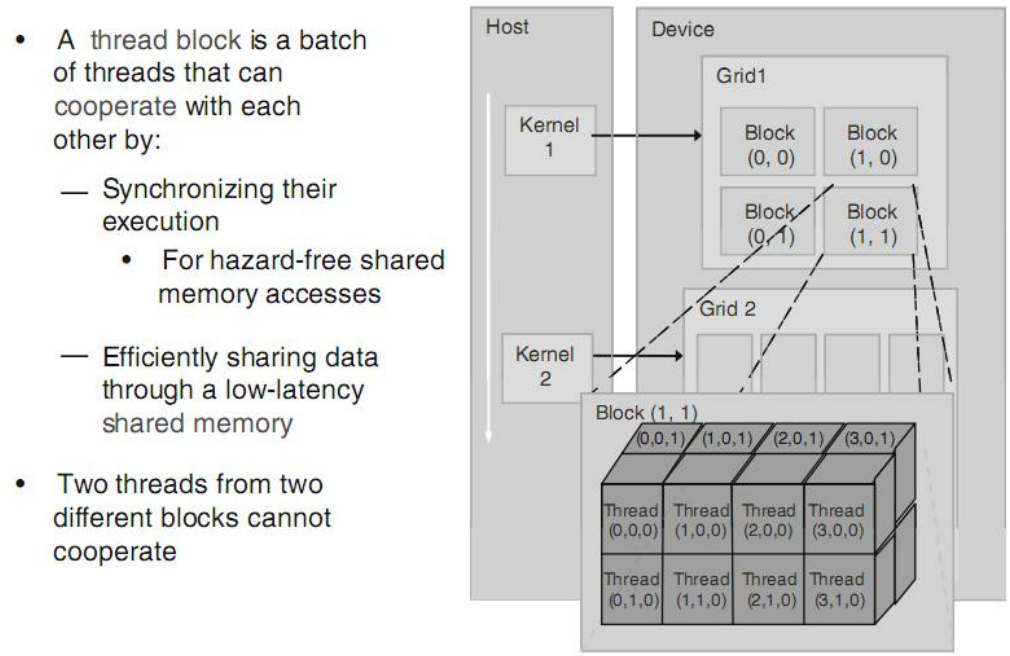

Figura 2.10. Organização dos fluxos no CUDA. Figura extraída de Kirk & Hwu, 2010, p.54

2.8 Organização dos fluxos no CUDA

Os fluxos de uma grade são organizados em uma hierarquia de dois níveis, conforme ilustrado na Figura 2.10. Para uma melhor visualização, um pequeno número de fluxos é mostrado. No primeiro nível, cada grade é composta de um ou mais blocos de fluxos. Todos os blocos em uma grade tem o mesmo número de fluxos, organizados da mesma maneira. Na Figura 2.10, uma grade é organizada como uma matriz 2 x 2, contendo um total de 4 blocos. Cada bloco tem uma coordenada bidimensional, dada pelas palavras-chave do CUDA *blockIdx.x* e *blockIdx.y*.

Quando um fluxo está em execução, as variáveis *blockIdx* e *threadIdx* retornam suas coordenadas. As variáveis adicionais *gridDim* e *blockDim* dão a dimensão da grade e de cada bloco, respectivamente.

Em geral, uma grade é organizada como uma matriz de blocos de duas dimensões. Cada bloco é organizado em uma matriz de três dimensões de fluxos. A

organização exata de uma grade é determinada pela configuração de execução, prevista na chamada do núcleo.

Segundo Nvidia Corporation (2006), qualquer chamada a uma função do tipo "__global__" deve especificar a configuração de execução para essa chamada, que define as dimensões da grade e dos blocos. A especificação é feita através da inserção de uma expressão da forma <<<Dg, Db, Ns, S>>> entre o nome da função e da lista de argumentos entre parênteses. Dentre estes quatro parâmetros, podemos destacar os dois primeiros.

O primeiro parâmetro especifica as dimensões da grade, em termos de número de blocos. O segundo parâmetro especifica as dimensões de cada bloco, em termos de número de fluxos. Cada um desses parâmetros é do tipo *DIM3*, que é uma estrutura de dados em C, com três campos inteiros sem sinal: x, y, e z. Como as grades possuem apenas duas dimensões, o terceiro campo desta estrutura de dados é ignorado.

De acordo com Kirk & Hwu (2010), os valores de *gridDim.x gridDim.y* podem variar de 1 a 65.535. Uma vez que o núcleo for iniciado, as dimensões dos blocos e da grade não podem ser alteradas. Todos os fluxos em um mesmo bloco tem o mesmo valor de *blockIdx*. O *blockIdx.x* valor varia entre 0 e *gridDim.x* - 1 e o valor *blockIdx.y* varia entre 0 e *gridDim.y* - 1.

A figura 2.11 exemplifica um código definido no hospedeiro para as definições das dimensões da grade e dos blocos, além ainda da chamada do núcleo, de acordo com a organização dos blocos e da grade presente na figura 2.10.

```
dim3 dimGrid(2, 2, 1);
dim3 dimBlock(4, 2, 2);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

Figura 2.11. Definições das dimensões e chamada de um núcleo. Figura extraída de Kirk & Hwu, 2010, p.62

O tamanho total de um bloco é limitado a 512 fluxos, podendo usar a flexibilidade na distribuição destes elementos em três dimensões. Por exemplo, (512, 1, 1), (8, 16, 2) e (16, 16, 2) são valores possíveis para as dimensões do bloco, mas (32, 32, 1) não é admissível, pois o número total de fluxos seria 1024.

2.9 Conclusão

Durante décadas, foram observados avanços significativos no poder de processamento dos processadores com apenas um núcleo. Devido a diversos problemas físicos, o crescimento no poder de processamento não pôde continuar avançando na

mesma taxa. Por este motivo, os fabricantes de processadores optaram por aumentar o número de núcleos disponíveis em cada processador. O emprego de programação paralela faz-se necessário para tirar proveito de um maior número de núcleos. Em um passado recente, esta forma de programação exigia uma arquitetura muito cara e poucas aplicações justificavam a compra de computadores com preços tão elevados.

A plataforma CUDA surgiu como uma alternativa para a programação massivamente paralela, dando a oportunidade para os programadores explorarem o poder de processamento das GPUs. Este poder foi alavancado pelo grande crescimento da indústria dos jogos.

Uma grande vantagem da plataforma CUDA é o fato da sintaxe de sua linguagem estar bem próxima das linguagens C e C++, o que reduz o impacto na forma de programar do programador. Além do mais, CUDA é uma excelente opção para o desenvolvimento de aplicações paralelas, pois sua arquitetura propicia explorar ao máximo o paralelismo de dados existente nas aplicações.

3 A biblioteca CSparse

3.1 Introdução

As matrizes esparsas são geralmente geradas a partir do processamento sobre certa modelagem de algum problema. Grandes matrizes esparsas aparecem geralmente em física e engenharia, como por exemplo na resolução de equações diferenciais parciais ou sistemas de equações lineares.

Ao armazenar e trabalhar com matrizes esparsas em um computador, muitas vezes é benéfico e necessário utilizar algoritmos e estrutura de dados especiais, que buscam minimizar o uso desnecessário dos recursos computacionais. O armazenamento destas matrizes pode ser feito guardando apenas os elementos diferentes de zero, além de criar mecanismos que relacione o elemento à sua posição na matriz original.

As operações com matrizes esparsas, armazenadas em um vetor de duas dimensões, consomem grande quantidade de memória desnecessária, principalmente quando o tamanho desta matriz cresce consideravelmente. Porém, existem formas simples e eficientes de comprimir matrizes esparsas, o que deixa o armazenamento menos custoso para os ambientes computacionais.

A biblioteca CSparse foi desenvolvida pelo CISE (*Department of Computer and Information Science and Engineering*) da Universidade da Flórida e, segundo Davis (2006), possui algoritmos que trabalham com matrizes esparsas para a solução direta de sistemas lineares esparsos. Os algoritmos desta biblioteca foram criados com o objetivo de incorporar toda a teoria por trás de algoritmos de matrizes esparsas. Outra característica é ser assintoticamente ótimos no seu tempo de execução e uso de memória. Estes algoritmos buscam ainda abranger um amplo espectro de operações destas matrizes, procurando ao máximo a precisão e robustez.

3.2 Estrutura de dados

Segundo Davis (2006), uma matriz é dita esparsa quando suas entradas são, na sua maioria, nulas. Há muitas maneiras de armazenar uma matriz esparsa. Independentemente da maneira escolhida, alguma estrutura de dados compacta é necessária para evitar o armazenamento de entradas numericamente nulas da matriz. Esta estrutura precisa ser simples e flexível para que possa ser usada em uma ampla gama de operações da matriz.

$$A = \begin{bmatrix} 4.5 & 0 & 3.2 & 0 \\ 3.1 & 2.9 & 0 & 0.9 \\ 0 & 1.7 & 3.0 & 0 \\ 3.5 & 0.4 & 0 & 1.0 \end{bmatrix}$$

Figura 3.1. Exemplo de uma matriz esparsa. Figura extraída de Davis, 2006, p.7

A mais simples estrutura de dados de uma matriz esparsa é uma lista de entradas diferentes de zero, chamada de forma *triplet*. Esta lista consiste em dois vetores de inteiros *i*, *j* e um vetor de números reais *x*, ambos de comprimento igual ao número de entradas na matriz. Por exemplo, a matriz presente na figura 3.1 é representa na forma *triplet* apresentada na figura 3.2.

```
int i [ ]    = { 2,   1,   3,   0,   1,   3,   3,   1,   0,   2   } ;
int j [ ]    = { 2,   0,   3,   2,   1,   0,   1,   3,   0,   1   } ;
double x [ ] = { 3.0, 3.1, 1.0, 3.2, 2.9, 3.5, 0.4, 0.9, 4.5, 1.7 } ;
```

Figura 3.2. Matriz armazenada na forma *triplet*. Figura extraída de Davis, 2006, p.8

A estrutura de dados mostrada acima para uma matriz *m* por *n*, contém índices de linha e coluna no intervalo de 0 a *m*-1 e 0 a *n*-1. Por essa característica, esta estrutura de dados ganha uma denominação especial, chamada *zero-based triplet*.

Segundo Davis (2006), a forma *triplet* é de fácil criação, mas difícil de usar na maioria dos algoritmos de matrizes esparsas. Por conta disso, a forma *compressed-column* é mais útil e é utilizada em quase todas as funções da CSparse. Nesta estrutura existe um campo inteiro chamado *nzmax*, indicando que uma matriz esparsa com *m* linhas e *n* colunas pode conter até *nzmax* entradas.

Existe ainda na representação *compressed-column* um vetor *p* de inteiros com tamanho *n*+1, além de um vetor *i*, também de inteiros, de comprimento *nzmax*, e um vetor *x* de valores reais de tamanho também *nzmax*.

O vetor *i* guarda quais os índices das linhas que contém os elementos na matriz original. Já o vetor *p* mostra em qual posição no vetor *i* está ocorrendo uma mudança de coluna.

A figura 3.3 mostra a matriz da figura 3.1 representada na forma *compressed-column*. Cabe ainda dizer que a primeira entrada *p* [0] é sempre zero, e o valor presente em *p* [*n*] é menor ou igual a *nzmax*, que é o número de entradas numéricas não nulas na matriz.

```
int p [ ]    = { 0,            3,            6,        8,       10 } ;
int i [ ]    = { 0,   1,   3,   1,   2,   3,   0,   2,   1,   3   } ;
double x [ ] = { 4.5, 3.1, 3.5, 2.9, 1.7, 0.4, 3.2, 3.0, 0.9, 1.0 } ;
```

Figura 3.3. Matriz armazenada na forma *compressed-column.* Figura extraída de Davis, 2006, p.8

Por exemplo, o índice da linha em que o primeiro valor não nulo presente na coluna 3 da matriz da figura 3.1 é dado por *i* [*p* [3]] = *i* [8] = 1 e o respectivo valor numérico será *x* [*p* [3]] = *x* [8] = 0,9.

De acordo com Davis (2006), a estrutura de dados que armazena tanto a forma *triplet* quanto a forma *compressed-column* é a mostrada na figura 3.4.

```
typedef struct cs_sparse
{
    int nzmax ;
    int m ;
    int n ;
    int *p ;
    int *i ;
    double *x ;
    int nz ;
} cs ;
```

Figura 3.4. Estrutura de dados da biblioteca CSparse. Figura extraída de Davis, 2006, p.8

Os campos desta estrutura de dados são definidos da seguinte maneira: *nzmax* é o número máximo de entradas na matriz. O campo *m* corresponde ao número de linhas e *n* ao número de colunas. O campo *i* é um ponteiro para o vetor dos índices das linhas, cujo tamanho é *nzmax*. Já o campo *x* é um ponteiro para o vetor de valores numéricos, onde o tamanho desta lista também é *nzmax*. O campo *nz* é o número de entradas da matriz quando ela for da forma *triplet*, e -1 quando for da forma *compressed-column*.

O ponteiro *p* contém um vetor das posições de onde as colunas se iniciam no vetor *i* quando a matriz está na forma *compressed-column*. O tamanho deste vetor é *n*+1. Quando a matriz está na forma *triplet*, este ponteiro equivale ao vetor de índices das colunas da matriz e possui o tamanho *nz*.

Segundo Davis (2006), qualquer função da biblioteca CSparse tem definida se ela vai receber como entrada uma matriz na forma *triplet* ou na forma *compressed-column*, com exceção das funções *cs_print, cs_spalloc, cs_spfree* e *cs_sprealloc* que podem operar com qualquer uma das duas formas.

3.3 Visão geral sobre alguns algoritmos.

A maioria dos algoritmos da biblioteca CSparse requer uma matriz esparsa no formato *cs* como entrada. Algumas funções são necessárias para criar esta estrutura de dados.

3.3.1 *Cs_malloc*, *cs_calloc*, *cs_realloc* e *cs_free*.

Segundo DAVIS (2006), as funções *cs_malloc*, *cs_calloc*, *cs_realloc* e *cs_free* são feitas de tal forma que fiquem em torno do equivalente a biblioteca ANSI C.

A função *cs_malloc* aloca um espaço de memória não inicializado, com o tamanho *n* passado por parâmetro. Ela retorna o bloco alocado no caso de sucesso e um ponteiro nulo no caso de erro.

A função *cs_calloc* também aloca um espaço de memória de tamanho *n* passado por parâmetro, porém essa memória é limpa pela função.

A função cs_free libera o bloco de memória *p* que foi passado por parâmetro. A figura 3.5 mostra os cabeçalhos destas funções.

```
void *cs_calloc (int n, size_t size)
void *cs_malloc (int n, size_t size)
void *cs_free (void *p)
```

Figura 3.5. Cabeçalhos das funções *cs_calloc, cs_malloc e cs_free*. Figura extraída de Davis, 2006, p.10

A função *cs_realloc* realiza mudanças no tamanho de um bloco de memória. No caso de sucesso, a função retorna um ponteiro para o novo bloco de memória, com o tamanho alterado para o valor da multiplicação dos parâmetros *n* e *size*. No caso de sucesso, a função ainda atribui o valor *true* ao *parâmetro ok*. Já no caso de falha, o ponteiro *p* continua com seus valores e dimensões originais e a função atribui *false* ao parâmetro *ok*. A figura 3.6 mostra o cabeçalho da função *cs_realloc*.

```
void *cs_realloc (void *p, int n, size_t size, int *ok)
```

Figura 3.6. Cabeçalho da função *cs_realloc*. Figura extraída de Davis, 2006, p.11

3.3.2 *Cs_spalloc*

A figura 3.7 mostra o cabeçalho da função *cs_spalloc.* Esta função cria uma matriz esparsa com *m* linhas e *n* colunas, que pode armazenar até *nzmax* entradas. O vetor de valores numéricos é alocado se o parâmetro *values* é verdadeiro. Com este parâmetro sendo falso criamos uma matriz binária. Para essas matrizes, é desnecessário

utilizar um vetor para armazenar valores, pois todos os elementos não nulos da matriz terão o valor 1. A matriz é definida da forma *triplet* ou *compressed-colunm*, dependendo se o parâmetro *triplet* é verdadeiro ou falso.

```
cs *cs_spalloc (int m, int n, int nzmax, int values, int triplet)
```

Figura 3.7. Cabeçalho da função *cs_spalloc*. Figura extraída de Davis, 2006, p.11

3.3.3 *Cs_spfree* e *cs_sprealloc*

A figura 3.8 mostra os cabeçalhos das funções *cs_spfree* e *cs_sprealloc*. A função *cs_spfree* libera todos os blocos de memória ocupados pela matriz esparsa *A*, que é passada por parâmetro. Já a função *cs_sprealloc* recebe a matriz esparsa *A* por parâmetro e altera o número máximo de entradas que ela pode conter. O novo valor será dado pelo parâmetro da função *nz*. Para as duas funções, a matriz pode ser tanto da forma *triplet* quando da forma *compressed_column*.

```
cs *cs_spfree (cs *A)
int cs_sprealloc (cs *A, int nzmax)
```

Figura 3.8. Cabeçalhos das funções *cs_spfree* e *cs_sprealloc*. Figura extraída de Davis, 2006, p.11

3.3.4 *Cs_csc* e *cs_triplet*

Dois testes muito importantes da biblioteca de matrizes esparsas são implementados como definições da linguagem C. O teste CS_CSC retorna 1 se a matriz de entrada for da forma *compressed-column*, e 0 se for de outra forma ou não existir. Analogamente, o teste CS_TRIPLET retorna 1 se a matriz de entrada for da forma *triplet*, e 0 se a for de outra forma ou não existir. A figura 3.9 mostra como foram definidos os seguintes testes. Vale lembrar que o campo *nz* da estrutura de dados *cs* equivale à contagem do número de entradas da matriz e tem valor -1 se a matriz for da forma *compressed-column*.

```
#define CS_CSC(A) (A && (A->nz == -1))
#define CS_TRIPLET(A) (A && (A->nz >= 0))
```

Figura 3.9. Definições dos testes *CS_CSC* e *CS_TRIPLET*. Figura extraída de Davis, 2006, p.10

3.3.5 *Cs_entry*

A função *cs.entry* adiciona um valor na matriz *T* passada por parâmetro. De acordo com Davis (2006), se não houver espaço suficiente para a adição da próxima entrada, o tamanho dos vetores *i*, *j* e *x* da matriz é dobrado. Já as outras dimensões da matriz *T*, dadas pelos valores de *nz* e *nzmax*, são aumentadas conforme necessário.
Esta função recebe por parâmetro a matriz *T* onde será adicionado um novo valor. Os outros parâmetros da função são os índices *i* e *j*, que correspondem à linha e coluna da posição onde o valor será inserido. A função recebe ainda o valor numérico *x* que será adicionado na matriz. Cabe ressaltar que função *cs_entry* só aceita uma matriz de entrada na forma *triplet*. *A* figura 3.10 mostra o cabeçalho desta função.

```
int cs_entry (cs *T, int i, int j, double x)
```

Figura 3.10. Cabeçalho da função *cs_entry*. Figura extraída de Davis, 2006, p.12

3.3.6 *Cs_compress*

A função *cs_compress* converte uma matriz na forma *triplet* em uma matriz na forma *compressed-column*. Esta função recebe por parâmetro a matriz *T* na forma *triplet* e retorna a nova matriz na forma *compressed-column*. O cabeçalho desta função é mostrado na figura 3.11.

```
cs *cs_compress (const cs *T)
```

Figura 3.11. Cabeçalho da função *cs_compress*. Figura extraída de Davis, 2006, p.13

3.3.7 *Cs_cumsum*

Segundo Davis (2006), o cálculo de uma soma cumulativa é útil em várias funções dentro da biblioteca CSparse, portanto isto é criada uma função específica para esse cálculo, a *cs_cumsum*. Esta função recebe por parâmetro dois vetores de inteiros *p* e *c*, além de um inteiro *n* que contém o tamanho dos dois vetores anteriores.

Os valores do vetor *p* são calculados de tal forma que *p* [i] será a soma de *c* [0] até *c* [*i*-1]. No final, os valores de *c* [0 ... n-1] são substituídos por *p* [0 ... n-1]. Esta função ainda retorna o valor do somatório de todos os valores de *p* [0 ... n-1]. *O* cabeçalho da função *cs_cumsum* é mostrado na figura 3.11.

```
double cs_cumsum (int *p, int *c, int n)
```

Figura 3.12. Cabeçalho da função *cs_cumsum*. Figura extraída de Davis, 2006, p.13

3.3.8 *Cs_print*

A função *cs_print* imprime o conteúdo da matriz *A* passada por parâmetro. O segundo parâmetro desta função tem o nome de *brief* e diz se apenas as primeiras entradas da matriz *A* serão impressas. Esta função retorna 0 se a matriz *A* não existir. Em todos os outros casos, a função retorna 1. O cabeçalho desta função é mostrado na figura 3.13.

```
int cs_print (const cs *A, int brief)
```

Figura 3.13. Cabeçalho da função *cs_print*. Figura extraída de Davis, 2006, p.23

3.3.9 *Cs_done*

A função *cs_done* libera todos os espaços de trabalho e pode retornar também um ponteiro nulo ou então uma matriz esparsa. O primeiro parâmetro é a matriz esparsa *C*, do tipo *cs*, que pode ser tanto da forma *triplet* quanto da *compressed-column*. O segundo e o terceiro parâmetros são os ponteiros genéricos *w* e *x* para os espaços de trabalho que serão liberados. Já o quarto parâmetro é um inteiro com nome *ok* que diz se a matriz esparsa passada no primeiro parâmetro será ou não retornada. No caso positivo, a matriz *C* é simplesmente retornada. Já no caso negativo, a função retorna um ponteiro nulo e a matriz *C* é liberada com a chamada da função *cs_spfree*. A figura 3.14 mostra o cabeçalho da função *cs_done*.

```
cs *cs_done (cs *C, void *w, void *x, int ok)
```

Figura 3.14. Cabeçalho da função *cs_done*. Figura extraída de Davis, 2006, p.13

3.3.10 *Cs_gaxpy*

Para Davis (2006), um dos mais simples algoritmos de matrizes esparsas é a multiplicação de matriz-vetor, $Z = Ax + y$, onde os parâmetros *y* e *x* são vetores densos e o parâmetro *A* é uma matriz esparsa.

O código da função *cs_gaxpy* é dado pela figura 3.15 abaixo.

```
int cs_gaxpy (const cs *A, const double *x, double *y)
{
    int p, j, n, *Ap, *Ai ;
    double *Ax ;
    if (!CS_CSC (A) || !x || !y) return (0) ;       /* check inputs */
    n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    for (j = 0 ; j < n ; j++)
    {
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            y [Ai [p]] += Ax [p] * x [j] ;
        }
    }
    return (1) ;
}
```

Figura 3.15. Código da função *cs_gaxpy.* Figura extraída de Davis, 2006, p.10

A função começa verificando se os parâmetros passados como entrada existem, retornando 0 no caso negativo. Isso protege contra um chamador que ficou sem memória. É importante dizer que a matriz *A* dada como entrada tem que ser da forma *compressed-column*. Após essa parte inicial, o cálculo $Ax + y$ é efetuado, retornando o resultado no ponteiro *y* passado por parâmetro. A função ainda retorna 1 no caso do cálculo ser efetuado com sucesso.

3.3.11 *Cs_transpose*

A função *cs_transpose* realiza a transposição de matrizes ($C = A^t$) utilizando a estrutura de dados de matrizes esparsas. Esta função será mostrada com mais detalhes do que as anteriores visto que esta foi uma das operações paralelizadas.

O algoritmo começa calculando a contagem do número de elementos das linhas da matriz esparsa *A* passada por parâmetro. Logo depois, o algoritmo calcula a soma cumulativa para obter os ponteiros das linhas, e, em seguida, itera sobre cada entrada diferente de zero em *A*, colocando-a em seu lugar apropriado no vetor de linhas da matriz *C*, que será o resultado final da transposição. É importante dizer que a matriz A deve ser da forma *compressed_column*.

Além da matriz *A* de entrada, a função *cs_transpose* recebe também o parâmetro *values* que informa se a matriz A contém o vetor *x* de valores reais. Se não tiver, temos um caso de transposição de uma matriz binária. Segue abaixo o código da função *cs_transpose*.

```
cs *cs_transpose (const cs *A, int values)
{
    int p, q, j, *Cp, *Ci, n, m, *Ap, *Ai, *w ;
    double *Cx, *Ax ;
    cs *C ;
    if (!CS_CSC (A)) return (NULL) ;
    m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    C = cs_spalloc (n, m, Ap [n], values && Ax, 0) ;
    w = cs_calloc (m, sizeof (int)) ;
    if (!C || !w) return (cs_done (C, w, NULL, 0)) ;
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    for (p = 0 ; p < Ap [n] ; p++) w [Ai [p]]++ ;
    cs_cumsum (Cp, w, m) ;
    for (j = 0 ; j < n ; j++)
    {
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            Ci [q = w [Ai [p]]++] = j ;
            if (Cx) Cx [q] = Ax [p] ;
        }
    }
    return (cs_done (C, w, NULL, 1)) ;
}
```

Figura 3.16. Código da função *cs_transpose*. Figura extraída de Davis, 2006, p.14

Para exemplificar o funcionamento deste algoritmo, seja a matriz *A*, passada como entrada para a função *cs_transpose*, dada a seguir:

$$A = \begin{bmatrix} 1.2 & 0 & 2.1 \\ 0 & 5.3 & 4.7 \\ 0 & 3.0 & 0 \end{bmatrix}$$

Os vetores de linha, coluna e valor para essa matriz ficariam com os seguintes dados:

$Ap = [0, 1, 3, 5]$
$Ai = [0, 1, 2, 0, 1]$
$Ax = [1.2, 5.3, 3.0, 2.1, 4.7]$

A função começa verificando se a matriz *A* é da forma *compressed-column*. Se não for, a transposição não é executada e a função retorna um ponteiro nulo. Após isso, são alocados um espaço de trabalho *w* de tamanho igual ao número de linhas da matriz original, além da matriz *C*, que será o resultado final da transposição de *A*.

Após isso, é calculado o somatório do número de elementos de cada linha de *A*, armazenado esse valor em *w*. Depois é realizada a soma parcial de *w*, colocando esse

valor no vetor *Cp* e também em *w*. Isso é feito porque, dado a soma do número de elementos de cada linha da matriz original, após realizar a transposição, esses valores corresponderão ao somatório dos elementos de cada coluna da matriz final. Sendo assim, após realizar as somas parciais destes valores, teremos os índices do vetor *Ci* onde se inicia cada uma das colunas, que é justamente o conteúdo que o vetor *Cp* deve conter.

Após essa parte, o algoritmo percorre os elementos da matriz *A*, coluna por coluna e os colocam nas suas respectivas posições na matriz *C*, ajustando os vetores *Ci* e *Cx*.

Com a conclusão do código, a matriz *C* e seus respectivos vetores de linha, coluna e valor ficam da seguinte forma:

$$C = \begin{bmatrix} 1.2 & 0 & 0 \\ 0 & 5.3 & 3.0 \\ 2.1 & 4.7 & 0 \end{bmatrix}$$

$$Cp = [0, 2, 4, 5]$$

$$Ci = [0, 2, 1, 2, 1]$$

$$Cx = [1.2, 2.1, 5.3, 4.7, 3.0]$$

Ao final da execução da função *cs_transpose*, a matriz *C* é retornada e o espaço de trabalho *w* é liberado. Isso tudo ocorre com a chamada da função *cs_done*, passando a matriz *C*, o vetor *w* e o valor 1 no parâmetro *ok*, que diz se a matriz deve ser liberada ou retornada, conforme discutido na seção 3.3.9.

3.4 Conclusão

Devido a grande presença de matrizes esparsas em vários problemas de diversos ramos da Ciência, além também da grande dificuldade de armazenar e operar com grandes matrizes com esta característica, surge a necessidade de formatos especiais de armazenamento de dados esparsos, onde somente os elementos não nulos são considerados. Estes formatos especiais são as estruturas de dados criadas com a finalidade de armazenar somente o que realmente interessa nas matrizes esparsas, ou seja, seus valores não nulos, evitando assim gastos de memória desnecessários.

Juntamente com as estruturas de dados, surgem também algoritmos especializados em trabalhar com tais estruturas, pois todas as operações realizadas em cima da forma de armazenamento padrão de matrizes (vetor bidimensional), não funcionarão com as novas estruturas de dados.

Por conta destes problemas e ainda com o objetivo de melhorar a eficiência nas operações sobre matrizes esparsas, surge a necessidade da criação de uma biblioteca específica, que tenha as mais variadas operações com este tipo específico de matrizes. A biblioteca em questão que destacamos neste trabalho é a CSparse, discutida ao longo deste capitulo.

4 A Paralelização

4.1 Introdução

A biblioteca CSparse possui módulos que realizam as mais diversas operações com matrizes esparsas, porém todos os algoritmos originais desta biblioteca são implementados na forma sequencial. Este capitulo traz o objetivo principal deste projeto, que é a criação de versões paralelas de alguns módulos da CSparse, comparando sempre o tempo de execução da versão paralela com a sequencial.

4.2 Transposição

Um dos módulos implementados foi a transposição de matrizes esparsas. O módulo se chama *cs_transpose* e faz parte da biblioteca CSparse. Este módulo recebe uma matriz na forma *compressed-column*, conforme visto na seção 3.3.11.

Para realizar o cálculo, foi criado um fluxo de computação para cada coluna da matriz original. A transposição da matriz esparsa é calculada então seguindo a lógica utilizada no código original da biblioteca.

Seja w um vetor alocado para ser um espaço de trabalho da função, e C a matriz que terá a transposição de A, passada por parâmetro. No vetor *w* é armazenado inicialmente o número de elementos de cada linha da matriz original. Depois, é calculada a soma acumulativa de *w*, armazenando o resultado em *w* e em *Cp*. Estes valores correspondem aos índices nos vetores *Ci* e *Cx* em que as colunas estão se iniciando. O valor de *w* é utilizado para iniciar a atribuição dos elementos de cada coluna da matriz transposta. Já o valor de *Cp* é o que ele deve conter após a transposição. Conforme visto anteriormente, o vetor *Cp* guarda os índices dos inícios das colunas em uma matriz na forma *compressed-column*.

Na versão sequencial, o vetor *w* é incrementado a cada iteração. Para realizar a operação em paralelo, isso não poderia acontecer, pois não é garantido que os incrementos ocorrerão da maneira correta. Sendo assim, um laço em cada fluxo de computação percorre *w*, calculando o deslocamento *d* que ele deve sofrer com base em quantos seriam os incrementos que *w* deveria ter sofrido até essa fase da operação.

Ainda na versão sequencial, um teste acontece para verificar se a matriz era binária ou não. No caso negativo, a matriz não teria o vetor *Cx*. Para diminuir os testes dentro de cada fluxo de computação, foram criados dois núcleos diferentes. Um para quando existir o vetor *Cx* na matriz e outro para o caso em que ele não existir. Esse teste agora acontece na parte do código que executa na CPU e, dependendo do resultado, um dos núcleos é chamado.

```
__global__ void transposeCX(int *ApC, int *CiC, int *wC, int *AiC,
double *AxC, double *CxC, int *nThreadsC){
    int p, q, d, i;
    int j = blockIdx.x*TAMBLOCK + threadIdx.x + threadIdx.y*DIMBLOCK +
    threadIdx.z*DIMBLOCK*DIMBLOCK;

    if (j < *nThreadsC){
        for (p = ApC [j] ; p < ApC [j+1] ; p++){
            d = 0;
            for (i = 0; i < p; i++)
                if (AiC[i] == AiC[p]) d++;
            q = wC [AiC [p]] + d;
            CiC [q] = j ;
            CxC [q] = AxC [p] ;
        }
    }
}
```

Figura 4.1. Núcleo da versão paralela da função *cs_transpose*.

A figura 4.1 apresenta o código do núcleo que é invocado quando a matriz não é binária. O outro núcleo difere deste exposto apenas no fato de não existir a linha que realiza a atribuição $CxC[\,q\,] = AxC\,[\,q\,]$.

Um exemplo dos valores em cada iteração é mostrado a seguir. Seja *A* uma matriz esparsa dada de entrada no módulo de transposição, com seus respectivos vetores *Ap*, *Ai* e *Ax*.

$$A = \begin{bmatrix} 1.2 & 0 & 2.1 \\ 0 & 5.3 & 4.7 \\ 0 & 3.0 & 0 \end{bmatrix}$$

$$Ap = [0, 1, 3, 5]$$

$$Ai = [0, 1, 2, 0, 1]$$

$$Ax = [1.2, 5.3, 3.0, 2.1, 4.7]$$

Da mesma maneira, seja *C* a matriz que terá o valor de A^t ao final dos cálculos. *Ci*, *Cx* e *Cp* são os vetores que armazenam a matriz *C*, na forma *compressed-column*.

Como existe um fluxo de computação para cada uma das colunas, existe um laço que percorre os elementos da coluna, variando a linha. Dentro deste laço é calculado o deslocamento dos índices dos vetores *Ci* e *Cx,* dado pelos elementos de *w*. Após essa fase, os vetores *Ci* e *Cx* recebem os índices das linhas e os valores numéricos, respectivamente.

Seja *j* o identificador do fluxo de computação, *p* o índice que percorre os elementos de uma certa coluna, *q* o índice de *Ci* e *Cx* que receberá os dados e *d* o deslocamento que o vetor *w* sofrerá. Abaixo segue os valores destas variáveis em cada um dos fluxos e também em cada um das iterações de *p*.

$j = 0$:

$p = 0$:

$d = 0;$

$q = 0;$

$Ci[0] = 0;$

$Cx[0] = 1.2;$

$j = 1$:

$p = 1$:

$d = 0;$

$q = 2;$

$Ci[2] = 1;$

$Cx[2] = 5.3;$

$p = 2$:

$d = 0;$

$q = 4;$

$Ci[4] = 1;$

$Cx[4] = 3.0;$

$j = 2$:

$p = 3$:

$d = 1;$

$q = 1;$

$Ci[1] = 2;$

$Cx[1] = 2.1;$

$p = 4$:

$d = 1;$

$q = 3;$

$Ci[3] = 2;$

$Cx[3] = 4.7;$

Após estes cálculos nos fluxos de computação, teremos a matriz *C*, com seus respectivos vetores *Cp*, *Ci* e *Cx*, como resultado da transposição da matriz *A*.

$$C = \begin{bmatrix} 1.2 & 0 & 0 \\ 0 & 5.3 & 3.0 \\ 2.1 & 4.7 & 0 \end{bmatrix}$$

$Cp = [0, 2, 4, 5]$

$Ci = [0, 2, 1, 2, 1]$

$Cx = [1.2, 2.1, 5.3, 4.7, 3.0]$

Vale lembrar que o vetor *Cp* é calculado da mesma maneira que a versão sequencial, com a soma acumulativa do vetor *w*. Este cálculo é efetuado na parte que executa na CPU, portanto se encontra fora do núcleo paralelo mostrado na figura 4.2.

4.3 Multiplicação de Matriz por Vetor

O outro módulo paralelizado nesse projeto foi o que implementa o cálculo $y = Ax + y$, onde *A* é uma matriz esparsa e os dois vetores *x* e *y* são densos. O módulo se chama *cs_gaxpy* e também faz parta da biblioteca CSparse.

A implementação original recebe uma matriz na forma *compressed-column*. Porém, para que fosse possível gerar uma versão paralela deste módulo, a matriz de entrada foi alterada para a forma *triplet*. Ambas as formas já foram tratadas no capítulo 3, onde a biblioteca CSparse é apresentada.

A estratégia utilizada foi criar um fluxo de computação para cada uma das posições do vetor *y*, onde o resultado final da operação é armazenado. Cada um dos fluxos criados percorre todos os elementos da matriz, descobrindo qual deles vai influenciar no cálculo da posição a que este fluxo se refere no vetor *y*. O núcleo paralelo que foi implementado é mostrado na figura 4.2.

```
__global__ void gaxpy(int *AiC, int *AjC, int *nzC, double *AxC,
double *xC, double *yC, int *nThreadsC){
    int i;
    int t = blockIdx.x*TAMBLOCK + threadIdx.x + threadIdx.y*DIMBLOCK +
    threadIdx.z*DIMBLOCK*DIMBLOCK;
    double soma = 0;

    if (t < *nThreadsC){
        for (i = 0; i < *nzC; i++){
            if (AiC[i] == t) soma += AxC[i] * xC[AjC[i]];
        }
        yC[t] += soma;
    }
}
```

Figura 4.2. Núcleo da versão paralela da função *cs_gaxpy*.

O cálculo foi realizado da seguinte forma:

Seja *t* o identificador de cada um dos fluxos de computação, *Ai* o vetor que guarda os índices das linhas que contém valores, *Aj* o vetor que guarda o índice das colunas que contém valores, *Ax* o vetor que guarda os valores da matriz e *nz* o número de elementos da matriz original. O vetor resultante *y* será gerado com a operação a seguir.

$$(\forall u)\,((u \in [\,0\,..\,nz-1]) \wedge (Ai[u] = t) \rightarrow y[t] \mathrel{+}= Ax[u] * X\big[Aj[u]\big])$$

Segue um exemplo para demonstrar os cálculos.

Sejam os parâmetros passados para o módulo em questão a matriz *A* com seus respectivos vetores *Ai*, *Aj* e *Ax*, além dos vetores *x* e *y*, apresentados a seguir.

$$A = \begin{bmatrix} 1.2 & 0 & 2.1 \\ 0 & 5.3 & 4.7 \\ 0 & 3.0 & 0 \end{bmatrix}$$

$$Ai = [0, 1, 2, 0, 1]$$

$$Aj = [0, 1, 1, 2, 2]$$

$$Ax = [1.2, 5.3, 3.0, 2.1, 4.7]$$

$$x = [2, 1, 5]$$

$$y = [3, 2, 2]$$

É importante dizer que as dimensões dos vetores *x* e *y* devem ser o número de colunas e de linhas da matriz *A*, respectivamente.

Como o vetor *y* tem três posições, serão criados três fluxos de computação para este cálculo.

Com estes valores, os fluxos farão os cálculos da seguinte maneira:

$t = 0 \Rightarrow y[0] := 3 + 1.2 * 2 + 2.1 * 5 \Rightarrow y[0] = 15.9$

$t = 1 \Rightarrow y[1] := 2 + 5.3 * 1 + 4.7 * 5 \Rightarrow y[1] = 30.8$

$t = 2 \Rightarrow y[2] := 2 + 3.0 * 1 \Rightarrow y[2] = 5.0$

Dessa forma, a chamada ao módulo *cs_gaxpy* passando a matriz *A* e os vetores *x* e *y*, ambos mostrados acima, geraria como resultado o seguinte vetor *y*:

$y = [15.9, 30.8, 5.0]$

4.4 Resultados

Apresar de todo esforço para que módulos da biblioteca original fossem paralelizados, as acelerações alcançadas foram quase nulas.

A aceleração *Sp* foi calculada com a fórmula a seguir.

$$Sp = \frac{T_1}{T_p}$$

De acordo com Scott, Clark & Bagheri (2005), *T1* deve ser o tempo de execução do código na forma sequencial e *Tp* deve ser o tempo de execução da solução paralela, em *p* processadores.

Para os testes, foram utilizadas matrizes com 20% delas contendo valores significativos. O código *cs_transpose* permitiu a execução com matrizes de até 800x800 elementos. Já o código *cs_gaxpy* permitiu a execução com matrizes maiores, de até 5000x5000 elementos.

Essa diferença de tamanhos acontece porque o desbalanceamento de carga dos fluxos de computação é muito grande no código da transposição. Sendo assim, quando as dimensões das matrizes crescem, cresce também a discrepância de carga dos fluxos, chegando ao ponto do código não executar corretamente.

O módulo *cs_transpose* obteve a aceleração simbólica de 1.4%. Já o módulo *cs_gaxpy* obteve a aceleração também pequena, em torno de 6%. Todos estes resultados foram obtidos utilizando as dimensões limites, mostradas acima.

4.5 Dificuldades

Diversos problemas foram encontrados durante a realização deste trabalho. As particularidades das estruturas de dados presentes na biblioteca CSparse impediram bastante que os módulos fossem criados de uma maneira que explorasse melhor a arquitetura paralela do CUDA.

Por exemplo, se a biblioteca armazenasse uma matriz com um vetor de duas dimensões, a transposição sairia de uma forma muito mais rápida, criando um fluxo de computação para cada uma das posições da matriz original. Porém, como já foi tratado em um momento anterior, essa forma de armazenamento gastaria recursos desnecessários do computador, motivo pelo qual estruturas de dados especiais são utilizadas para o armazenamento interno de uma matriz esparsa.

Outra grande dificuldade foi a programação em CUDA. Apresar da linguagem estar muito próxima do C, a forma de se programar é completamente diferente, o que leva a uma alteração na maneira de desenvolver a solução dos problemas propostos.

4.6 Conclusão

O objetivo deste projeto, que é paralelizar alguns módulos da biblioteca CSparse, foi alcançado. Porém, a aceleração alcançada com estes módulos paralelos não foi satisfatória. O CUDA pode prover uma aceleração dos códigos muito maior que a gerada neste trabalho, devido a toda sua arquitetura paralela.

Para que melhores resultados sejam alcançados, será preciso uma reestruturação dos módulos criados para que o paralelismo de dados presentes em matrizes fosse melhor aproveitado.

Será preciso também diminuir a colisão de acessos aos dados pelos fluxos de computação, de modo a se utilizar melhor a hierarquia de memória do CUDA. Caso os fluxos acessem as mesmas posições de memória ao mesmo tempo, o acesso é serializado, o que por sua vez limita muito o desempenho dos módulos paralelos.

Outro ponto importante é diminuir o tempo em que os fluxos de computação ficam ociosos. Da maneira que está implementado, existem muitos fluxos ociosos grande parte do tempo de execução da aplicação.

No exemplo da transposição, fica claro também o desbalanceamento de carga de cada um dos fluxos de computação. Enquanto o fluxo com o identificador 0 executa somente uma iteração de p, os outros fluxos executam, cada um deles, duas iterações de p. Este desbalanceamento também é um fator que limita a busca da aceleração na versão paralela.

5 Considerações Finais

O aumento no poder de processamento que surgia a cada lançamento de um novo *chip* pelos fabricantes começou a diminuir, devido a diversos fatores físicos existentes na fabricação dos *chips*. Dessa maneira, os fabricantes começaram a procurar outras maneiras de, ainda assim, conseguirem alcançar melhorias no poder de processamento de seus novos *chips*. Duas estratégias foram encontradas pelos projetistas: a *multicore* e a *many-core*.

O CUDA (*Compute Unified Device Architecture*) é um exemplo de arquitetura que utiliza a plataforma *many-core.* Este foi desenvolvido pela NVIDIA e foi escolhido como arquitetura alvo para a paralelização dos códigos deste projeto. Ele utiliza GPUs (Unidades de Processamento Gráfico), que possuem um grande poder de processamento provido pelos seus diversos núcleos, que foi alavancado pela grande indústria dos jogos.

Para a paralelização proposta neste trabalho, foram escolhidos módulos de uma biblioteca de operações sobre matrizes esparsas chamada CSparse. Esta biblioteca possui estruturas de dados e diversas operações, todos estes aplicados a classe de matrizes esparsas. Um dos módulos paralelizados foi o *cs_transpose,* que realiza a transposição de uma matriz esparsa. Outro módulo que também foi paralelizado foi o *cs_gaxpy* que realiza a operação $y = Ax + y$, onde *A* é uma matriz esparsa e *x* e y são dois vetores densos.

Para o código *cs_gaxpy* foi alterada a estrutura de dados da matriz de entrada *A*. No código original, a matriz *A* estava na forma *compressed-column* e, após a paralelização, a matriz *A* deve ser passada na forma *triplet*. Só assim foi possível gerar versões paralelas deste módulo. Já no código *cs_transpose,* a estrutura de dados da matriz de entrada foi mantida na mesma forma, mas a paralelização foi feita parcialmente, criando um fluxo para cada coluna e não um fluxo para cada elemento da matriz transposta, o que seria o ideal.

O objetivo deste trabalho, que era paralelizar os módulos da biblioteca de matrizes esparsas, foi alcançado. Porém, a aceleração obtida não foi satisfatória, pois sabe-se que o CUDA pode prover uma aceleração muito maior que a alcançada.

Diversos problemas foram encontrados durante a realização deste trabalho, como a dificuldade de explorar, de uma maneira mais eficiente, o paralelismo dos dados presentes em matrizes.

Foram mostrados neste projeto quais pontos importantes os futuros desenvolvedores de módulos paralelos desta biblioteca deverão se preocupar para que melhores resultados sejam alcançados.

REFERÊNCIAS

DAVIS, Timothy A. Direct Methods for Sparse Linear Systems: Fundamentals of Algorithms. Florida: SIAM, 2006. 217p.

KIRK, David B.; HWU, Wen-mei W. Programming Massively Parallel Processors: A Hands-on Approach. Burlington: Elsevier, 2010. 258p.

NVIDIA CORPORATION. GPU Programming Guide: Version 2.5.0. Disponível na Internet. http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide.pdf.19 ago. 2010.

SCOTT, L. Ridgway; CLARK, Terry; BAGHERI, Babak. Scientific Parallel Computing. New Jersey: Princeton University Press, 2005. 416p.