Universidade Federal dos Vales do Jequitinhonha e Mucuri
Faculdade de Ciências Exatas e Tecnológicas
Departamento de Computação
Bacharelado em Sistemas de Informação

Paralelização de Algoritmos e Estruturas de Dados utilizando GPU(Graphics Processing Unit)

Douglas Januário Silva

Diamantina, Outubro de 2012

Universidade Federal dos Vales do Jequitinhonha e Mucuri
Faculdade de Ciências Exatas e Tecnológicas
Departamento de Computação
Bacharelado em Sistemas de Informação

Paralelização de Algoritmos e Estruturas de Dados utilizando GPU(Graphics Processing Unit)

Douglas Januário Silva

Monografia submetida à Banca Examinadora designada pelo curso de Sistemas de Informação da Universidade Federal dos Vales do Jequitinhonha e Mucuri, como requisito para obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Alexandre Ramos Fonseca

Diamantina, Outubro de 2012

Douglas Januário Silva

Paralelização de Algoritmos e Estruturas de dados utilizando GPU(Graphics Processing Unit)

Monografia submetida à Banca Examinadora designada pelo curso de Sistemas de Informação da Universidade Federal dos Vales do Jequitinhonha e Mucuri, como requisito para obtenção do título de Bacharel em Sistemas de Informação.

COMISSÃO EXAMINADORA

Prof. Dr. Alexandre Ramos Fonseca (Orientador)
Universidade Federal dos Vales do Jequitinhonha e Mucuri

Prof. MSc. Cristiano Grijó Pitangui
Universidade Federal dos Vales do Jequitinhonha e Mucuri

Prof. MSc. Raphael Santin
Universidade Federal dos Vales do Jequitinhonha e Mucuri

Diamantina, Outubro de 2012

Agradecimentos

Agradeço primeiramente aos meus pais, que nos momentos de dificuldades sempre me apoiaram, não medindo esforços para que este momento chegasse.

Agradeço a 3ª turma de Sistemas de Informação da UFVJM pelos quatro anos maravilhosos de convivência e aprendizado, principalmente aos amigos Guilherme, Gustavo, Henrique e Igor.

Agradeço a todo Departamento de Computação da UFVJM, pelo esforço para formação de ótimos profissionais da área de tecnologia. Agradecimentos especiais a Oscar e Alan, que por diversas vezes me ajudaram com algumas necessidades nos laboratórios. E um agradecimento especial ao Professor Alexandre, que no momento em que eu estava meio perdido, me propôs o tema deste trabalho e ainda me ajudou a "financiá-lo". No início foi difícil tratar de um tema relativamente novo no mercado, mas que ao final trouxe grande aprendizado e interesse pelo assunto.

Resumo

A busca por aumento no processamento de computadores, fez com que as empresas que criam processadores aplicassem muitos dos seus recursos, na busca de melhor desempenho para seus produtos. Mas devido a dificuldades financeiras e até energéticas, o desenvolvimento de processadores cada vez mais potentes não está em ritmo tão acelerado nos tempos atuais. Paralelo a tudo isso, o desenvolvimento de dispositivos gráficos, está em absoluta ascensão, principalmente devido ao crescente mercado de *games*. Neste trabalho iremos fazer uma contextualização de programação paralela e correlacionar a área de dispositivos gráficos com a mesma. Iremos também mostrar como a empresa NVIDIA desenvolveu uma nova arquitetura de dispositivos gráficos chamada CUDA, que se tornou uma alternativa de grande valor para a otimização de processamento de aplicativos de propósitos gerais. Várias áreas como Tomografia Médica, Processamento de Imagens e Sistema Financeiro ja estão investindo na montagem de máquinas que trabalhem com CUDA, e os resultados são muito impressionantes.

Foi feito uma introdução ao ambiente, recursos e ferramentas utilizados para o desenvolvimento de algoritmos com CUDA C, assim como um explicação do Modelo de arquitetura da tecnologia que trabalha com indexação de *threads* e blocos. Em paralelo, foi proposto o desenvolvimento de um algoritmo para busca de vizinhos, que é um dos gargalos nos algoritmos usados por Métodos Numéricos sem Malha. O Desenvolvimento do algoritmo visou o aprendizado e aplicação da tecnologia, fazendo ao fim do desenvolvimento, uma comparação da mesma implementação feita em um computador sem a utilização de paralelismo. Os ganhos do algoritmo paralelizado pela GPU chegaram a 15 vezes em um dispositivo gráfico de 336 núcleos.

Abstract

The search for improvement in the computers processing, made the companies that create these processors apply plenty of their resources in the inquiry of better development for their processors. However, due to some financial and even energetic difficulties, the development of more powerful processors is not in a so accelerate rhythm nowadays. Parallel to all on this the development of graphic devices is in absolute ascension, mainly due to the crescent games market. In this work we will make a contextualization of the parallel programming and correlate the area of graphic devices with it. We will also demonstrate how the company NVIDIA developed a new architecture for the graphic devices called CUDA, which has became an alternative of great value to the optimization of the processing of the general proposing apps. Several areas as the Medical Tomography, Image Processing and Financial System are already investing in an assembling machines that work with CUDA and the results are very impressive. It was done an introduction of the environment, resources and tools used to the development of algorithm with CUDA C, as well the explanation of the CUDA's architectural model that works with indexing of threads and blocks. In parallel, it was proposed the development of one algorithm to search for neighbors, which is one of the predicaments in the algorithms used by Meshless Numerical Methods. The development of the algorithm aimed the learning and the application of the technology, making in the end of that a comparison of the same implementation made in a computer without the utilization of the parallelism. The gains of the parallelized algorithm for the GPU reached to 15 times in a graphic device using 336 nucleus.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Símbolos

ALU *Arithmetic Logic Unit*

API *Application Programming Interface*

CGAL *Computational Geometry Algorithms Library*

CPU *Central Processing Unit*

CUBLAS *CUDA Basic Linear Algebra Subroutines*

CUDA *Compute Unified Device Architecture*

CUFFT *CUDA Fast Fourier Transform*

DRAM *Dynamic Random-Access Memory*

GPGPU *General Purpose Graphics Processing Unit*

GPU *Graphics Processing Unit*

MIMD *Multiple Instruction, Multiple Data*

MISD *Multiple Instruction, Single Data*

PDE *Partial Diferential Equations*

SIMD *Single Instruction, Multiple Data*

SIMT *Single Instruction Multiplethread*

SISD *Single Instruction, Single Data*

STL *Standard Template Library*

WPA *Wi-Fi Protected Access*

Capítulo 1

Introdução

Este trabalho visa a disseminação da programação paralela em placas de vídeos para propósito gerais, também conhecida como *General Purpose Graphics Processing Unit* (GPGPU), juntamente com a introdução da nova tecnologia para a mesma, a *Compute Unified Device Architecture* (CUDA), além de sua aplicação em algoritmos de alto custo computacional. Este capítulo tem o objetivo de contextualizar o problema (Seção 1.1), apresentando os objetivos a serem alcançados (Seção 1.2), bem como a motivação e justificativa para a sua realização (Seção 1.3). Também será apresentado neste capítulo algumas aplicações práticas da tecnologia CUDA (Seção 1.4) e a estrutura do trabalho (Seção 1.5).

1.1 Apresentação

As vantagens que a computação trouxe para a humanidade são inúmeras, estando todas relacionadas à velocidade na execução de determinadas tarefas do nosso cotidiano, as quais o computador consegue realizar de forma muito mais rápida que um homem. Tais vantagens são geradas graças ao intenso desenvolvimento de *hardwares* de computador ao longo dos anos, impulsionado principalmente pelo aumento da capacidade de processamento das CPU's (*Central Processing Unit*). Uma das principais estratégias utilizadas para este aumento até pouco tempo, era o incremento de *clock* nos *hardwares*. O *clock* em computadores consiste em um sinal elétrico regular que fornece uma referência de tempo para todas as atividades e permite sincronismo nas operações internas do computador.

Em 2004, foi anunciado o fim do incremento de *clock* nas CPU's, que atingiram o seu limite devido ao alto consumo de energia e aquecimento de *hardware* (Flynn, 2004). Este

fato obrigou os desenvolvedores a investir em paralelização da execução de instruções como uma das formas de se reduzir o tempo de computação de algoritmos. Essa característica de processamento paralelo é oferecida por um conjunto de processadores capazes de trabalhar cooperativamente para resolver um dado problema (Foster, 1995), aplicando-se a técnica "dividir para conquistar". Desde então, para permitir o processamento paralelo nas CPU's, tem-se trabalhado muito no aumento de núcleos do processador e na capacidade que os *softwares* têm para utilizar estes recursos. Diferentes formas de arquiteturas paralelas têm sido estudadas através dos anos, sempre buscando o aumento da eficiência e a diminuição do tempo de processamento dos algoritmos. Dentre as principais opções de técnicas de paralelismo que existiam até pouco tempo, podemos citar os supercomputadores e os *clusters* (Schepke, 2009).

Juntamente com este cenário da necessidade de aumento do processamento computacional, tem-se a crescente demanda de processamento gráfico, impulsionada principalmente pelo mercado de jogos. As GPU's (*Graphics Processing Unit*) que podemos encontrar hoje no mercado, são dispositivos de arquiteturas extremamente paralelas. Enquanto os computadores pessoais da atualidade possuem em sua arquitetura, processadores com no máximo oito núcleos, uma GPU Geforce GTX460 da NVIDIA, possui 336 núcleos em sua arquitetura, característica que demonstra a sua grande capacidade para realização de processamento paralelo. A Figura 1.1 ilustra a evolução da capacidade de processamento em uma GPU face à mesma evolução no CPU.

Já no início de 2007, tendo em vista esse enorme potencial computacional oferecido pelas GPU's, a empresa NVIDIA apresentou uma nova plataforma de *software* que causou uma grande revitalização tecnológica para as placas de vídeo: a CUDA. Com esta nova tecnologia é possível adaptar aplicações que normalmente seriam executadas em uma CPU, para serem computadas em GPU's (Sanders and Kandrot, 2010), oferecendo um novo modelo de desenvolvimento para melhoria de desempenho computacional em aplicações de propósitos gerais.

Desde então, cada vez mais pesquisadores tem se aproveitado das vantagens que a tecnologia CUDA trouxe para o ramo de programação de propósitos gerais em placas de vídeo, área de desenvolvimento que era pouco explorada, devido ao alto grau de dificuldade e consequentemente, pouco interesse de se trabalhar com as antigas ferramentas que ofereciam essa linha de desenvolvimento.

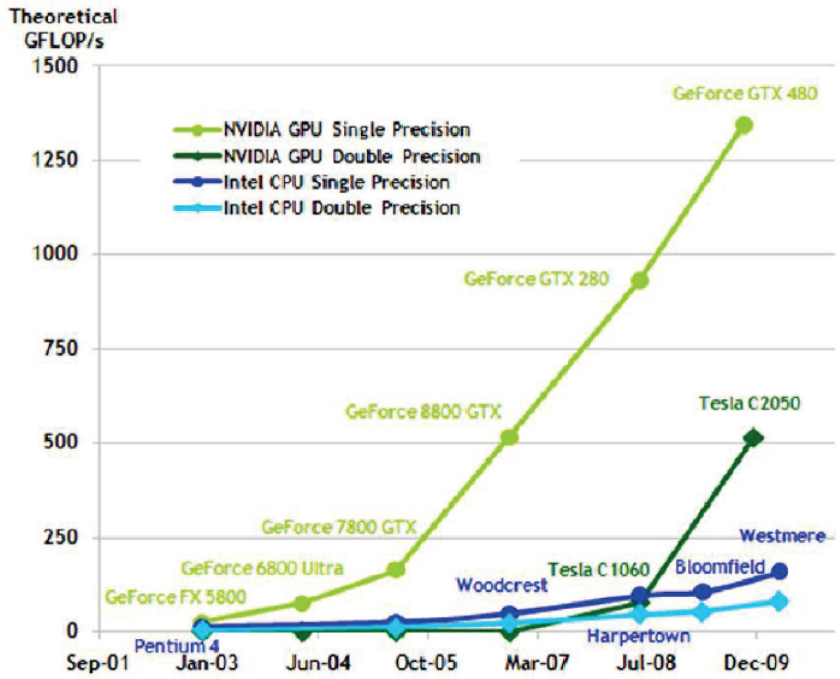


Figura 1.1: Evolução da capacidade do GPU em face da mesma evolução no CPU (Nvidia, 2012)

1.2 Objetivos

Existem áreas de estudo onde a necessidade de cálculos está além da capacidade de processamento dos computadores, como por exemplo, técnicas de simulação computacional que são frequentemente usadas para modelar e investigar fenômenos físicos. O objetivo deste trabalho é estudar a tecnologia CUDA e aplicá-la em algoritmos, buscando sempre a diminuição do custo computacional e aumento em sua eficiência.

Para que os objetivos principais sejam alcançados, é necessário que objetivos específicos sejam seguidos:

- Estudar os modelos e técnicas oferecidas pela tecnologia CUDA;
- Aplicar a tecnologia CUDA em um algoritmo de alto custo computacional;
- Alinhar e comparar os resultados obtidos com CUDA e versões de código sequencial, buscando identificar as relações de custo/benefício da nova tecnologia;
- Verificar a eficiência da tecnologia proposta.

1.3 Motivação e Justificativa

Tanto no meio acadêmico quanto na concorrência de mercado, existe uma necessidade crescente de otimizar os algoritmos de computador que auxiliam nas tarefas do dia-dia. Por mais avançados que os programas de computador estejam, sempre existem cálculos onde a melhoria de desempenho e custo se faz necessária.

Desde a criação da tecnologia CUDA no início de 2007, diferentes tipos de aplicativos e indústrias têm desfrutado de um grande sucesso ao optar por construir suas aplicações em GPU/CUDA, que até então, eram restritas à computação gráfica. Dentre as melhorias está incluso o elevado ganho de desempenho computacional no processamento das aplicações. Além disso, executar aplicações em GPU's tem mostrado melhor aproveitamento de desempenho quando comparado a custo financeiro e custo de energia (*watt*), do que aplicações implementadas exclusivamente na tradicional tecnologia de processamento central (CPU).

1.4 Aplicações do Problema

Casos de sucesso de aplicação da tecnologia CUDA já podem ser encontrados. Dentre estas aplicações podemos incluir:

- A tecnologia CUDA está ajudando pesquisadores a identificar sinais de câncer de forma mais rápida, precisa e minimamente invasiva. A ultrassonografia, que é um processo mais seguro que a radiografia, é utilizada em conjunto da mamografia para ajudar nos cuidados e diagnósticos de câncer de mama. Mas o ultrassom convencional de mama possui suas limitações, fato que levou a criação da empresa *Techniscan*, que desenvolveu um promissor método tridimensional de ultrassom, mas com sérias limitações computacionais. Converter os dados de ultrassom em imagens era considerado demorado e caro para uso prático. Após aplicar a tecnologia CUDA em seus *softwares*, *Techniscan* conseguiu com que eles processassem 35GB de dados em 20 minutos, ao invés de dias como era anteriormente, graças ao poder computacional das GPU's (Nvidia, 2012).
- Hackers russos tem utilizado GPU's para realizar um ataque de força bruta em redes WPA e WPA2 com o objetivo de quebrá-las. A placa de vídeo utilizada aumentou a operação da descoberta da senha em até 10.000 por cento, fato que deixa os especialistas em segurança com um grave problema em suas mãos e evidencia o grande poder computacional das GPU's

Entre outras áreas que já adotaram a tecnologia de programação em GPU/CUDA podemos citar algumas em destaque: Dinâmica de Fluídos (Corrigan et al., 2009), Método Lattice-Boltzmann (Vasconcellos, 2009), Medicina através de tomografia médica (Noel et al., 2008), Problemas Térmicos (ANSONI, 2010), dentre vários outros. Este trabalho tem como influência os ótimos resultados obtidos com programação paralela massiva em GPU nestas diversas áreas.

1.5 Estrutura do Trabalho

No Capítulo 2 são apresentados os conceitos que envolvem a programação paralela em geral. São discutidos os assuntos relativos a *Graphics Processing Unit* (GPU), abordando a trajetória da aplicação da programação paralela na mesma, que resultaram no surgimento do conceito de *General Purpose Graphics Processing Unit* (GPGPU), e consequentemente, no surgimento da tecnologia *Compute Unified Device Architecture* (CUDA), cuja função é explorar este novo meio de otimizar a programação paralela de propósitos gerais em placas de vídeo. É feito também, um pequeno relato de como estas novas ferramentas podem ajudar na área de Métodos Numéricos sem Malha.

No Capítulo 3 apresentamos os dispositivos e ferramentas disponíveis, que foram utilizadas neste trabalho, para explorar a tecnologia CUDA. É mostrado também, o modelo de arquitetura da tecnologia, juntamente de um guia da utilização da linguagem CUDA/C, com os principais recursos disponíveis até o momento.

O Capítulo 4 apresenta uma implementação proposta para o desenvolvimento de um algoritmo para busca de vizinhos, muito utilizada em Métodos Numéricos sem Malha.

Por fim, no Capítulo 5 são apresentadas as conclusões e propostas para trabalhos futuros.

Capítulo 2

Referencial Teórico

Neste capítulo será apresentado os conceitos usados neste trabalho com a intenção de proporcionar um maior entendimento sobre o tema tratado.

2.1 Programação Paralela

A paralelização de instruções de computador é uma técnica que visa reduzir o tempo de computação de algoritmos. A capacidade de processamento paralelo é oferecida por um conjunto de processadores capazes de trabalhar cooperativamente para resolver um dado problema (Foster, 1995), aplicando-se a técnica conhecida como "dividir para conquistar". Essa busca pelo alto desempenho dos computadores ocasionou o surgimento de vários modelos de arquiteturas paralelas. Essas arquiteturas possuem uma característica fundamental para permitir o processamento paralelo que é a presença de vários núcleos em seu processador, evoluindo para concepção de multiprocessadores e culminando com o uso de *clusters e grids* (Schepke, 2009).

Outro ponto fundamental é a capacidade dos *softwares* em utilizar estes recursos oferecidos. E para que um sistema inteiro possa suportar a computação paralela, é necessário que os softwares também tenham capacidade de interpretar o paralelismo. Isso inclui linguagens de programação paralelas, compiladores, bibliotecas, sistemas de comunicação e sistemas de entrada e saída paralelo. Os modelos de arquitetura paralelas são classificados pelo fluxo de instruções e dados que se apresentam. Essa classificação é definida como taxonomia de Flynn (Flynn, 2004), e é divida em quatro categorias: SISD, SIMD, MISD e MIMD.

2.2 GPU

As GPU's (*Graphics Processing Unit*) ou placas de vídeo, são componentes que enviam sinais de uma computador para um monitor. Inicialmente, elas foram projetadas para fazer cálculos de ponto flutuante, mas com o passar dos anos, percebeu-se que elas tinham características que facilitavam o processamento de imagens, devido às estas serem compostas de matrizes de números, onde cada posição i x j da matriz constitui um pixel. Para os usuários comuns de hoje, a GPU é responsável pela qualidade gráfica do dispositivo, seja ele um computador, um videogame ou até mesmo um celular (Viana, 2011). Com estes objetivos iniciais, as GPU's tornaram-se ferramentas poderosas e eficientes em manipular gráficos computacionais onde, a partir de objetos vindos da CPU (*Central Processing Unit*), as transformam e geram imagens as quais o usuário vê na tela.

O grande motivador da propagação das GPU's foi o crescimento da popularidade dos sistemas operacionais gráficos conduzidos pela Microsoft Windows, onde os usuários começaram a adquirir aceleradores de exibição 2D para os computadores de mesa. Fato que levou a popularização dos aceleradores gráficos a uma variedade de mercados na década de 1980, incluindo governo, aplicações e técnicas de visualização científica, bem como ferramentas para criar impressionantes efeitos cinematográficos (Sanders and Kandrot, 2010).

Já entre os desenvolvedores, o grande marco para a evolução do desenvolvimento das GPU's foi a partir de 1992, quando a empresa Silicon Graphics abriu a interface de programação de seu *hardware*, liberando a biblioteca OpenGL. Esta por sua vez se destinava a ser uma linguagem padrão para programar aplicações gráficas em 3D e que seria independente de plataforma e métodos. A partir desse momento estava lançada a contagem de tempo para que as tecnologias relacionadas às GPU's atingissem o caminho para aplicações de consumo. O principal produto gerado pelas tecnologias em GPU's com certeza são os jogos em 3D. A demanda por jogos alavancou os desenvolvimentos gráficos.

2.3 GPGPU

Em meio ao desenvolvimento massivo das aplicações gráficas, a empresa NVIDIA lançou o primeiro chip da indústria de computação (GeForce Série 3) para implementar o então novo Microsoft Directx 8.0, possibilitando pela primeira vez que os desenvolvedores tivessem algum controle sobre os cálculos exatos que seriam realizados em suas GPU's. Começava então as primeira tentativas de GPGPU, que é o nome dado para a programação de propósitos gerais

em uma GPU. As primeiras abordagens nos primeiros dias de computação em GPU foram extremamente complicadas, porque APIs gráficas padrões como OpenGL e DirectX, ainda eram as únicas maneiras de se interagir com uma GPU, e quaisquer tentativas de realizar cálculos arbitrários em GPU, estariam sujeitas às restrições de programação dentro de uma API gráfica (Sanders and Kandrot, 2010).

A primeira abordagem que os programadores criaram para tentar sanar as dificuldades foi a de 'enganar' a GPU, transmitindo dados de cálculos gerais para a mesma, encapsulados em pixels de cores como se fossem renderizações padrões. Tal técnica era inteligente, mas muito complicada por causa do rendimento aritmético elevado das GPU's e a dificuldade de prever como dados de pontos flutuantes seriam tratados, impondo fortes limitações de recursos. Somado a tudo isso, ainda era obrigatório que os programadores soubessem programar em OpenGL ou DirectX, pois estas eram as únicas formas de se interagir com a GPU. Passados cinco anos após o lançamento da série Geforce 3 e intensos investimentos em pesquisa, a NVIDIA apresentou a nova série GeForce 8800, a primeira GPU construída com arquitetura CUDA. Esta arquitetura trouxe para o mercado novos componentes projetados estritamente para a computação em GPU e simplificou parte das complicações que impediam os processadores gráficos anteriores de serem realmente úteis para a computação de propósito geral.

2.4 CUDA

A principal mudança das novas placas gráficas da NVIDIA a partir da série 8800 em relação as anteriores, estava em como tratar os recursos de computação que não eram de finalidade gráfica. Enquanto as antigas GPU's transformavam os dados em *"pixels shaders"*, a tecnologia CUDA possui um *pipeline* de sombreamento unificado, fazendo com que toda a unidade lógica aritmética (ALU) do *chip* pudesse ser empacotada por um programa que fizesse os cálculos de propósitos gerais (Sanders and Kandrot, 2010). Outra mudança importante foi a inclusão de capacidades que cumprem os requisitos de precisão aritmética de pontos flutuantes nestas ALU's, adaptando áreas para computação em geral separadas da computação gráfica. Além disso, a GPU agora pode ler e escrever na memória através do recurso de memória compartilhada. Através destas novas características, as GPU's passaram a ser um disposto extremamente eficiente em computação gráfica e execução de tarefas básicas e tradicionais.

Apesar de todos os esforços da NVIDIA de simplificar a arquitetura e fornecer um produto autônomo que serviria para computação em geral, ainda era preciso saber usar OpenGL e DirectX para acessar os recursos do CUDA, o que restringia o número de pesquisadores na

área. Levando este fato em conta, a NVIDIA trouxe a tecnologia para a indústria que utilizava a linguagem C como padrão, adicionando um número baixo de palavras-chave do CUDA, com o objetivo de aproveitar as principais características da nova tecnologia.

O interesse de desenvolvedores tornou-se grande, levando a empresa a lançar, juntamente da sua nova série de GPU's (GeForce 8800 GTX), um compilador que mescla as duas tecnologias, criando a primeira linguagem especificamente desenhada para programação de propósitos gerais em uma GPU, a CUDA C. Com esta nova linguagem, os desenvolvedores não são obrigados mais a conhecer a programação em OpenGL e DirectX, e nem saber sobre tarefas de computação gráfica. Bastaria agora aos desenvolvedores, obter uma placa de vídeo que suporte a tecnologia CUDA, juntamente do driver e ferramentas que auxiliam no seu desenvolvimento, tudo isso disponibilizados pela NVIDIA. É desejável também que o usuário tenha o mínimo de conhecimento em linguagem C/C++, que até hoje são as linguagens mais utilizadas no mundo.

Desde que foi lançada, a tecnologia CUDA tem sido amplamente utilizada em várias áreas e por várias empresas que relatam ótimos ganhos em desempenho computacional, economia financeira e até economia energética, quando comparados ao uso de CPU's. No Brasil, temos um dos poucos cêntros de excelência em CUDA no mundo, como pode ser visto em (InformationWeek, 2012).

2.5 Métodos Numéricos sem Malha

Problemas computacionais de Engenharia, geralmente estão envolvidos com simulações de fenômenos físicos naturais. Métodos numéricos são utilizados para realizar tais simulações, sendo que as aplicações de engenharia possuem características que envolvem um alto grau de precisão, alinhada ao tempo de execução das tarefas. Para a resolução numérica de equações diferenciais (PDEs), associados à Mecânica Computacional, consideram-se basicamente três grupos clássicos de métodos numéricos: Métodos das diferenças finitas, Métodos de Volumes finitos e Métodos dos elementos finitos (GUEDES, 2006).

Métodos sem Malha, assim como o clássico método dos elementos finitos, são métodos de aproximação numérica para resolução numérica de equações diferenciais. Para aplicações gerais, nenhuma dessas técnicas é tão eficiente quanto os métodos tradicionais baseados em malhas, como o método dos elementos finitos, enquanto para engenharia, percebe-se uma crescente tendência de aplicações sobre métodos sem malha. Sua principal diferença para com

os métodos baseados em malha está caracterizado no uso de um conjunto de nós espalhados pelo domínio do problema, ao invés de uma malha ou grade (Fonseca, 2011). Outra dificuldade está baseada na imposição das condições de contorno de Dirichlet quando as funções de forma não apresentam a propriedade do delta de Kronecker. Estes métodos vêm ganhando destaque e um número grande de interessados em seu desenvolvimento, principalmente desde um trabalho pioneiro da área apresentado em (Belytschko et al., 1996).

Capítulo 3

CUDA

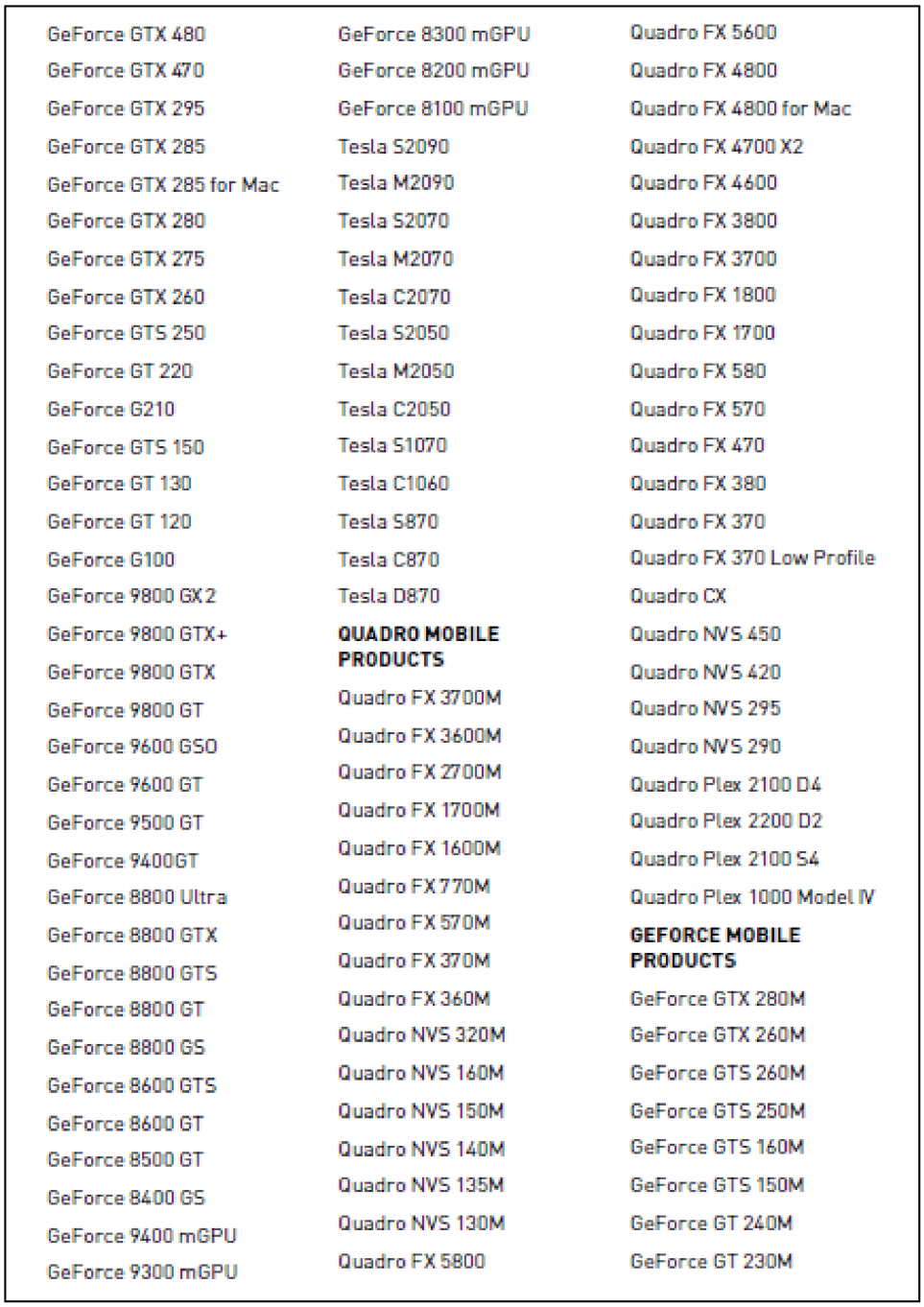
Neste capítulo, serão apresentados os componentes de *hardware* e *software* que serão necessários para o início da programação em CUDA C, assim como o ambiente em que este trabalho foi produzido junto de suas respectivas configurações. Os pré requisitos são:

- Uma Placa de Vídeo com suporte a CUDA
- Um *Driver* NVIDIA para o dispositivo
- Um *kit* de Ferramentas de desenvolvimento CUDA
- Um compilador C padrão

3.1 Dispositivos

Nem todas as GPU's irão funcionar com a programação em CUDA-C. A primeira restrição, é a do fabricante das GPU's, que até o momento da conclusão deste trabalho, só poderiam ser GPU's da marca NVIDIA, criadora da arquitetura CUDA e posteriormente da API juntamente da linguagem CUDA-C. Apesar de serem fáceis de encontrar, nem todas as GPU's NVIDIA são construídas sobre a arquitetura CUDA. Das que foram lançadas a partir de 2006, depois do lançamento da Geforce 8800 GTX, a maioria tem sido compatível com CUDA. A Figura 3.1 apresenta a maioria delas:

Para este trabalho, foi utilizada um GPU NVIDIA, Geforce GTX 460, 1GB de memória e 336 núcleos cuda. Para ter uma lista completa das GPU's, e os seus respectivos drivers estão disponíveis em (Nvidia, 2012).



GeForce GTX 480	GeForce 8300 mGPU	Quadro FX 5600
GeForce GTX 470	GeForce 8200 mGPU	Quadro FX 4800
GeForce GTX 295	GeForce 8100 mGPU	Quadro FX 4800 for Mac
GeForce GTX 285	Tesla S2090	Quadro FX 4700 X2
GeForce GTX 285 for Mac	Tesla M2090	Quadro FX 4600
GeForce GTX 280	Tesla S2070	Quadro FX 3800
GeForce GTX 275	Tesla M2070	Quadro FX 3700
GeForce GTX 260	Tesla C2070	Quadro FX 1800
GeForce GTS 250	Tesla S2050	Quadro FX 1700
GeForce GT 220	Tesla M2050	Quadro FX 580
GeForce G210	Tesla C2050	Quadro FX 570
GeForce GTS 150	Tesla S1070	Quadro FX 470
GeForce GT 130	Tesla C1060	Quadro FX 380
GeForce GT 120	Tesla S870	Quadro FX 370
GeForce G100	Tesla C870	Quadro FX 370 Low Profile
GeForce 9800 GX2	Tesla D870	Quadro CX
GeForce 9800 GTX+	**QUADRO MOBILE PRODUCTS**	Quadro NVS 450
GeForce 9800 GTX	Quadro FX 3700M	Quadro NVS 420
GeForce 9800 GT	Quadro FX 3600M	Quadro NVS 295
GeForce 9600 GSO	Quadro FX 2700M	Quadro NVS 290
GeForce 9600 GT	Quadro FX 1700M	Quadro Plex 2100 D4
GeForce 9500 GT	Quadro FX 1600M	Quadro Plex 2200 D2
GeForce 9400GT	Quadro FX 770M	Quadro Plex 2100 S4
GeForce 8800 Ultra	Quadro FX 570M	Quadro Plex 1000 Model IV
GeForce 8800 GTX	Quadro FX 370M	**GEFORCE MOBILE PRODUCTS**
GeForce 8800 GTS	Quadro FX 360M	GeForce GTX 280M
GeForce 8800 GT	Quadro NVS 320M	GeForce GTX 260M
GeForce 8800 GS	Quadro NVS 160M	GeForce GTS 260M
GeForce 8600 GTS	Quadro NVS 150M	GeForce GTS 250M
GeForce 8600 GT	Quadro NVS 140M	GeForce GTS 160M
GeForce 8500 GT	Quadro NVS 135M	GeForce GTS 150M
GeForce 8400 GS	Quadro NVS 130M	GeForce GT 240M
GeForce 9400 mGPU	Quadro FX 5800	GeForce GT 230M
GeForce 9300 mGPU		

Figura 3.1: Lista de algumas GPU's que foram construídas na plataforma CUDA

3.2 Ferramentas de Desenvolvimento

Com a GPU instalada com seu respectivo *driver*, o computador ja está pronto para executar código CUDA compilado, mas como a intensão aqui é o aprendizado da linguagem e o desenvolvimento de novos códigos, será necessário a instalação de um *kit* de ferramentas para

auxiliar na tarefa. É necessário o uso de dois compiladores: um para compilar o código da CPU e outro para compilar o código da GPU. Assim como o *driver*, a NVIDIA disponibiliza o compilador para GPU em seu site juntamente de um pacote de softwares que auxiliam no desenvolvimento, que é chamado de CUDA *Toolkit*. A versão utilizada neste trabalho foi a CUDA Toolkit 4.0 que possui os seguintes componentes:

- Compilador CUDA C/C++
- *Software* auxiliar Visual Profiler
- Bibliotecas de suporte criadas especificamente para GPU: BLAS(Basic Linear Algebra Subroutines), FFT(Fast Fourier Transform), Sparse Matrix, RNG
- Documentação e ferramentas adicionais

Existem versões dos *softwares* para as plataformas Windows, Linux e Mac. Neste trabalho foi utilizado um computador DELL Studio, com processador Intel Core 2 Quad 2.34 GHz, 3 GB de memória RAM, sistema operacional Microsoft Windows 7, versão 64 bits. Recomenda-se também, a instalação do software NVIDIA Nsight Visual Studio Edition, que consiste em um depurador de código CUDA integrado ao Visual Estudio da Micrososft.

3.2.1 Configuração do Ambiente de Desenvolvimento

Como dito anteriormente, os *softwares* que auxiliam no desenvolvimento de código CUDA-C estão disponíveis nas Plataformas Windows, Linux e Mac. Aqui será apresentado a configuração do Microsoft Visual Studio 2010 para Windows 7 64 bits, para que este possa suportar e compilar códigos CUDA-C. Para seguir os seguintes passos, julga-se o entendimento mínimo no uso do referido *software*.

O primeiro passo, é criar um novo projeto no Visual Studio C++, do tipo "*Win32 Console Application*", selecionando a opção "*empty project*" nas configurações. Após a criação do projeto, criar um único arquivo do tipo "cpp" de nome qualquer, mas adicionando a extensão ".cu", como na Figura 3.2.

Com o projeto criado, vamos atribuir o projeto para rodar tanto em plataformas 32bits e 64 bits, clicando com o botão direito em "*Solution*", selecionar "*Configuration Manager*".

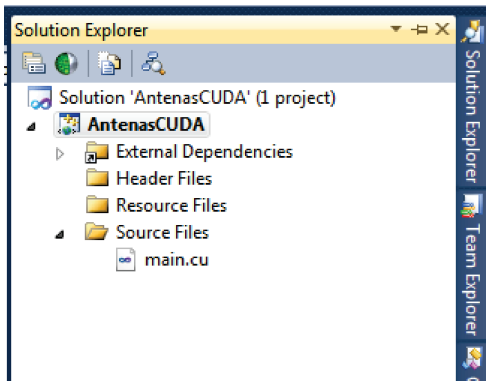


Figura 3.2: Estado de um projeto CUDA inicial

Dentro da nova janela, na caixa de nome "*Plataform*", selecionar a opção "*New*" e irá surgir a Figura 3.3. Na caixa de diálogo selecione x64 em "*New Plataform*" e Win32 em "*Copy settings from*".

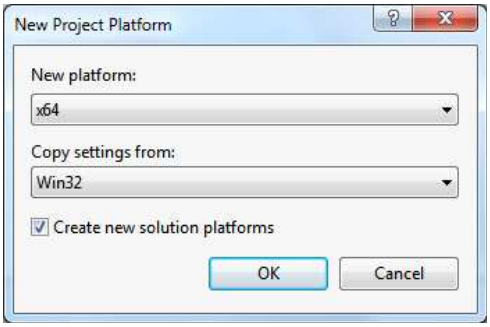


Figura 3.3: Local onde deve-se atribuir que o projeto rode em plataformas 32 e 64 bits

O próximo passo, é configurar o compilador CUDA para os arquivos de extensão ".cu". Clicando com o botão direito no nome do Projeto e selecionando "*Build Customizations*". Dentro da caixa de diálogo que aparece, selecionar "*CUDA 4.0 targets*" como na Figura Figura 3.4.

Em seguida, clicar com o botão direito no arquivo de extensão ".cu" e selecionar "*Properties*". Na aba superior "*Configuration*" marcar "*All configuration*" e na aba "*Plataform*" selecionar "*All Plataforms*". Dentro da caixa de opções, no item "*Type Field*" selecionar "CUDA C/C++" como na Figura 3.5.

Após aplicar a última instrução, deve-se clicar com o botão direito no projeto e acessar as "*Properties*". Ná árvore de itens a esquerda acessar "*Configuration Properties* -> CUDA C/C++". Após, na aba "*Configuration*" selecionar "*All Configuration*" e na aba "*Plata-*

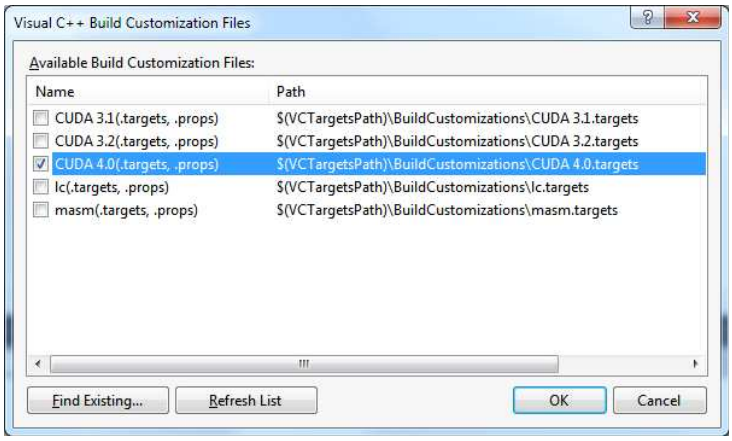


Figura 3.4: Local de configuração do compilador CUDA no projeto

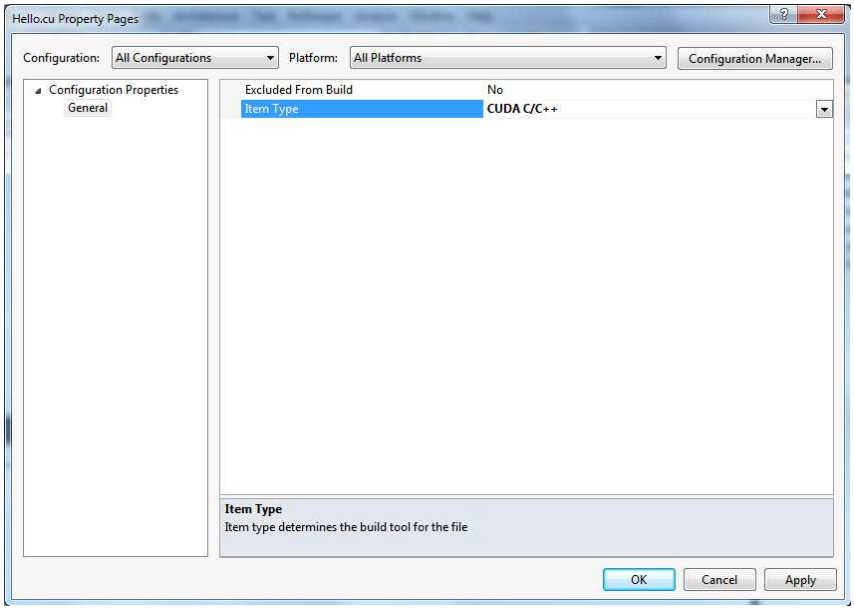


Figura 3.5: Definindo o tipo do arquivo .cu

form" marcar "x64". Dentro da caixa de opções a direita, selecionar "*x64-bit(-machine64)*" na opção "*Machine Plataform*", como exemplificado na Figura 3.6. Após aplicar estas modificações, mantenha a mesma janela aberta!

Com a mesma janela de opções aberta, é necessário configurar as opções de *linker* das bibliotecas CUDA. Na aba superior "*Plataform*" marcar "*All plataform*" e em seguida, selecionar o seguinte caminho na árvore de opções na esquerda: *Configuration Properties* -> *Linker* -> *Input*. Dentro do campo "*Additional Dependencies field*" na caixa de texto a direita, adicionar "cudart.lib" como na Figura 3.7

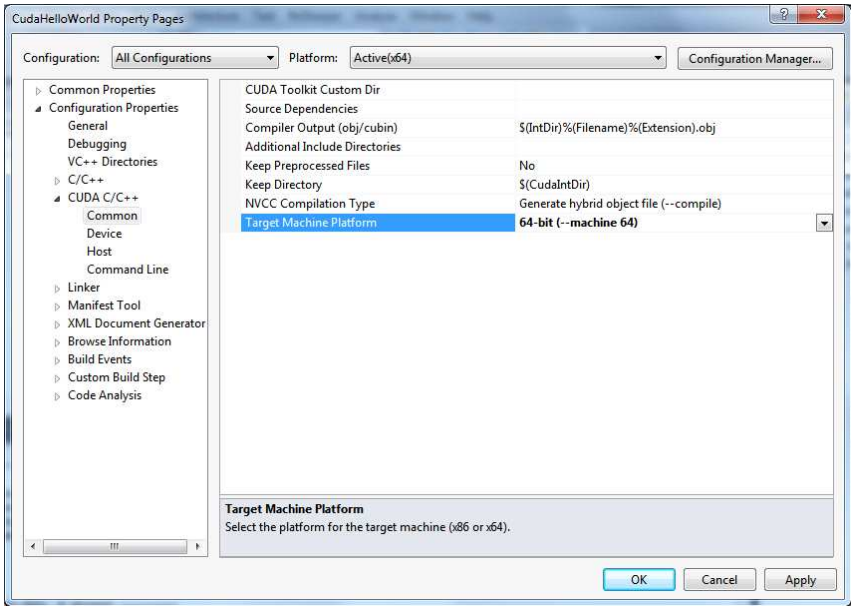


Figura 3.6: Configurando o compilador NVCC CUDA para plataformas x64

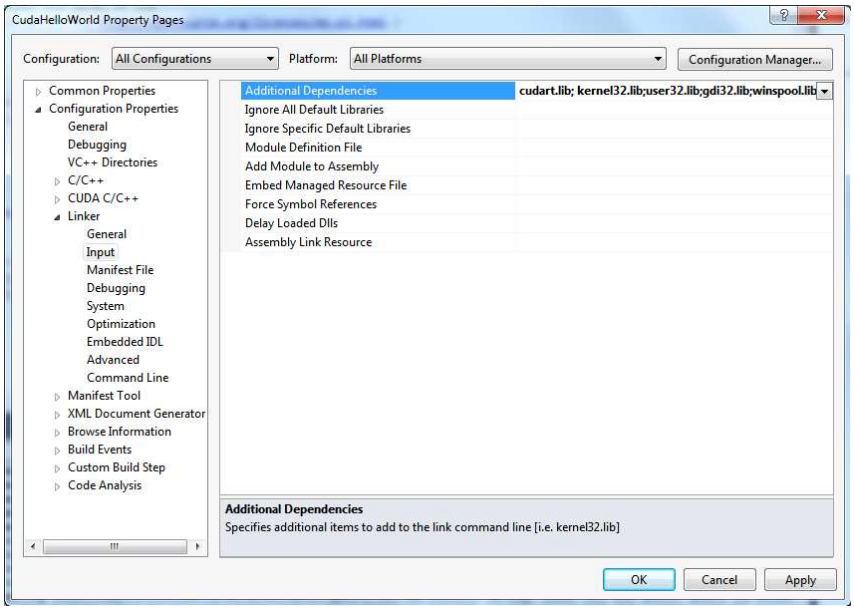


Figura 3.7: Adicionando o linker das bibliotecas CUDA

Com todos estes passos, o Visual Studio 2010 já está configurado para compilar e executar os códigos CUDA. Para finalizar, vamos configurar o editor para que ele reconheça os arquivos ".cu" como arquivos do *software* Microsoft Visual C/C++ 2010. Na barra de menus na tela principal do Visual, selecionar "*Tools*" e em seguida "*Options*". Dentro da janela que

irá surgir, seguir o seguinte caminho na árvore de opções a esquerda: "*Text Editor -> File Extension*". Adicionar a extensão ".cu" ao Microsoft Visual C++ como na Figura 3.8

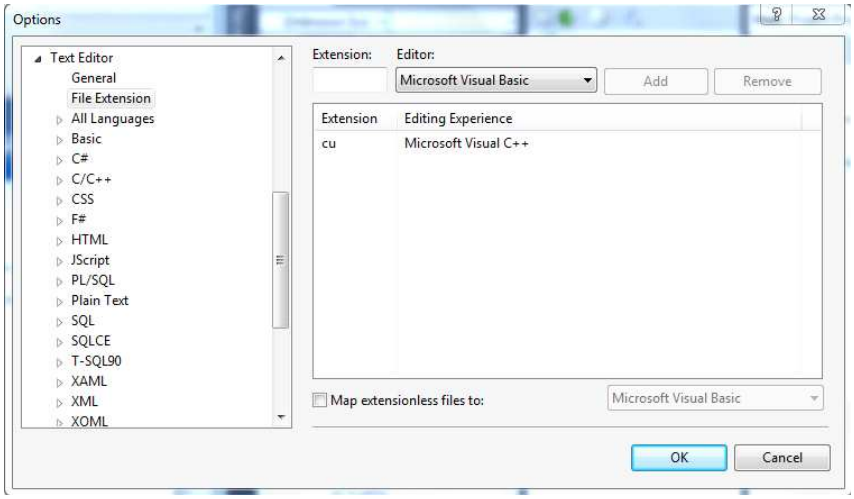


Figura 3.8: Adicionando os arquivos .cu ao editor do Microsoft Visual C/C++

Vale ressaltar que, os passos onde envolvem a adição de opções relacionadas a plataforma 64bits, não precisam ser executadas caso o trabalho seja realizado em máquinas de 32 bits. Mas como as máquinas de 64 bits já estão bem difundidas é bom avaliar se vale a pena já deixar a configuração para ambas.

3.3 Modelo de Arquitetura e Fluxo de um Programa

A evolução das arquiteturas das GPU's, permitiu a otimização na execução de uma grande quantidade de *threads*. Apesar de possuírem um memória global lenta, as GPU's conseguem sobrepor a latência da memória global com um recurso de chaveamento entre blocos e *threads*, que resulta em um alto número de instruções executadas por unidade de tempo. Como meio de comparação, poderíamos dizer que, enquanto a CPU utiliza grande parte do seu chip para *cache* e controle de fluxo, a GPU usa para mais unidades de processamento (núcleo) visando melhorar a vazão (instruções por segundo). A Figura 3.9 mostra o modelo de arquitetura de uma GPU CUDA em relação a uma CPU.

A comunicação entre a GPU e a CPU em aplicações gerais, ou seja, que não sejam para propósitos gráficos, se dá através do fluxo de 5 passos básicos:

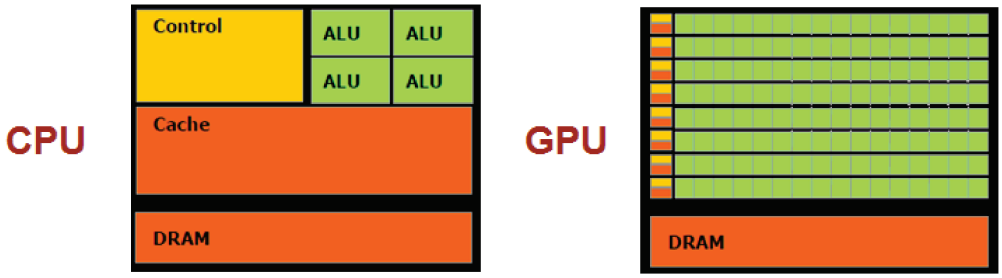


Figura 3.9: Comparação do desenho de uma arquitetura de GPU frente a de uma CPU

- 1. A CPU envia e escreve dados na memória global da GPU.
- 2. A CPU invoca a função que irá executar na GPU
- 3. A GPU divide o fluxo de dados e executa em paralelo dentro dos seus núcleos.
- 4. A GPU devolve os dados para sua memória global.
- 5. Após o término da execução da função, a CPU copia os dados para sua memória global.

Somando-se as funções gráficas que uma GPU oferece a uma CPU, com essa nova função de troca de informações com a CPU para programação de aplicação de propósitos gerais, temos um modelo identificando a API em comunicação com o sistema na Figura 3.10. Temos ilustrado a pilha de *softwares* da plataforma CUDA, composta pelo *driver* de acesso ao *hardware*, que faz toda a comunicação do *Host* (CPU) com o *Device* (GPU), seja para trabalhos gráficos ou de propósitos gerais. Em seguida, tem-se um componente *runtime* e as bibliotecas matemáticas como a CUBLAS e a CUFFT, e por fim, no topo da pilha encontra-se a API da plataforma CUDA.

3.4 Hierarquia de Blocos e Threads

Com a arquitetura CUDA, a NVIDIA introduziu um novo conceito na computação paralela, o SIMT: *single-instruction multiplethread*. Tal arquitetura cria e gerencia automaticamente *threads* paralelas que serão alocadas em um ou mais blocos paralelos, que por sua vez, estarão inseridos em um *Grid* (Grade). Um bloco é a unidade básica de organização das *threads* e de mapeamento para o *hardware*. Um bloco de *threads* é alocado a um multiprocessador da GPU. Pode-se pensar nestes blocos, como sendo cópias de um dado *kernel* (função que

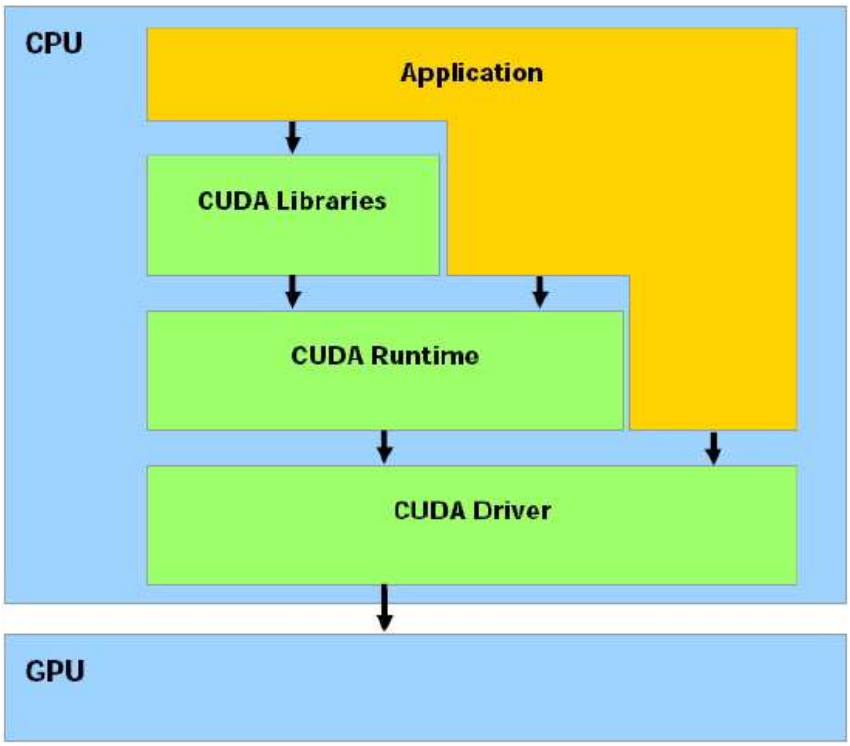


Figura 3.10: Conjunto de Comunicação da API CUDA com Host

será executada pela GPU), que irão executar cada um, o mesmo pedaço de código com um determinado número de *threads* em cada bloco, sendo possível a criação de blocos de até três dimensões. Já o *grid*, é a unidade básica onde estão distribuídos os blocos e a estrutura completa de distribuição das *threads* que executam um *kernel*, e podem ter até duas dimensões. É nele que está definido o número total de blocos e de *threads* que serão criados e gerenciados pela GPU. A Figura 3.11 mostra um exemplo de um *grid* com seus blocos e *threads*, enquanto no Apêndice A é apresentada uma explicação mais detalhada.

Tal modelo de hierarquia, exige que tenhamos uma forma de identificar os índices dos blocos e suas respectivas *threads*, para que possamos tratá-las de forma individual dentro de cada *kernel*. A maneira certa de se fazer isso, é através de algumas variáveis *built-in* (variáveis definidas em tempo de execução CUDA) que serão vistas na Subseção 3.5.4. Um grupo de *threads* dentro de um mesmo bloco podem sincronizar sua execução e se comunicarem através da memória compartilhada (Seção 3.7), essas duas características geram grande ganho na computação paralela.

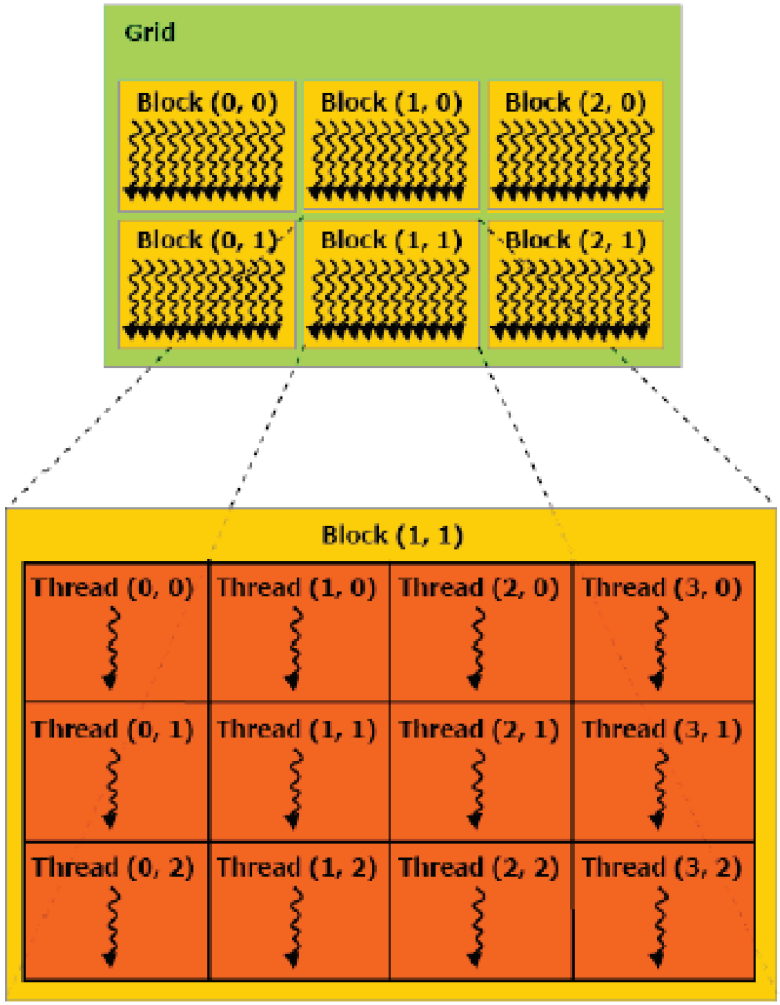


Figura 3.11: Modelo da organização de um *grid* com seus blocos e *threads*

Outra característica importante em entender na arquitetura, é o conceito de *warp*. Um *warp* faz referência a um conjunto de 32 *threads* que estão "entrelaçadas" e são executadas em conjunto, em cada linha de um programa, onde cada *thread* em um warp executa as mesmas instruções em dados diferentes (Sanders and Kandrot, 2010). Este conceito é importante para entender os benefícios e desvantagens da **Memória Constante** e será visto na Seção 3.7.

3.5 Extensões CUDA para C/C++

Junto da arquitetura e do compilador, a NVIDIA trouxe algumas particularidades do CUDA para a linguagem C/C++. A começar pelas nomenclaturas "*Host*" e "*Device*", que significam

CPU e GPU respectivamente. Por diversas vezes teremos expressões como: "copiar da memória do *host* para a memória do *device*" nas literaturas envolvendo CUDA.

Outra palavra muito utilizada será a palavra "*Kernel*". A modularização está presente nos códigos acelerados utilizando CUDA, o que leva os códigos a serem implementados em forma de funções. Dentro do contexto CUDA, uma função executada na GPU (device) que é invocada pela CPU (host) é denominada **Kernel**.

Na parte da codificação em si, a API da plataforma CUDA introduz quatro novas "extensões" a linguagem C/C++: Qualificadores de tipo de função; Qualificadores de tipo de Variável; Nova sintaxe de chamada de Função; Variáveis internas para acessar índices.

3.5.1 Qualificadores de tipo de Função

Os qualificadores de funções foram adicionados a plataforma CUDA para indicar o local de invocação de um *Kernel*, seja no *host* ou *device*. São três qualificadores, que serão colocados sempre na declaração do *kernel*: `__global__, __device__, __host__` .

- `__global__` : Usado para indicar que um *kernel* vai ser executado no *device*, mas deve ser chamado pelo *host*. Funções deste tipo não suportam recursão, não podem possuir variáveis estáticas dentro de seu corpo, seu número de argumentos deve ser fixo, deve ter retorno do tipo *void* e possuir uma configuração de execução que será visto na Subseção 3.5.3.

- `__device__` : Usado para indicar que um *kernel* vai ser executado no *device* e chamado pelo *device*. Além de não suportar recursão, variáveis estáticas em seu corpo e número fixo de argumentos, este tipo de função não pode ter seu endereço recuperado. Não é necessário uma configuração de execução como no tipo `__global__`.

- `__host__` : Usado para indicar que uma função será executada e chamada a partir do *host*. Geralmente não é utilizado sozinho, pois a ausência de um qualificador representa o tipo `__host__` por padrão.

- `__device__ __host__` : Podem ser usados combinados, para os casos onde o *kernel* deve ser compilado tanto pelo *device* quanto pelo *host*.

3.5.2 Qualificadores de tipo de Variável

Assim como acontece com as funções *(*kernels), algumas variáveis precisam que seja especificado o local onde será armazenada, na GPU ou na CPU. São três qualificadores de variáveis que aparecem na sua declaração: `__device__`, `__constant__` e `__shared__`.

- `__device__` : Indica que a variável irá ser alocada na memória global da GPU. Poderá ser acessada por todas as *threads* de um *grid* e também pela CPU através da biblioteca *runtime* do CUDA, tendo seu tempo de vida igual ao da aplicação.
- `__constant__` : Possui todas as características do qualificador `__device__`, exceto pela sua localização, que será na memória constante da GPU, que será explorada na Seção 3.7. Implica na declaração de uma variável estática. Não podem ser atribuídas a partir da GPU, sendo permitida sua atribuição pela CPU através da biblioteca *runtime*.
- `__shared__` : Este tipo de variável possui algumas características diferenciais. Elas residem na memória compartilhada da GPU (Seção 3.7), estando acessíveis apenas para as *threads* de um mesmo bloco e possui tempo de vida igual ao do bloco. Implica na declaração de uma variável estática. Este tipo de variável não pode ser inicializada em sua declaração.

Algumas considerações devem ser feitas para estes últimos três tipos de qualificadores de variáveis. Não podem ser usados em declarações de *structs, union*, parâmentros formais e variáveis locais dentro de uma função que é executada pelo *host*. As variáveis destes tipos de qualificadores não podem ser definidas como a palavra **extern**. Tendo como exceção as variáveis do tipo `__device__` alocadas dinâmicamente.

3.5.3 Sintaxe de um Kernel

Utilizando os qualificadores de tipo de função (Subseção 3.5.1) é possível escrever a assinatura e o corpo de um *Kernel* que será executado na GPU. A forma de se escrever um *kernel* é praticamente igual a uma função normal escrita em C, tendo como diferencial os qualificadores. Abaixo a estrutura e um exemplo de um kernel:

qualificador tipoRetorno nomeKernel(paramentros);

```
__global__ void kernel(int a, int b);
__device__ int kernel_2(double a, int b);
```

Figura 3.12: Exemplo de assinatura de kernels

O *kernel* recebe o qualificador `__global__`, o que o transforma em uma função que será chamada pelo *host* e executada no *device*. Já `kernel_2` recebe o qualificador `__device__` e será chamado e executado pelo *device*. Todos os detalhes, restrições e características dos qualificadores foram explicados na Subseção 3.5.1.

Para que um *kernel* possa ser invocada a partir do *host* e seja executado no *device*, é preciso especificar alguns parâmetros de configuração na hora de sua chamada. Estes novos parâmetro, devem aparecer nas chamadas de funções do tipo `__global__`, e entre os símbolos "<<<>>>", da seguinte forma:

```
<<< Dg, Db, NS, S >>>
```

Destes 4 parâmetros, os dois primeiros são obrigatórios:

Dg especifica uma variável do tipo **dim3** (Subseção 3.5.4) que informa ao *kernel* a dimensão e o tamanho do grid, ou seja, o número de blocos usados para tratar o *kernel*;

Db especifica uma variável do tipo **dim3** que informa ao *kernel* a dimensão e o tamanho de cada bloco, ou seja, o número de *threads* que cada bloco irá conter para tratar o *kernel*;

Os outros dois parâmetros (NS, S), são opcionais:

NS especifica uma variável do tipo "`size_t`" que informa o tamanho, em *bytes*, do espaço que será reservado dinamicamente na memória compartilhada por cada bloco (Seção 3.7);

S especifica uma variável do tipo "`cudaStream_t`" relativa a um *stream* adicional (Seção 3.10). A Tabela 3.1 traz um resumo dos parâmetros.

Parâmetro	Tipo	Obrigatório	Função
Dg	dim3	Sim	Dimensão e tamanho do grid
Db	dim3	Sim	Dimensão e tamanho de cada bloco
Ns	`size_t`	Não	Número de bytes da memória compartilhada em cada bloco
S	dim3	Não	Especifica uma *stream* adicional

Tabela 3.1: Parâmentros de Configuração de Execução. Fonte (Lopes and Azevedo, 2008)

O tipo dim3 corresponde a um vetor de inteiros especializado para a definição de dimensões. Ao invocar um kernel onde os parâmetros de configuração são do tipo *"int"*

```
kernel<<<1,1>>>(a, b);
```

estamos automaticamente informando ao *device* que os blocos e *threads* serão configurados apenas com uma dimensão. Para trabalharmos com mais de uma dimensão, deve-se criar uma variável do tipo **dim3**, informando a dimensões dos blocos e *threads* na hora de sua criação. Por exemplo, caso queira criar um grid de 16 blocos bidimensionais, sendo cada bloco composto de 20 *threads* bidimensionais, devem-se seguir os passos abaixo:

```
dim3 blocos(16, 16);
dim3 threads(20, 20);
  kernel<<<blocos, threads>>>(a,b);
```

Vale ressaltar que, cada GPU tem um número limite de blocos (65.535 durante o desenvolvimento deste trabalho) e threads (512 por bloco na placa utilizada neste trabalho) que podem ser lançados em um *kernel*, forçando o desenvolvedor a conhecer as especificações de sua GPU, para que não viole qualquer tipo de restrição da mesma. O tipo dim3 está disponível na biblioteca de *runtime* do CUDA. Para nenhuma das variáveis *Built-in* é possível realizar atribuição de valores (Rocha and Filho, 2010).

3.5.4 Variáveis Internas

A API do CUDA nos fornece uma série de variáveis *Built-in*, que são variáveis definidas em tempo de execução e servem para representar o *grid*, a dimensão dos blocos e os índices das

threads, sendo permitidas apenas dentro de um *kernel*. Abaixo temos as principais variáveis de índices e na tabela 3.2 um quadro com um resumo de suas características.

- threadIdx.x, threadIdx.y e threadIdx.z armazem o indice para as três dimensões que uma thread pode ter
- blockIdx.x e blockIdx.y armazem para as duas dimensões que um bloco pode ter
- blockDim que armazena a dimensão do bloco, ou seja, número de threads
- gridDim que armazena a dimensão do grid, ou seja, número de blocos

Variável	Tipo	Especificação
gridDim	dim3	Dimensão do *grid*
blockIdx	unit3	Índice do bloco no *grid*
blockDim	dim3	Dimensão do bloco
threadIdx	unit3	Índice da *thread* no bloco
warpSize	int	Contêm o tamanho do *warp* em *threads*

Tabela 3.2: Variáveis *Built-in*. Fonte (Lopes and Azevedo, 2008)

3.6 Modelo de Programação CUDA C

Após conhecer as extensões e novidades que a tecnologia CUDA adiciona à linguagem C, já é possível escrever código para computação de propósitos gerais em CUDA C. Assim como as novidades, a programação é bem intuitiva e fácil de aprender. Quem já tem familiarização com a linguagem C, irá encontrar semelhanças que ajudarão muito na hora do desenvolvimento.

O código para executar tarefas na GPU, é constituído de basicamente de um fluxo de passos padrão: criar ponteiros na GPU, alocar espaço na GPU para estes ponteiros, copiar os dados que serão processados para a GPU, executar o(s) *kernel(s)*, copiar os dados processados de volta para a CPU e liberar o espaço reservado na GPU. Todas as aplicações irão seguir este padrão, o número de vezes que isto acontece e algumas tarefas adicionais oferecidas pelo CUDA, vão depender da aplicação e do desenvolvedor. Para exemplificar estes passos, será considerada uma aplicação de soma de dois vetores de números inteiros.

Para a criação de ponteiros para o *device*, o processo é o idêntico ao de criar ponteiros para o *host*. Mas costuma-se seguir uma convenção que ajuda a identificar se o ponteiro está sendo utilizado na GPU ou na CPU. Ao criar um vetor "a" para o *host*, o ponteiro do *device* que irá receber estes mesmos dados, será chamado de ``a_d''. Trata-se apenas de uma convenção de boas práticas, qualquer um pode criar seu próprio padrão. A Figura 3.13 apresenta a criação dos ponteiros para soma de vetores. Como dito, a criação dos ponteiros é feita de maneira comum, apenas com uma identificação no nome para facilitar na hora de lembrar que este ponteiro será utilizado na GPU.

```
int a[N], b[N], c[N];
int* a_d, *b_d, *c_d;
```

Figura 3.13: 1º Passo: Criação de Ponteiros

Após a criação dos ponteiros, deve-se preencher os vetores do *host* (a, b, c) da forma que for conveniente. Neste exemplo, a alocação do espaço de memória no *host* foi feita já na criação do vetor. Para alocarmos o mesmo tamanho destes vetores na GPU, iremos utilizar uma função da API CUDA chamada "cudaMalloc()". Quem já está familiarizado com a função "malloc" da biblioteca C padrão, notará que é muito semelhante e simples a alocação na GPU. A função cudaMalloc() necessita de 2 parâmetros, o ponteiro que fará referência ao espaço alocada na GPU e o tamanho do espaço que será alocado. A Figura 3.14 apresenta a alocação dos nossos três ponteiros criados para a GPU `(a_d, b_d, c_d)`.

```
cudaMalloc((void**)&a_d, N * sizeof(int) );
cudaMalloc((void**)&b_d, N * sizeof(int) );
cudaMalloc((void**)&c_d, N * sizeof(int) );
```

Figura 3.14: 2º Passo: Alocação dos ponteiros da GPU

Com o espaço de memória devidamente reservado na GPU, precisamos enviar os dados que estão no *host* para estes espaços que acabamos de alocar no *device*. A função da biblioteca CUDA que realiza tal tarefa também é bastante intuitiva para os usuários da linguagem C. Iremos utilizar a função "cudaMemcpy()" para copiar os dados dos vetores da CPU (a, b, c) para os vetores alocados na GPU (`a_d, b_d, c_d`). A função requisita quatro parâmetros:

cudaMemcpy(destino, origem, tamanho, sentido);

Onde **destino** identifica o ponteiro para onde vai a cópia; **Origem** identifica o ponteiro dos dados que serão copiados; **Tamanho** identifica o tamanho em bytes dos dados que serão copiados e; **Sentido**, que é um parâmetro especial do CUDA. Ele identifica o sentido da cópia dos dados que podem ser:

- cudaMemcpyHostToDevice: a cópia será feita do *host* para o *device*.
- cudaMemcpyDeviceToHost: a cópia será feita do *device* para o *host*.
- cudaMemcpyDeviceToDevice: a cópia será feita do *device* para o próprio *device*.

A Figura 3.15 mostra a cópia dos dois vetores que irão ser somados na GPU.

```
cudaMemcpy( a_d, a, N * sizeof(int), cudaMemcpyHostToDevice );
cudaMemcpy( b_d, b, N * sizeof(int), cudaMemcpyHostToDevice );
```

Figura 3.15: 3º Passo: Cópia de dados Ponteiros

Neste momento, os dois vetores que terão seus valores somados, já foram alocados e devidamente copiados para a GPU. O próximo passo, é a execução do *kernel* que realizará a soma dos vetores na GPU. Como descrito na Subseção 3.5.3, é feita uma chamada de função como na Figura 3.16. Os parâmetros de execução específicos do CUDA (`<<<10, 100>>>`) que se referem ao número de blocos e threads utilizados durante a execução do *kernel*, ficam

```
add<<<10,100>>>(a_d, b_d, c_d);
```

Figura 3.16: 4º Passo: Chamada do kernel "add"

a critério do desenvolvedor, analisar e verificar quais as melhores configurações para o seu problema de acordo com as características de sua GPU.

A partir desta chamada, o comando da tarefa passa a ser da API CUDA, enquanto o *host* fica livre para realizar outras tarefas ao mesmo tempo que a GPU executa o *kernel*. Para este caso específico, o *kernel* que será executado é apresentado na Figura 3.17. A partir daqui, é a GPU quem gerencia e controla os índices dos blocos e *threads* definidos pela variável "tid". Cada bloco irá receber uma "cópia" do *kernel*, e juntamente de suas *threads* irá realizar a soma de um pedaço dos vetores. Após um bloco terminar um pedaço, o comando de repetição *while* testa se ainda existem índices dos vetores que não foram somados, se sim, o bloco assume outro pedaço do vetor, e assim será até que o fim do vetor seja encontrado. Todo esse processo, é a GPU quem gerencia, cabendo ao desenvolvedor apenas definir os parâmetros de execução que irão ser atribuídos ao *kernel*. Um detalhe importante no corpo da função é a linha de "incremento" da variável **tid**:

```
tid += blockDim.x * gridDim.x;
```

é nesta linha de comando, que é atribuído um novo índice para uma *thread* após ela ter terminado de executar sua tarefa em um determinado índice anterior.

```
__global__ void add(int *a, int *b, int *c){
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N){
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}
```

Figura 3.17: Corpo do Kernel "add"

Finalizando a tarefa que lhe foi solicitada, o *device* retornará o comando ao *host*, que tem que copiar de volta os dados resultantes, calculados no *kernel* **add**. Este passo é praticamente idêntico ao da cópia dos dados da CPU para a GPU, exceto pelo último parâmetro, o sentido da cópia, que agora será do *device* para o *host*, como mostra a Figura 3.18. Neste caso,

a cópia é inversa, ou seja, o último parâmetro será **cudaMemcpyDeviceToHost**, indicando que a cópia tem que ser feita da GPU para a CPU.

```
cudaMemcpy( c, c_d, N * sizeof(int), cudaMemcpyDeviceToHost );
```

Figura 3.18: 5º Passo: Cópia dos dados calculados na GPU para a CPU

Após a realização da cópia dos resultados para o *host*, resta fazer apenas mais uma coisa relacionada à GPU: liberar o espaço de memória. Assim como na linguagem C, onde temos a função **free()** para liberar memória, também temos que liberar o espaço utilizado no *device* antes de encerrar o aplicativo. Como as outras funções da biblioteca CUDA, a função de liberar memória na GPU é a mesma da biblioteca C padrão, com o acréscimo da palavra "cuda" no início, ilustrado na Figura 3.19.

```
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
```

Figura 3.19: 6º Passo: Liberação de memória da GPU

O código completo da soma de vetores em CUDA está disponível no Apêndice A.

3.7 Hierarquia de Memória

Como já comentado, GPU's modernas estão equipadas com enormes quantidades de poder de processamento aritmético. Na verdade, a vantagem computacional que processadores gráficos têm em relação à CPUs, ajudou a precipitar o interesse inicial no uso de processadores gráficos para computação de propósito geral. Com centenas de unidades aritméticas na GPU, muitas vezes, o gargalo não é o rendimento aritmético dos chips, mas sim a largura de banda de memória do chip. Há tantos ALU's (Arithmetic Logic Unit) nos processadores gráficos que, às vezes simplesmente não podemos manter a entrada rápida o suficiente para sustentar as taxas elevadas de computação. Mas felizmente, CUDA nos oferece vários espaços de memória diferentes no *device* durante a execução de um *kernel*. São elas: **Memória Global**, **Memória Local**, **Memória Compartilhada**, **Memória constante** e **Memória de textura**

Toda *thread* que está em execução no programa, possui sua própria **Memória Local**, acessada somente por ela, para fim de armazenamento de dados.

A **Memória Global** é a principal memória da GPU, ou seja, todo o *grid* tem acesso a ela, fazendo com que *threads* de blocos diferentes possam compartilhar informações e dados processados. Mas ao fazer com que milhares de *threads* acessem o mesmo espaço de memória, o processo pode se tornar lento, fazendo o programador buscar alternativas para melhorar o desempenho.

Um dos recursos, é o uso da **Memória compartilhada**. O compilador CUDA C trata as variáveis em memória compartilhada diferentemente do que variáveis típicas. Ele cria uma cópia da variável para cada bloco que você inicie na GPU. Todas as *threads* de um determinado bloco compartilhão tal memória, mas não podem ver ou modificar a cópia desta variável que está em outros blocos. Isto proporciona um excelente meio pelo qual *threads* dentro de um mesmo bloco podem comunicar e colaborar em cálculos. Além disso, *buffers* de memória compartilhada residem fisicamente no GPU ao invés de residirem na DRAM (Dynamic Random-Access Memory), fora do *chip*. Devido a isso, a latência de acesso à *buffers* de memória compartilhada tende a ser muito mais baixa do que de costume, fazendo uma memória compartilhada eficaz para cada bloco. Para uma variável pertencer à memória compartilhada, ela deve receber o qualificador (3.5.2) `__shared__` em sua declaração.

Como o nome pode indicar, o uso de **Memória Constante** é para dados que não irão mudar ao longo de uma execução de *kernel*. O *hardware* da NVIDIA fornece 64KB de memória constante, para que ele trate este tipo de memória de maneira diferente do que trata a memória global. Quando se trata de manipulação de memória constante, o hardware CUDA pode transmitir uma única leitura de memória para cada meio *warp* (3.4), que é um grupo de 16 *threads* (metade de um *warp* que possui 32 *threads*). Se as *threads* em um *half-warp* fizerem pedidos do mesmo endereço de memória, a GPU irá gerar apenas um único pedido de leitura e, posteriormente, transmitir os dados para cada *thread*. Se a GPU está lendo certo número de dados da memória constante, vai gerar apenas 1/16 (cerca de 6 por cento) do tráfego de memória que seria gasto utilizando a memória global. E não é só isso, após a leitura de um endereço de memória de um *half-warp*, este endereço ficará no *cache* de dados constantes, o que possibilitará que outros *half-warp* tenham acesso ainda mais rápido ao mesmo endereço, diminuindo ainda mais o tráfego de memória adicional.

Mas infelizmente, nem sempre a utilização de memória constante vai garantir a melhoria de desempenho. Este recurso de retransmissão entre *half-warp* é um faca de dois gumes. Embora possa acelerar drasticamente o desempenho quando todas as 16 *threads* estão lendo o mesmo endereço, ele pode resultar em diminuição de desempenho quando as 16 *threads*

de um *half-warp* forem ler endereços diferentes, pois as 16 *threads* estão autorizadas a fazer apenas um único pedido de leitura por vez na memória constante (Sanders and Kandrot, 2010). Em algumas situações, usando a memória constante ao invés de memória global, podemos reduzir a largura de banda de memória necessária, tudo vai depender do desenvolver analisar o problema e verificar se o uso da mesma irá trazer benefícios à aplicação.

Em questão de programação, ha uma diferença na hora de copiar os dados para a GPU. No momento de copiar os dados para a memória constante da GPU, a função é diferente de **cudaMemcpy()**, sendo utilizado a função *cudaMencpyToSymbol()*. Um código de soma entre vetores utilizando memória constante é apresenta no Apêndice A.

Assim como a Memória Constante, a **Memória de Textura** é armazenada em cache no *chip*, por isso em algumas situações, proporcionará maior largura de banda efetiva, reduzindo os pedidos para a Memória Global. Especificamente, *caches* de textura são projetados para aplicações gráficas, onde padrões de acesso à memória exibem uma grande quantidade de localidade espacial. Em uma aplicação de computação, isto implica que uma *thread* provavelmente vá querer ler de um endereço "perto" do seu atual endereço, na qual uma outra *thread* está lendo, como mostrado na Figura 3.20

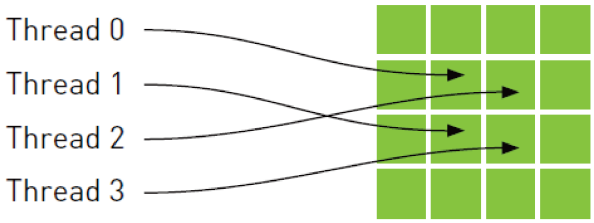


Figura 3.20: *Threads* utilizando a Memória de Textura. Fonte (Sanders and Kandrot, 2010)

Aritmeticamente, os quatro endereços indicados não são consecutivos, para que eles não sejam armazenados juntos em um *cache*, o que ficaria parecido com um esquema típico de *cache* em CPU. No entanto, este tipo de memória é projetado para acelerar os padrões de acesso, tendo um aumento no desempenho ao usar memória de textura no lugar de memória global. A memória de textura oferece diferentes modos de endereçamento, bem como dados de filtragem para alguns formatos de dados específicos (Rocha and Filho, 2010). Na verdade, esse tipo de padrão de acesso não é tão comum em computação de propósito geral, mas pode ser de grande ajuda se bem utilizado. Para maiores detalhes e um exemplo de simulação de transferência de calor utilizando memória de textura, consultar (Sanders and Kandrot, 2010).

A Figura 3.21 mostra a arquitetura das memórias na GPU. Com esta estrutura, a Memória Constante e a de Textura possuem melhor desempenho do que a Memória Local e a Global, devido à utilização de *cache* no acesso aos dados. Já as Memórias Compartilhadas tem o melhor desempenho entre todos os tipos por estarem no mesmo chip do microprocessador. Sendo a Memória Global a mais custosa no acesso, por estar mais distante fisicamente do processador.

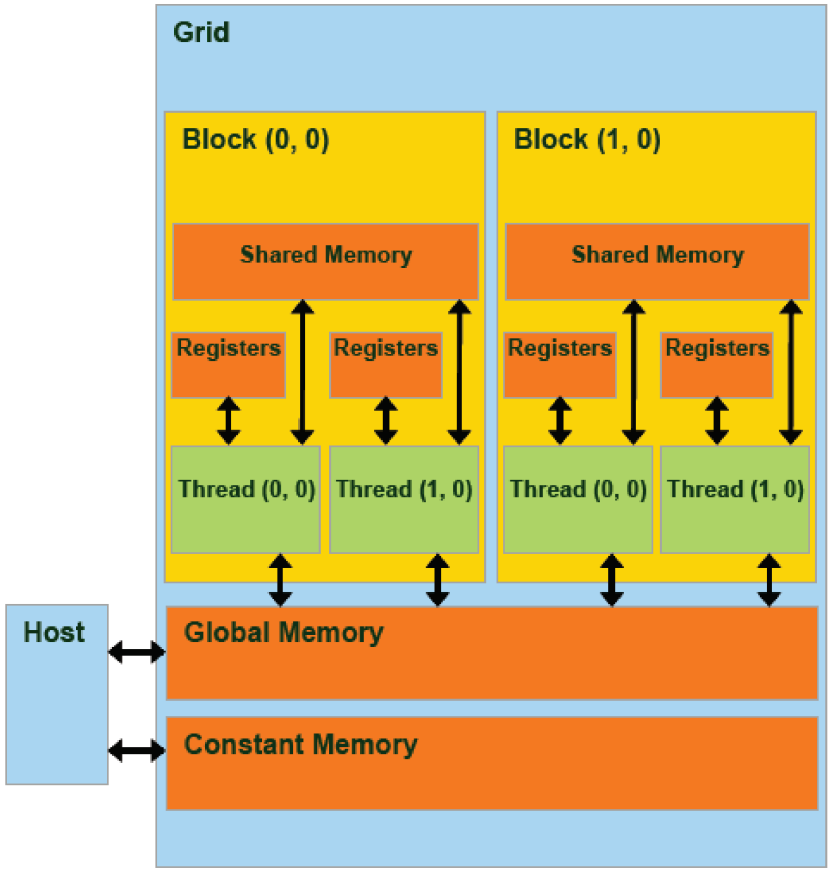


Figura 3.21: Arquitetura dos variados tipos de memória da GPU. Fonte (Martins and Lucas, 2010)

Todo o gerenciamento de memória passa pelo *runtime* do CUDA, incluindo a alocação, liberação e transferência de dados entre as memórias do *host* e do *device*. As diretivas responsáveis por esse gerenciamento estão disponíveis na biblioteca CUDA (Rocha and Filho, 2010).

3.7.1 Sincronização

A perspectiva de comunicação entre as *threads* é um modelo que traz grandes benefícios para a paralelização de algoritmos, mas tal comunicação precisa de alguns cuidados em certas situações. Se queremos uma comunicação entre *threads*, precisamos também de um mecanismo para sincronização entre *threads*. Por exemplo, se dada *thread* A escreve um valor para memória compartilhada e queremos que certa *thread* B faça algo com este valor, não podemos deixar a *thread* B iniciar os seus trabalhos até que saibamos que o trabalho da *thread* A está completo. Sem sincronização, criamos uma condição de corrida, onde a correção dos resultados de execução dependem dos detalhes do *hardware*. Existe um mecanismo para tratar esse tipo de situação no CUDA, o comando :

`__syncthreads()`.

Mas alguns cuidados têm de ser tomados. A arquitetura CUDA garante que nenhuma *thread* vá avançar para uma instrução além do `__syncthreads()` até que cada *thread* no bloco tenha executado o `__syncthreads()`. Infelizmente, se o `__syncthreads()` entrar em uma divergência (um "if" por exemplo), algumas das *threads* nunca vão chegar ao comando `__syncthreads()`. Portanto, por causa da garantia de que nenhuma instrução após um `__syncthreads()` possa ser executada antes que cada *thread* tenha executado a mesma, o *hardware* simplesmente continua a esperar por essas *threads*. E espera para sempre. O comando `__syncthreads()` é um mecanismo poderoso para garantir que sua aplicação massivamente paralela calcule resultados corretos. Mas por causa do potencial para causar consequências não intencionais, é preciso tomar cuidado ao usá-lo. Um código de produto de vetores, que faz uso da memória compartilhada e da sincronização de *threads*, está disponível no Apêndice A.

3.8 Eventos

Desenvolvedores estão acostumados a criar suas aplicações, e no fim, testar e comparar com outras aplicações que executam a mesma tarefa, para mensurar o seu desempenho em relação ao tempo. Podem-se usar os temporizadores da CPU ou do Sistema Operacional, mas com isso vamos incluir a latência e variação de qualquer tipo de fonte (*threads* do sistema operacional por exemplo). Se a finalidade é mensurar toda a aplicação em conjunto, este método está de acordo. Mas caso queiramos medir o tempo que uma GPU gasta para executar determinada tarefa, temos de utilizar os *Events* da API CUDA.

Um **Evento** em CUDA é como um carimbo de tempo da GPU, que é gravado em determinado ponto do tempo. Este recurso consiste em apenas duas etapas: criar um Evento e posteriormente, gravar o Evento.

- Criar as variáveis para marcar o evento com o tipo `cudaEvent_t`: start, stop.
- Criar o evento com a chamada `cudaEventCreate(&start)`
- Iniciar o evento com a chamada cudaEventRecord(start, 0). O segundo parâmetro será explicado na seção 3.10.
- Parar o evento com a chamada cudaEventRecord(stop, 0).
- Sincronizar o evento com a chamada cudaEventSynchronize(stop).
- Calcular a diferença entre "start" e "stop" para ter o tempo gasto pela tarefa feita pela GPU com `cudaEventElapsedTime(&variavelFloat, start, stop)`
- Destruir as variáveis de evento com cudaEventDestroy(start) e cudaEventDestroy(stop);

O último passo se faz necessário, devido a certa característica da GPU. Ela só irá retornar a marcação do tempo final de um evento, após terminar todas as suas operações, ou seja, ao solicitar o fim de um evento ao mesmo tempo em que a GPU executa outras tarefas, este tempo não será retornado de imediato. Ele ficará em uma fila de eventos, aguardando que todas as tarefas da GPU terminem. Este procedimento só é possível através da instrução **cudaEventSynchronize(stop)**.

Logo em seguida, para calcular o tempo que o kernel demorou para realizar a tarefa, lançamos uma chamada para `cudaEventElapsedTime(&variavelFloat, start, stop)`, com uma variável *float* que irá receber o resultado e as duas variáveis do tipo `cudaEvet_t`. E por fim, destruímos as variáveis de evento com chamadas para cudaEventDestroy(). No Apêndice A é apresentado um *kernel* de soma de vetores utilizando eventos CUDA.

3.9 Atomics

Existem algumas situações onde, algo incrivelmente simples de implemetação no modo *single-thread*, apresenta um sério problema quando tentamos implementar em uma arquitetura massivamente paralela (GPU). Assim como existem diferentes modelos de CPU com diferentes

capacidades e conjuntos de instruções, a arquitetura CUDA também tem suas diferenças de modelos de placas diferentes. Tais diferenças são apresentadas pela Capacidade de Computação (Compute Capability) de uma GPU NVIDIA, que até a produção deste trabalho poderiam variar entre 1.0 e 3.0. As Capacidades de Computação de cada GPU da NVIDIA podem ser encontradas em (Nvidia, 2012).

Antes de trabalhar com operações atômicas, deve-se procurar saber se a GPU dá suporte para tais operações, através da sua Capacidade de Computação que deve ser 1.1 ou superior, para suportar a referida tarefa na memória global da GPU. Caso a intenção seja utilizar operações atômicas na memória compartilhada, a GPU tem de ter uma capacidade de computação 1.2 ou superior. Além disso, é preciso informar ao compilador que o código exige capacidade de computação 1.2 ou superior. Como este trabalho foi desenvolvido em ambiente Windows e Microsoft Visual Studio 2010, segue abaixo os passos para configurar a Capacidade de Computação no Visual Studio 2010.

- Na janela principal do Visual, clicar em: *Properties* do menu *Project*.
- Na janela que ira aparecer, navegar pela árvore de itens da esquerda pelo seguinte caminho: `Configuration Properties -> CUDA C/C++ -> Device`.
- Nas opções da esquerda, no campo *Code Generation* modificar o valor "‘`compute_10,sm_10`" pelo valor da capacidade de computação da GPU utilizada. Neste trabalho foi utilizado uma GPU de capacidade de computação 2.0, desta forma, o campo fica com o valor "`compute_20,sm_20`". A Figura 3.22 apresenta o resultado desta configuração.

O melhor exemplo de operações atômicas, se dá através do operador de incremento "$x++$" da biblioteca C. Onde após executar essa expressão, o valor de "x" deve ser maior do que antes de executar o incremento. Tal operação exige três passos:

- Ler o valor de x.
- Adicionar 1 ao valor lido na primeira etapa.
- Escrever o resultado de volta para x.

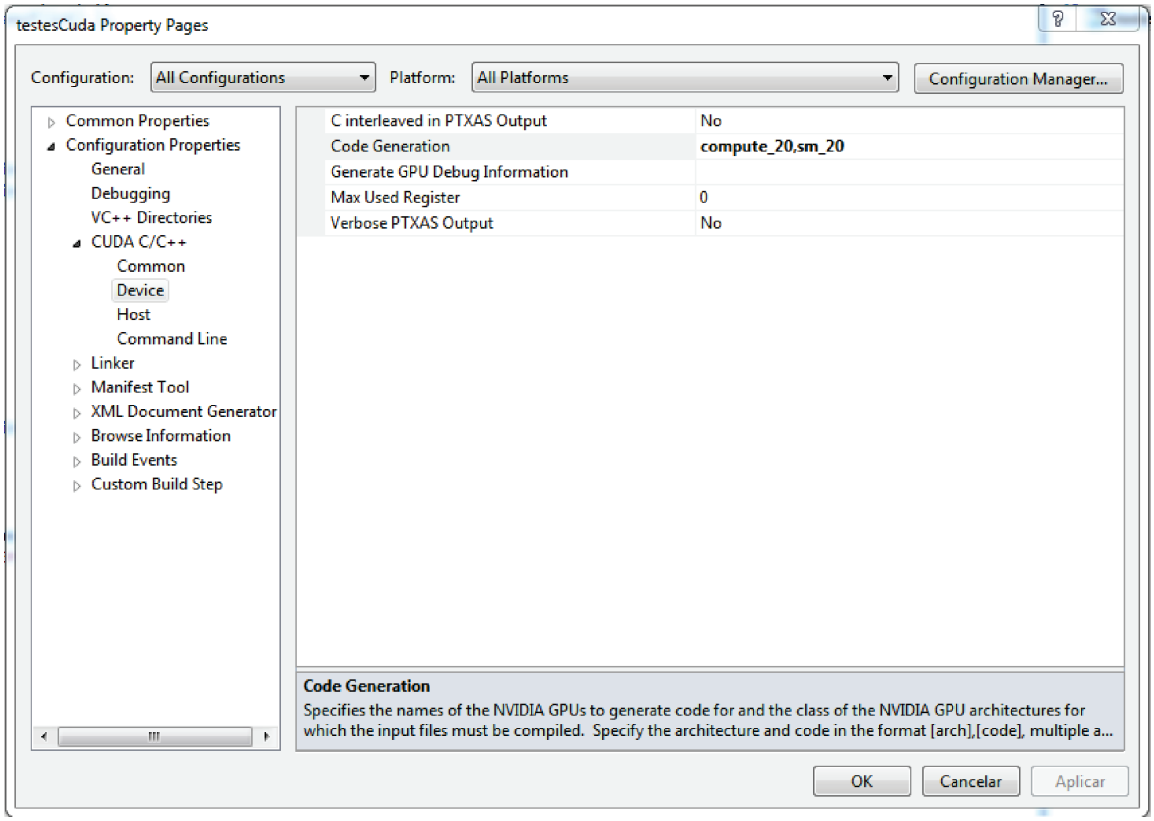


Figura 3.22: Modificando a Capacidade de Computação da GPU no Visual Studio 2010

Este processo pode ter sérias complicações quando duas *threads* necessitam realizar o mesmo incremento na variável "x", resultando em um resultado desfavorável como no exemplo da Tabela 3.3, onde o valor final de "x" deveria ser 9 e acabou sendo 8. Quando transportamos um código *single-thread* para uma versão *multithreaded*, surge um grande potencial para resultados imprevisíveis se as múltiplas *threads* precisarem ler ou escrever valores em uma mesma variável.

Passo	Exemplo
Thread A lê o valor de x	A lê 7 de x
Thread B lê o valor de x	B lê 7 de x
Thread A adiciona 1 ao valor lido	A resulta 8
Thread B adiciona 1 ao valor lido	B resulta 8
Thread A escreve o resultado em x	x = 8
Thread B escreve o resultado em x	x = 8

Tabela 3.3: Operações atômicas, concorrência entre *threads*. Fonte (Sanders and Kandrot, 2010)

A execução destas operações não pode ser quebradas em partes menores por outras threads, o que leva a nomeá-las como operações atômicas. A API CUDA suporta várias operações atômicas que permitem operar com segurança na memória, mesmo quando inúmeras *threads* estão concorrendo pelo mesmo acesso. No Apêndice A está presente um algoritmo para cálculo de histogramas (Sanders and Kandrot, 2010) utilizando as diretivas de operações atômicas em GPU.

3.10 Streams

Processadores gráficos da NVIDIA ainda oferecem outra classe de paralelismo, semelhante ao paralelismo de tarefas encontrado em aplicações *multithread* de CPU's. Ao invés de computar simultaneamente a mesma função em lotes de dados como se faz com o paralelismo de dados, o paralelismo de tarefas consiste em um aplicativo fazer duas ou mais tarefas completamente diferentes de forma paralela, como por exemplo, baixar algo da rede enquanto faz um cálculo aritmético qualquer. As tarefas estariam em paralelo, apesar de não ter ligação uma com a outra. O paralelismo de tarefas na GPU não é tão flexível como em um processador de propósitos gerais, mas oferece algumas vantagens que podem ajudar o desenvolvedor em suas implementações.

Uma **Stream** CUDA é uma fila de operações de GPU que são executadas em uma ordem específica. Essas operações podem ser tratar de um lançamento de *kernel* ou até cópias de memória. A ordem em que as tarefas serão executadas é a ordem em que elas são adicionadas à Stream específica, sendo possível que elas sejam executadas em paralelo. Os Eventos, vistos na Seção 3.8, estão diretamente relacionados com as Streams. Nas chamadas para **cudaEventRecord(start, 0)**, o segundo parâmetro serve para indicar uma Stream no qual o evento irá ser relacionar, ou seja, o evento irá iniciar na primeira tarefa da Stream referida, e terminar na última tarefa da mesma.

O primeiro passo para incluir uma Stream, é verificar se o *device* utilizado, suporta o recurso chamado de Sobreposição de Dispositivos. Este recurso permite que a GPU execute simultaneamente um *Kernel* CUDA e uma cópia de dados entre as memórias do *device* e do *host*. É possível recuperar tal informação, através da propriedade **deviceOverlap** da estrutura **cudaDeviceProp** presente na API CUDA e exemplificada no Apêndice A. A seguir, as chamadas para criar uma Stream CUDA:

- Criar uma variável do tipo stream: `cudaStream_t nomeVariavel`
- Criar a stream: `cudaStreamCreate(&nomeVariavel)`

Após criada, podemos adicionar tarefas à Stream, como cópia de dados entre as memórias e lançamentos de *kernels*. Para adicionar uma cópia de memória na fila de uma stream, é preciso utilizar uma versão diferente de **cudaMemcpy()** ao copiar os dados entre GPU e CPU, além de uma função diferente para alocar memória no *host*. Para fazer a cópia entre as memórias, tanto para copiar da CPU para GPU, quanto o sentido inverso, tem-se que utilizar

cudaMemcpyAsync(destino, origem, tamanho, sentido, stream)

que irá fazer com que a cópia tenha característica assíncrona, ou seja, temos a certeza que ela irá acontecer, mas não se sabe em que momento irá ocorrer. Isso nos leva ao segundo caso, a alocação no *host* tem que ser do tipo

cudaHostAlloc(endereço, tamanho, cudaHostAllocDefault)

onde o último parâmetro neste caso é *default*, podendo assumir outros valores. Essa última chamada é necessária, pois não se sabe em que momento a cópia será feita. Com esta chamada os dados e seu endereço de memória não serão paginados pelo sistema operacional, evitando que elas vão para o disco rígido em algum momento, fazendo com que fiquem disponíveis o tempo todo e não correndo o risco de estarem indisponíveis no momento em que a Stream invocar a cópia entre os dados. Este recurso, assim como o recurso de memória compartilhada, pode trazer alguns problemas. O desenvolvedor tem que estar atento, pois a utilização de memória não paginada, pode comprometer o desempenho de outras aplicações em uso no computador. Se bem utilizada, trará grandes benefícios aplicação, assim como prejuízos se não aplicada corretamente.

O paralelismo de tarefas, não se da apenas com o uso de uma Stream. Temos que utilizar duas ou mais Streams para se alcançar o paralelismo de tarefas. No Apêndice A é apresentado um aplicativo onde é feito a soma de dois vetores e aplicado o resultado em um terceiro, ao mesmo tempo em que, outros 3 vetores fazem a mesma operação, com o uso de Streams.

Capítulo 4

Implementações e Resultados

Para aprofundar os conceitos da tecnologia CUDA, além da aplicação dos algoritmos presentes em (Sanders and Kandrot, 2010), foi proposto o desenvolvimento de um algoritmo de busca de vizinhos onde, dado um determinado conjunto de pontos cartesianos, devem-se localizar os vizinhos de cada ponto de acordo com suas respectivas coordenadas e seu raio. A procura por vizinhos, as vezes se torna um dos gargalos de desempenho de vários algoritmos que necessitam de uma otimização elevada, como por exemplo em Métodos Numéricos sem Malha (Fonseca, 2011), que são métodos utilizados para simular fenômenos físicos naturais de alto grau de precisão, e que necessitam da etapa de localização de vizinhos, que por sua vez, torna-se uma etapa muito mais lenta ao se comparar com os clássicos Métodos de Elementos Finitos (Hughes, 2000).

A busca de vizinhos em Métodos Numéricos sem Malha é baseada a partir do conceito de Domínio de suporte. O Domínio de Suporte determina o número de nós usados para aproximar uma função em um ponto **x**, podendo ter diferentes formas e tamanhos, além de variar entre diferentes pontos de domínio, como na Figura 4.1.

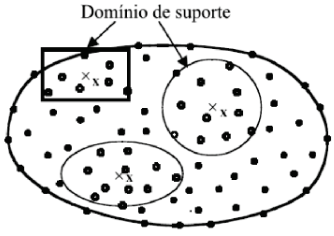


Figura 4.1: Domínio de Suporte em diferentes pontos e formas. Fonte (Lima, 2011)

Em (Lima, 2011) são apresentados três estratégias de buscas de vizinhos baseadas em Domínio de Suporte: *ShapeSupportDomain, InfluenceSupportDomain, KnodesSupportDomain*. Todos elas implementadas na biblioteca Computational Geometry Algorithms Library (CGAL), que é uma biblioteca de geometria computacional, cujo propósito principal é fornecer acesso fácil a algoritmos geométricos eficientes e confiáveis. Ela é escrita em linguagem C++, e usada em várias áreas como computação gráfica, sistemas de informações geográficas e outros (CGAL, 2012). Os algoritmos que realizam estas estratégias, são todos otimizados com árvores de buscas, sendo extremamente eficientes. O objetivo aqui, é aplicar a estratégia *shapeSupportDomain* sem a utilização de árvores de busca, e logo depois, aplicar a paralelização deste algoritmo para GPU's utilizando CUDA.

4.1 Implementação na CPU

A Figura 4.2 apresenta o algoritmo implementado para a busca de vizinhos para a CPU. Iniciamos o processo lendo um arquivo de entrada contendo as informações dos nós como: o número de nós, as coordenadas cartesianas de cada nó (x, y, z) e os respectivos raios de cada nó. Após a leitura do arquivo, é construído o domínio do problema, atribuindo além das características lidas no arquivo, a coordenada máxima possível para um ponto. A cada ponto adicionado, são atribuídos os valores referentes a construção do domínio de suporte sobre *ShapeSupportDomain*, que neste trabalho irá ser representado por um retângulo. Com todos os pontos adicionados com seus respectivos domínios de suporte caracterizados, podemos aplicar a equação da distância entre dois pontos, para distinguir os pontos que são realmente vizinhos do ponto analisado naquele momento, adicionado os mesmos a uma lista de vizinhos.

O *ShapeSupportDomain* consiste em: a partir de um ponto **x**, realizar uma busca pelos possíveis vizinhos deste ponto demarcados por uma figura geométrica qualquer que é baseada no raio do ponto **x**, como na Figura 4.3, onde **C** delimita o domínio de suporte e **R** o raio de **x**.

A Partir disso, é feita uma busca dos verdadeiros vizinhos do ponto **x**, utilizando a equação da distância entre dois pontos sobre os pontos que estão dentro domínio de suporte, como apresentado na Figura 4.4.

```
1  Leitura dos nós (CPU);
2  para cada leitura de um nó sobre o domínio faça
3  |  Adicionar o ponto ao domínio;
4  |  Calcular o retângulo que irá caracterizar o domínio de suporte;
5  fim para cada
6  Localizando Vizinhos (CPU);
7  para cada nó sobre o domínio faça
8  |  Localizar os possíveis vizinhos do nó atual através do domínio de suporte;
9  |  Localizar os verdadeiros nós vizinhos que exercem influência ao raio do nó atual;
10 |  Adicionar os nós vizinhos na lista de vizinhança;
11 fim para cada
12 Escrever a solução;
```

Figura 4.2: Visão geral do algoritmo implementado para a CPU

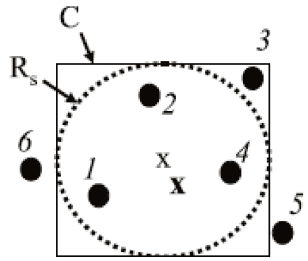


Figura 4.3: Domínio de Suporte do ponto **x**. Fonte (Lima, 2011)

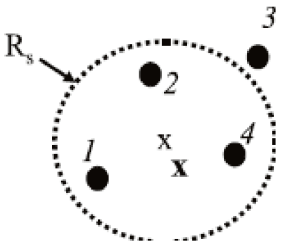


Figura 4.4: Busca dos verdadeiros vizinhos do ponto **x**. Fonte (Lima, 2011)

4.2 Implementação na GPU

A princípio, todo o código implementado para a CPU foi feito com o paradigma de orientação a objetos, mas devido algumas restrições da arquitetura CUDA, muitos dos conceitos de orientação a objeto tiveram que ser retirados, para efeito de compatibilidade no momento

de comparações de desempenho. Em (Thrust, 2012) é possível encontrar um projeto de uma biblioteca de algoritmos paralelos que se assemelha ao C++ Standard Template Library (STL), que busca integração da computação em GPU com os conceitos de orientação a objeto.

A parte inicial do algoritmo, adição dos nós ao domínio, não foi paralelizada para a GPU, devido a falta de complexidade da operação. Não há restrição alguma para tal, mas os ganhos não seriam perceptíveis. Desta forma, somente a segunda parte do algoritmo foi paralelizada, onde a GPU faz o trabalho de localizar os possíveis vizinhos de dado nó e sequencialmente os verdadeiros vizinhos de influência do nó. A visão geral do algoritmo com a adição de um *kernel* que irá executar na GPU fica como o algoritmo descrito na Figura 4.5.

```
1  Leitura dos nós (CPU);
2  para cada leitura de um nó sobre o domínio faça
3  |   Adicionar o ponto ao domínio;
4  |   Calcular o retângulo que irá caracterizar o domínio de suporte;
5  fim para cada
6  Localizando Vizinhos (GPU);
7  para cada nó sobre o domínio faça
8  |   Localizar os possíveis vizinhos do nó atual através do domínio de suporte;
9  |   Localizar os verdadeiros nós vizinhos que exercem influência ao raio do nó atual;
10 |   Adicionar os nós vizinhos na lista de vizinhança;
11 fim para cada
12 Escrever a solução (CPU);
```

Figura 4.5: Visão geral do algoritmo implementado para a GPU

Para transportar o código para a GPU, foram utilizados os passos básicos da programação CUDA tratada neste trabalho. Após a leitura dos nós por parte da CPU, é feita uma alocação e uma cópia do vetor de pontos para a GPU através de **cudaMalloc() e cudaMemcpy()** respectivamente. Logo após, é feita uma chamada para o *kernel* `localizaPossiveisVizinhosQuadrante_d()`, que irá realizar as operações destinadas à GPU citadas em 4.5. Em seguida, após ter calculado as operações, o resultado é enviado de volta para a CPU através de outra chamada **cudaMemcpy()**. A partir deste momento, a CPU pode realizar quaisquer operação com os nós e seus respectivos vizinhos encontrados. Uma visão do *kernel* executado na GPU (passos 6 a 10 no algoritmo 4.5) é apresentado na Figura 4.6, lembrando que o endereçamento de *threads* e blocos são feitos automaticamente pela GPU. O *Kernel* completo é encontrado no Apêndice A.

```
1  Atribuição dos índices das threads;
2  enquanto thread atual for menor que número de pontos faça
3      para cada possível vizinho no domínio faça
4          se possível vizinho difere do ponto atual então
5              se possível vizinho está no domínio de suporte então
6                  Calcula a distancia entre os pontos;
7                  se for vizinho então
8                      Adiciona à lista de vizinho do ponto atual;
9                  senão
10                     Não é vizinho;
11                 fim se
12             senão
13                 Não é vizinho;
14             fim se
15         senão
16             O ponto não é vizinho dele mesmo;
17         fim se
18     fim para cada
19     Atualiza a quantidade de Vizinhos do ponto atual;
20     Atribui novo índice a thread atual;
21 fim enqto
```

Figura 4.6: Visão geral do algoritmo implementado para a GPU

4.3 Resultados

Para fins de testes e comparação da aplicação feita na GPU em relação aos testes feitos na CPU, foram criadas cinco instâncias de pontos cartesianos: 1.000, 5.000, 10.000, 20.000 e 30.000 pontos, geradas aleatoriamente pelo função **rand()** da biblioteca padrão C. Os únicos parâmetros controlados foram o valor máximo das coordenadas e o raio. Estes dois últimos, receberam os valores padrões 100 e 5 respectivamente para todos os pontos e instâncias. Na Tabela 4.1 os resultados da aplicação do algoritmo 4.2 rodado na CPU para as cinco instâncias.

Nº Pontos	1.000	5.000	10.000	20.000	30.000
Tempo(seg)	0,031	0,46	1,9	13,7	35,18

Tabela 4.1: Resultados obtidos na CPU

É nítido o crescimento do tempo de processamento do código sequencial na CPU ao aumentarmos o número de pontos na instância.

Para os testes no código da GPU (4.5), além da variação de número de pontos, também foram variados os números de blocos e *threads* lançados no kernel durante a execução do algoritmo, para analisar o comportamento do processamento. Segue os resultados (em segundos) na Tabela 4.2 .

Nº Pontos	1.000	5.000	10.000	20.000	30.000
30 Blocos - 500 Threads	0.76	0.83	0.96	1.54	2,67
60 Blocos - 500 Threads	0.73	0.79	0.96	1.52	2,55
300 Blocos - 100 Threads	0.73	0.80	0.93	1.47	2,38
500 Blocos - 60 Threads	0.77	0.80	0.92	1.44	2,29
500 Blocos - 500 Threads	0.75	0.82	0.95	1.55	2,54

Tabela 4.2: Resultados obtidos na GPU

Ao compararmos as duas implementações, percebe-se que o algoritmo executado pela GPU, só começa a ser melhor a partir de 10.000 pontos, exemplificando as questões relativas a carga de dados. Só se torna interessante o processamento na GPU, em casos onde a carga de dados cresce de forma muito grande. Comparando a instância de melhor caso com 30.000 pontos, na qual o algoritmo executado na GPU foi executado em 2,29 segundos, temos um ganho de mais de 15 vezes em relação ao algoritmo da CPU, que executou em 35,18 segundos.

Como comentado anteriormente, o número de blocos e *threads* que são lançados no *kernel* influenciam diretamente no seu processamento. No algoritmo proposto acima, percebe-se uma melhoria no tempo de execução do algoritmo, ao acrescentarmos mais blocos e diminuirmos as *threads*. A melhor combinação encontrada foi a de 500 blocos e 60 *threads*. A combinação correta de blocos e *threads* é uma questão de testes e análise. Cabe ao desenvolvedor conhecer as especificações da sua GPU, e o modelo de negócio da sua aplicação, para escolher os valores que irão agregar maior capacidade de processamento.

Capítulo 5

Conclusões

O uso de GPUs para programação paralela de propósitos gerais está crescendo em ritmo acelerado, de acordo com a necessidade de processamento do mercado. A arquitetura CUDA, juntamente com a linguagem e ferramentas que o acompanham, esta conseguindo trazer muitos desenvolvedores para explorar a área que antes era pouco atrativa, devido a alta complexidade das ferramentas e linguagens disponíveis. Uma das evidências deste fato, é que durante o término deste trabalho, a tecnologia já estava presente em três dos cinco supercomputadores do mundo.

Ficam evidentes as grandes facilidades que a tecnologia proporciona para os desenvolvedores. Nenhum conhecimento relacionado a gráficos é exigido para que se trabalhe com CUDA, ficando abstrato para os desenvolvedores, assim como a comunicação entre o *Host* e o *Device*. Tendo um mínimo de conhecimento da linguagem C, a maior dificuldade encontrada está no aprendizado das questões relacionadas à indexação de blocos e *threads*, juntamente em como extrair o máximo de desempenho da aplicação. Esta é uma das questões a serem tratadas em trabalhos futuros. A análise profunda da configuração de blocos e *threads* é uma área que pode ser bastante explorada.

As CPU's ainda tem uma certa vantagem de processamento, quando tratamos de cálculos complexos com necessidade de precisão de mais de 6 dígitos ou com baixa carga de dados. As GPU's ainda tem dificuldade para tratar cálculos com esta taxa de precisão, e a partir de testes, pôde-se perceber que em aplicações com baixa carga de dados, os ganhos não são muito significativos devido às cópias entre as memórias durante a execução de uma *kernel*. Mas a partir do momento em que a carga de dados cresce, o uso das GPU's já se torna interessante, pela razão de sua arquitetura ser massivamente paralela. Muito se falou em

comparações entre GPU e CPU, mas devemos perceber que ao final das contas, a programação paralela de propósitos gerais com a GPU depende da CPU a todo momento. A GPU a princípio, não tem o objetivo de substituir a CPU, mas sim trabalhar em conjunto como um processador adicional para oferecer melhor desempenho nas aplicações.

Métodos Numéricos sem Malha, são apenas uma das várias áreas que necessitam de melhoria no desempenho de seus algoritmos. Apesar dos ótimos resultados obtidos, a implementação apresentada neste trabalho tinha como objetivo principal a aplicação da tecnologia para aprendizado, além de oferecer uma nova forma de implementar a estratégia de busca de vizinhos baseadas em Domínio de Suporte: *ShapeSupportDomain* . Até o momento, esta tarefa é executada através do uso da biblioteca CGAL (CGAL, 2012), que utiliza árvores de busca. São algoritmos extremamente otimizados, e fica como trabalho futuro, a busca por implementações paralelas que possam equiparar aos algoritmos da CGAL.

Outra questão para trabalho futuros, é a utilização de várias GPU's trabalhando juntas. A tecnologia CUDA oferece formas de se tirar proveito de um *cluster* entre GPU's. Se com uma GPU os ganhos de performace já são grandes, trabalhando em conjunto, as melhorias devem ser de grande valia para a áre de otimização de algoritmos.

Referências Bibliográficas

ANSONI, J. L. (2010). Resolução de um problema térmico inverso utilizando processamento paralelo em arquiteturas de memória compartilhada. Master's thesis, Universidade de São Paulo - Escola de Engenharia de São Carlos. [citado na(s) páginas(s) 5]

Belytschko, T., Krongauz, Y., Organ, D., Fleming, M., and Krysl, P. (1996). Meshless methods: An overview and recent developments. computer methods in applied mechanics and engineering. International Journal for Numerical Methods in Engineering. [citado na(s) páginas(s) 10]

CGAL (2012). Homepage computational geometry algorithms library. http://www.cgal.org/. Acesso em Junho, 2012. [citado na(s) páginas(s) 40, 46]

Corrigan, A., Camelli, F., Lohner, R., and Wallin, J. (2009). Running unstructured grid based cfd solvers on modern graphics hardware. 19th AIAA Computational Fluid Dynamics. [citado na(s) páginas(s) 5]

Flynn, L. J. (2004). Intel halts development of 2 new microprocessors. Technical report, The New York Times. [citado na(s) páginas(s) 1, 6]

Fonseca, A. R. (2011). Algoritmos Eficientes em Métodos sem Malha. PhD thesis, Universidade Federal de Minas Gerais. [citado na(s) páginas(s) 10, 39]

Foster, I. T. (1995). Designing and building parallel programs: concepts and tools for parallel software engineering. Addison Wesley - 1 edition. [citado na(s) páginas(s) 2, 6]

GUEDES, C. M. (2006). Métodos Sem Malha em Problemas de Mecânica Computacional. PhD thesis, Faculdade de Engenharia, Universidade do Porto. [citado na(s) páginas(s) 9]

Hughes, T. J. R. (2000). The finite element method: Linear static and dynamic finite element analysis. Dover Publications. [citado na(s) páginas(s) 39]

InformationWeek (2012). Uff se torna primeiro centro de excelência em cuda da américa latina. http://informationweek.itweb.com.br/voce-informa/uff-se-torna-primeiro-centro-de-excelencia-em-cuda-da-america-latina. Acesso em Junho, 2012. [citado na(s) páginas(s) 9]

Lima, N. Z. (2011). Desenvolvimento de um framework para métodos sem malha. Master's thesis, Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Minas gerais. [citado na(s) páginas(s) 39, 40, 41]

Lopes, B. C. and Azevedo, R. J. (2008). Computação de alto desempenho utilizando cuda. 20th International Symposium on Computer Architecture and High Performance Computing. [citado na(s) páginas(s) xi, 24, 25]

Martins, W. S. and Lucas, D. C. S. (2010). Programação cuda - minicurso conpeex 2010. [citado na(s) páginas(s) 32, 65, 66]

Noel, P. B., Walczak, A. M., Hoffmann, K. R., Xu, J., Corso, J. J., and Schafer, S. (2008). Clinical evaluation of gpu-based cone beam computed tomography. MICCAI Workshop: High-Performance Medical Image Computing and Computer Aided Intervention. [citado na(s) páginas(s) 5]

Nvidia (2011). Nvidia cuda library online. http://developer.download.nvidia.com/compute/cuda/4_2/rel/toolkit/docs/online/structcudaDeviceProp.html. Acesso em Junho, 2012. [citado na(s) páginas(s) 52]

Nvidia (2012). Site nvidia - zona cuda. http://www.nvidia.com.br/object/cuda_home_new_br.html. Acesso em Maio, 2012. [citado na(s) páginas(s) 3, 4, 11, 35]

Rocha, K. A. P. and Filho, L. J. V. (2010). Introdução ao cuda utilizando métodos numéricos. [citado na(s) páginas(s) 24, 31, 32]

Sanders, J. and Kandrot, E. (2010). CUDA By Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional - 1 edition. [citado na(s) páginas(s) xi, 2, 7, 8, 20, 31, 36, 37, 39, 50, 59]

Schepke, C. (2009). Ambientes de programação paralela. Technical report, Universidade Federal do Rio Grande do Sul, Instituto de Informática - Programa de Pós de Graduação. [citado na(s) páginas(s) 2, 6]

Thrust (2012). Homepage thrust library. http://code.google.com/p/thrust/. Acesso em Junho, 2012. [citado na(s) páginas(s) 42]

Vasconcellos, F. B. (2009). Programando com gpu's: Paralelizando o método lattice-boltzmann com cuda. Master's thesis, Universidade Federal do Rio Grande do Sul. [citado na(s) páginas(s) 5]

Viana, R. M. (2011). Programação em gpu: Passado, presente e futuro. [citado na(s) páginas(s) 7]

Apêndice A

Algoritmos - Kernels

Esse apêndice apresenta alguns códigos escrito em CUDA/C de forma completa em relação a trechos de exemplos descritos nos capítulos deste trabalho. Muitos deles foram baseados nos exemplos propostos em (Sanders and Kandrot, 2010), e executados em um computador com as configurações descritas em 3.2.1.

A.1 Kernel de adição de vetores

O algoritmo de adição de vetores apresenta duas listas de números, onde queremos somar elementos correspondentes de cada lista e armazenar o resultado em uma terceira lista. É possível notar alguns padrões comuns que são empregamos em CUDA:

- Alocamos três vetores no dispositivo usando chamadas para cudaMalloc(): dois vetores, `a_d` e `b_d`, para manter as entradas, e um array, `c_d`, para manter o resultado
- Como estamos lidando com a memória do *device*, temos que limpar com cudaFree().
- Usando cudaMemcpy(), copiamos os dados de entrada para o dispositivo com o parâmetro cudaMemcpyHostToDevice e os dados do resultado de volta para o host com cudaMemcpyDevicetoHost.
- Executamos o código no *device* no kernel add() do código main() no *host*, usando a nova sintaxe de chamada de um kernel com os símbolos ‘‘<<<>>>’’.
- Foi escrito uma função chamada add() que executa no *device*. Isso é feito a partir de um código C e adicionando um qualificador `__global__` ao nome da função.

Listing A.1: Kernel de adição de vetores

```
#include <iostream>
#include <cuda_runtime.h>
#include <cuda_runtime_api.h>

#define N (33*1024)

//Kernel add
__global__ void add(int *a, int *b, int *c){
        int tid = threadIdx.x + blockIdx.x * blockDim.x;
        while (tid < N){
                c[tid] = a[tid] + b[tid];
                tid += blockDim.x * gridDim.x;
        }
}

int main(void){

        int a[N], b[N], c[N];
        int* a_d, *b_d, *c_d;

        cudaMalloc((void**)&a_d, N * sizeof(int) );//alocação dos vetores na GPU
        cudaMalloc((void**)&b_d, N * sizeof(int) );
        cudaMalloc((void**)&c_d, N * sizeof(int) );

        for (int i=0; i<N; i++){
                a[i] = -i;
                b[i] = i* i;
        }
        //Cópia dos vetores da CPU para a GPU
        cudaMemcpy( a_d, a, N * sizeof(int), cudaMemcpyHostToDevice );
        cudaMemcpy( b_d, b, N * sizeof(int), cudaMemcpyHostToDevice );

        add<<<N+127/128,128>>>(a_d, b_d, c_d); //execução do kernel

        //cópia do resultado calculado na GPU para a CPU
        cudaMemcpy( c, c_d, N * sizeof(int), cudaMemcpyDeviceToHost );

        cudaFree(a_d);//desalocando memória da GPU
        cudaFree(b_d);
        cudaFree(c_d);
        return 0;
}
```

A.2 Recuperando propriedades do Device

Em algumas situações, o programador terá a necessidade ou até mesmo obrigação de saber algumas informações relevantes sobre o *device* na qual ele está trabalhando. Pode ser útil saber o quanto de memória e quais os tipos de recursos o *device* oferece. Em casos onde são usados

mais de um *device*, tem que existir uma maneira de determinar qual deles está operando em certo momento. Existe alguns mecanismos para realizar estes últimos questionamento através de chamadas de funções específicas, para recuperar informações da estrutura **cudaDevicePro**, que contém uma série de informações sobre o(s) dispositivos que estão ativos na máquina em questão. Segue abaixo um kernel que recuperar algumas das mais importantes informações de um *device* e em (Nvidia, 2011) é possível encontrar todos os tipos presentes na estrutura.

Listing A.2: Aplicativo que retorna informações sobre o device

```
1
2 #include <cuda_runtime_api.h>
3 #include <iostream>
4
5 int main(void){
6         cudaDeviceProp prop;
7
8         int count;
9         cudaGetDeviceCount (&count);
10
11        for (int i=0; i<count; i++){
12                cudaGetDeviceProperties(&prop, i);
13                printf( " --- INformacao Geral do Dispositivo %d ---\n", i );
14                printf( "Nome: %s\n", prop.name );
15                printf( "Capacidade de Computacao: %d.%d\n", prop.major, prop.minor );
16                printf( "Taxa de clock: %d\n", prop.clockRate );
17
18                printf( " --- Informacao da memória do dispositivo %d ---\n", i );
19                printf( "Total Memoria global: %ld\n", prop.totalGlobalMem );
20                printf( "Total Memoria constante: %ld\n", prop.totalConstMem );
21
22                printf( " --- Informaçoes MP do device %d ---\n", i );
23                printf( "Threads em 1 warp: %d\n", prop.warpSize );
24                printf( "Maximo de threads por bloco: %d\n",
25                prop.maxThreadsPerBlock );
26                printf( "Maximo de threads por dimensions: (%d, %d, %d)\n",
27                prop.maxThreadsDim[0], prop.maxThreadsDim[1],
28                prop.maxThreadsDim[2] );
29                printf( "Maximo de blocos por gride: (%d, %d, %d)\n",
30                prop.maxGridSize[0], prop.maxGridSize[1],
31                prop.maxGridSize[2] );
32                printf( "\n" );
33        }
34        return 0;
35 }
```

A.3 Kernel com Memória Compartilhada e Sincronização

No Produto escalar de vetores, primeiramente nós multiplicamos os elementos correspondentes de dois vetores de entrada. É muito semelhante à adição de vetores, utilizando multiplicação em vez de adição. No entanto, em vez de em seguida armazenar esses valores para um terceiro vetor de saída, temos que somar todos os valores do terceiro vetor, para produzir uma única saída escalar. Como precisamos que o resultado seja a soma de todos esses produtos, cada thread mantém uma soma parcial dos pares que adicionou. As novidades no código ficam por conta de:

- Foi declarado um buffer de memória compartilhada identificado como *cache*. Este **buffer** é usado para armazenar a soma parcial de cada thread.
- No ponto onde é preciso somar todos os valores temporários que já foram colocados no cache, é preciso tomar certo cuidado, pois é uma operação perigosa. Foi Preciso a utilização de um método para garantir que todas já tenham escrito para o *cache* compartilhado, antes que alguém tente ler o mesmo *buffer*. O método utilizado foi o de sincronizar as *threads* com o comando `__syncthreads()`, que garante que cada *thread* no bloco tenha concluído suas instruções antes do hardware executar a próxima instrução em qualquer *thread*.

Listing A.3: Kernel Produto Escalar de Vetores com memória compartilhada e sincronização

```
1  #include <cuda_runtime_api.h>
2  #include <cstdlib>
3  #include <iostream>
4
5  #define imin(a,b) (a<b?a:b)
6
7  const int N = 26 * 1024;
8  //Numero de threads padronizado. Depende da placa, geralmente no maximo 512
9  const int threadsPerBlock = 256;
10  //Garante que o numero de blocos de acordo com o número de threads.
11 const int blocksPerGrid = imin( 32, (N+threadsPerBlock-1) / threadsPerBlock);
12
13 __global__ void dot( float *a, float *b, float *c){
14
15         //Variável compartilhada por todas threads do mesmo bloco.
16         __shared__ float cache[threadsPerBlock];
17
18         //Atribui o Índice da thread - Indice Thread/Indice bloco/Tamanho bloco.
```

```
19          int tid = threadIdx.x + blockIdx.x * blockDim.x;
20          //Índice para a variável compartilhada, irá armazenar a soma de todas threads da
                mesma linha.
21          int cacheIndex = threadIdx.x;
22          //Armazena a soma temporária deste bloco
23          float temp = 0;
24
25          //Garante que só executa até o número certo de dados. Caso hajam threads sobrando
                , ignora.
26          while(tid < N){
27                  temp += a[tid] * b[tid]; //Multiplicação dos valores dos 2 vetores
28                  tid += blockDim.x * gridDim.x; //Atribuição dos índices.
29          }
30
31          cache[cacheIndex] = temp; //Atribui a soma total de um bloco no chache de índice
                threadIdx.x
32
33          //Garante que nenhum comando será executado enquanto todas as threads do bloco
                não terminarem.
34          __syncthreads();
35
36          int i = blockDim.x/2;
37
38          while (i != 0){
39                  if(cacheIndex < i)
40                          cache[cacheIndex] += cache[cacheIndex + i];
41                  __syncthreads();
42                  i /= 2;
43          }
44
45          if(cacheIndex == 0)
46                  c[blockIdx.x] = cache[0];
47  }
48  int main(){
49
50          float *a, *b, c, *partial_c;
51
52          float *dev_a, *dev_b, *dev_partial_c;
53
54          //alocação de memória na CPU
55          a = new float[N];
56          b = new float[N];
57          partial_c = new float[blocksPerGrid];
58
59          //alocação de memória na GPU
60          cudaMalloc ( (void**)&dev_a, N*sizeof(float) );
61          cudaMalloc ( (void**)&dev_b, N*sizeof(float) );
62          cudaMalloc ( (void**)&dev_partial_c, blocksPerGrid*sizeof(float) );
63
64          //preenche a memória do host com dados
65          for(int i=0; i<N; i++){
66                  a[i] = 1;
67                  b[i] = 1;
68          }
```

```
69
70          //Copia dos arrays 'a' e 'b' para a GPU
71          cudaMemcpy( dev_a, a, N*sizeof(float), cudaMemcpyHostToDevice );
72          cudaMemcpy( dev_b, b, N*sizeof(float), cudaMemcpyHostToDevice );
73
74          dot<<<blocksPerGrid, threadsPerBlock>>>(dev_a, dev_b, dev_partial_c);
75
76          //copia do array c da GPU para a CPU
77          cudaMemcpy( partial_c, dev_partial_c, blocksPerGrid*sizeof(float),
                cudaMemcpyDeviceToHost );
78
79          //fim na CPU
80          c = 0;
81          for(int i=0; i<blocksPerGrid; i++){
82                  c += partial_c[i];
83          }
84
85          printf(" O valor e %.6g \n", c);
86
87
88          //libera memoria da GPU e CPU
89          cudaFree(dev_a);
90          cudaFree(dev_b);
91          cudaFree(dev_partial_c);
92
93          free(a);
94          free(b);
95          free(partial_c);
96
97          return 0;
98  }
```

A.4 Kernel com Memória Constante

Este é um kernel simples de soma entre vetores, onde todos os vetores recebem o valor do vetor anterior somado sempre ao 1° vetor(vetor a). Este é um kernel muito simples, com o intuito de apresentar o uso da memória constante em GPU. Como todos os vetores vão ser somados ao primeiro, que não terá seu valor alterado, alocamos este primeiro vetor na memória constante da GPU. As novidades são comentadas abaixo.

- Foi declarado um vetor `__contant__` no ínicio do algoritmo. Este é o primeiro vetor, que não terá seus dados alterados e será apenas para leitura.
- Como o primeiro vetor já foi alocado, não é preciso chamar **cudaMalloc()** para o mesmo, e no momento de copiar os dados para a memória constante da GPU, a função

é diferente de **cudaMemcpy()**(função que copia para a memória global da GPU), sendo utilizado a função **cudaMencpyToSymbol()**.

Listing A.4: Kernel de soma ao anterior com memória constante

```
#include <cuda_runtime_api.h>
#include <iostream>

//variaveis: N = tamanho array, blocksize = numero de blocos
const int N = 100;
const int blocksize = 2;

//Variavei armazenada na memoria constante da GPU
__constant__ float d_a[N*N];

//kernel de soma ao anterior com o primeiro
__global__ void add_anterior(float* b, float* c, float* d, float *e, int N){
        int i = threadIdx.x + blockIdx.x * blockDim.x;
        int j = threadIdx.y + blockIdx.y * blockDim.y;
        int index = i + j * N;

        if(i < N && j < N){
                b[index] = d_a[index] + d_a[index];
                c[index] = b[index] + d_a[index];
                d[index] = c[index] + d_a[index];
                e[index] = d[index] + d_a[index];
        }
}

int main(){
        float* a = new float[N*N];
        float* b = new float[N*N];
        float* c = new float[N*N];
        float* d = new float[N*N];
        float* e = new float[N*N];

        for (int i=0; i < N*N; ++i){
                a[i] = 1.5;
                b[i] = 2.5;
                c[i] = 3.5;
                d[i] = 4.5;
        }

        float *d_b, *d_c, *d_d, *d_e;
        const int size = N*N*sizeof(float);

        //Cópia do array a para a memória consntante da GPU
        cudaMemcpyToSymbol( d_a, a, size );

        //Alocação de memória para a memória global da GPU
        cudaMalloc((void**)&d_b, size);
        cudaMalloc((void**)&d_c, size);
        cudaMalloc((void**)&d_d, size);
```

```
49          cudaMalloc((void**)&d_e, size);
50
51          //Cópia dos dados para a memória global da GPU
52          cudaMemcpy( d_b, b, size, cudaMemcpyHostToDevice);
53          cudaMemcpy( d_c, c, size, cudaMemcpyHostToDevice);
54          cudaMemcpy( d_d, d, size, cudaMemcpyHostToDevice);
55          cudaMemcpy( d_e, e, size, cudaMemcpyHostToDevice);
56
57          //Criação de blocos e threads de 2 dimensões
58          dim3 dimBlock( blocksize, blocksize);
59          dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);
60
61          //Chamada ao kernel add_anterior
62          add_anterior<<<dimGrid, dimBlock>>>(d_b, d_c, d_d, d_e, N);
63
64          //Cópia do resultado calculado na GPU para a CPU
65          cudaMemcpy( e, d_e, size, cudaMemcpyDeviceToHost);
66
67          //Imprime os dois primeiros indices do resultado
68          std::cout << e[0] <<" - " << e[1] << "\n";
69
70          //Libera espaço na GPU
71          cudaFree( d_b );
72          cudaFree( d_c );
73          cudaFree( d_d );
74          cudaFree( d_e );
75
76          return 0;
77  }
```

A.5 Kernel com Eventos

Este kernel é o mesmo apresentado na seção 1 deste apêndice, acrescentando as funcionalidades dos Eventos em CUDA.

Listing A.5: Kernel de adição de vetores com Eventos

```
1  #include <iostream>
2  #include <cuda_runtime.h>
3  #include <cuda_runtime_api.h>
4
5  #define N (33*1024)
6
7  //kernel de adição de vetores
8  __global__ void add(int *a, int *b, int *c){
9          int tid = threadIdx.x + blockIdx.x * blockDim.x;
10         while (tid < N){
11                 c[tid] = a[tid] + b[tid];
12                 tid += blockDim.x * gridDim.x;
13         }
14 }
```

```
15
16 int main(void){
17
18         int a[N], b[N], c[N];
19         int* a_d, *b_d, *c_d;
20
21         //alocação dos vetores na GPU
22         cudaMalloc((void**)&a_d, N * sizeof(int) );
23         cudaMalloc((void**)&b_d, N * sizeof(int) );
24         cudaMalloc((void**)&c_d, N * sizeof(int) );
25
26         for (int i=0; i<N; i++){
27                 a[i] = -i;
28                 b[i] = i* i;
29         }
30
31         //cópia dos vetores da CPU para GPU
32         cudaMemcpy( a_d, a, N * sizeof(int), cudaMemcpyHostToDevice );
33         cudaMemcpy( b_d, b, N * sizeof(int), cudaMemcpyHostToDevice );
34
35         cudaEvent_t start, stop; //variáveis tipo Evento CUDA
36         cudaEventCreate(&start); //Criação do Evento start
37         cudaEventRecord(start, 0); //Marcação do Evento Start
38
39         add<<<10,100>>>(a_d, b_d, c_d); //Chamada ao kernel add
40
41         cudaEventRecord(stop, 0); //Marcação do fim do Evento
42         cudaEventSynchronize(stop); //Coloca o Evento na fila, aguardando o término das
               tarefas da GPU
43
44         //cópia do resultado calculado na GPU para a CPU
45         cudaMemcpy( c, c_d, N * sizeof(int), cudaMemcpyDeviceToHost );
46
47         std::cout << c[N-1] << "\n";
48
49         //Calculo do tempo gasto no kernel através de Eventos
50         float elapsedTime;
51         cudaEventElapsedTime(&elapsedTime, start, stop);
52         std::cout << "Tempo gasto pela GPU para executar o kernel: " << elapsedTime;
53
54         //Destruindo os eventos
55         cudaEventDestroy(start);
56         cudaEventDestroy(stop);
57
58         //desalocando memória da GPU
59         cudaFree(a_d);
60         cudaFree(b_d);
61         cudaFree(c_d);
62
63         return 0;
64 }
```

A.6 Kernel com Atomics

Para exemplificar o uso de operações atômicas, será apresentado um algoritmo para cálculo de histogramas encontrado em (Sanders and Kandrot, 2010). Um histograma consiste em: dado um conjunto de dados que possui vários elementos distintos ou não, o histograma representa o número de vezes que cada elemento aparece no conjunto de dados, ou seja, a sua frequência. A tabela A.1 apresenta um histograma da expressão "Programação Paralela com CUDA".

Elementos	7	3	1	1	1	2	2	3	2	3	1
Frequência	A	C	D	E	G	L	M	O	P	R	U

Tabela A.1: Histograma da expressão: "Programação Paralela com CUDA".

Os dados que serão analisados por um histograma podem significar qualquer coisa, qualquer tipo de dado, mas neste aplicativo de exemplo, foi utilizado um fluxo aleatório de bytes, utilizando uma função auxiliar chamada `big_random_block()`. Foram criados 100 *megabytes* de dados aleatórios. Uma vez que cada *byte* de 8 *bits* aleatórios pode ser qualquer um dos 256 valores diferentes contidos na arquitetura do computador (de 0x00 a 0xFF), o nosso histograma precisa conter 256 caixas para os elementos, a fim de incrementar o número de vezes que cada valor é encontrado nos dados. Foi criada uma matriz de inteiros com 256 índices e todos inicializados com zero.

Com o histograma criado e todos os campos inicializados com zero, resta encontrar a frequência com que cada valor aparece nos dados contidos no `buffer[]`. A ideia aqui é que sempre que vemos algum valor "z" no `buffer[]`, queremos incrementar o valor "z" do nosso histograma representado por `histo[]`. Desta forma, está sendo contado o número de vezes em que aparece o valor de "z".

Como ja dito, o problema com cálculo de um histograma dos dados de entrada decorre do fato de que várias *threads*podem querer incrementar o mesmo índice do histograma de saída, ao mesmo tempo. Nesta situação, é preciso usar incrementos atômicos para evitar uma situação de erro como explicado anteriormente. Neste caso específico, foi necessário utilizar também, os recursos de memória compartilhada, pois a utilização de recursos de operações atômicas na memória global traz grande queda de desempenho da aplicação. Utilizando a memória compartilhada, o kernel se divide em duas partes. Primeiramente, cada bloco irá calcular um histograma individualmente, podendo assim calcular estes na memória compartilhada. Em

seguida, tem-se que mesclar todos os histogramas temporários dos blocos para o *buffer histo* global. As novidades relativas a programação CUDA nesta aplicação ficam por conta de:

- Utilização da função **cudaMemset()** que assim como a função **menset()** na biblioteca C padrão, irá atribuir valores a uma certa posição de memória alocada.
- Utilização de operação atômica de incremento com a chamada: atomicAdd(elemento, quantidade). Onde a variável "elemento"será incrementada com o parâmetro "quantidade".

Segue abaixo a aplicação completa que calcula histograma com utilização de operações atômicas CUDA.

Listing A.6: Kernel de calculo de histogramas utilizando atomics na GPU

```
1
2 #include <iostream>
3 #include <cuda_runtime_api.h>
4
5 #define SIZE    (500*1024*1024)//tamanho dos dados de entrada
6
7 void* big_random_block( int size );//função auxiliar que gera dados
8
9 __global__ void histo_kernel( unsigned char *buffer, long size, unsigned int *histo ){
10        //variável compartilhada para cada bloco calcular um semi-histograma
11    __shared__  unsigned int temp[256];
12    temp[threadIdx.x] = 0;
13    __syncthreads();//sincroniza as threads do bloco
14
15    int i = threadIdx.x + blockIdx.x * blockDim.x;//atribui indice às threads
16    int stride = blockDim.x * gridDim.x;
17    while (i < size) {
18            //garante aromicidade das operações entre threads do mesmo bloco
19        atomicAdd( &temp[buffer[i]], 1 );
20        i += stride;
21    }
22    __syncthreads();
23        //garante atomicidade das operações entre todos os blocos
24    atomicAdd( &(histo[threadIdx.x]), temp[threadIdx.x] );
25 }
26
27 int main( void ) {
28    unsigned char *buffer = (unsigned char*)big_random_block( SIZE );
29
30    cudaEvent_t     start, stop;//Criando, iniciando e gravando um evento
31    cudaEventCreate( &start ) ;//parar calcular o tempo da aplicação na GPU
32    cudaEventCreate( &stop ) ;
33    cudaEventRecord( start, 0 );
```

```
34
35      // alocando e copiando memória na GPU
36      unsigned char *dev_buffer;
37      unsigned int *dev_histo;
38      cudaMalloc( (void**)&dev_buffer, SIZE );
39      cudaMemcpy( dev_buffer, buffer, SIZE, cudaMemcpyHostToDevice );
40      cudaMalloc( (void**)&dev_histo, 256 * sizeof( int ) );
41      cudaMemset( dev_histo, 0, 256 * sizeof( int ) );
42
43          //invocação do kernel
44      histo_kernel<<<128,256>>>( dev_buffer, SIZE, dev_histo );
45
46          //cópia do histograma calculado na GPU para a CPU
47      unsigned int     histo[256];
48      cudaMemcpy( histo, dev_histo, 256 * sizeof( int ), cudaMemcpyDeviceToHost );
49
50      // recuperação do tempo gasto
51      cudaEventRecord( stop, 0 );
52      cudaEventSynchronize( stop );
53      float   elapsedTime;
54      cudaEventElapsedTime( &elapsedTime, start, stop );
55      printf( "Tempo gasto:  %3.1f ms\n", elapsedTime );
56
57          //destruindo eventos e desalocando memória
58      cudaEventDestroy( start );
59      cudaEventDestroy( stop );
60      cudaFree( dev_histo );
61      cudaFree( dev_buffer );
62      free( buffer );
63      return 0;
64  }
65
66  void* big_random_block( int size ) {
67      unsigned char *data = (unsigned char*)malloc( size );
68      for (int i=0; i<size; i++)
69          data[i] = rand();
70
71      return data;
72  }
```

A.7 Kernel com Streams

As Streams oferecem o paralelismo de tarefas na programação de propósitos gerais em GPU. Para alcançar tal objetivo, temos que utilizar duas ou mais streams. No aplicativo abaixo nós temos seis vetores. Os dois primeiros serão somados e o resultado é recebido pelo terceiro, enquanto o quarto e o quinto vetor, são somados e o resultado atribuído ao sexto e último vetor. As duas operações de soma são feitas de forma simultânea, com o uso de duas Streams CUDA. As Stream são basicamente uma fila de operações que a GPU certamente irá executar,

mas sem a confirmação de em que momento isso ocorrerá. Por causa dessa característica, duas chamadas especiais são diferentes quanto aos outros aplicativos apresentados neste apêndice:

- cudaMemcpyAsync(destino, origem, tamanho, sentido, stream): Onde os parâmetros são os mesmo de **cudaMemcpy()**, com exceção do último, que contém o nome da stream que irá receber a cópia de dados como tarefa. O termo "Async"da chamada vem da palavra assíncrona, que indica que esta cópia não tem o momento correto de ocorrer, apenas que irá acontecer em algum momento.
- cudaHostAlloc(endereço, tamanho, cudaHostAllocDefault): Para que o item anterior possa ocorrer, a alocação da memória correspondente no *host* tem que ser com a chamada **cudaHostAlloc()**, sendo o último parâmetro um argumento *default* para o nosso exemplo. Com este tipo de alocação, nós garantimos que o sistema operacional não irá "paginar"o endereço de memória e seus dados, ou seja, mandá-los para o disco rígido em algum momento de inatividade. Esse comportamento é necessário, pois a cópia que solicitamos através de uma stream, não tem momento certo para acontecer. Desta forma evitamos qualquer problema na hora da cópia ocorrer. O desenvolvedor tem que estar atento ao uso de memória não paginada, pois pode trazer sérios problemas de desempenho de outras aplicações que possam estar rodando no computador e para a própria memória da CPU.

Listing A.7: Kernel de paralelismo de tarefas com Streams

```
1
2 #include <cuda_runtime_api.h>
3 #include <iostream>
4
5 #define N (33*1024)
6 #define FULL_DATA_SIZE (N*20)
7
8 __global__ void add(int *a, int *b, int *c){
9         int tid = threadIdx.x + blockIdx.x * blockDim.x;
10        while (tid < N){
11                c[tid] = a[tid] + b[tid];
12                tid += blockDim.x * gridDim.x;
13        }
14 }
15
16 int main( void ) {
17
18        cudaDeviceProp prop;
19        int whichDevice;
20        cudaGetDevice( &whichDevice );
21        cudaGetDeviceProperties( &prop, whichDevice );
```

```
22
23          if (!prop.deviceOverlap) {
24                  printf( "Device não suporta a sobreposição \n" );
25                  return 0;
26          }
27
28          cudaEvent_t start, stop;
29          float elapsedTime;
30          // Inicializa temporizadores
31          cudaEventCreate( &start );
32          cudaEventCreate( &stop );
33          cudaEventRecord( start, 0 );
34
35          // Inicializa stream's
36          cudaStream_t stream0, stream1;
37          cudaStreamCreate( &stream0 );
38          cudaStreamCreate( &stream1 );
39
40          int *host_a, *host_b, *host_c;
41          int *dev_a0, *dev_b0, *dev_c0;
42          int *dev_a1, *dev_b1, *dev_c1;
43
44          // Aloca memória na GPU
45          cudaMalloc( (void**)&dev_a0, N * sizeof(int) );
46          cudaMalloc( (void**)&dev_b0, N * sizeof(int) );
47          cudaMalloc( (void**)&dev_c0, N * sizeof(int) );
48          cudaMalloc( (void**)&dev_a1, N * sizeof(int) );
49          cudaMalloc( (void**)&dev_b1, N * sizeof(int) );
50          cudaMalloc( (void**)&dev_c1, N * sizeof(int) );
51
52          // Aloca Memória não paginável na CPU
53          cudaHostAlloc( (void**)&host_a, FULL_DATA_SIZE * sizeof(int),
54          cudaHostAllocDefault );
55          cudaHostAlloc( (void**)&host_b, FULL_DATA_SIZE * sizeof(int),
56          cudaHostAllocDefault );
57          cudaHostAlloc( (void**)&host_c, FULL_DATA_SIZE * sizeof(int),
58          cudaHostAllocDefault );
59
60          for (int i=0; i<FULL_DATA_SIZE; i++) {
61                  host_a[i] = rand();
62                  host_b[i] = rand();
63          }
64
65          //Cada uma das operações na GPU serão feitas FULL_DATA_SIZE vezes
66          for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
67                  // Cópia dos vetores "a" para as respectivas streams
68                  cudaMemcpyAsync( dev_a0, host_a+i,
69                  N * sizeof(int),
70                  cudaMemcpyHostToDevice,
71                  stream0 );
72                  cudaMemcpyAsync( dev_a1, host_a+i+N, N * sizeof(int),
73                  cudaMemcpyHostToDevice,
74                  stream1 );
75                  // Cópia dos vetores "b" para as respectivas streams
```

```
76                  cudaMemcpyAsync( dev_b0, host_b+i, N * sizeof(int),
77                  cudaMemcpyHostToDevice, stream0 );
78                  cudaMemcpyAsync( dev_b1, host_b+i+N, N * sizeof(int),
79                  cudaMemcpyHostToDevice,
80                  stream1 );
81                  //Chamada do kernel add para cada stream
82                  add<<<N/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );
83                  add<<<N/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );
84
85                  //Cópia dos resultados parciais para os vetores "c"
86                  cudaMemcpyAsync( host_c+i, dev_c0, N * sizeof(int),
87                  cudaMemcpyDeviceToHost,
88                  stream0 );
89                  cudaMemcpyAsync( host_c+i+N, dev_c1, N * sizeof(int),
90                  cudaMemcpyDeviceToHost,
91                  stream1 );
92          }
93
94          //Sincroniza as Streams, fazendo a CPU aguardar a GPU
95          cudaStreamSynchronize( stream0 );
96          cudaStreamSynchronize( stream1 );
97
98          cudaEventRecord( stop, 0 );
99          cudaEventSynchronize( stop );
100         cudaEventElapsedTime( &elapsedTime, start, stop );
101         printf( "Tempo: %3.1f ms\n", elapsedTime );
102
103         //Desalocação de memória da GPU e CPU
104         cudaFreeHost( host_a );
105         cudaFreeHost( host_b );
106         cudaFreeHost( host_c );
107         cudaFree( dev_a0 );
108         cudaFree( dev_b0 );
109         cudaFree( dev_c0 );
110         cudaFree( dev_a1 );
111         cudaFree( dev_b1 );
112         cudaFree( dev_c1 );
113         //Destruição das streams
114         cudaStreamDestroy( stream0 );
115         cudaStreamDestroy( stream1 );
116         return 0;
117 }
```

A.8 Identificadores Globais de Threads

A arquitetura CUDA trabalha com um novo modelo que envolve *threads*, blocos e uma grade. Onde uma grade é um conjunto de blocos de no máximo duas dimensões(x, y), e os blocos são contituídos de um conjunto de *threads* de no máximo três dimensões(x, y, z). Esta divisão é realizada de forma automática pela GPU, bastando apenas informar as configurações de

execução como número de *threads* e blocos que serão lançados em dado kernel. Mas apesar dessa divisão, no final das operações, podemos dizer que tudo resulta em um grande vetor de *threads*, onde cada uma possui um identificador global, baseado nos índices e dimensões de blocos e *threads* que são representados pelas variáveis built-in (3.5.4). Na Figura A.1 temos um exemplo de uma grade que possui 3 blocos, que por sua vez, possuem 5 *threads* cada um. Temos exposto na figura, os índices individuais de cada bloco e *thread*, e podemos constatar também que a dimensão do bloco (blockDim) é igual a 5, pois em cada bloco temos 5 *threads*. Com essas informações, conseguimos o identificador global de cada *threads* aplicando a seguinte operação:

int i = blockId.x * blockDim.x + threadId.x

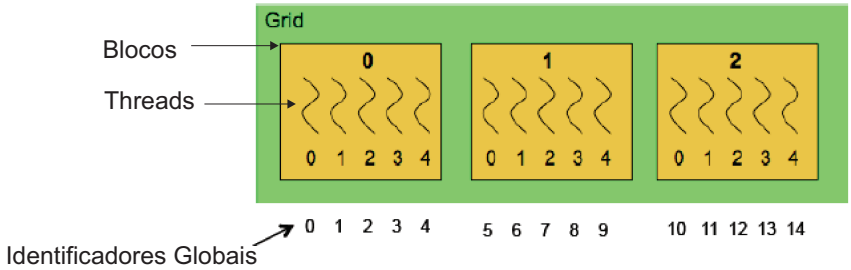


Figura A.1: Modelo de um grid com blocos e threads. Fonte: (Martins and Lucas, 2010)

Este último exemplo, mostra os identificadores globais das *threads* onde blocos e *threads* possuem apenas uma dimensão. Para aplicações onde as dimensões são maiores, o cálculo para localizar os identificadores globais muda um pouco. O primeiro passo é idêntico ao anterior, localizando o índice global da dimensão "x" com: **int i = blockId.x * blockDim.x + threadId.x**. Logo após, temos que fazer a mesma operação, mas com os operadores da dimensão "y":

int j = blockId.y * blockDim.y + threadId.y

Com o índice de cada dimensão calculada, encontra-se o índice global multiplicando o índice de "y" (j) pelo dimensão de "x" (blockDim.x) e pela dimensão da grade(gridDim.x). Somando ao final, o índice de "x" (i):

int index = i + j * blockDim.x + gridDim.x

A Figura A.2 apresenta um exemplo de uma Grade com quatro blocos de duas dimensões cada, distintos pelas cores verde, amarelo, roxo e laranja. Cada um destes blocos possuem quatro threads de duas dimensões cada. E ao lado, é apresentado um quadro com os valores de cada variável par ao cálculo dos identificadores globais de cada *thread* que variam de 0 a 15.

blockIdx		threadIdx		I	J	index
x	y	x	y			
0	0	0	0	0	0	0
		0	1	0	1	4
		1	0	1	0	1
		1	1	1	1	5
0	1	0	0	0	2	8
		0	1	0	3	12
		1	0	1	2	9
		1	1	1	3	13
1	0	0	0	2	0	2
		0	1	2	1	6
		1	0	3	0	3
		1	1	3	1	7
1	1	0	0	2	2	10
		0	1	2	3	14
		1	0	3	2	11
		1	1	3	3	15

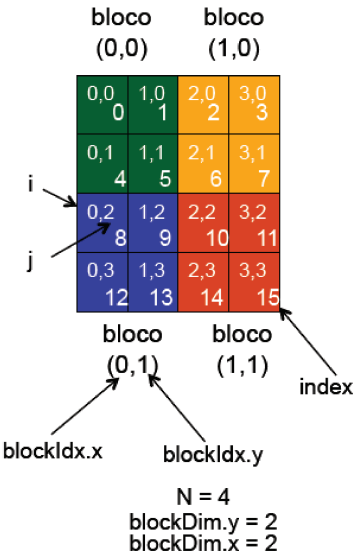


Figura A.2: Modelo de um grid com blocos e threads bidimensionais. Fonte: (Martins and Lucas, 2010)

A.9 Kernel de localização de vizinhos shapeSupportDomain

Este kernel apresenta o algoritmo que foi implementa durante a execução deste trabalho, para fins de prática e desenvolvimento dos conceitos utilizados durante os capítulos iniciais de introdução ao CUDA. Este kernel em particular, é apenas para apresentar como foi desenvolvido a ideia da localização dos vizinhos de um conjunto de pontos. Ele não irá compilar, devido a falta de algumas partes do código não estarem contidas neste trabalho.

Listing A.8: Kernel localização de vizinhos

```
1
2 //Código meramente ilustrativo. Este código não irá compilar devido a falta de estruturas
      não adicionadas neste trabalho
3
4 #include <cuda_runtime_api.h>
5
6 //kernel de localização de vizinhos pela técnica ShapeSupportDomain
```

```
7  __global__ void localizaPossiveisVizinhosQuadrante_d(Ponto* vetorDePontos, int
       numeroCoordenadas){
8  //atribuição de índice das threads
9  int x = threadIdx.x + blockIdx.x * blockDim.x;
10 int contVizinhos;
11 double resultado;
12
13 //Enquanto o indice da thread for menor que o número de pontos
14 while(x < numeroCoordenadas){
15         contVizinhos = 0;
16         //enquanto o número de possíveis vizinhos for menor que o número de pontos
17         for(int j=0; j<numeroCoordenadas; ++j){
18                 //Um ponto não é vizinho dele mesmo
19                 if(x != j){
20                         //Se está no domínio de suporte do ponto atual
21                         if( ( vetorDePontos[j].quadranteX <= vetorDePontos[x].xmax) &&
22                                         ( vetorDePontos[j].quadranteX >= vetorDePontos[x
                                            ].xmin) &&
23                                         ( vetorDePontos[j].quadranteY <= vetorDePontos[x
                                            ].ymax) &&
24                                         ( vetorDePontos[j].quadranteY >= vetorDePontos[x
                                            ].ymin) &&
25                                         ( vetorDePontos[j].quadranteZ <= vetorDePontos[x
                                            ].zmax) &&
26                                         ( vetorDePontos[j].quadranteZ >= vetorDePontos[x
                                            ].zmin)  )
27                         {
28                                 //Distancia entre dois pontos
29                                 resultado = sqrt( pow( (vetorDePontos[x].x -
                                        vetorDePontos[j].x), 2 ) + pow( (vetorDePontos[x].y
                                        - vetorDePontos[j].y), 2 ) + pow( (vetorDePontos[x].
                                        z - vetorDePontos[j].z), 2 ) ) ;
30                                 //Se está dentro do raio do ponto atual, é vizinho
31                                 if( resultado <= vetorDePontos[x].raio ){
32                                         vetorDePontos[x].indexVizinhos[contVizinhos] = j;
33                                         contVizinhos++;
34                                 }
35                         }
36                 }
37         }
38         vetorDePontos[x].quantidadeVizinhos = contVizinhos;//Atualiza a qtd de vizinhos
               do ponto
39         x += blockDim.x * gridDim.x; //atualiza o índice da thread
40 }
41 }
42
43 int main(){
44
45 cudaMalloc((void**)&vetorDePontos_d, dominio.numeroDePontos * sizeof(Ponto));
46 cudaMemcpy(vetorDePontos_d, dominio.vetorDePontos, dominio.numeroDePontos * sizeof(Ponto)
       , cudaMemcpyHostToDevice );
47 localizaPossiveisVizinhosQuadrante_d<<<300, 500>>>(vetorDePontos_d, dominio.
       numeroDePontos);
```

```
48 cudaMemcpy(dominio.vetorDePontos, vetorDePontos_d, dominio.numeroDePontos * sizeof(Ponto)
       , cudaMemcpyDeviceToHost );
49
50 return 0;
51 }
```