Um estudo do uso eficiente de programas em placas gráficas

Patricia Akemi Ikeda

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciência da Computação

Orientador: Prof. Dr. Alfredo Goldman vel Lejbman

São Paulo, Agosto de 2011

Um estudo do uso eficiente de programas em placas gráficas

Esta dissertação trata-se da versão original
da aluna Patricia Akemi Ikeda.

Resumo

Inicialmente projetadas para processamento de gráficos, as placas gráficas (GPUs) evoluíram para um coprocessador paralelo de propósito geral de alto desempenho. Devido ao enorme potencial que oferecem para as diversas áreas de pesquisa e comerciais, a fabricante NVIDIA destaca-se pelo pioneirismo ao lançar a arquitetura CUDA (compatível com várias de suas placas), um ambiente capaz de tirar proveito do poder computacional aliado à maior facilidade de programação.

Na tentativa de aproveitar toda a capacidade da GPU, algumas práticas devem ser seguidas. Uma delas consiste em manter o *hardware* o mais ocupado possível. Este trabalho propõe uma ferramenta prática e extensível que auxilie o programador a escolher a melhor configuração para que este objetivo seja alcançado.

Palavras-chave: CUDA, NVIDIA, GPU Computing

Abstract

Initially designed for graphical processing, the graphic cards (GPUs) evolved to a high performance general purpose parallel coprocessor. Due to huge potencial that graphic cards offer to several research and commercial areas, NVIDIA was the pioneer lauching of CUDA architecture (compatible with their several cards), an environment that take advantage of computacional power combined with an easier programming.

In an attempt to make use of all capacity of GPU, some practices must be followed. One of them is to maximizes hardware utilization. This work proposes a practical and extensible tool that helps the programmer to choose the best configuration and achieve this goal.

Keywords: CUDA, NVIDIA, GPU Computing

Sumário

1 Introdução **8**

2 *GPU Computing* **10**

2.1 Breve Histórico dos Processadores Gráficos 10

2.1.1 GPU - Primeira Geração 10

2.1.2 GPU - Segunda Geração 11

2.1.3 GPU - Terceira Geração 12

2.1.4 GPU - Quarta Geração 13

2.1.5 GPU - Quinta Geração 14

2.2 GPGPU 14

2.2.1 *Pipeline* Gráfico 15

2.2.2 *Shaders* Gráficos 17

2.3 *GPU Computing* 17

2.4 GPU vs. CPU 17

2.5 Evolução das GPUs e a arquitetura Fermi 21

2.5.1 Endereçamento virtual de 64-bits 25

2.5.2 *Caches* 25

2.5.3 ECC - *Error Correcting Codes* 25

2.5.4 Instruções atômicas mais rápidas 26

2.5.5 Espaço de memória unificado 26

2.5.6 Suporte a depuração de código no *kernel* 26

2.5.7 Aritmetica de ponto flutuante 27

2.5.8 Execução simultânea de múltiplos *kernels* 27

2.6 Top 500 27

3 CUDA **29**

3.1 *Stream Processing* 29

3.2 Arquitetura da GPU 30

3.2.1 NVIDIA GeForce 8 31

3.3 Arquitetura CUDA 34
3.4 Modelo de Programação CUDA 35
3.4.1 *Kernels* e hierarquia de *threads* 36
3.4.2 Hierarquia de memória 38
3.4.3 API 41
3.4.4 *Compute Capability* 44
3.4.5 Exemplo 45

4 Desempenho e otimização **48**
4.1 Partição dinâmica de recursos 48
4.2 Registradores 48
4.3 Blocos de *threads* 49
4.4 Ocupação do multiprocessador 50

5 Trabalhos relacionados **52**
5.1 Occupancy Calculator 52
5.2 Compute Visual Profiler 53
5.3 NVIDIA Parallel Nsight™ 54
5.4 CUDA Profiling Tools Interface - CUPTI 55
5.4.1 CUPTI Callback API 55
5.4.2 CUPTI Event API 55
5.5 PAPI CUDA Component 56
5.6 Vampir/VampirTrace 56
5.7 TAU Performance System 57

6 Descrição do *plug-in* desenvolvido **63**
6.1 Eclipse 63
6.1.1 *Plug-ins* 63
6.2 JNI 66
6.3 Funcionalidades 66
6.3.1 Entrada de dados 67

6.3.2 *Capability characteristics* 68
6.3.3 *Allocation per thread block* 68
6.3.4 *Maximum thread blocks per multiprocessor* 69
6.3.5 *GPU occupancy* 70
6.3.6 Informações adicionais da GPU 70

7 Conclusão **73**

Lista de Figuras

1 *Pipeline* Gráfico programável [42] 16
2 CPU vs. GPU - operações de ponto flutuante [3] 19
3 CPU vs. GPU - largura de banda [3] 20
4 Arquitetura Fermi [57] 23
5 Arquitetura do multiprocessador na Fermi [57] 24
6 Alocação de transistores na CPU e GPU [3] 30
7 *Shaders* separados e o *Unified Shader* [33] 31
8 Arquitetura da NVIDIA G80 32
9 Tipos de memória do multiprocessador [3] 33
10 Panorama do CUDA [5] 34
11 *Grid* e bloco de *threads* [3] 37
12 Código sequencial é executado no *host* e código paralelo no *device* [3] 39
13 Tipos de memória [4] 40
14 Memória e *threads* [3] 42
15 Memória global e *grids* [3] 42
16 CUDA Occupancy Calculator 59
17 Visual Profiler 60
18 Visual Profiler - detalhamento de um *kernel* 61
19 Parallel Nsight 62
20 Eclipse SDK [53] 65
21 *Plug-in* - visão geral 66
22 Entrada de dados 67
23 *Capability characteristics* 68
24 *Allocation per thread block* 68
25 *Maximum thread blocks per multiprocessor* 69
26 *GPU occupancy* 70
27 Informações adicionais da GPU 71
28 *Plug-in - warnings* 72

1 Introdução

O crescimento de poder computacional deixou de se basear no aumento da velocidade dos processadores para dar lugar ao acréscimo de núcleos. O desenvolvimento de CPUs (*Central Processing Unit*), tendo atingido limites físicos e de consumo energético, passou a tirar proveito da computação paralela ao deixar de ter apenas um núcleo.

A GPU (*Graphic Processing Unit*), por outro lado, desde o princípio possui suporte ao processamento paralelo, em virtude de originalmente ser concebida para aplicações gráficas. Tal tipo de aplicação exige que milhões de cálculos sejam realizados, sobre milhares de dados independentes, ao mesmo tempo, formando uma classe particular de aplicações notoriamente diferente em comparação com a CPU, tanto em arquitetura quanto em modelo de programação.

O aumento da demanda por GPUs cada vez mais eficientes e flexíveis deu início à uma nova abordagem, a *GPU Computing* (também chamada GPGPU - *General-Purpose computation on GPU*), conceito que visa explorar as vantagens das placas gráficas modernas com aplicações de propósito geral altamente paralelizáveis que exigem intenso fluxo de cálculos. Com seu crescente potencial no futuro dos sistemas computacionais, várias aplicações com particularidades semelhantes tem sido identificadas e mapeadas com sucesso na execução em GPUs [1]

Embora disponível desde 2002, foi somente em 2007, com o lançamento oficial da arquitetura CUDA, que a *GPU Computing* ganhou notoriedade e passou a ser largamente utilizada na computação de alto desempenho. Acrônimo para *Compute Unified Device Architecture*, o CUDA é um modelo de programação paralela para uso geral de GPUs criado pela NVIDIA e inicialmente disponível nas suas placas da série GeForce 8. Estas modernas

GPUs são formadas por unidades de processamento chamadas multiprocessadores, com registradores e memória compartilhada próprios, capazes de executar milhares de *threads*.

Para que o potencial máximo da programação em GPUs seja atingido, algumas práticas devem ser seguidas. Uma das recomendações consiste em manter o *hardware* o mais ocupado possível, utilizando uma métrica chamada ocupação. A proposta deste trabalho é estudar e desenvolver um *plug-in* para o Eclipse que auxilie o programador na escolha da melhor ocupação para determinada configuração além de indicar possíveis pontos de falha.

O texto a seguir está organizado da seguinte forma:

- Capítulo 2 - contém um breve histórico das placas gráficas além de mostrar como se deu a transição das GPUs tradicionais para as programáveis; apresentação da mais recente arquitetura da NVIDIA: Fermi;
- Capítulo 3 - visão geral de CUDA;
- Capítulo 4 - aspectos relevantes em relação a desempenho e otimização em CUDA;
- Capítulo 5 - trabalhos relacionados - descrição de algumas ferramentas de análise de desempenho (*profiling*) de programas CUDA disponíveis;
- Capítulo 6 - descrição do *plug-in* desenvolvido;
- Capítulo 7 - conclusão do trabalho.

2 *GPU Computing*

2.1 Breve Histórico dos Processadores Gráficos

Nos últimos anos, o *hardware* especializado em computação gráfica tem avançado a níveis sem precedentes. Foi em 1970 que surgiram os primeiros aceleradores gráficos 2D, disponíveis na família Atari. De 1995 a 1998, tem-se a época da aceleração 3D (pré-GPU) que, apesar de não ser acessível a grande parte dos consumidores devido aos preços altos, tiveram um importante papel na história do desenvolvimento das GPUs, fornecendo a base para transformações de vértices e mapeamento de textura.

Diferentemente dos aceleradores gráficos, a chegada da GPU trouxe o conceito de processamento gráfico. Com o passar dos anos, deixaram de ter funções de processamento fixas para dar lugar às GPUs programáveis, sendo possível aproveitar o enorme potencial de computação paralela que vem embutido e que melhora exponencialmente a cada novo lançamento. Os próximos tópicos descrevem as fases de evolução destas placas até o momento.

2.1.1 GPU - Primeira Geração

Em 1999, após vários fabricantes fracassarem com seus produtos, três grandes competidores se consolidaram no mercado: ATI, NVIDIA e 3Dfx. A ATI lançou a última revisão da família Rage, o Rage 128 (e suas várias versões), compatível com os recursos Direct3D, trazendo todos os principais recursos de seus antecessores, com melhorias de processamento e acrescentando recursos voltados à reprodução de DVDs [24]. A RIVA TNT2 (NV5), da NVIDIA, era muito semelhante a sua antecessora (RIVA TNT), mas contava com um processo de fabricação mais avançado, possibilitando atingir velocidade de processamento superior. A Voodoo3, da 3Dfx, combinava o melhor que a empresa tinha a oferecer em uma única placa e dividia com a RIVA TNT2 o posto de melhor placa de vídeo.

Na primeira geração, a placa de vídeo servia apenas como dispositivo que enviava dados para o monitor, ou seja, nenhum processamento era feito internamente e não havia a possibilidade de desenvolver programas nelas.

2.1.2 GPU - Segunda Geração

Em 31 de agosto de 1999, a NVIDIA lança a GeForce 256 (NV10) e a proclama como a primeira GPU do mundo. Ela se destacou por introduzir o conceito de *pipeline gráfico* no qual a GPU é responsável não apenas pela rasterização e texturização dos polígonos enviados, mas também pela transformação dos vértices do polígono e iluminação dos mesmos. Com isso, passou-se a enviar polígonos não mais nas coordenadas da tela, mas nas chamadas coordenadas de mundo [25]. Além disso, possibilitou um avanço considerável no desempenho de jogos e foi o primeiro acelerador gráfico compatível com o padrão Direct3D 7. O grande sucesso da GeForce 256 projetou a NVIDIA como líder de mercado e contribuiu para a queda da 3Dfx.

O atraso no lançamento dos sucessores do Voodoo3 (Voodoo4 e Voodoo5) e a acirrada concorrência foram decisivos para a decadência da 3Dfx. Ainda assim, a empresa forneceu uma grande contribuição para o segmento: o SLI, no qual duas placas de vídeo são interligadas permitindo o processamento paralelo em GPUs e oferecendo melhora considerável no desempenho. Tal tecnologia ainda é utilizada nos melhores modelos de placa da atualidade pela NVIDIA (SLI) e ATI (Crossfire). A 3Dfx foi adquirida pela NVIDIA em dezembro de 2000.

Em 2000, a ATI reage ao lançamento da NVIDIA e lança a Radeon R100 em duas versões (com e sem entrada/saída de vídeo), ambas superiores à GeForce 256. A NVIDIA, em resposta, coloca sua linha GeForce 2 GTS (NV15, NV16) no mercado, superando a rival. Com este lançamento, a empresa

resolveu entrar em vários segmentos, tendo as versões de baixo custo (GeForce 2 MX - NV11) - que se tornaram muito populares - , as intermediárias (GeForce 2 GTS, GeForce 2 Pro e GeForce 2 Ti) e a de alto desempenho (GeForce 2 Ultra). A reação da ATI foi rápida: o R100 foi rebatizado como Radeon 7200, sendo um concorrente direto da GeForce 2 Ti, e o Radeon 7000 (RV100) foi lançado para concorrer com o GeForce 2 MX. Inicia-se assim, uma acirrada disputa entre as duas empresas que dura até os dias atuais.

Nesta segunda geração de GPUs, muitas funcionalidades que antes eram feitas pela CPU passaram a ser feitas diretamente no hardware da GPU, principalmente transformações geométricas e iluminação. Tal fato tornou a velocidade de renderização ainda maior além de aliviar o processamento da CPU, visto que objetos e operações seriam apenas transportados para a GPU, que se encarregaria do cálculo pesado [18]. Dessa forma, houve uma considerável melhora nas aplicações gráficas em tempo real. Embora as possiblidades dos programadores para combinar texturas e colorir os pixels tivessem aumentado em relação à geração anterior, ainda eram limitadas. As placas eram mais configuráveis, mas não verdadeiramente programáveis [28].

2.1.3 GPU - Terceira Geração

Em 2001, a NVIDIA lança a GeForce 3 (NV20), representando um grande salto para a indústria dos jogos, não só pelo desempenho mas também pela nova proposta que trazia, sendo a primeira GPU programável do mercado [18]. A placa era compatível com o novo Direct3D 8 (que não durou muito e logo foi substituído pelo 8.1) e serviu de base para o NV2A, utilizado no video game Xbox, da Microsoft. Poucos meses depois a ATI lança o Radeon 8500 (R200), superando a linha GeForce 3 em até 20%, além de ser totalmente compatível com o Direct3D 8.1.

Em 2002 a NVIDIA lança a linha GeForce 4 (NV25), tendo sua versão

de alto desempenho (GeForce 4 Ti) fracassado na competição com a Radeon 8500. Apesar disso, devido a boa relação entre custo e desempenho em relação aos modelos maiores, a GeForce 4 Ti4200 tornou-se muito popular na época. A versão de baixo custo (GeForce 4 MX - NV17) foi muito criticada pela crítica especializada por se tratar, na verdade, de uma atualização da geração NV15 e sem os recursos na NV20, o que não impediu ser sucesso de vendas graças ao razoável desempenho por preços baixos. Alguns meses depois, uma nova revisão (NV28) é lançada, conseguindo, finalmente, superar a rival.

Nesta terceira geração começa a ser possível implementar aplicações que executem diretamente na GPU ao incorporar técnicas de *pixel shading*, onde cada pixel poderia ser processado por um pequeno programa que incluía texturas adicionais (sendo feito de forma similar nos vértices geométricos), antes de serem projetados na tela [17]. As funções aplicadas nos vértices (*vertex shading*) eram limitadas por 128 instruções e aceitavam até 96 parâmetros. A nível de pixel, as limitações estavam na forma como os dados de textura eram acessados e disponibilizados, além de ter suporte apenas para variáveis de ponto fixo.

Não havia uma linguagem de programação específica, sendo necessário utilizar a linguagem de montagem da GPU (assembly), o que dificultava o desenvolvimento. Mas, apesar de limitado, começa-se a vislumbrar a GPU como hardware programável e, ainda, com características que a aproximaria de uma máquina vetorial ou de processamento paralelo [18].

2.1.4 GPU - Quarta Geração

No fim de 2002, a ATI lança o R300, base para vários modelos, dentre os quais Radeon 9600, 9700 e 9800. Com essa linha consegue grande sucesso e, pela primeira vez, supera a rival NVIDIA, que lançou a série FX (NV30).

Em resposta, a NVIDIA lançaria a GeForce 6, se tornando uma das mais bem sucedidas placas da história. Após este fato, houve um empate com os novos lançamentos da ATI (linha X1***) e NVIDIA (GeForce 7).

A quarta geração representa as placas que suportam milhares de instruções, aceitam variáveis de ponto flutuante, praticamente não possuem mais limitações na utilização dos dados de textura, possibilitando implementações de propósito geral.

2.1.5 GPU - Quinta Geração

No fim de 2006 a NVIDIA lança sua nova linha de GPUs, a série GeForce 8, primeira a suportar a nova arquitetura de programação paralela (lançada em 2007 também pela NVIDIA): CUDA. Com este lançamento, a computação de alto desempenho em GPUs começou a ganhar destaque e interesse não apenas de pela comunidade científica, mas também mercado comercial.

A quinta geração é formada pelas placas que suportam arquiteturas bem estruturadas para aproveitar o poder de computação paralela das GPUs. Há ainda a tentativa da AMD de explorar este promissor mercado com sua ATI Stream, além do surgimento recente de uma abordagem aberta e compatível com ambas tecnologias: o OpenCL. Apesar disso, o CUDA e, consequentemente a NVIDIA, possuem maior visibilidade atualmente, fato pelo qual esta arquitetura foi escolhida como objeto de estudo deste trabalho.

2.2 GPGPU

Na 4ª geração de GPUs, foi introduzida a possibilidade de criar programas através de APIs (HLSL, GLSL, Cg) que rodassem diretamente na placa gráfica, resultando em melhora de desempenho e maior flexibilidade. Porém, apesar deste grande avanço, ainda existiam muitas dificuldades na aborda-

gem do GPGPU: o modelo de programação era improvisado e malfeito, a codificação era complexa e exigia conhecimento profundo da API e da arquitetura da placa, além dos *shaders* programáveis não serem ideais para computação de propósito geral.

A seguir é mostrada a principal mudança ocorrida nesta geração que tornou possível o surgimento da GPGPU, um passo intermediário ao *GPU Computing*.

2.2.1 *Pipeline* Gráfico

O pipeline gráfico consiste em uma representação conceitual das etapas pelas quais passam os dados processados pela GPU, desenvolvido para manter alta frequência de computação através de execuções paralelas.

Os dados de entrada de um pipeline geralmente são vértices com seus respectivos atributos, como posição no espaço, cor e textura. Em cada etapa da transformação, tais vértices são submetidos a cálculos matemáticos, sendo posteriormente mapeados na tela de um dispositivo a fim de gerar a imagem.

***Pipeline* Gráfico convencional** .

O *pipeline* gráfico convencional é composto por vários estágios, executados através de parâmetros definidos por uma API associada. São eles [1]:

- Operações com vértices: cada vértice deve ser mapeado na tela e transformado levando em consideração sua interação com a luz da cena. Esta etapa é altamente paralelizável pois cada vértice (dentre os milhares que compõem uma cena) pode ser calculado de forma independente.
- Montagem de primitivas: os vértices são transformados em triângulos, a primitiva fundamental de uma GPU moderna.

- Rasterização: determinação de quais pixels serão cobertos por cada triângulo.
- Operações com fragmentos: os vértices são combinados dando origem aos fragmentos, cuja cor final é calculada nesta etapa, de forma paralela. Costuma ser o passo de maior custo computacional.
- Composição: os fragmentos são transformados na imagem final.

***Pipeline* Gráfico programável** .

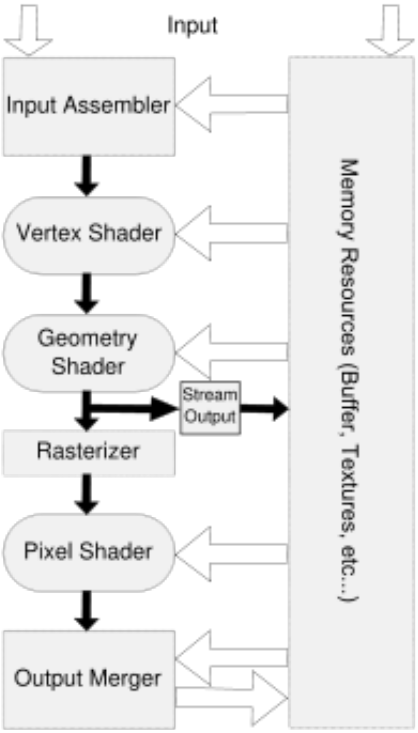


Figura 1: *Pipeline* Gráfico programável [42]

As funções fixas do *pipeline*, definidas na fabricação das placas, passaram a ser entraves na busca por efeitos mais complexos. Visando mais flexi-

bilidade, os estágios de manipulação de vértices e fragmentos se tornaram programáveis. Além disso, um novo estágio foi adicionado ao pipeline: o *geometry shader*.

- *Vertex Shader*: função gráfica que adiciona efeitos em objetos presentes em um ambiente 3D, através da manipulação de propriedades como posição, cor e textura.
- *Geometry Shader*: responsável pela renderização de uma cena, gerando novas primitivas gráficas (como pontos, linhas e triângulos) a partir dos dados recebidos pelo *vertex shader*.
- *Fragment Shader*: também conhecido por *Pixel Shader*, é responsável por adicionar efeitos de luz e cor aos pixels de uma imagem 3D.

2.2.2 *Shaders* Gráficos

Em computação gráfica, *shaders* são programas executados na GPU ou em qualquer outro processador compatível. Sua utilização marcou a transição dos *pipelines* fixos para os programáveis, permitindo maior flexibilidade e novos efeitos gráficos [30].

2.3 *GPU Computing*

Alguns autores ainda utilizam o termo GPGPU para designar o uso de GPUs em programas de propósito geral (que não seja gráfico). Apesar disso, o termo mais usual é o *GPU Computing*, caracterizando as GPUs totalmente programáveis da última geração.

2.4 GPU vs. CPU

Movido pela insaciável demanda por aplicações em tempo real e gráficos 3D de alta definição, a GPU evoluiu para um processador com vários núcleos altamente paralelo, com um gigantesco poder de computação e grande largura

de barramento de memória [39]. Apesar da CPU também estar em constante evolução, o poder de cálculo dos processadores gráficos obteve uma melhora substancialmente melhor.

A melhora de desempenho da CPU está limitada por problemas físicos e consumo de energia. Atualmente, o desenvolvimento de novos processadores concentra-se na adição de núcleos ao invés de melhorar o desempenho de apenas um. Paralelismo é o futuro da computação [1]. Mas enquanto as CPUs modernas possuem no máximo 4 ou 8 núcleos, uma GPU é formada por centenas; desde o princípio são especializadas em tarefas de computação intensiva e altamente paralelas.

O gráfico da figura 2 compara o desempenho em operações de ponto flutuante por segundo de GPUs NVIDIA e CPUs Intel. Na figura 3 é feita uma comparação de largura de banda (taxa de transferência de dados).

Por trás desta comparação existem diferentes abordagens e desafios:

- Objetivo: o núcleo da CPU foi desenvolvido para executar, à velocidade máxima, uma única thread composta de instruções sequenciais. Já a GPU executa, paralelamente, milhares de threads o mais rápido possível.
- Transistores: a CPU utiliza seus transistores com o intuito de melhorar o desempenho de execução em uma sequência de tarefas. A GPU é especializada em executar milhares de instruções paralelas. A maior parte de seus transistores trabalha no processamento do dado em si ao invés de ocupar-se no controle de fluxo.
- *Cache*: a CPU possui grandes caches, cujo acesso é feito de forma aleatória. Tais estruturas são responsáveis por acelerar a execução

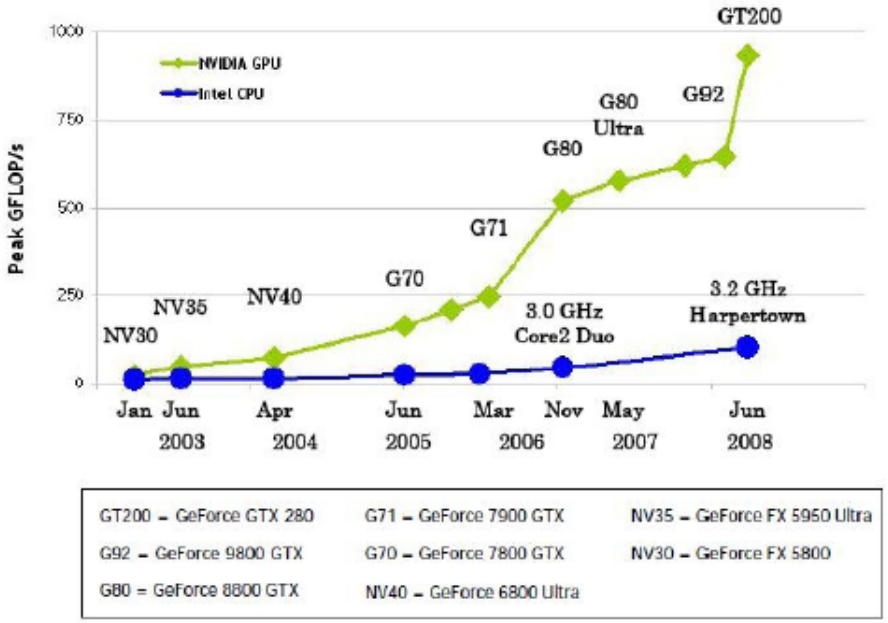


Figura 2: CPU vs. GPU - operações de ponto flutuante [3]

e alguns poucos comandos e reduzir a latência de memória, embora consumam muita energia.

Na GPU encontramos controladores de memória para vários canais e um *cache* pequeno e mais rápido, possibilitando maior largura de banda e o processamento de milhares de *threads* simultaneamente. O acesso à essas unidades é previsível: tanto a leitura quanto a escrita é feita sequencialmente, ou seja, o vizinho do dado lido/escrito, será o próximo na fila de processamento.

- *Multithreading*: a CPU é capaz de executar 1 ou 2 *threads* por núcleo, e a mudança de uma *thread* para outra custa centenas de ciclos, enquanto que a GPU pode manter até 1024 por multiprocessador (há vários em uma GPU) além de tipicamente fazer trocas de *thread* a cada ciclo.

Abaixo uma tabela comparativa entre um modelo de CPU e outro de

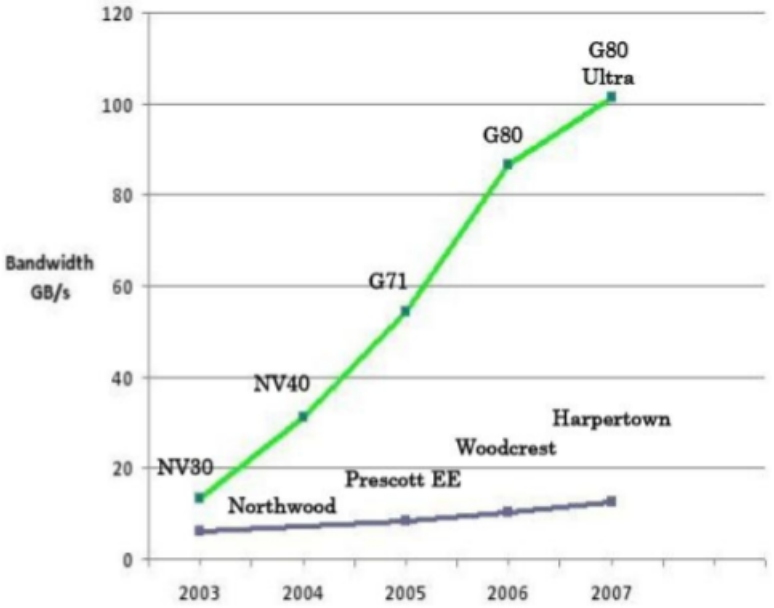


Figura 3: CPU vs. GPU - largura de banda [3]

GPU (dados de 2008):

	Intel Core 2 Quad 3.0 Ghz	NVIDIA GeForce 8800 GTX
Poder computacional	96 GFlops	330 GFlops
Largura de banda	21 GB/s	55.2 GB/s
Crescimento anual	1.4x	1.7x (*fragment shader*) 2.3x (*vertex shader*)
Preço	US$ 1100	US$ 550

Mas para que os objetivos sejam alcançados, novos desafios no processo de desenvolvimento dos sistemas, escalonamento de trabalho e gerenciamento de recursos devem ser bem analisados:

- Volume de dados e processamento: aplicações gráficas em tempo real trabalham com bilhões de *pixels* por segundo e cada *pixel* necessita de centenas de operações aritméticas. Com isso, a GPU deve corresponder com alto desempenho para satisfazer a grande demanda que estas aplicações exigem.

- Paralelismo: a independência permite que pacotes múltiplos de dados sejam processados em uma única solicitação.

- Vazão sobre latência: execução de trabalho útil enquanto espera pelo retorno de operações mais lentas.

2.5 Evolução das GPUs e a arquitetura Fermi

A demanda por gráficos mais rápidos e de alta definição continua a exigir o desenvolvimento de GPUs altamente paralelizáveis. A tabela abaixo mostra os marcos significantes na evolução das placas gráficas da NVIDIA. O número de transistores aumentou exponencialmente, dobrando a cada 18 meses; desde sua introdução em 2006, o número de núcleos CUDA também dobrou a cada 18 meses [57].

Evolução da tecnologia das GPUs NVIDIA

Data	Produto	Transistores	núcleos CUDA	Tecnologia
1997	RIVA 128	3 milhões	-	aceleradores gráficos 3D
1999	GeForce 256	25 milhões	-	primeira GPU
2001	GeForce 3	60 milhões	-	primeiro *shader* GPU programável
2002	GeForce FX	125 milhões	-	GPU programável com ponto flutuante 32-bit
2004	GeForce 6800	222 milhões	-	GPGPU
2006	GeForce 8800	681 milhões	128	primeira placa com CUDA
2008	GeForce GTX 280	1.4 bilhões	240	CUDA C e OpenCL
2009	Fermi	3 bilhões	512	arquitetura de computação GPU, endereçamento 64-bit, *caching*, CUDA C, OpenCL

A mais recente geração de GPUs da NVIDIA, com a arquitetura Fermi, introduz diversas funcionalidades que entregam melhor desempenho, melhora a forma de programação empregada e amplia a variedade de aplicações que podem se beneficiar dela. Baseada nas experiências de usuários das gerações anteriores, é dirigida às mais variadas áreas, com o objetivo de tornar a computação em GPU (*GPU computing*) mais ampla [57] e aumentar sua uti-

lização.

A figura 4 mostra a arquitetura Fermi, com seus 512 núcleos CUDA, organizados em 16 multiprocessadores (cada um com 32 núcleos CUDA), compartilhando um *cache* (L2), seis interfaces DRAM de 64-bits e uma interface para comunicação com o a CPU (*host*). O escalonador GigaThread distribui os blocos de *threads* para os multiprocessadores (SM) disponíveis, fazendo balanceamento de trabalho na GPU e executando múltiplos *kernels* em paralelo quando apropriado [57]. Cada multiprocessador executa até 1536 *threads* concorrentemente para esconder a latência no acesso à memória DRAM. A medida que cada bloco completa sua tarefa e libera os recursos, o escalonador designa um novo bloco de *thread* para ocupar o multiprocessador livre.

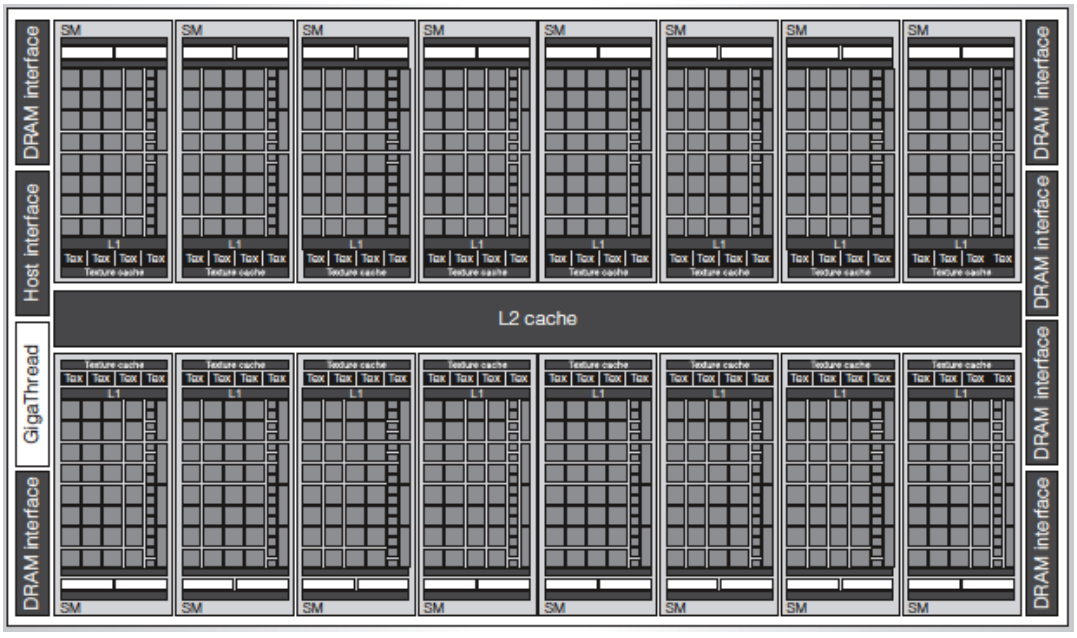


Figura 4: Arquitetura Fermi [57]

A figura 5 mostra em detalhes a arquitetura de um multiprocessador na Fermi com seus 32 núcleos CUDA, memória compartilhada e *cache* (L1).

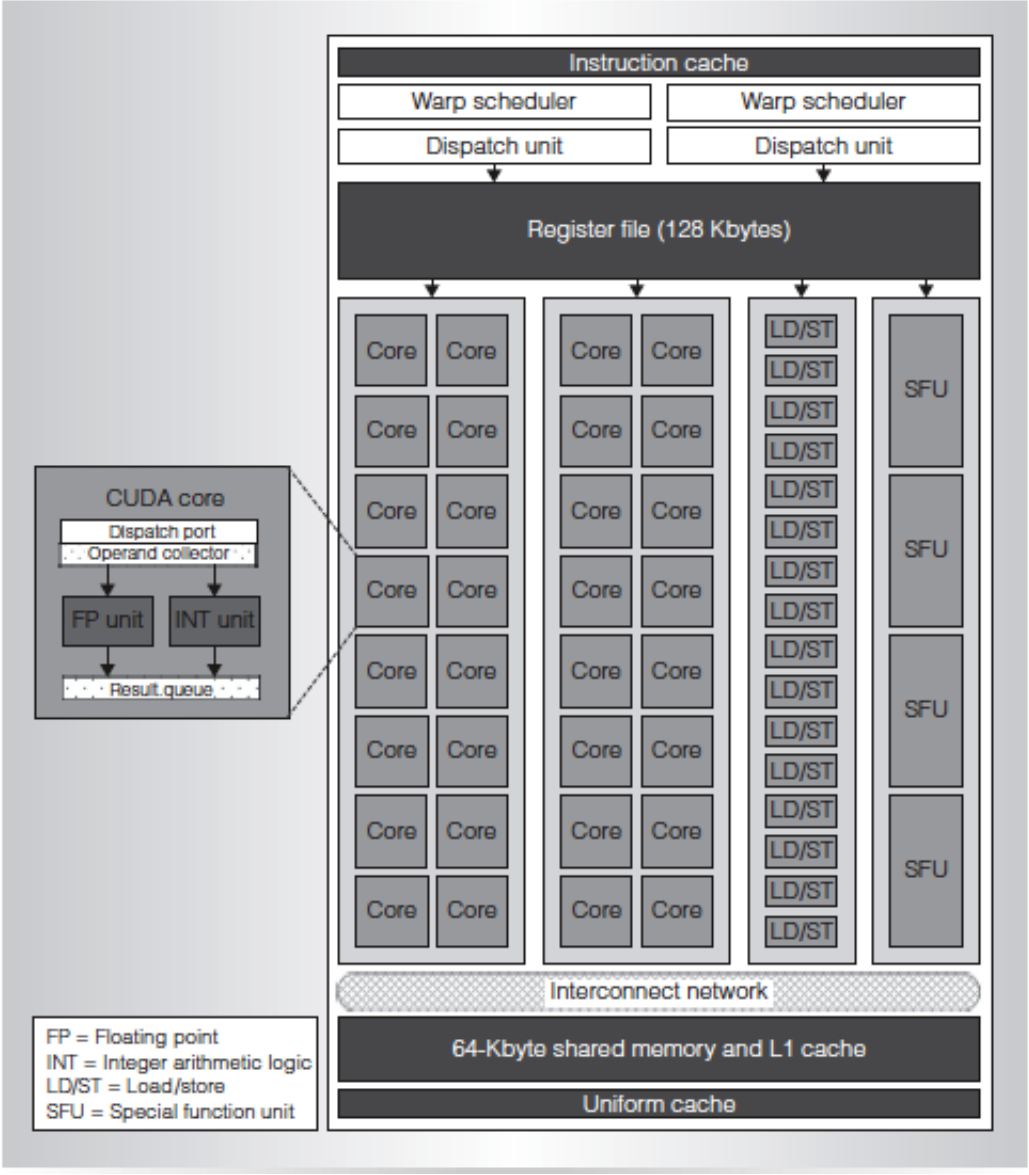


Figura 5: Arquitetura do multiprocessador na Fermi [57]

Várias melhorias foram feitas nesta nova arquitetura e algumas das mais significativas são descritas nos tópicos seguintes.

2.5.1 Endereçamento virtual de 64-bits

As GPUs tradicionalmente utilizam endereçamento virtual de 32-bits, que limita o tamanho da memória (DRAM) da placa em 4 *gigabytes*, sendo mais que suficiente para as aplicações gráficas. As CPUs, por outro lado, possuem endereçamento de 64-bits há anos.

A arquitetura Fermi adota o mesmo estilo da CPU possibilitando que a GPU incorpore mais que 4 *gigabytes* de memória virtual. O principal benefício é a possibilidade dos *kernels* CUDA operarem grandes conjuntos de dados, além de ser possível mapear as memórias físicas da CPU e GPU em um espaço virtual compartilhado de endereços, permitindo que todas as variáveis da aplicação possuam um endereço único.

2.5.2 *Caches*

Um dos principais argumentos para o tamanho limitado do *cache* (por ser um recurso caro) na GPU é utilizar a paralelização com suas milhares de *threads*, compensando a latência no acesso à memória (DRAM). Tal fato, porém, é verdadeiro apenas para uma parcela de programas, não sendo suficiente para outros que não possuem paralelismo de dados suficiente ou que possuem padrões imprevisíveis de reuso de dados. A Fermi possui 64KB de memória *cache* privativo por multiprocessador (L1) e 768KB para uso compartilhado (L2).

2.5.3 ECC - *Error Correcting Codes*

Foi introduzido a proteção de memória ECC, melhorando a integridade dos dados em sistemas com grande número de GPUs. Ele permite integrar mi-

lhares de GPUs em um sistema enquanto mantém tempo médio alto entre falhas.

2.5.4 Instruções atômicas mais rápidas

As GPUs historicamente possuem baixo suporte para ajudar na cooperação de tarefas paralelas. O desafio do *hardware* é fazer uma primitiva que permita fazer essa sincronização de forma segura sem que prejudique o desempenho. A solução clássica para esse problema consiste nas instruções atômicas, que lêem de um local compartilhado, verificam seu valor e escrevem um novo valor sem que outro processo seja capaz de alterá-lo durante o procedimento [57]. Na Fermi, as instruções atômicas podem ser de 5 a 20 vezes mais rápidas que na sua antecessora G80, reduzindo a necessidade de envolver a CPU onde blocos de *threads* atualizam grandes estruturas de dados compartilhadas, diminuindo pressão na comunicação com a GPU [7].

2.5.5 Espaço de memória unificado

No tradicional modelo CUDA, cada tipo de memória (constante, compartilhada, local e global) possui seu próprio endereçamento de espaço. O programador podia usar ponteiros apenas na memória global. Na Fermi, todas as memórias são parte de um espaço unificado, permitindo que linguagens baseadas em ponteiros, como C e C++, sejam mais fáceis de rodar e executar na GPU, aumentando o número de programas beneficiados com a tecnologia [59].

2.5.6 Suporte a depuração de código no *kernel*

Ao contrário de seus antecessores, a arquitetura Fermi permite tratamento de exceções e chamadas de funções de suporte, como printf(), fornecendo um importante suporte no desenvolvimento de *softwares* que executam em GPUs.

2.5.7 Aritmetica de ponto flutuante

As GPUs da geração anterior eram capazes de executar aritmeticas com precisão dupla no *hardware* com significante redução de velocidade (por volta de 8 vezes mais devagar) comparado com a de precisão simples. Já na arquitetura Fermi, a mesma operação é feita na metade do tempo da precisão simples, assim como ocorre nos principais processadores disponíveis. Desta forma, os programadores podem comparar os *tradeoffs* entre estes dois tipos do mesmo jeito que fariam nos processadores ao qual estão acostumados.

2.5.8 Execução simultânea de múltiplos *kernels*

Versões anteriores de CUDA permitiam que apenas um *kernel* fosse executado por vez em cada GPU. Já a Fermi é capaz de executar vários ao mesmo tempo, evitando que o programador faça grandes *kernels* a fim de melhor aproveitar a placa gráfica.

2.6 Top 500

O projeto Top 500 consiste em um *ranking* dos 500 sistemas de computação mais poderosos a nivel mundial. Iniciado em 1993, publica duas listas por ano, uma em junho, coincidindo com a Internacional Supercomputing Conference, e outra em novembro, no ACM/IEEE Supercomputing Conference. O projeto tem por objetivo prover uma base confiável para detectar tendências na computação de alto desempenho [61]. A medição é feita através de uma implementação do LINPACK Benchmark, com o cálculo de um denso sistema linear de equações.

A NVIDIA apareceu na lista pela primeira vez em 2008 com o TSUBAME, supercomputador construído em parceria com o Tokyo Institute of Technology (Tokyo Tech). Composto por 170 GPUs Tesla S1070, o TSUBAME possui desempenho teórico de 170 teraflops, assim como 77.48 teraflops me-

didos no *benchmark* Linpack, colocando-o na 29ªposição do ranking.

Já em 2011, 3 entre os 5 mais rápidos computadores utilizam GPUs NVIDIA. Todas possuem processadores Intel Xeon mas a maior parte do poder de processamento vem das placas gráficas. O TianHe-1A, construído pelo National University of Defense Technology of China (NUDT), ocupa o segundo lugar (tendo sido o primeiro, na lista anterior) e possui 14336 CPUs Intel Xeon X5670 e 7168 GPUs NVIDIA Tesla M2050, sendo de 4.7 petaflops seu desempenho teórico e 2.5 petaflops medidos no *benchmark* Linpack. A NVIDIA sugere que necessitaria de 50000 CPUs e o dobro de espaço para entregar o mesmo desempenho apenas com CPUs. Além disso, o consumo de energia seria quase o triplo (12 megawatts versus 4.04 megawatts).

O supercomputador está instalado no National Supercomputing Center (Tianjin - China) e é utilizado com sucesso em vários campos como exploração de petróleo, pesquisas biomédicas, exploração de novas fontes de energia, simulações, análise de risco no mercado financeiro etc.

3 CUDA

A NVIDIA, enxergando a oportunidade de crescimento na utilização das GPUs, se empenhou na criação de um modelo de programação e *shaders* melhores, passando a chamá-los de *stream processors*. Não foi apenas um golpe de *marketing*, mas também o surgimento de uma nova arquitetura.

Acrônimo para *Compute Unified Device Architeture*, o CUDA é um modelo de programação paralela para uso geral de GPUs, criado pela NVIDIA em 2007 e inicialmente disponível na série GeForce 8. Foi um avanço da empresa no GPGPU e na computação de alto desempenho, com o benefício da maior facilidade de uso.

Através dele, a GPU é vista como um conjunto de multiprocessadores capaz de executar um grande número de *threads* em paralelo [29]. Utilizando uma extensão da linguagem C como interface de programação, a API esconde a complexidade da placa e permite que os programadores tirem proveito de seu poder de processamento, sem a necessidade de conhecer os detalhes do *hardware*.

3.1 *Stream Processing*

O modelo de programação no qual se baseia o *GPU Computing* em CUDA, é o *Stream Processing* (também conhecido como *Thread Processing*), no qual um conjunto uniforme de dados que podem ser operados em paralelo (*stream*), é processado por várias instruções, chamadas de *kernel*. Este modelo constitui uma forma simples e restrita de paralelismo em que não há necessidade de coordenar a comunicação nem de sincronização explícita, sendo ideal para aplicações com alta demanda aritmética e dados paralelos.

3.2 Arquitetura da GPU

Originalmente desenvolvida para executar aplicações gráficas, a arquitetura da GPU evoluiu seguindo um caminho diferente da CPU. Tais aplicações executam uma mesma operação para uma grande quantidade de dados, fazendo com que as GPUs sejam projetadas para executar várias instruções em paralelo (método SIMD - *Single Instruction, Multiple Data*).

Além disso, a GPU utiliza a maior parte de seus transistores para cálculos, restando poucos para a parte de controle e cache, como ilustrado na figura 6. Isso aumenta o poder de computação, embora torne o fluxo do programa mais limitado.

Figura 6: Alocação de transistores na CPU e GPU [3]

No acesso à memória, a GPU procura maximizar a taxa de transferência de dados (*throughput*) em detrimento à latência. A demora no acesso de um elemento não é tão importante quanto transferir um conjunto de elementos de maneira eficiente [25].

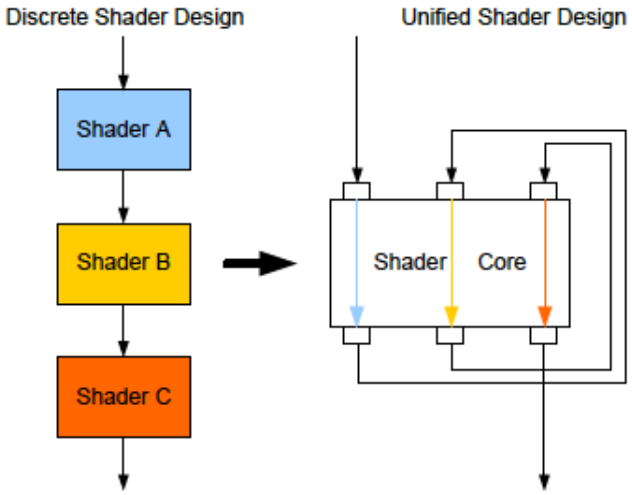


Figura 7: *Shaders* separados e o *Unified Shader* [33]

3.2.1 NVIDIA GeForce 8

No início das placas programáveis, o processador vetorial (*vertex shader*) e o processador de fragmentos (*fragment shader*) atuavam de forma independente, devido às diferenças no conjunto de instruções de ambos. Tal característica causava problemas no balanceamento de carga pois, como em qualquer *pipeline*, o desempenho dependia do estágio mais lento. Com a convergência das características destas etapas, surgiu uma nova estrutura na qual todas as unidades programáveis compartilham o mesmo *hardware*: o *unified shader* (iniciada na GeForce 8800 GTX), que pode ser utilizado dinamicamente como qualquer um dos *shaders* possíveis (figura 7). Agora os programadores podem tirar melhor proveito da nova arquitetura, sem a necessidade de dividir o trabalho entre as unidades de *hardware*.

Além disso, ao contrário das GPUs anteriores, cada *Stream Processor* possui seus próprios registradores e memória, não dependendo mais apenas da memória global da placa, o que resulta em melhora de desempenho.

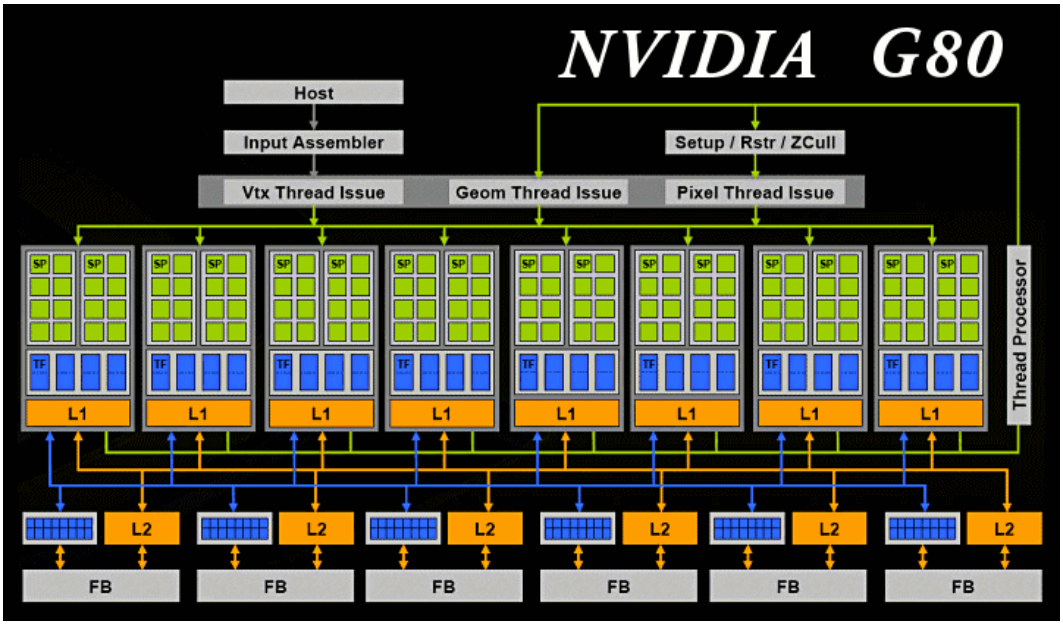


Figura 8: Arquitetura da NVIDIA G80

A figura 8 ilustra a arquitetura da NVIDIA GeForce 8800 GTX, com suas unidades unificadas programáveis altamente paralelas.

Esta GPU possui 128 *Stream Processors* (SP), núcleo capaz de manipular 96 threads e responsável por executar as operações básicas de ponto flutuante [8]. Cada grupo com 8 destes processadores, juntamente com uma memória local compartilhada de 16KB, forma o *Streaming Multiprocessor* (SM). Tal estrutura é responsável por criar, gerenciar e executar as *threads*, utilizando uma nova arquitetura chamada SIMT - *Single Instruction, Multiple Thread*. O SIMT divide as *threads* que recebe em blocos de 32, chamado de *warp*, e distribui para os SMs disponíveis, que executam uma instrução comum por vez.

Cada muliprocessador possui 4 tipos de memória como ilustrado na figura 9:

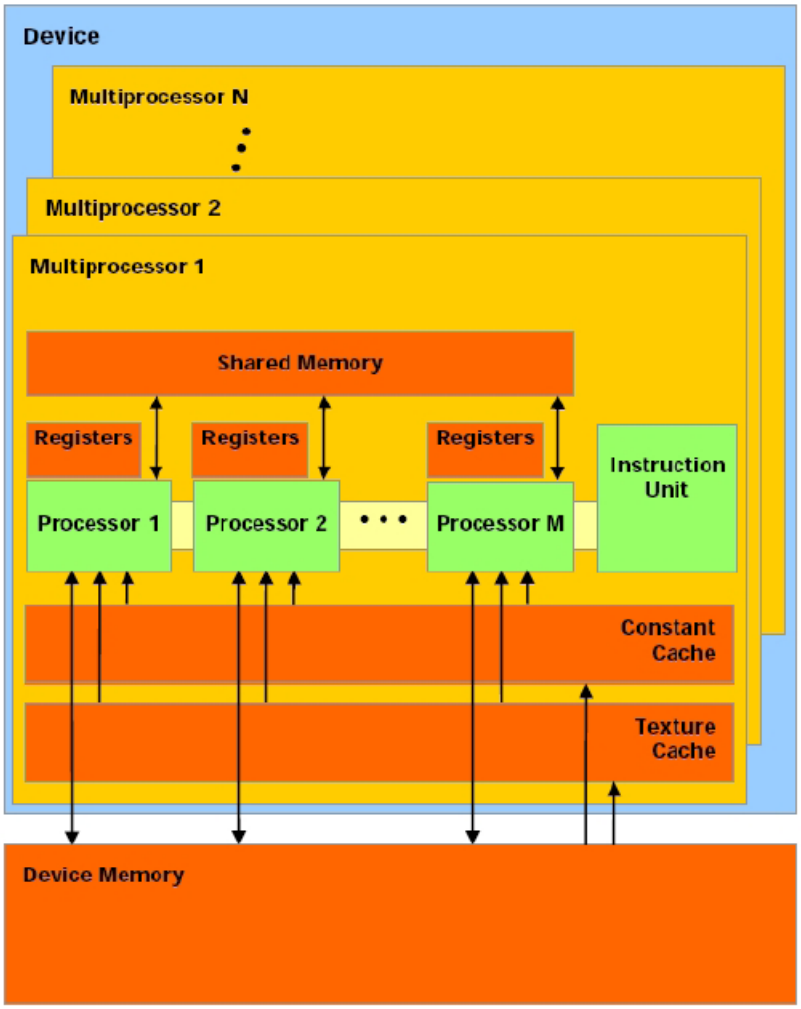


Figura 9: Tipos de memória do multiprocessador [3]

- Um conjunto de registradores 32 *bits* para cada processador.
- Uma memória compartilhada com todos os processadores que compõe o multiprocessador.
- Uma memória *cache*, apenas com permissão de leitura, para otimizar o acesso à memória de constantes (localizada no *device*), compartilhada com todos os processadores do multiprocessador.

- Uma memória *cache*, apenas com permissão de leitura, para otimizar o acesso à memória de texturas (localizada no *device*), compartilhada com todos os processadores do multiprocessador.

3.3 Arquitetura CUDA

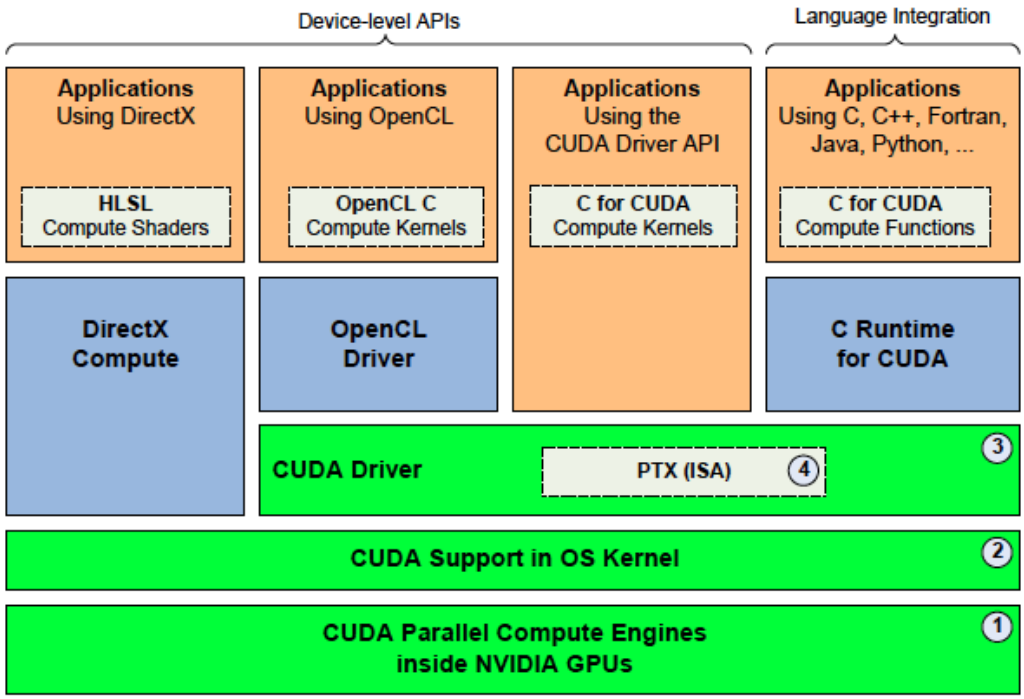


Figura 10: Panorama do CUDA [5]

A revolucionária arquitetura CUDA permite tirar proveito do poder de processamento paralelo das GPUs NVIDIA para aplicações de propósito geral. Como ilustrado na figura 10, ela é composta por vários componentes:

1. Uma GPU NVIDIA capaz de rodar CUDA.

2. Suporte CUDA no nível do *kernel* para inicialização de *hardware*, configuração etc.

3. CUDA *driver* que fornece a API para desenvolvedores.

O ambiente de desenvolvimento do CUDA proporciona todas as ferramentas, exemplos e documentação necessária para que os desenvolvedores criem aplicações GPGPU:

- Bibliotecas otimizadas para uso em CUDA (BLAS, FFT e outras).
- *C Runtime for CUDA* permite a execução de funções em C padrão.
- Ferramentas: compilador (*nvcc*), *debugger* (*cudagdb*) etc.
- Exemplos de código e documentação que mostra boas práticas para uma variedade de algoritmos e aplicações.

Além disso, possui duas interfaces de programação: uma a nível de *device*, no qual é possível utilizar o padrão DirectX (linguagem HLSL), ou OpenCL (ambos independetes da GPU sendo, portanto, aplicáveis a GPUs AMD também); a outra alternativa é utilizar o *C Runtime for CUDA*, uma abordagem de alto nível que requer menos codificação, com um conjunto de extensões para indicar se a instrução será executada na GPU ou na CPU.

3.4 Modelo de Programação CUDA

O modelo de programação do CUDA é altamente escalável, ou seja, está preparado para manipular um volume crescente de trabalho de forma uniforme.

O gerenciamento automático das *threads* reduz de forma significativa a complexidade da programação paralela, permitindo definir explicitamente o paralelismo de determinadas partes do código [30].

Na API criada pela NVIDIA, a GPU é vista como co-processador (*device*) para a CPU (*host*), capaz de executar várias threads em paralelo. A CPU

é utilizada para controlar o fluxo de execução do programa, enquanto que a GPU fica responsável pelos inúmeros cálculos sobre os dados.

Um programa CUDA possui o seguinte fluxo básico [40]:

- O *host* inicializa um vetor com dados;
- O vetor é copiado da memória do *host* para a memória do *device*;
- O *device* realiza cálculos sobre o vetor de dados;
- O vetor contendo os dados modificados é copiado novamente para o *host*.

3.4.1 *Kernels* e hierarquia de *threads*

O princípio básico da programação em CUDA consiste em definir funções chamadas de *kernels* (utilizando uma extensão da linguagem C), que especifica o código a ser executado N vezes em paralelo por N diferentes *threads* na GPU. Estas *threads* são extremamente leves, resultando em baixo custo de criação.

Tal modelo estimula a divisão dos problemas em dois passos:

- primeiro separar em sub-problemas independentes, formando os *grids*
- depois dividir o *grid* em blocos independentes de mesmo tamanho (*thread block*), cujas *threads* podem trabalhar de forma cooperativa e compartilham uma memória visível apenas entre elas

Cada *thread* no bloco possui um identificador único tridimensional (*threadIdx.x*, *threadIdx.y*, *threadIdx.z*) e cada bloco em um *grid* também possui seu próprio identificador bidimensional (*blockIdx.x*, *blockIdx.y*). Apesar disso,

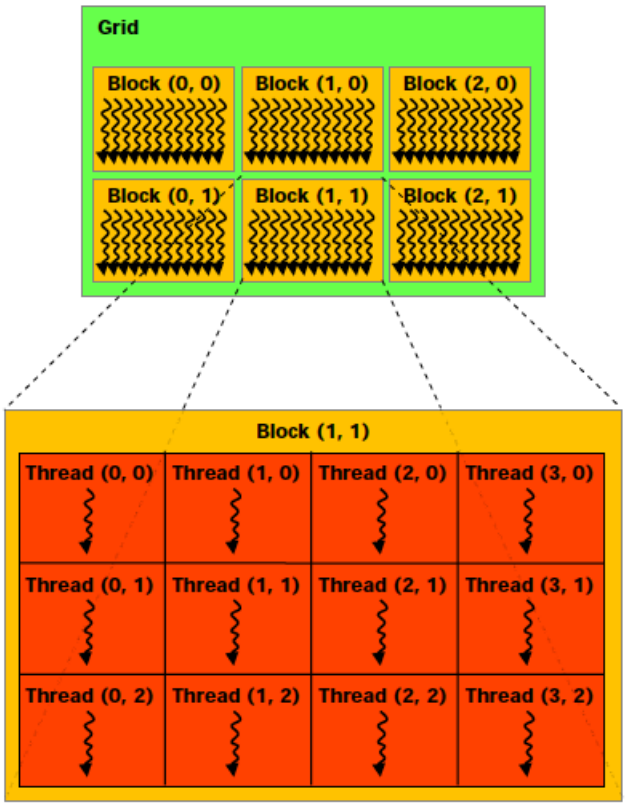


Figura 11: *Grid* e bloco de *threads* [3]

nem todas as aplicações usarão todas as dimensões disponíveis.

No exemplo da figura 11, o *grid* é formado por 6 blocos de *threads* que são organizados em um *array* bidimensional 2x3. Cada bloco possui uma coordenada única, dada pelas palavras-chave *blockId.x* e *blockId.y*. Todos blocos devem ter o mesmo número de *threads* organizadas da mesma forma. Apenas para ilustrar o conceito, o exemplo mostrado possui dimensões reduzidas; na realidade, um *grid* é formado por milhares (ou milhões) de *threads* a cada invocação de um *kernel*. A criação de *threads* em número suficiente para utilizar o *hardware* em sua totalidade, geralmente requer grande paralelismo

de dados.

A figura 12 ilustra a sequência de execução de um programa no qual código sequencial é executado no *host* enquanto que o código paralelo é executado no *device*.

Para criar uma função *kernel*, deve-se utilizar o especificador __global__ e, ao executá-la, especificar a dimensão do *grid* (número de blocos) e a dimensão do bloco (número de *threads*).

Código 1: *Kernel*

```
//funcao kernel
__global__ void nomeDoKernel (lista_de_parametros)
{
  ...
}
```

Código 2: Chamada de um *kernel*

```
nomeDoKernel<<<gridDim, blockDim>>>(lista_de_parametros);
```

3.4.2 Hierarquia de memória

Otimizar o desempenho de aplicações CUDA geralmente envolve otimizações no acesso de dados, que inclui o uso apropriado dos diferentes tipos de memória (figura 13):

- registrador - cada *thread* possui sua própria memória, que é a mais rápida (pois fica dentro do multiprocessador) e possui o mesmo tempo de vida dela;
- compartilhada - cada bloco possui uma memória (localizada dentro do multiprocessador) compartilhada entre todas as *threads* que o compõe, possuindo o mesmo tempo de vida do bloco;

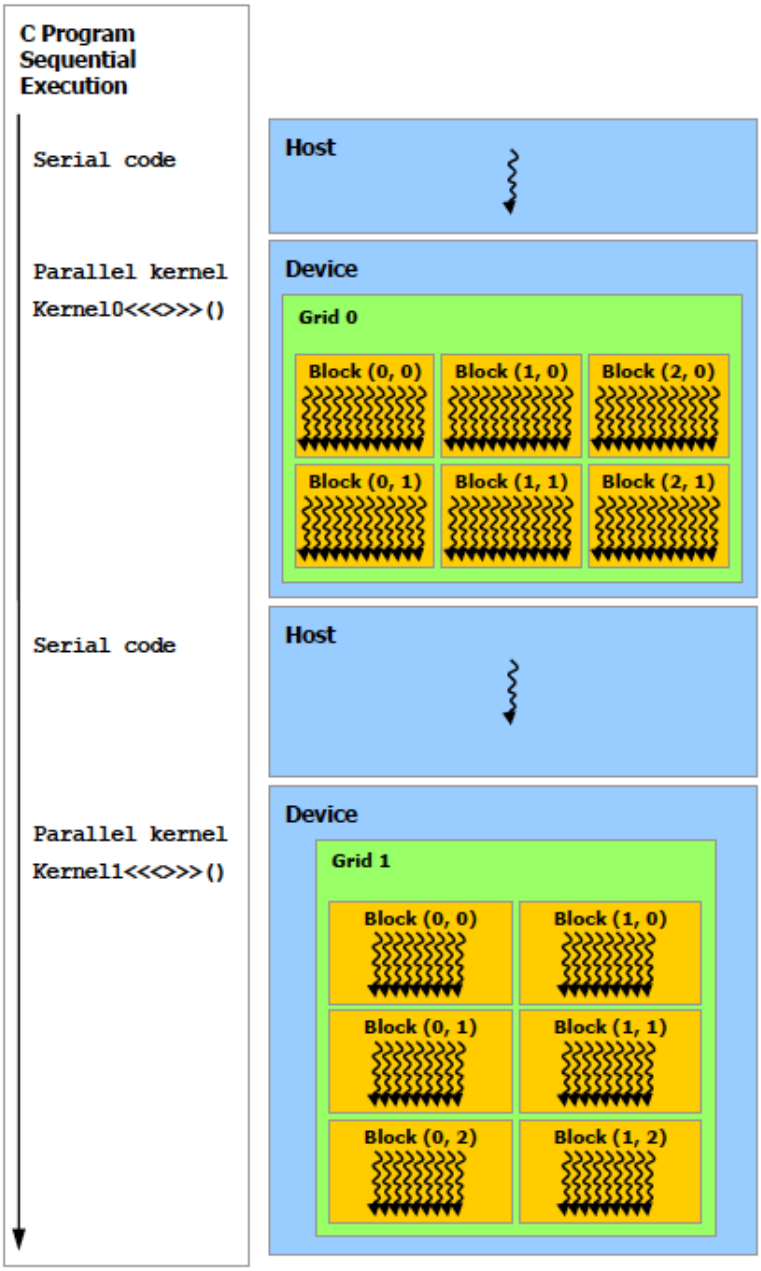


Figura 12: Código sequencial é executado no *host* e código paralelo no *device* [3]

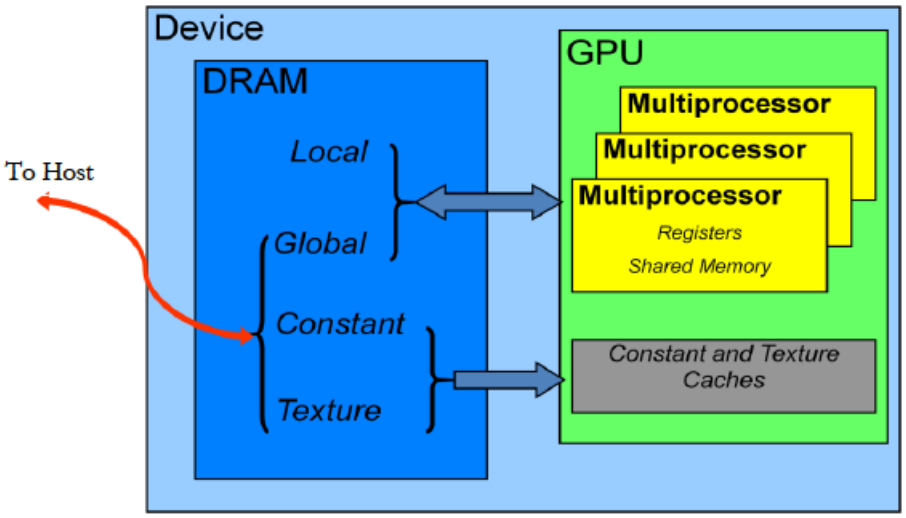


Figura 13: Tipos de memória [4]

- global - todas as *threads* da aplicação tem acesso à mesma memória global, permitindo a comunicação entre blocos de *grids* diferentes. Seu tempo de vida é o mesmo da aplicação e é cerca de 150 vezes mais lenta que o registrador e a memória compartilhada;

- global (apenas para leitura) - memória de constantes e memória de textura, acessível à todas as *threads*, otimizadas para diferentes tipos de utilização. Seu tempo de vida é o mesmo da aplicação;

- local - abstração (não é um componente de *hardware*) de uma memória local no escopo de cada *thread*, que fica localizada na memória global (possuindo, portanto, o mesmo desempenho). É utilizada quando o conjunto de dados locais ultrapassam o limite da memória mais rápida (dentro do multiprocessador) por alguma razão, possuindo o mesmo tempo de vida da *thread*;

- textura - memória que pode ser utilizada como alternativa à memória global, obtendo melhor desempenho em determinados casos;

A tabela abaixo compara as principais características dos diferentes tipos de memória do *device*. *On-chip* é a memória que fica dentro do multiprocessador (logo, possui baixa latência), e *off-chip* é a que fica fora.

Memória	*on/off chip*	Acesso	Escopo	Tempo de vida
Registrador	*on*	leitura/escrita	1 *thread*	*thread*
Local	*off*	leitura/escrita	1 *thread*	*thread*
Compartilhada	*on*	leitura/escrita	todas *threads* do bloco	*bloco*
Global	*off*	leitura/escrita	todas *threads* + *host*	alocação do *host*
Constante	*off*	leitura	todas *threads* + *host*	alocação do *host*
Textura	*off*	leitura	todas *threads* + *host*	alocação do *host*

Levando em consideração que o acesso à memória global (e, consequentemente, à local) demora cerca de 100 - 150 vezes mais que nos registradores e na compartilhada, é fundamental maximizar o uso das memórias do multiprocessador com o objetivo de otimizar o desempenho das aplicações.

3.4.3 API

Através da API criada pela NVIDIA, a complexidade da GPU fica escondida ao fornecer funções que permitem sua operação. Além disso, a flexibilidade torna possível a mudança e evolução do *hardware* sem que *softwares* existentes se tornem obsoletos.

Qualificadores de funções determinam onde a função será executada (*host* ou *device*) e por quem poderá ser chamada:

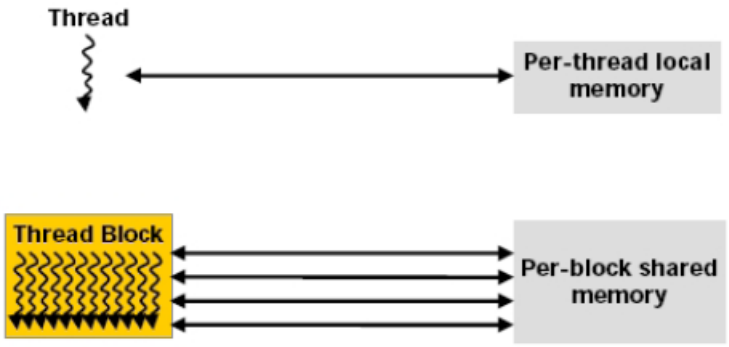


Figura 14: Memória e *threads* [3]

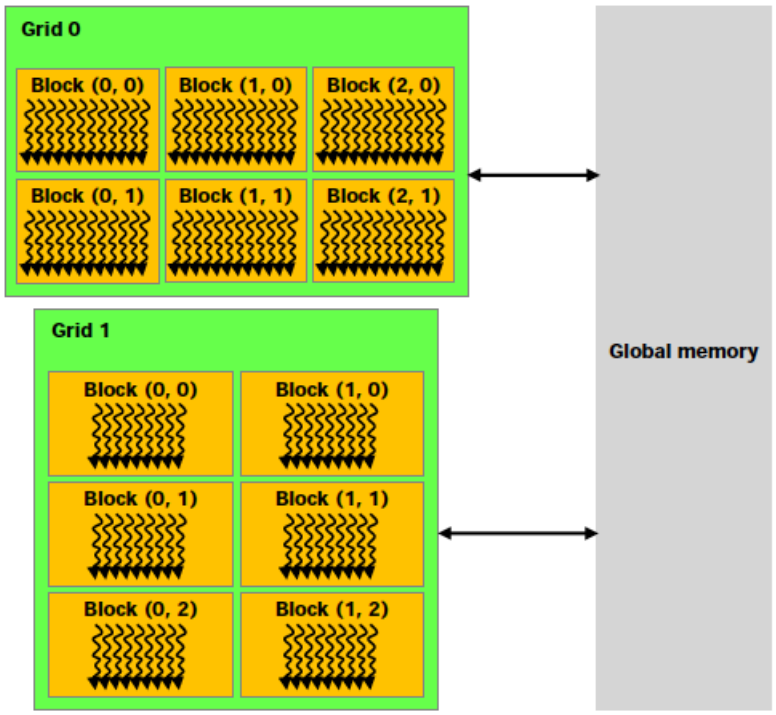


Figura 15: Memória global e *grids* [3]

- __device__ especifica que a função será executada no *device* e somente poderá ser invocada a partir do mesmo.

- __global__ especifica um *kernel*, que será executado no *device* e somente poderá ser invocado a partir do *host*. Obrigatoriamente retorna *void* e deve ter uma configuração de execução (número de blocos e *threads*) quando for chamada, como visto anteriormente.

- __host__ especifica uma função que somente será executada e invocada a partir do *host*.

Por padrão, quando uma função não tiver um especificador, será considerada __global__.

Qualificadores de variáveis determinam em que tipo de memória do *device* (GPU) a variárel será alocada:

- __device__ especifica uma variável que reside no *device*, acessível a todas as *threads* e possui o mesmo tempo de vida da aplicação. Por não ter *cache*, possui alta latência.

- __constant__ especifica uma variável que reside na memória constante e possui o mesmo tempo de vida da aplicação. É acessível por todas as *threads* (apenas com permissão de leitura) de um *grid* e pelo *host* (permissão de leitura e escrita), através da biblioteca do *runtime*.

- __shared__ especifica uma variável que reside na memória de um bloco, tendo o mesmo tempo de vida e somente acessível pelas *threads* que o compõem.

- variáveis sem qualificadores ficam na memória local.

Variáveis *built-in* especificam a dimensão dos *grids* e dos blocos e dos índices dos blocos e *threads*. São válidos apenas dentro de funções executadas no *device*.

- *gridDim* - variável do tipo *dim3* que contém a dimensão do *grid*.

- *blockIdx* - variável do tipo *uint3* que contém o índice do bloco dentro do *grid*.
- *blockDim* - variável do tipo *dim3* que contém a dimensão do bloco
- *threadIdx* - variável do tipo *uint3* que contém o índice da *thread* dentro do bloco.
- *warpSize* - variável do tipo *int* que contém o número de *threads* no *warp*.

dim3 designa um vetor de inteiros (baseado no *uint3*), utilizado para especificar dimensões (*struct dim3*{*int x,y,z;*} - valor padrão (1,1,1)).

3.4.4 *Compute Capability*

Os principais recursos disponíveis em cada GPU são expostos através de um sistema padrão chamado *compute capability*, formado por um *major number* e um *minor number*. Ele descreve as características do *hardware*, que reflete em um conjunto de instruções suportadas pelo dispositivo, assim como outras especificações [4]. Dispositivos com um mesmo *major number* possuem a mesma arquitetura central. O *minor number* corresponde a melhorias acrescentadas à essa arquitetura. Quanto maior a *compute capability*, mais nova é a geração da GPU, assim como o conjunto de funcionalidades é maior e, geralmente, uma característica de determinada *capability* é suportada por qualquer outra mais atual.

A tabela abaixo mostra algumas especificações das diferentes *capability* disponíveis até o momento:

	Compute Capability				
Especificações técnicas	1.0	1.1	1.2	1.3	2.x
Dimensão máxima de um *grid*	2				3
Dimensão máxima de um *bloco*	3				
Número máximo de *threads* por bloco	512				1024
Tamanho do *warp*	32				
Número máximo de blocos residentes em um multiprocessador	8				
Número máximo de *warps* residentes em um multiprocessador	24		32		48
Número máximo de *threads* residentes em um multiprocessador	768		1024		1536
Número de registradores 32-bit por multiprocessador	8 K		16 K		32 K
Quantidade máxima de memória compartilhada por multiprocessador	16 KB				48 KB
Quantidade de memória local por *thread*	16 KB				512 KB
Número máximo de instruções por *kernel*	2 milhões				

3.4.5 Exemplo

Código 3: Soma dois vetores A e B, de dimensão N, e coloca o resultado no vetor C

```
//definicao do kernel
__global__ void vecAdd(float* A, float* B, float* C)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if(i < N)
```

```
    C[i] = A[i] + B[i];
}

//codigo do host (CPU)
int main()
{
  //tamanho do vetor
  int N = ...;
  size_t size = N * sizeof(float);

  //aloca vetores na memoria do host
  float* host_A = malloc(size);
  float* host_B = malloc(size);
  float* host_C = malloc(size);

  //aloca vetores na memoria do device (GPU)
  float* device_A;
  cudaMalloc((void**)&device_A, size);
  float* device_B;
  cudaMalloc((void**)&device_B, size);
  float* device_C;
  cudaMalloc((void**)&device_C, size);

  //copia vetores da memoria do host para a do device
  //funcao cudaMemcpy realiza a transferencia de dados entre
      host e device
  cudaMemcpy(device_A, host_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(device_B, host_B, size, cudaMemcpyHostToDevice);

  //chamada do kernel, informando tamanho de bloco e grid
  int threadsPerBlock = 256;
  int blocksPerGrid = (N + threadsPerBlock - 1) /
      threadsPerBlock;
  vecAdd<<<blocksPerGrid, threadsPerBlock>>>(device_A, device_B,
       device_C);
```

```
    //copia o resultado da memoria do device para a memoria do
        host
40  //host_C contem o resultado na memoria do host
    cudaMemcpy(host_C, device_C, size, cudaMemcpyDeviceToHost);

    //libera memoria do device
    cudaFree(device_A);
45  cudaFree(device_B);
    cudaFree(device_C);
}
```

4 Desempenho e otimização

A chave para obter melhor desempenho de uma aplicação CUDA consiste em utilizar *multithreading* massivamente. O desafio para atingir esta meta é conciliar os recursos disponíveis e o número de *threads* ativas, encontrando um ponto de equilíbrio que favoreça a aplicação. Diferentes aplicações possuem restrições diferentes que se tornam fatores limitantes. Administrá-los requer entendimento da arquitetura e conhecimento de estratégias já estabelecidas, evitando que se torne um trabalho de adivinhação. Ainda assim é necessário que o programador experimente algumas alternativas antes de escolher a melhor alternativa que se encaixe no problema.

Os tópicos seguintes abordarão aspectos relacionados ao gerenciamento de *threads* e recursos disponíveis, que é o foco deste trabalho.

4.1 Partição dinâmica de recursos

Uma das principais características da arquitetura CUDA é a flexibilidade na partição de recursos, como registradores e memória local, entre as *threads*. Tais recursos são particionados dinamicamente, fazendo com que os multiprocessadores sejam extremamente versáteis. É possível executar vários blocos compostos por poucas *threads*, assim como pode-se ter poucos blocos com muitas *threads*, permitindo ajustar a configuração mais adequada para cada problema.

4.2 Registradores

Embora acessar um registrador geralmente não represente custo extra, há atraso quando uma instrução utiliza um resultado alocado por uma instrução anterior. Tal latência pode ser completamente escondida se houver pelo menos 192 *threads* ativas (isto é, 6 *warps*) para dispositivos com *capability* 1.x.

Já nas de 2.0, que possuem 32 núcleos CUDA por multiprocessador, são necessários no mínimo 768 *threads* [7].

Além disso, apesar de cada multiprocessador possuir milhares de registradores, estes devem ser compartilhados entre as várias *threads*, o que nem sempre é suficiente dependendo da tarefa a ser executada. Quanto menos registadores um *kernel* utilizar, mais *threads* poderão residir em um multiprocessador. O compilador é responsável pela distribuição destes recursos tendo como base heurísticas de otimização. Com o intuito de melhorar o desempenho e impedir que muitos registradores sejam alocados, o desenvolvedor pode prover informação adicional ao compilador na forma de *launch bounds*.

Código 4: Especificando um *Launch bound*.

```
__global__ void
__launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor
    )
MyKernel(...)
{
    ...
}
```

- maxThreadsPerBlock possibilita limitar o número máximo de *threads* por bloco no lançamento do *kernel*.
- minBlocksPerMultiprocessor é um campo opcional que possibilita especificar o número mínimo de blocos por multiprocessador.

4.3 Blocos de *threads*

Na execução de um *kernel*, a principal preocupação ao escolher o número de blocos por *grid* deve ser manter a GPU o mais ocupada possível. Desta forma, se o número de blocos for maior que o de multiprocessadores, cada

multiprocessador terá pelo menos um bloco para executar. Mais ainda, o ideal é ter vários blocos ativos para aqueles que não estiverem esperando por uma sincronização mantenham o dispositivo ocupado [4].

Já a escolha do tamanho do bloco, embora envolva vários fatores, deve seguir algumas regras [4]:

- o número de *threads* por bloco deve ser múltiplo do tamanho do *warp* a fim de evitar desperdício em processamento com *warps* não totalmente preenchidos.
- um mínimo de 64 *threads* por bloco deve ser utilizado, mas apenas se houver múltiplos blocos simultâneos por multiprocessador.
- entre 128 e 256 *threads* por bloco é um bom ponto de partida para experimentar diferentes tamanhos de bloco.
- utilize vários (3 a 4) pequenos blocos ao invés de 1 grande bloco por multiprocessador se a latência afetar o desempenho.

4.4 Ocupação do multiprocessador

Na busca por melhor desempenho, é fundamental manter os multiprocessadores da GPU o mais ocupados possível. O conceito chave por trás deste esforço é a ocupação (*occupancy*), métrica que representa a razão entre o número de *warps* ativos por multiprocessador sobre o número máximo de *warps* possível. Para maximizar a utilização de *hardware*, a aplicação deve ser executada seguindo uma boa configuração (variável de acordo com a arquitetura) que irá balancear o número de *threads* e blocos com os registradores e memória compartilhada (pois ambos possuem baixa latência). Quanto maior for a taxa, mais completamente o *hardware* estará sendo utilizado, escondendo a latência causada pelos acessos à memória global [32].

Embora altas taxas nem sempre representem melhora de desempenho (há um ponto na qual ocupação adicional não melhora o desempenho), uma baixa taxa de ocupação sempre interfere de forma negativa [4]. O ideal é que seja alcançado o valor de pelo menos 50%.

Disponibilidade de registradores é uma das maneiras de determinar a ocupação pois tais dispositivos permitem acesso com baixa latência às variáveis locais. Cada multiprocessador possui um conjunto limitado de registradores e memória compartilhada, que são compartilhados pelos blocos de *threads* em execução. Se cada bloco utilizar muitos registradores, o número de blocos em um multiprocessador será reduzido, acarretando em uma taxa baixa. A ocupação pode ser melhorada com a diminuição de recursos utilizados por cada bloco ou diminuindo o número de *threads* em cada bloco.

5 Trabalhos relacionados

Com o objetivo de auxiliar o desenvolvedor na complexa tarefa de programar para a GPU, ferramentas de análise de desempenho tem surgido, criando um ambiente mais amigável e facilitando o entendimento de aspectos que não existem na programação convencional.

Devido a complexidade do tema, as ferramentas costumam ser complementares, sendo mais proveitoso utilizar mais de uma na análise do programa. Nos tópicos seguintes são apresentadas algumas das alternativas disponíveis.

5.1 Occupancy Calculator

O número de registradores disponíveis e o número máximo de *threads* simultâneas em cada multiprocessador varia entre as diferentes arquiteturas. Devido às nuances na alocação de registradores e no fato de que a memória compartilhada é particionada entre os blocos, a relação exata entre a utilização de registradores e a ocupação pode ser difícil de se determinar [4].

Por exemplo, GPUs com capacidade computacional 1.0 possuem 8192 registradores 32-bit por multiprocessador, que suportam até 768 *threads* simultaneamente (24 *warps* x 32 *threads* por *warp*). Dessa forma, para que a ocupação seja de 100%, cada *thread* pode utilizar no máximo 10 registradores. Contudo, tal abordagem não leva em consideração a granularidade da alocação dos registradores. Considerando blocos de 128 *threads* com 12 registradores para cada *thread*, obtemos uma ocupação de 83%, com 5 blocos ativos por multiprocessador. Por outro lado, se os blocos tiverem 256 *threads* com os mesmos 12 registradores, a ocupação será de apenas 66%, pois apenas 2 blocos poderão residir no multiprocessador.

A *Occupancy Calculator* é uma planilha em Excel criada pela NVIDIA

que permite calcular mais facilmente a ocupação do multiprocessador da GPU para determinado *kernel* CUDA [2], ajudando a selecionar uma boa configuração para diferentes arquiteturas e cenários. A planilha permite explorar os *trade-offs* entre o número de *threads* e blocos ativos e a quantidade de registradores e memória compartilhada.

A figura 16 mostra como é possível testar várias combinações para tamanho de bloco, registradores disponíveis para cada *thread* e quantidade de memória compartilhada por bloco (etapa 2 da planilha). Os resultados para a combinação escolhida são mostrados logo abaixo (etapa 3 da planilha).

5.2 Compute Visual Profiler

O Compute Visual Profiler é uma ferramenta utilizada para medir o desempenho e encontrar oportunidades de otimização com o intuito de obter máximo desempenho das GPUs NVIDIA [45]. A ferramenta é a mais completa opção disponível atualmente, fazendo parte do CUDA *Toolkit* e embora esteja presente para todas as plataformas, nem todas funcionalidades estão disponíveis em todas elas. Fornece métricas na forma de gráficos e contadores.

Os contadores são utilizados para identificar diferenças de desempenho entre um código otimizado e outro sem otimização. Eles representam eventos de um *warp*, e não a atividade de uma *thread* individualmente. Além disso a análise é baseada em apenas um multiprocessador, não correspondendo ao total de *warps* lançados para determinado *kernel*. Por esta razão, é recomendável que sejam lançados blocos de *threads* em número suficiente de forma a entregar uma porcentagem consistente de trabalho ao multiprocessador [45].

Os resultados dos contadores podem ser diferentes para a mesma aplicação em diferentes execuções pois depende do número de blocos executados em

cada multiprocessador. Para resultados consistentes, o ideal é que o número de blocos para cada *kernel* lançado seja igual ou múltiplo ao total de multiprocessadores disponíveis, permitindo a distribuição uniforme de trabalho.

A figura 17 mostra a visão geral da análise dos *kernels* que fazem parte da aplicação a ser analisada, informando o tempo gasto para cada um ser executado na GPU, o número de vezes que são chamados etc. Cada teste feito é chamado de *session*, sendo possível configurá-lo de acordo com o interesse do programador: escolher a GPU a ser utilizada (no caso de haver mais de uma), tempo de execução máximo da aplicação, quais configurações serão visualizadas (tamanho de bloco, tamanho de *grid*, quantidade de memória compartilhada etc), dentre outras.

A figura 18 mostra um maior detalhamento de um *kernel* escolhido na tela anterior.

5.3 NVIDIA Parallel Nsight™

O NVIDIA Parallel Nsight™ é um ambiente de desenvolvimento para aplicações em CUDA integrado ao Microsoft Visual Studio. Ele estende as capacidades de depuração de código e análise de desempenho disponíveis no Visual Studio para computação em GPU.

Através da ferramenta de análise de desempenho é possível acompanhar as atividades e eventos entre CPU e GPU, além de analisar o desempenho a nível de *kernel*. Suas principais funcionalidades são:

- Depuração de *kernels* CUDA, permitindo a inclusão de *breakpoints*, examinar variáveis na GPU e verificar erros de memória
- Coleta de dados (*trace*/*log*) de forma que o programador possa entender onde e como a aplicação está consumindo tempo (na GPU, transferindo

dados ou na CPU)

Além de estar disponível apenas para a plataforma Windows, a maior parte de suas funcionalidades estão presentes apenas na versão paga.

5.4 CUDA Profiling Tools Interface - CUPTI

Inserido junto com o CUDA Tools SDK (Toolkit 4.0), o CUPTI é uma biblioteca dinâmica que permite criar ferramentas para analisar o desempenho de aplicações em CUDA. Através de duas APIs, *Callback API* e *Event API*, é possível desenvolver ferramentas que ajudam a compreender o comportamento de aplicações CUDA na CPU e GPU [56].

5.4.1 CUPTI Callback API

O *Callback API* permite que o desenvolvedor insira seu próprio código na entrada e saída de cada chamada à CUDA *runtime* ou *driver* API. Uma funcão *callback* é associada com uma ou mais funções da CUDA API. Quando estas funções são chamadas, a função *callback* criada é chamada também [56].

5.4.2 CUPTI Event API

O *Event API* permite pesquisar, configurar, iniciar, parar e ler os contadores de evento presentes em um dispositivo CUDA.

Há três tipos de eventos, cada um indicando como a atividade ou ação associada é coletada:

- SM - indica que o evento é coletado por uma ação ou atividade que ocorre em um ou mais multiprocessadores (*streaming multiprocessor - SM*).

- TPC - indica que o evento é coletado por uma ação ou atividade que ocorre no primeiro multiprocessador de um *Texture Processing Cluster* (TPC). GPUs com *capability* 1.x possuem dois por TPC, já as de *capability* 2.0, possuem três multiprocessadores por TPC.

- FB - indica que o evento é coletado por uma ação ou atividade que ocorre em uma partição da DRAM.

5.5 PAPI CUDA Component

O projeto PAPI (*Performance Application Programming Interface*) visa proporcionar uma interface consistente e metodologia para medição do desempenho do *hardware* através de contadores presentes na maior parte dos microprocessadores. Ele permite a visualização, quase que em tempo real, da relação entre desempenho do *software* e eventos do processador [47].

O PAPI CUDA Component é uma ferramenta feita para a tecnologia da NVIDIA, permitindo o acesso aos contadores presentes na GPU. Baseado no suporte CUPTI presente na *NVIDIA driver library*, é capaz de prover informações a respeito da execução de *kernels* na GPU. Atualmente está disponível apenas para a plataforma Linux.

No componente, a inicialização, a gestão do dispositivo e de contexto é feito pelo pelo *CUDA driver API*, enquanto que a gestão de domínio e evento é feito pelo CUPTI. Devido as diferenças de contadores de *hardware* entre as várias GPUs, o PAPI provê uma lista de eventos disponíveis em uma GPU específica.

5.6 Vampir/VampirTrace

O Vampir é uma ferramenta de análise de desempenho que permite a visualização gráfica e análise de aplicações paralelas (por exemplo, MPI) através

da coleta de dados relativos à execução do programa (*trace*). Foi desenvolvida para ser de fácil uso, permitindo que os desenvolvedores possam analisar rapidamente o comportamento da aplicação em qualquer nível de detalhe [48]. Sua metodologia consiste em [46]:

- instrumentação (modificação do código e estrutura de um programa após este ter sido compilado) do código fonte da aplicação para ser estudado e conectado ao VampirTrace (biblioteca que permite a gravação dos eventos em um arquivo *trace*);
- executar a aplicação instrumentada para gerar arquivo detalhado de *log* (*trace*);
- análise do arquivo de *trace* através das ferramentas disponíveis.

O VampirTrace possui suporte à CUDA, possibilitando gerar arquivos de *trace* relativos à utilização da GPU através da interceptação das chamadas na biblioteca CUDA. São gravadas informações como o número de vezes que um *kernel* é executado e transferência de dados entre CPU e GPU.

Através do CUPTI, contadores de desempenho são acessados pelo VampirTrace a fim de obter mais informações a respeito da execução de um *kernel*. Os arquivos de *trace* gerados são analisados sem a necessidade de mudanças na visualização do Vampir. Apesar disso, a reutilização de métricas (a ferramenta foi feita para o MPI) com a adição de eventos da GPU polui as estatísticas obtidas. Tal limitação seria resolvida apenas com mudanças mais trabalhosas tanto no Vampir quanto no VampirTrace.

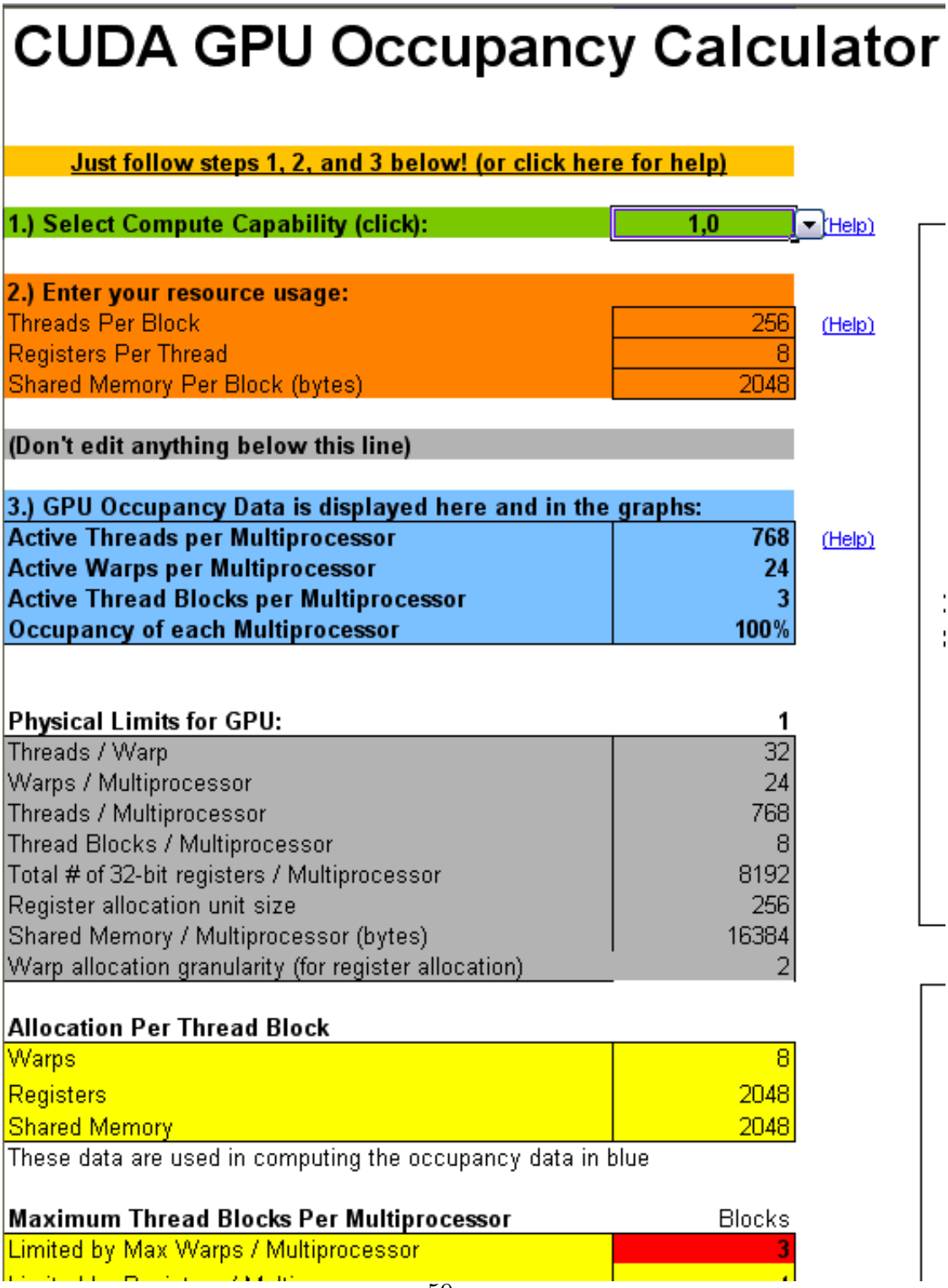
5.7 TAU Performance System

TAU Performance System®[49] é uma ferramenta de código aberto capaz de medir e analisar o desempenho de aplicações paralelas, informando o tempo total gasto em cada rotina (*profiling*) e quando determinado evento ocorre ao

longo da linha do tempo (*tracing*). O suporte a CUDA foi originado através de uma versão experimental, TAUcuda, compatível apenas com uma versão específica do Linux CUDA *driver library*, que fornecia suporte experimental a ferramentas.

Possui suporte a ferramentas para instrumentação de código e compilador, além de uma biblioteca de encapsulamento que permite o monitoramento dos eventos na CPU.

O principal desafio para a ferramenta é ser capaz de observar a memória da GPU e as operações do *kernel* e associar as informações de desempenho da GPU com a aplicação sendo executada na CPU. É uma tarefa complicada pois as operações da GPU ocorrem de forma assíncrona com as atividades da CPU. Enquanto o VampirTrace é capaz de gerar eventos *trace* que pode serem associados no tempo com um evento na CPU, analisar um *kernel* da GPU necessita de suporte extra para contextualizar seus eventos com a execução da CPU. Para fazer essa associação de GPU com a aplicação que fez a chamada à rotina do *driver* CUDA, é recuperado o *TAU context*, que representa o evento mais próximo da CPU que o TAU está medindo no momento do lançamento do *kernel*. Desta forma, a ferramenta é capaz de distinguir *kernels* lançados de diferentes partes da aplicação.



CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	1,0	(Help)
2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	8	
Shared Memory Per Block (bytes)	2048	
(Don't edit anything below this line)		
3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	**768**	(Help)
Active Warps per Multiprocessor	**24**	
Active Thread Blocks per Multiprocessor	**3**	
Occupancy of each Multiprocessor	**100%**	

Physical Limits for GPU:	**1**
Threads / Warp	32
Warps / Multiprocessor	24
Threads / Multiprocessor	768
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	8192
Register allocation unit size	256
Shared Memory / Multiprocessor (bytes)	16384
Warp allocation granularity (for register allocation)	2

Allocation Per Thread Block	
Warps	8
Registers	2048
Shared Memory	2048

These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	3

Figura 16: CUDA Occupancy Calculator

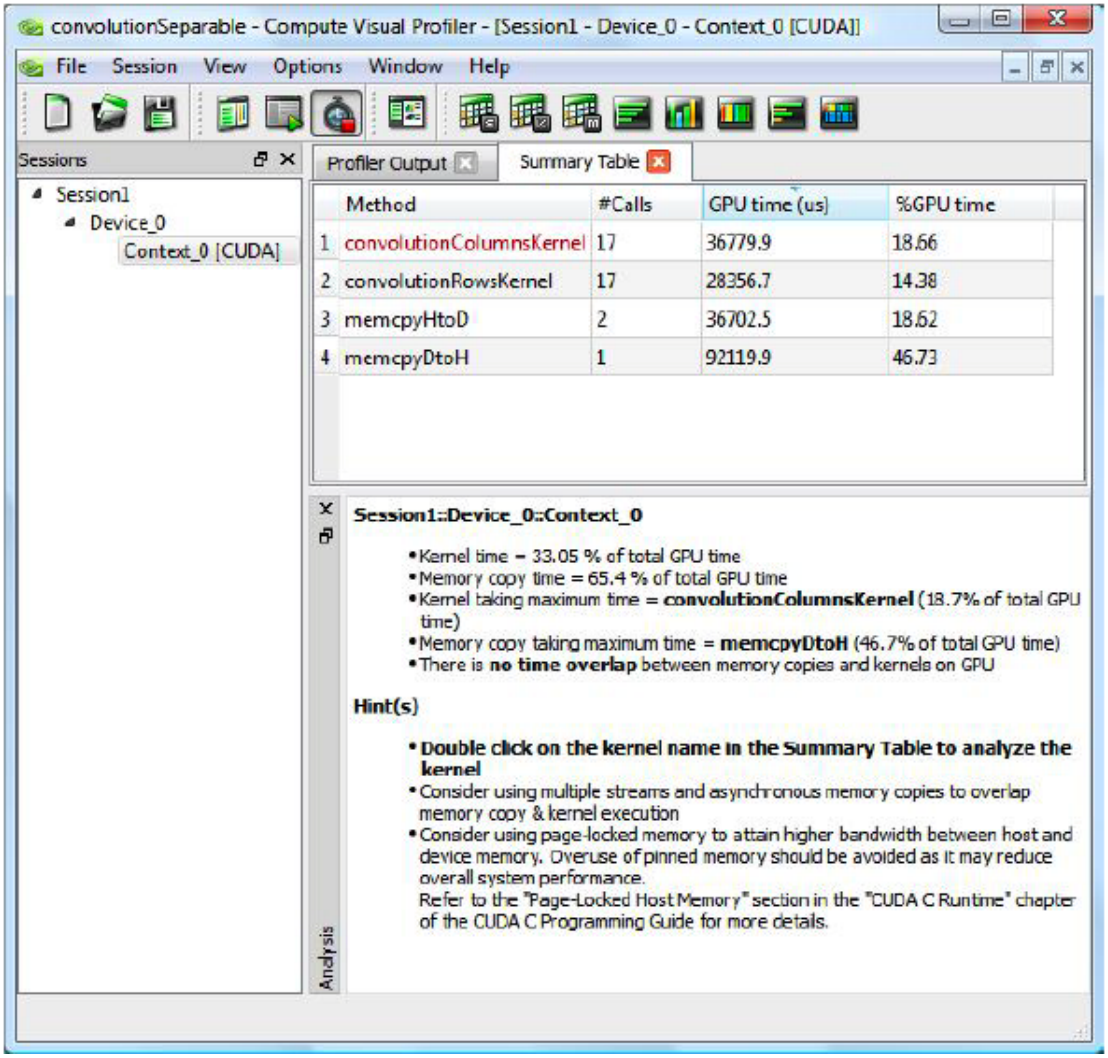


Figura 17: Visual Profiler

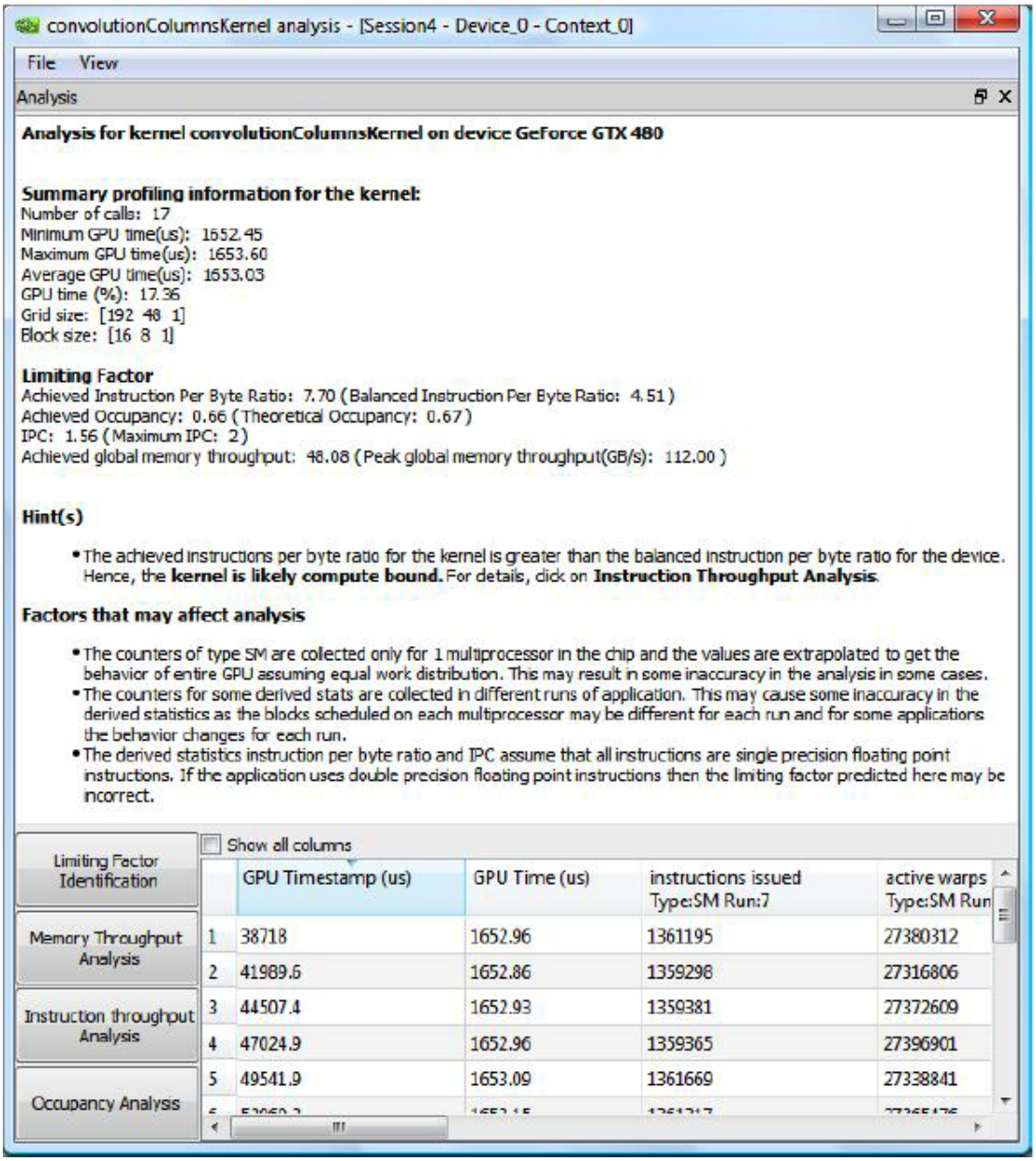


Figura 18: Visual Profiler - detalhamento de um *kernel*

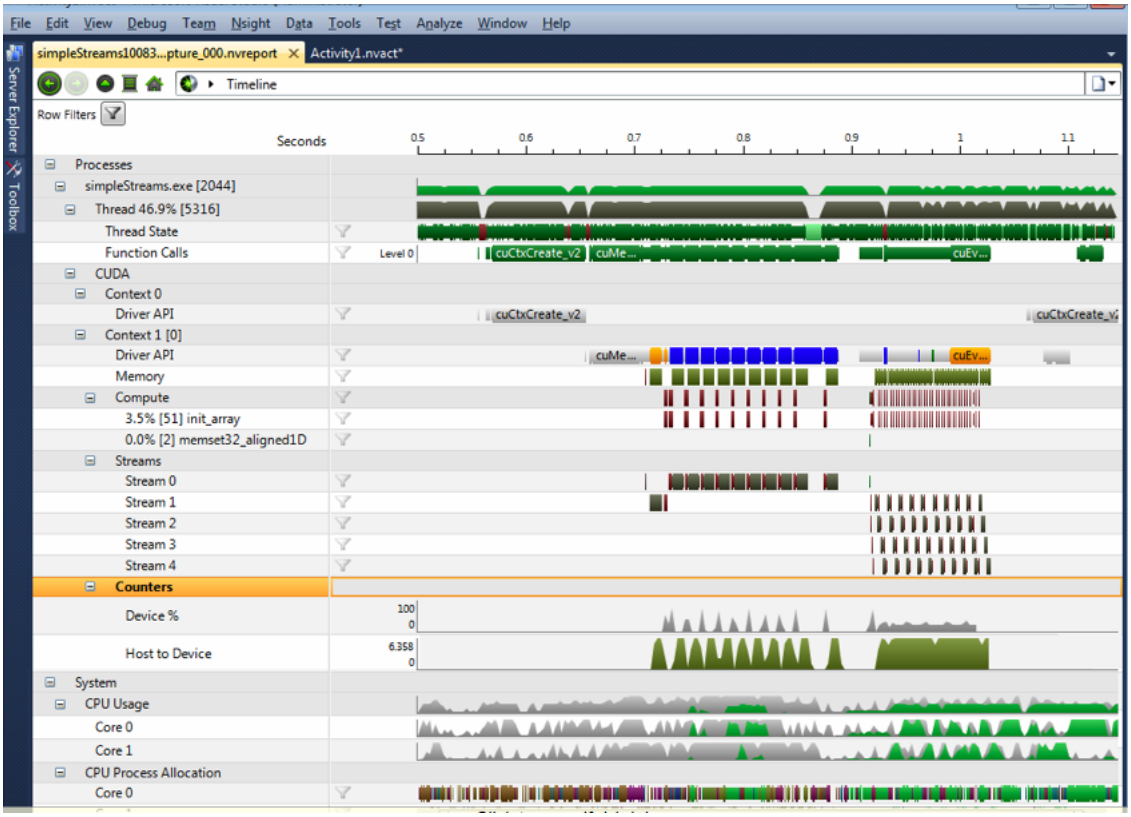


Figura 19: Parallel Nsight

6 Descrição do *plug-in* desenvolvido

A ferramenta para auxílio na programação em CUDA desenvolvida neste trabalho foi feita na forma de um *plug-in* para o Eclipse. Os tópicos seguintes descrevem as tecnologias utilizadas, algumas vantagens e funcionamento do *plug-in*.

6.1 Eclipse

Eclipse é um arcabouço *opensource* para desenvolvimento de sistemas, escrito em Java. Doada à comunidade de *software* livre pela IBM, possui uma arquitetura madura, bem projetada e estensível.

Através de *plug-ins*, diversas ferramentas podem ser combinadas criando uma IDE (ambiente de desenvolvimento integrado), oferecendo suporte a várias linguagens (além de Java) como C, C++, Python, Perl e PHP. Tal característica permite estender suas capacidades, além da possibilidade dos usuários criarem seus próprios *plug-ins*.

Levando em consideração tais propriedades, o Eclipse foi escolhido para o desenvolvimento de uma ferramenta que auxilie na otimização da ocupação, maximizando o uso do *hardware* (GPU). Sugestões de melhorias serão fornecidas no próprio ambiente de programação em CUDA (implementado na forma de um *plug-in* para o Eclipse).

6.1.1 *Plug-ins*

A arquitetura extensível do Eclipse se baseia na criação de *plug-ins*, que são integrados seguindo o conceito de extensão e ponto de extensão. Fazendo uma analogia, a entrada elétrica seria o ponto de extensão e o *plug* de um aparelho qualquer, a extensão. Apenas *plugs* de determinado tipo podem ser encaixados em uma entrada. Da mesma forma, um *plug-in* requer um ponto

de extensão ao qual se conectar para que funcione.

Com excessão do *kernel* (conhecido como *Platform Runtime*), tudo no Eclipse é *plug-in*. Cada sub-sistema na plataforma é estruturado como um conjunto de *plug-ins* que implementam alguma função chave. Alguns adicionam características visíveis, outros fornecem classes bibliotecas para serem implementadas [53]. Como mostrado na figura 20, a estrutura padrão do Eclipse é formada pelos seguintes componentes:

- *Workspace* - central onde se encontram todos os projetos do usuário, onde cada projeto mapeia para seu sistema de arquivos correspondente. Contém pontos de extensão que permitem interagir com recursos (projetos, pastas e arquivos).
- *Workbench* - ambiente no qual o trabalho será realizado, incluindo menu e barra de tarefas, editores de texto e operações de arrastar e colar.
- JDT (*Java Development Tools*) - conjunto de *plug-ins* de ferramentas que formam um efetivo ambiente integrado (IDE) para desenvolvimento de aplicações Java, incluindo editores de texto com assistência para codificação, vizualizações, compilador, debugador e gerenciador de projetos.
- PDE (*Plug-in Development Environment*) - recursos e facilidades necessários para o desenvolvimento de *plug-ins* no Eclipse.
- *Team* - permite que os projetos sejam submetidos a controle de versão e sejam associados a um repositório.
- *Help* - permite fornecer documentação e ajuda *on-line*.

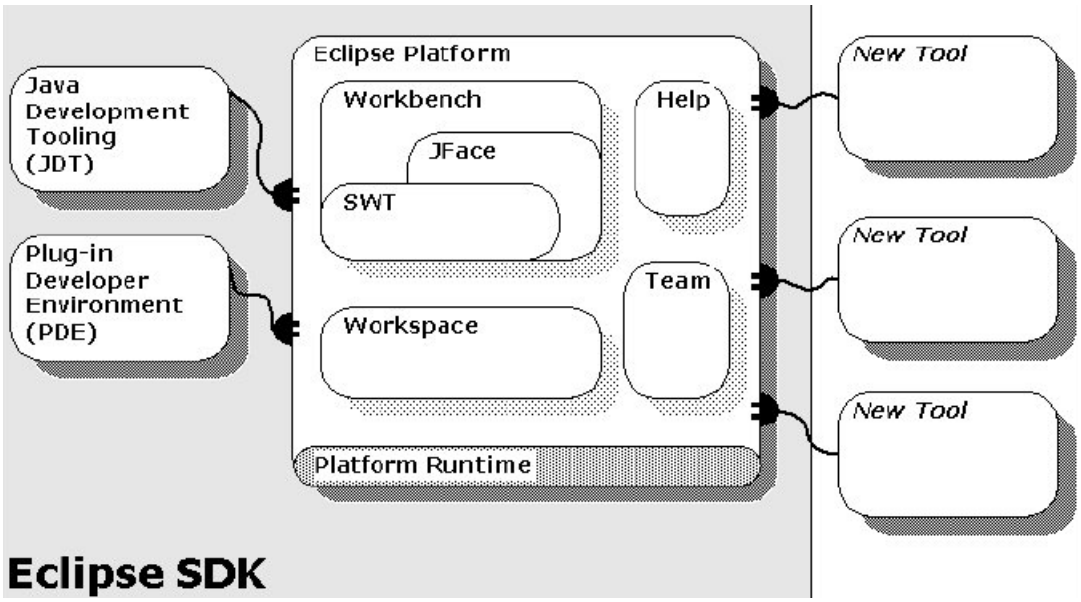


Figura 20: Eclipse SDK [53]

6.2 JNI

JNI (Java Native Interface) é um *framework* que permite que a máquina virtual da linguagem Java acesse bibliotecas construídas com o código nativo (específico de um *hardware* ou sistema operacional) de um sistema e bibliotecas escritas em liguagens como C e C++ [55].

Através do JNI, funções específicas de CUDA (escrito em C++) foram chamadas a partir do *plug-in*.

6.3 Funcionalidades

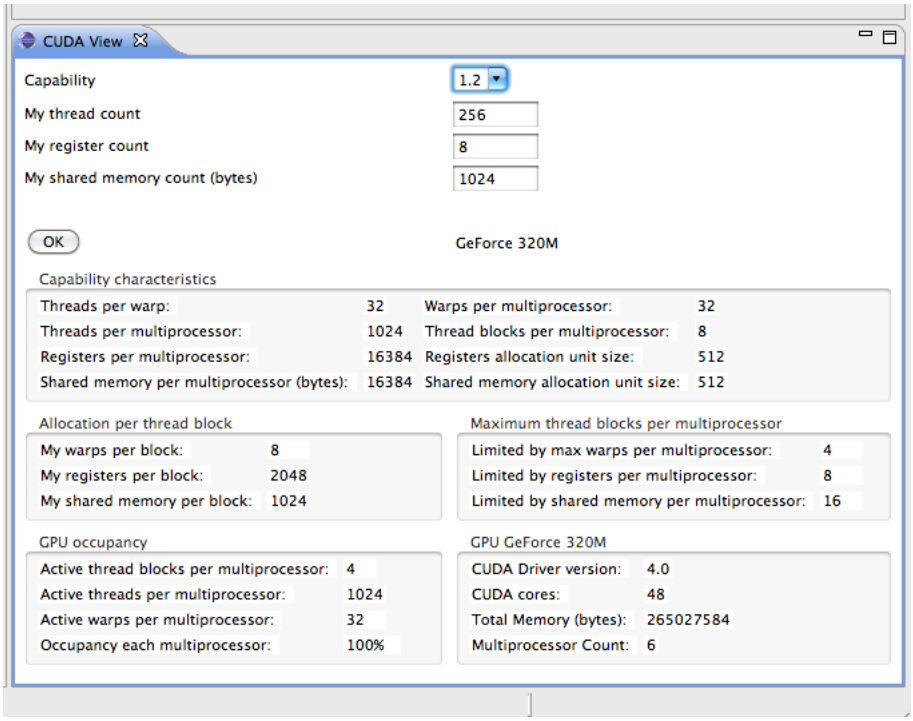


Figura 21: *Plug-in* - visão geral

Feito como um *plug-in* do Eclipse, a ferramenta está integrada ao ambiente de desenvolvimento, sendo facilmente utilizada durante a programação.

Ele é composto por 6 partes que serão detalhadas nos próximos tópicos.

6.3.1 Entrada de dados

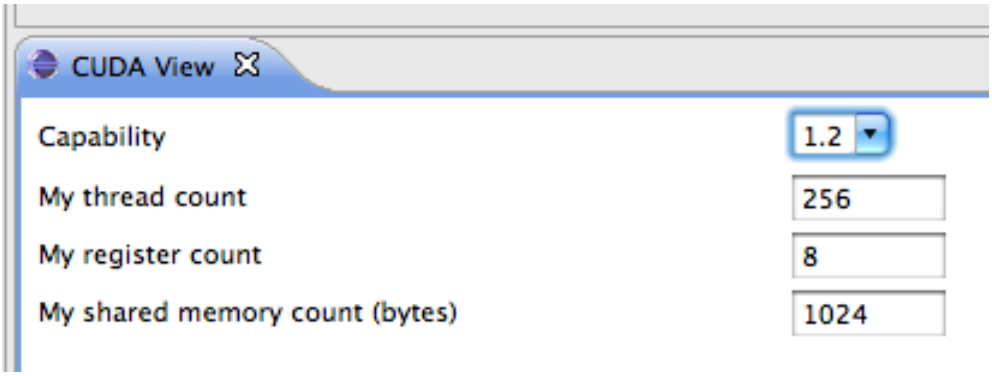


Figura 22: Entrada de dados

A primeira parte do *plug-in* é onde o usuário irá inserir a configuração escolhida para seu *kernel*. Com o CUDA instalado e uma GPU compatível, a *capability* correspondente àquela placa já vem selecionada. Para que os cálculos sejam feitos, são necessários mais três parâmetros:

- *My thread count* - tamanho do bloco de *thread* escolhido para o *kernel*. Como mostrado no capítulo 3, na chamada de um *kernel* é preciso passar o tamanho do *grid* (blocos por *grid*) e o tamanho do bloco (*threads* por bloco). Dessa forma, neste campo deve ser informado quantas *threads* um bloco possui, que corresponde ao segundo valor passado no *kernel*.

- *My register count* - quantidade de registradores alocados pelo compilador para cada *thread*. Embora não seja possível escolher esse parâmetro, há a possibilidade de limitar o número de registradores através de *launch bounds*, como visto no capítulo 4, pois nem sempre queremos a quantidade alocada pelo compilador. Para descobrir este valor, basta acrescentar a opção *–ptxas-options=-v* ao compilar o *kernel* com o nvcc.

- *My shared memory count* - quantidade de memória compartilhada (em *bytes*) utilizada pelo *kernel*. Este valor é mostrado juntamente com o número de registradores, ao compilar utilizando a opção *–ptxas-options=-v*. Caso seja utilizada mais memória compartilhada dinamicamente, esta deve ser acrescentada ao valor obtido.

6.3.2 *Capability characteristics*

Figura 23: *Capability characteristics*

Nesta parte são mostradas características da GPU de acordo com sua *capability*. Representa os limites físicos daquela placa, uma GeForce 320M com *capability* 1.2, no exemplo mostrado na figura 23.

6.3.3 *Allocation per thread block*

Figura 24: *Allocation per thread block*

Nesta parte são feitos os primeiros cálculos utilizando os dados fornecidos pelo usuário juntamente com as propriedades da GPU reconhecida. Os resultados obtidos são relativos à alocação de um bloco:

- *My warps per block* - número de *warps* (bloco de 32 *threads*) em cada bloco. No exemplo, cada bloco possui 256 *threads*, ou seja, 8 *warps*.
- *My registers per block* - número de registradores por bloco.
- *My shared memory per block* - quantidade de memória compartilhada por bloco. Para *capability* 1.x, o valor mínimo é de 512, já para 2.0, é de 128. Portanto, apenas se o valor informado no campo *My shared memory count* for maior que o mínimo, ele será utilizado.

6.3.4 *Maximum thread blocks per multiprocessor*

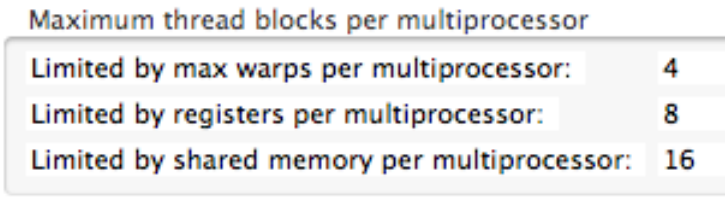


Figura 25: *Maximum thread blocks per multiprocessor*

Nesta parte, são feitos cálculos de limite de blocos por multiprocessador. No item seguinte, será utilizado o menor valor dentre esses três:

- *Limited by max warps per multiprocessor* - mínimo entre o total possível de blocos por multiprocessador e total de *warps* no multiprocessador sobre a quantidade de *warps* no bloco.
- *Limited by registers per multiprocessor* - total de registradores disponíveis sobre número de registradores por bloco.
- *Limited by shared memory per multiprocessor* - total de memória compartilhada disponível sobre a quantidade de memória por bloco.

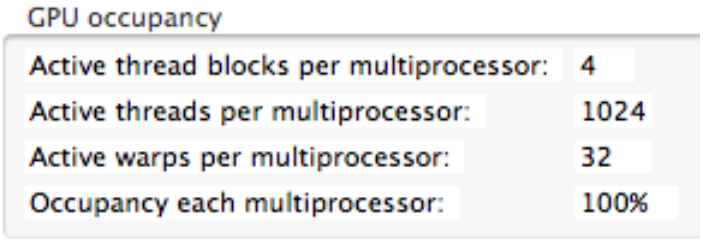


Figura 26: *GPU occupancy*

6.3.5 *GPU occupancy*

Nesta parte do *plug-in* são feitos cálculos relativos à ocupação da GPU:

- *Active thread blocks per multiprocessor* - blocos ativos por multiprocessador. Pega o menor valor dentre os três calculados no item anterior.
- *Active threads per multiprocessor* - *threads* ativas por multiprocessador. Calculado pela multiplicação entre quantidade de blocos ativos e o tamanho de bloco escolhido pelo usuário.
- *Active warps per multiprocessor* - *warps* ativos por multiprocessador. Como um *warp* corresponde a um conjunto de 32 *threads*, o valor é obtido multiplicando a quantidade de blocos ativos e a quantidade de *warps* em cada bloco.
- *Occupancy each multiprocessor* - ocupação do multiprocessador, ou seja, a quantidade de *warps* ativos por multiprocessador sobre o máximo possível.

6.3.6 Informações adicionais da GPU

Nesta última parte são apresentados alguns dados relativos à GPU que está sendo utilizada (número de núcleos CUDA, total de memória disponível e quantidade de multiprocessadores presentes), além da versão do *driver*

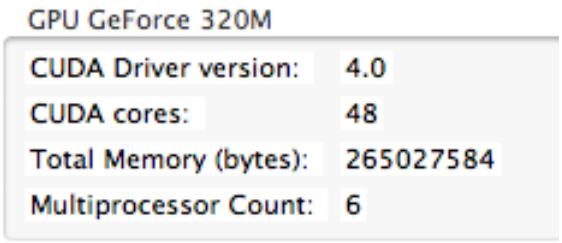


Figura 27: Informações adicionais da GPU

CUDA instalado. Estas informações são fornecidas pelas funções CUDA, cujo acesso é feito através do JNI.

No exemplo mostrado nas figuras é feito o cálculo da ocupação para blocos com 256 *threads*, 8 registradores para cada *thread* e 1024 *bytes* de memória compartilhada. O resultado é mostrado no quadro "GPU occupancy": ocupação de 100%, 4 blocos ativos por multiprocessador, 1024 *threads* ativas por multiprocessador e 32 *warps* por multiprocessador. Pode-se fazer várias experimentações com diferentes tamanhos de bloco e registradores por *thread*, comparando como as mudanças afetam esses parâmetros.

A figura 28 mostra um *warning* sendo disparado ao tentar especificar um tamanho de bloco maior que o permitido. De acordo com os dados passados, outras mensagens podem aparecer, orientando o programador a melhorar sua escolha de configuração.

Esta é uma ferramenta complementar às já existentes, como o Visual Profiler. Suas vantagens estão relacionadas principalmente à praticidade e possibilidade de estender as funcionalidades utilizando um ambiente (o Eclipse) com ampla adesão. À medida que novas práticas e parâmetros forem surgindo, poderão ser acrescentados na ferramenta, de forma a mantê-la sempre atualizada, além de enriquecer as informações apresentadas ao programador

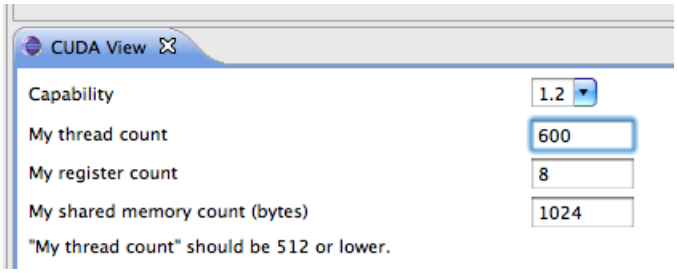


Figura 28: *Plug-in - warnings*

menos experiente no assunto.

7 Conclusão

O mercado de aceleradores gráficos baseados em GPU para computação de alto desempenho cada vez mais ganha notoriedade. Sua utilização para propósitos além dos gráficos oferece inúmeros benefícios, tanto em melhoras consideráveis na velocidade de resolução de problemas computacionais quanto em custo. Com o lançamento da tecnologia CUDA pela NVIDIA, programar para a GPU deixou de ser uma tarefa para poucos experientes no assunto, e se tornou um poderoso processador programável utilizando linguagens amplamente difundidas (C/C++), diminuindo a curva de aprendizagem.

A flexibilidade de alocação dos recursos presentes na GPU da NVIDIA, ao mesmo tempo que permite melhor se adaptar a diferentes problemas, acarreta em maior complexidade para encontrar a configuração mais adequada. Desenvolver programas paralelos não é trivial e muitas vezes gera resultados inesperados. Levando em consideração esses desafios, este trabalho estudou algumas ferramentas para analisar o desempenho (*profiling*) de programas em CUDA e apresentou uma ferramenta complementar, feito sob a forma de um *plug-in* do Eclipse, objetivando auxiliar o programador a testar e encontrar uma boa configuração de *kernel* de forma a aproveitar melhor os recursos da GPU. O *plug-in* foi feito pensando na praticidade, integrado ao Eclipse, uma plataforma de desenvolvimento amplamente utilizada, e escalabilidade, permitindo que novos recursos sejam acrescentados.

Como trabalho futuro fica a adição de gráficos, que permitiriam uma comparação entre diferentes configurações, acrescentar o suporte a computadores com multi GPUs, aperfeiçoar o espaço de *warnings* e tornar a interface mais amigável e intuitiva.

Referências Bibliográficas

[1] Owens, John D.; Houston, Mike; Luebke, David; Green, Simon; Stone, John E.; Phillips, James C. - "GPU Computing"(2008)

[2] NVidia GPU Computing Developer Home Page - http://developer.nvidia.com/object/gpucomputing.html

[3] NVidia CUDA Programming Guide Version 3.0 - www.nvidia.com/object/cuda_get.html

[4] NVidia CUDA Best Practices Guide Version 3.0 - www.nvidia.com/object/cuda_get.html

[5] NVidia CUDA Architecture Overview Version 1.1 - http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf

[6] Kirk, David; Hwu, Wen-mei - Textbook of UIUC's class - http://courses.ece.illinois.edu/ece498/al/Syllabus.html

[7] Kirk, David; Hwu, Wen-mei - Programming Massively Parallel Processors - A Hands-on Approach (2010)

[8] Ferraz, Samuel B. - "GPUs"(2009)

[9] Barthels, Andreas - "GPU Computing"(2009)

[10] NVidia CUDA - http://ixbtlabs.com/articles3/video/cuda-1-p1.html

[11] NVidia CUDA - http://www.behardware.com/articles/659-1/nvidia-cuda-preview.html

[12] General-purpose computing on the GPU (GPGPU) - http://www.think-techie.com/2009/09/general-purpose-computing-on-gpu-gpgpu.html

[13] Göhner, Uli - ”Computing on GPUs”(2009)

[14] CPU GPU diferences - http://hi.baidu.com/dabaohaitun/blog/item/b561101ef88582174034172d.html

[15] Mocelin, Charlan Luís; Silva, Eduardo Fialho Barcellos da; Tobaldini, Geison Igor - ”GPU: Aspectos Gerais”(2009)

[16] Graphics Processing Unit - http://en.wikipedia.org/wiki/Graphics_processing_unit

[17] Cirne, Marcos Vinícius Mussel - ”Introdução à Arquitetura de GPUs”(2007)

[18] Viana, José Ricardo Mello - ”Programação em GPU: Passado, presente e futuro”(2009)

[19] Galeria: Evolução das Placas de Vídeo - http://www.baboo.com.br/conteudo/modelos/Galeria-Evolucao-das-Placas-de-Video_a34168_z0.aspx

[20] From Voodoo to GeForce: The Awesome History of 3D Graphics - http://www.maximumpc.com/article/features/graphics_extravaganza_ultimate_gpu_retrospective

[21] A evolução das Placas de Vídeo - http://www.baixaki.com.br/info/2314-a-evolucao-das-placas-de-video.htm

[22] História das placas de vídeo - http://forum.clubedohardware.com.br/historia-placas-video/509834?s=20823bf50aedf23b20a5576cfab9b161&amp

[23] RIVA 128 - http://pt.wikipedia.org/wiki/RIVA_128

[24] ATI Rage - http://pt.wikipedia.org/wiki/ATI_Rage

[25] Reis, David; Conti, Ivan; Venetillo, Jeronimo - "GPU - Graphics Processor Units"(2007)

[26] Comba, João Luiz Dihl; Dietrich, Carlos A.; Pagot, Christian A.; Scheidegger, Carlos E. - "Computation on GPUs: From a Programmable Pipeline to an Efficient Stream Processor"(2003)

[27] Boer, Dirk Vanden - "General-Purpose Computing on GPUs"(2005)

[28] Randy Fernando; Mark Harris; Mathias Wloka; Cyril Zeller - "Programming Graphics Hardware"(2004)

[29] Borgo, Rita; Brodlie, Ken - "State of the Art Report on GPU Visualization"(2009)

[30] Zibula, Alexander - "General Purpose Computation on Graphics Processing Units (GPGPU) using CUDA"(2009)

[31] Halfhill, Tom R. - "Parallel Processing with CUDA"(2008)

[32] Harvey, Jesse Patrick - "GPU Acceleration of Object Classification Algorithms Using NVidia CUDA"(2009)

[33] Zibula, Alexander - "General Purpose Computation on Graphics Processing Units (GPGPU) using CUDA"(2009)

[34] Vasconcellos, Felipe Brito - "Programando com GPUs: Paralelizando o Método Lattice-Boltzmann com CUDA"(2009)

[35] Grammelsbacher, Alessandro Vieira; Medrado, João Carlos Campoi - "Comparação de Desempenho entre GPGPU e Sistemas Paralelos"(2009)

[36] Martins, Deivid Cunha - "Análise Comparativa entre Ambientes de Programação para Multi-cores e GPGPUs"(2009)

[37] Dr Dobb's - CUDA, Supercomputing for the Masses - Part 1 to 16 - http://www.drdobbs.com/high-performance-computing/207200659;jsessionid=JXIAGQ3LMPWILQE1GHRSKHWATMY32JVN

[38] Parallel Numerical Methods for Partial Differential Equations - CUDA - http://www.caam.rice.edu/~timwar/RMMC/CUDA.html

[39] CUDA: Modelo de Programação Paralela - http://johntortugo.wordpress.com/2008/12/30/cuda-modelo-de-programacao-paralela

[40] CUDA: Compute Unified Device Architecture - http://www-pet.inf.ufsm.br/periodico/index.php/tutoriais/6-mensais/51-cuda-compute-unified-device-architecture

[41] The Cg Tutorial Chapter 1 - http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html

[42] Microsoft Direct3D - http://en.wikipedia.org/wiki/Microsoft_Direct3D

[43] Wynters, Erik - "Parallel Processing on NVIDIA Graphics Processing Units Using CUDA"(2010)

[44] Ryoo, Shane; Rodrigues, Christopher I.; Baghsorkhi, Sara S.; Stone, Sam S.; Kirk, David; Hwu, Wen-mei - "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA"(2008)

[45] NVIDIA - "Compute Visual Profiler - User Guide"(2011)

[46] Malony, Allen D.; Biersdorff, Scott; Shende, Sameer; Jagode, Heike; Tomov, Stanimire; Juckeland, Guido; Dietrich, Robert; Poole, Duncan; Lamb, Christopher - "Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs"(2010)

[47] PAPI - Performance Application Programming Interface - http://icl.cs.utk.edu/papi/index.html

[48] Müller, Matthias S.; Knüpfer, Andreas; Jurenz, Matthias; Lieber, Matthias; Brunst, Holger; Mix, Hartmut; Nagel, Wolfgang E. - "Developing Scalable Applications with Vampir, VampirServer and VampirTrace"(2007)

[49] TAU (Tuning and Analysis Utilities) - http://www.cs.uoregon.edu/research/tau/home.php

[50] Malony, Allen D.; Biersdorff, Scott; Spear, Wyatt; Mayanglambam, Shangkar - "An Experimental Approach to Performance Measurement of Heterogeneous Parallel Applications using CUDA"(2010)

[51] Coutinho, Bruno; Teodoro, George; Sachetto, Rafael; Guedes, Dorgival; Ferreira, Renato - "Profiling General Purpose GPU Applications"(2010)

[52] Langdon, William B. - "Performing with CUDA"(2011)

[53] Eclipse - http://www.eclipse.org

[54] Eclipse Plugin Development - http://www.eclipsepluginsite.com

[55] JNI - Java Native Interface - http://en.wikipedia.org/wiki/Java_Native_Interface

[56] NVIDIA - "CUDA Tools SDK - CUPTI User's Guide"(2011)

[57] Nickolls, John; Dally, William J. - "The GPU Computing Era"(2010)

[58] NVIDIA - "NVIDIA's Next Generation CUDA Compute Architecture: Fermi"(2009)

[59] Patterson, David - "The Top 10 Innovations in the New NVIDIA Fermi Architecture, and The Top 3 Challenges"(2009)

[60] TOP 500 Supercomputer Sites - http://www.top500.org

[61] TOP 500 - http://en.wikipedia.org/wiki/TOP500

[62] Yang, Xue-Jun; Liao, Xiang-Ke; Lu, Kai; Hu, Qing-Feng; Song, Jun-Qiang; Su, Jin-Shu - The TianHe-1A Supercomputer: Its Hardware and Software (2011)