UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DANILO FUKUDA CONRAD

Análise da Hierarquia de Memórias em GPGPUs

Trabalho de Conclusão apresentado como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Prof. Dr. Philippe Olivier Alexandre Navaux
Orientador

Porto Alegre, julho de 2010

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Conrad, Danilo Fukuda

Análise da Hierarquia de Memórias em GPGPUs / Danilo Fukuda Conrad. – Porto Alegre: Graduação em Ciência da Computação da UFRGS, 2010.

43 f.: il.

Trabalho de Conclusão (bacharelado) – Universidade Federal do Rio Grande do Sul. Curso de Bacharelado em Ciência da Computação, Porto Alegre, BR–RS, 2010. Orientador: Philippe Olivier Alexandre Navaux.

1. GPGPU. 2. CUDA. 3. Memória. 4. Processamento Paralelo. 5. Análise de Desempenho. I. Olivier Alexandre Navaux, Philippe. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Profa. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do Curso: Prof. João César Netto
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“Só sei que nada sei.”
— SÓCRATES

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS . . . 6

LISTA DE FIGURAS . . . 8

LISTA DE TABELAS . . . 9

RESUMO . . . 10

ABSTRACT . . . 11

1 INTRODUÇÃO . . . 12

2 TRABALHOS RELACIONADOS . . . 15
2.1 Análise do Desempenho em GPUs . . . 15
2.2 Análise do Desempenho de Memórias . . . 16

3 *GPGPU* . . . 17
3.1 CUDA . . . 17
3.1.1 Arquitetura . . . 18
3.1.2 Modelo de Programação . . . 19
3.1.3 Capacidade Computacional . . . 20
3.1.4 Arquitetura Fermi . . . 21
3.2 Intel Larrabee . . . 21
3.2.1 Arquitetura . . . 22
3.2.2 Modelo de Programação . . . 22
3.3 ATI Stream . . . 23
3.3.1 Arquitetura . . . 23
3.4 OpenCL . . . 25
3.4.1 Modelo de Execução . . . 25
3.5 Discussão . . . 26

4 AVALIAÇÃO DA HIERARQUIA DE MEMÓRIA . . . 27
4.1 Metodologia . . . 27
4.2 Análise da Memória Global . . . 28
4.2.1 Acesso Coalescido . . . 28
4.2.2 Testes . . . 31
4.2.3 Resultados . . . 32
4.3 Análise da Memória Compartilhada . . . 32
4.3.1 Conflitos de Banco . . . 33

4.3.2 Testes 33
4.3.3 Resultados 34
4.4 Particionamento da Memória Global 35
4.4.1 Diagonalização de Blocos 36
4.4.2 Resultado 38

5 CONCLUSÕES 39

REFERÊNCIAS 41

LISTA DE ABREVIATURAS E SIGLAS

ALU	*Arithmetic Logic Unit*
API	*Application Programming Interface*
BLAS	*Basic Linear Algebra Subprograms*
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
DMA	*Direct Memory Access*
ECC	*Error-Correcting Code*
EDP	Equações Diferenciais Parciais
EP	*Embarrassingly Parallel*
FFT	*Fast Fourier Transform*
Flops	*Floating Point Operations Per Second*
FMA	*Fused Multiply-Add*
GEMM	*General Matrix Multiply*
GPGPU	*General-Purpose computing on Graphics Processing Units*
GPU	*Graphics Processing Unit*
HPC	*High-Performance Computing*
ILP	*Instruction Level Parallelism*
IMT	*Interleaved Multithreading*
I/O	*Input/Output*
MAD	*Multiply-Add*
Mops	Milhões de Operações por Segundo
NaN	*Not a Number*
NPB	*NAS Parallel Benchmarks*
PCIe	*Peripheral Component Interconnect Express*
SIMD	*Single Instruction, Multiple Data*
SIMT	*Single Instruction, Multiple Threads*

SM	*Streaming Multiprocessor*
SMP	*Symmetric Multiprocessor*
SP	*Scalar Processor*
SSE	*Streaming SIMD Extension*
TLB	*Translation Lookaside Buffer*
VLIW	*Very Large Instruction Word*

LISTA DE FIGURAS

Figura 1.1: Comparação entre GPUs e CPUs (MFLOPs). 12
Figura 1.2: Comparação das áreas no chip de GPUs e CPUs. 13
Figura 1.3: Comparação entre largura de banda da memória de GPUs e CPUs (GB/s). 14

Figura 3.1: Modelo de hardware da arquitetura CUDA. 18
Figura 3.2: Hierarquia de *threads* no modelo de programação. 19
Figura 3.3: Mapeamento de threads para processadores da arquitetura CUDA. . . 20
Figura 3.4: Organização da arquitetura Intel Larabee 22
Figura 3.5: Organização da arquitetura ATI Stream. 24
Figura 3.6: Fluxo de dados entre memórias do dispositivo e hospedeiro. 24
Figura 3.7: Mapeamento dos elementos de OpenCL para arquitetura ATI. 25

Figura 4.1: Acessos coalescidos na memória global. 29
Figura 4.2: Acessos não alinhados na memória global. 30
Figura 4.3: Acesso a memória global para dispositivos com capacidade computacional 1.2 ou superior. 31
Figura 4.4: Cópia na memória global. 32
Figura 4.5: Acesso a memória compartilhada sem conflitos. 33
Figura 4.6: Acesso a memória compartilhada com conflitos. 34
Figura 4.7: Largura de banda na transposição de matrizes. 35
Figura 4.8: Saturação de partições na memória global. 36
Figura 4.9: Indexação dos blocos na diagonal. 37
Figura 4.10: Diagonalização dos blocos. 37
Figura 4.11: Cópia na memória global. 38

LISTA DE TABELAS

Tabela 3.1: Propriedades das Memórias nas Arquiteturas NVIDIA séries G80 e G200 20
Tabela 3.2: Mudanças significativas na arquitetura Fermi 21
Tabela 3.3: Relação entre elementos de hardware da NVIDIA e ATI 26
Tabela 3.4: Relação entre elementos do modelo de programação da NVIDIA e ATI 26

Tabela 4.1: Configuração da Plataforma de Testes 28

RESUMO

O uso de placas gráficas como elementos de co-processamento para obter alto desempenho em aplicações paralelas tem crescido cada vez mais nos últimos tempos. Diversos são os exemplos de aplicações que, fazendo uso dessas arquiteturas, obtiveram *speedups* de até duas ordens de magnitude. Isto é possível pois as GPUs (*Graphics Processing Units*) possuem uma arquitetura altamente paralela, especializada em processamento de elementos gráficos, que pode ser feito em grande parte paralelamente. Além disso, o surgimento de linguagens de programação e *frameworks* que facilitam a programação nessas arquiteturas tem sido outro fator chave em sua popularização. Entretanto, ainda é necessário um aprofundado conhecimento da arquitetura utilizada a fim de dimensionar as aplicações nela executadas. Este trabalho tem como objetivo analisar o impacto de diferentes otimizações no acesso à memória da GPU utilizado como caso de estudo a arquitetura CUDA da empresa NVIDIA.

Palavras-chave: GPGPU, CUDA, Memória, Processamento Paralelo, Análise de Desempenho.

ABSTRACT

The use of graphics cards as coprocessing elements in order to achieve high performance in parallel applications is growing in the last couple of years. There are many examples of applications which, using such architectures, have achieved up to 2 orders of magnitude speedups in their performance. This is possible because GPUs (*Graphics Processing Units*) have a highly parallel architecture, specialized in processing graphics elements, which are highly independent and can often be processed separately, in parallel. Tools for aiding developers such as higher level programming languages and frameworks are factors which helped GPUs gain popularity for general purpose computing. However, a deep knowledge of its underlying architecture is needed in order to achieve a good performance. This work aims at analyzing the impact of different optimizations in accesses to the GPU memories using the CUDA (*Compute Unified Device Architecture*) architecture from NVIDIA as a case study.

Keywords: GPGPU, Parallel Computing, Performance Analysis, CUDA, Memory.

1 INTRODUÇÃO

Há décadas técnicas que exploram o paralelismo em sistemas de computação para obter um maior desempenho vêm sendo utilizadas e, mais recentemente, essa ideia migrou para dentro dos microprocessadores com arquiteturas *multi-core*. Isto ocorreu como uma alternativa a limitações tais como a dissipação de calor, a dificuldade de se explorar ainda mais o paralelismo a nível de instrução (ILP), e o pouco avanço na latência de memória (ASA 2006).

Nesse contexto, uma outra alternativa que tem se popularizado cada vez mais para obter maior desempenho é o uso de placas gráficas para computação de propósitos gerais (GPGPU - *General Purpose computing on Graphics Processing Units*[1]). Isto ocorre pois as GPUs possuem uma arquitetura especializada para o processamento de elementos gráficos, contendo até centenas de núcleos, permitindo assim que elas alcancem um pico teórico de desempenho muito superior ao de CPUs (*Central Processing Units*) convencionais. Uma das arquiteturas em evidência atualmente é a CUDA(NVI 2009) da NVIDIA. O gráfico da Figura 1.1 apresenta uma comparação entre o pico teórico de desempenho

[1] http://www.gpgpu.org

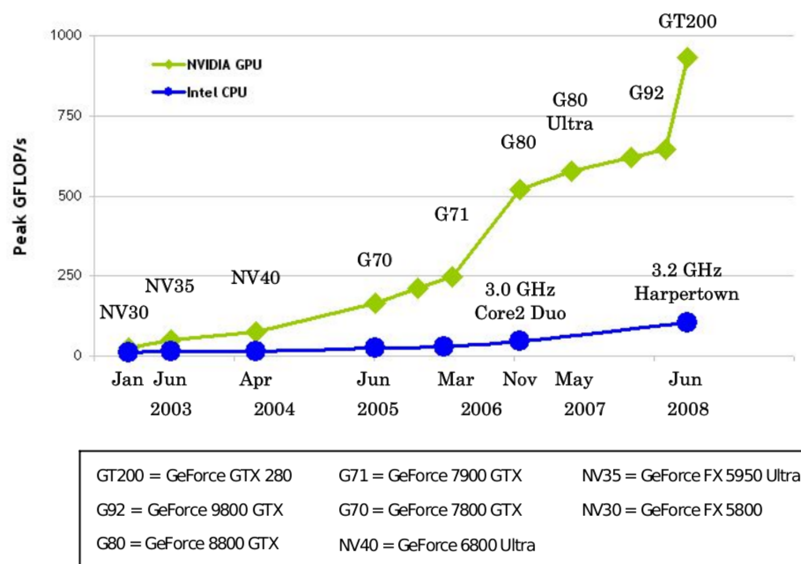

Figura 1.1: Comparação entre GPUs e CPUs em número de operações de ponto flutuante. (NVI 2009)

(em operações de ponto flutuante - Flops) de diferentes gerações de processadores da empresa Intel e GPUs da NVIDIA.

Essa diferença no desempenho é possível pois as placas gráficas dedicam muito mais transistores ao processamento de dados do que ao seu controle e armazenamento em memória *cache*, como ocorre na maioria dos microprocessadores atualmente. Isto é, as GPUs contém muitas ALUs simplificadas, podendo assim realizar um grande número de operações em paralelo sobre os mesmos dados, e, consequentemente, necessitando de menor controle para aplicaçoes com alto grau de paralelismo. A Figura 1.2 ilustra de maneira simplificada essa distribuição nos chips.

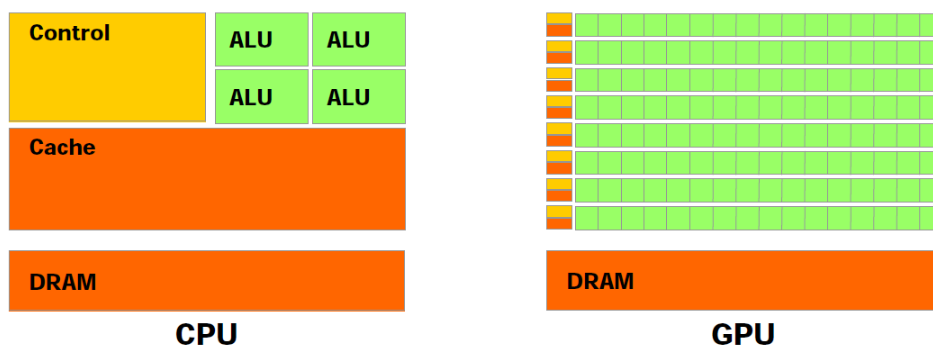

Figura 1.2: Comparação entre espaço dedicado ao processamento, controle e memória em GPUs e CPUs. (NVI 2009)

Ferramentas que facilitam o desenvolvimento nessas arquiteturas, tais como BrookGPU (BUC 2005), CUDA e OpenCL, permitiram abstrair a complexidade de mapear conceitos computacionais tradicionais para conceitos de computação gráfica. Utilizando tais ferramentas é possível desenvolver em placas gráficas como se elas fossem processadores massivamente paralelos (KIR 2010). No entanto, compreender a evolução arquitetural das GPUs facilita o entendimento do tipo de padrões computacionais que são mapeados mais facilmente para essa arquitetura, pois nem todas aplicações são apropriadas para serem nela executadas. Para melhor explorar o paralelismo por ela oferecido, é necessário um perfil de execução altamente paralelo, sendo muitas vezes necessário reescrever algoritmos já existentes. É preciso também um controle mais refinado da memória. Os dados precisam ser transferidos da memória do computador para a memória da placa de vídeo, onde devem ser dimensionados manualmente em uma hierarquia de memória de acordo com o problema.

Assim como a capacidade de processamento máxima teórica oferecida pela GPU é superior a das CPUs, sua largura de banda também apresenta uma taxa de transferência muito maior. A Figura 1.3 ilustra uma comparação entre a largura de banda das duas arquiteturas. É importante observar, no entanto, que a memória das placas gráficas é geralmente menor que a disponível no computador, e pode ser utilizada por até centenas de núcleos. Além disso, as transferências entre a memória do computador e a memória da placa gráfica são uma operação custosa, geralmente limitada pela taxa de transferência do barramento PCI-express, sendo necessário um elevado ganho no tempo de processamento na GPU para compensar pelo tempo da transferência. É possível também a utilização de técnicas como a sobreposição do tempo de transferência de dados com computação, e o uso da memória compartilhada localizada no chip da GPU, para diminuir a latência do acesso aos dados.

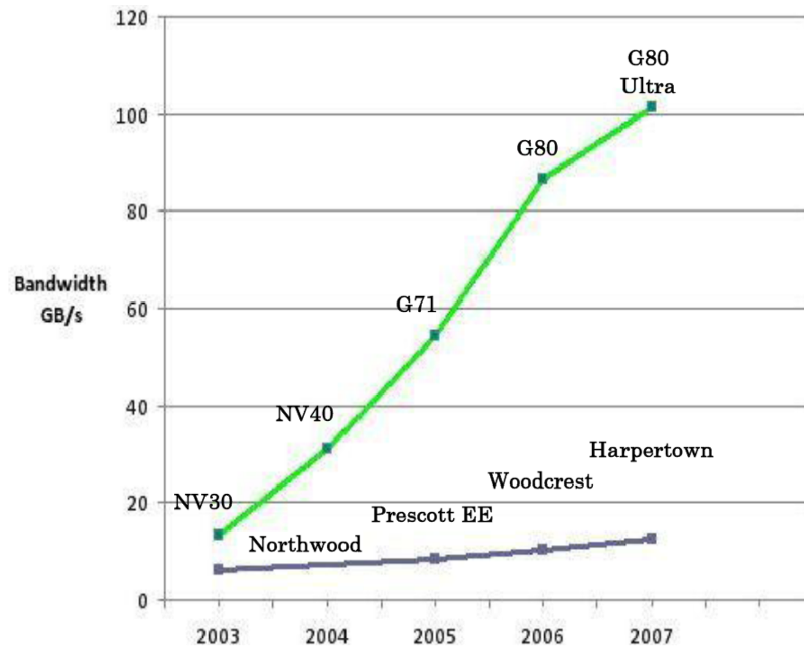

Figura 1.3: Comparação entre largura de banda da memória de GPUs e CPUs. (NVI 2009)

Este trabalho tem por objetivo analisar os diferentes níveis da hierarquia da memória das GPUs usando como caso de estudo a arquitetura CUDA da NVIDIA. O trabalho visa executar uma série de pequenos *benchmarks* com o intuito de obter informações sobre o impacto de diferentes otimizações no acesso à memória global e à memória compartilhada, para diferentes tamanhos de problema, utilizando a largura de banda como métrica de avaliação.

O próximo capítulo apresenta alguns trabalhos relacionados na área de GPGPU e análise de desempenho. No Capítulo 3 são apresentadas algumas soluções para computação de propósitos gerais em placas gráficas, bem como a arquitetura utilizada no presente trabalho. A metodologia dos testes executados são descritas no Capítulo 4, seguidas por uma análise dos resultados. Por fim, no Capítulo 5, são apresentadas as conclusões tiradas do trabalho e eventuais possibilidades de continuação do estudo realizado.

2 TRABALHOS RELACIONADOS

Com o crescente aumento do uso de GPUs para aplicações de propósitos gerais nos últimos anos, diversos são os trabalhos nessa área. Neste capítulo são apresentados alguns destes trabalhos, abordando principalmente a análise de desempenho na arquitetura de GPUs, técnicas para otimização do desempenho nessa arquitetura, e a análise de desempenho de memórias em diferentes arquiteturas.

2.1 Análise do Desempenho em GPUs

Em (VOL 2008) são realizados diversos *microbenchmarks* a fim de inferir informações sobre a arquitetura da GPU. Com as informações obtidas, foram reescritas rotinas para álgebra linear (BLAS), obtendo-se um desempenho até 90% superior ao da biblioteca CUBLAS fornecida pela própria fabricante (NVIDIA). Para obter esses resultados eles utilizaram a GPU como uma unidade de processamento vetorial *multicore* com *multithreading*, aplicando otimizações já conhecidas para computadores vetoriais. Além disso, foram desafiados alguns dos princípios de programação sugeridos pela própria fabricante, relativos a quantidade de *threads* que devem ser executadas, e que memórias oferecem melhor desempenho.

A fim de otimizar os diversos parâmetros possíveis ao dimensionar problemas em diferentes arquiteturas *multicore*, (DAT 2008) utiliza uma técnica chamada de *autotuning*. Essa técnica consiste em uma busca heurística automatizada, que realiza *benchmarks* no espaço de otimizações possíveis, selecionando a combinação que gera o melhor resultado. Em seu estudo foi utilizado como exemplo o cálculo de Equações Diferenciais Parciais (EDPs) através de computações de *stencils*. São apresentadas em (LI 2009) técnicas de *autotuning* para álgebra linear densa (DLA), mais especificamente da rotina de multiplicação de matrizes (GEMM). Obtendo resultados próximos aos limites teóricos da arquitetura, argumenta-se que através dessa técnica é possível obter um elevado desempenho que poderá ser facilmente portado para futuras gerações da arquitetura. Em (ASA 2006) é reforçada a ideia do uso de *autotuners* como uma alternativa a compiladores para gerar código otimizado e portável para diferentes arquiteturas *multi-core*.

Em (RYO 2008) é feita uma análise de como diversas otimizações interagem com a arquitetura GeForce 8800 GTX da NVIDIA. Através de uma análise estática do código, são criadas métricas com o intuito de reduzir o espaço de busca das otimizações. Dentre as métricas utilizadas estão a eficiência do *kernel* (código executado na GPU) e a utilização dos seus recursos de computação. Utilizando essas métricas o autor pode reduzir as com-

binações de configurações possíveis, restando apenas aquelas que se encontram na curva ótima da eficiência de Pareto. Isso reduziu o espaço de otimizações em até 98%, ainda possibilitando encontrar a configuração ótima para cada aplicação por eles estudada.

No trabalho de (PIL 2009) é feita uma análise para verificar a compatibilidade da arquitetura CUDA com algoritmos da classificação **Dwarf Mine** (ASA 2006). Para isto foram utilizados os *NAS Parallel Benchmarks* que podem ser facilmente mapeados para essas classes de problemas. Foram portados os *benchmarks* EP (*Embarrassingly Parallel*) e FT (*Fast Fourier Transform*), obtendo-se ganhos no desempenho de até $21\times$ e $3\times$, respectivamente. Deste modo foi comprovada a compatibilidade das categorias de algoritmos *Map Reduce* e *Spectral Methods* com a arquitetura CUDA.

2.2 Análise do Desempenho de Memórias

Uma avaliação do compartilhamento das memórias *cache* em arquiteturas *multi-core* é feita em (ALV 2009). Dentre os fatores analisados encontram-se: a quantidades de núcleos compartilhando a memória *cache* L2; os níveis de hierarquia na memória; o tamanho e associatividade da *cache*. Foi utilizado como carga de trabalho *NAS Parallel Benchmarks* (NPB).

No trabalho de (RYO 2008a), são destacados diversos princípios de otimização para a arquitetura CUDA utilizando como exemplo uma multiplicação de matrizes em uma placa GeForce 8800 GTX. Dentre as otimizações destaca-se o uso das memórias de baixa latência e o uso massivo de *threads* na GPU para ocultar a latência da memória global. Os autores apresentam os resultados das otimizações em uma grande gama de aplicações com resultados obtendo *speedups* de até $400\times$. São apresentados tanto os resultados para a execução unicamente do *kernel* quanto para toda a aplicação, ressaltando os fatores considerados como gargalos na arquitetura para executar as aplicações.

Em (VOL 2008a) é feita uma análise minuciosa da arquitetura de memória da GPU utilizando *microbenchmarks*. Através dos testes realizados o autor verificou a presença de diferentes níveis de *cache* na memória, o tamanho dessas memórias e a organização. Foi inferida também a presença de uma TLB (*Translation Lookaside Buffer*), seu tamanho e a suas possíveis associatividades. Outros testes realizados nesse trabalho incluem o custo de inicialização da funçao executada na GPU (*kernel*) e taxas de transferência entre CPU e GPU.

O problema de má distribuição nos acessos à memória global chamado de *partition camping*, e também contemplado neste trabalho, é descrito em (RUE 2009). É nele também apresentada a solução através da indexação dos blocos de modo diagonal, a fim de melhor distribuir os acessos.

No trabalho de (HE 2007) é feita uma análise das operações de *gather* e *scatter* em placas gráficas. Devido à natureza randômica dos acessos dessas operações, implementações sem qualquer otimização fazem um mal uso da largura de banda e consequentemente, não conseguem esconder a latência da memória. Para solucionar tal problema é proposto um modelo que divide as operações em etapas (passos) a fim de melhorar a localidade nos acessos. Os resultados obtidos apresentam ganhos de $2-7\times$ comparados ao desempenho em CPUs *multi-core* com 4 núcleos.

3 *GPGPU*

O uso de GPUs para computação de propósitos gerais (*General Purpose Computing on Graphics Processing Units*) é uma ideia que vem sendo explorada há algum tempo no meio científico com o objetivo de obter ganhos de desempenho na execução de aplicações paralelas (HAR 2002; BUC 2004; OWE 2005).

Por se tratar de uma arquitetura especializada em processar elementos gráficos, que possuem pouca dependência de dados, as GPUs possuem um grau de paralelismo muito maior que o de CPUs multi-core convencionais. Entretanto, a fim de programar em tais arquiteturas antigamente, era necessário mapear conceitos de computação convencional para elementos de computação gráfica tais como pixels e texturas, oferecidos pelas APIs das placas de vídeo (HAR 2005; HIM 2006; GOV 2006; GOV 2007).

Este capítulo apresenta as principais alternativas que surgiram com o objetivo de atenuar a complexidade de programação em GPUs. Serão apresentadas nas subseções seguintes as soluções da NVIDIA, Intel e AMD, respectivamente. Também será dedicado um capítulo ao *framework* OpenCL, do grupo Khronos, que permite o desenvolvimento em plataformas heterogêneas. Um enfoque especial será dado para a arquitetura CUDA, que foi a utilizada para o presente trabalho.

3.1 CUDA

A alternativa para computação de propósitos gerais em GPUs da empresa NVIDIA é a arquitetura **CUDA**[1] - *Compute Unified Device Architecture*. Ela permite utilizar a placa de vídeo como um co-processador para execução de aplicações em um ambiente heterogêneo, utilizando tanto a CPU quanto a GPU. Para isso, são oferecidas aos desenvolvedores extensões e *wrappers* para linguagens de programação como C, C++ e FORTRAN, com o intuito de facilitar a migração de algorítmos e programas já existentes.

A arquitura CUDA se baseia em três elementos chave: uma hierarquia de *threads*, memórias compartilhadas e barreiras de sincronização. Através deles é possível explorar o paralelismo das aplicações em diferentes níveis de granularidade, com primitivas simples de programação.

Nas subseções seguintes são apresentados os detalhes da arquitetura CUDA. Primeiramente, é dado um enfoque aos componentes de *hardware* que a compõem. Em seguida, são apresentados as abstrações fornecidas em nível de *software*, correlacionando com

[1] http://www.nvidia.com/cuda

os componentes previamente vistos. Na sequência, é apresentada a nova geração da arquitetura CUDA, chamada de Fermi. Por fim, é feita uma análise das características da arquitetura, relacionando as diferentes gerações.

3.1.1 Arquitetura

Cada placa gráfica compatível com CUDA possui até dezenas de multiprocessadores chamados de *Streaming Multiprocessors* (**SM**). Eles são considerados elementos de processamento escaláveis, ou seja, quanto maior sua quantidade, maior a capacidade de processamento paralelo da GPU. Até centenas de *threads* são executadas em cada SM, em um modelo que se assemelha à classificação SIMD, da Taxonomia de Flynn, chamado pela NVIDIA de **SIMT** (*Single Instruction - Multiple Thread*). Os multiprocessadores podem ter 8 ou 32 núcleos de processamento, de acordo com a geração da sua arquitetura. Esses núcleos também são chamados de *Scalar Processors* (**SP**) ou de **CUDA cores**. A Figura 3.1 ilustra essa organização hierárquica dos processadores.

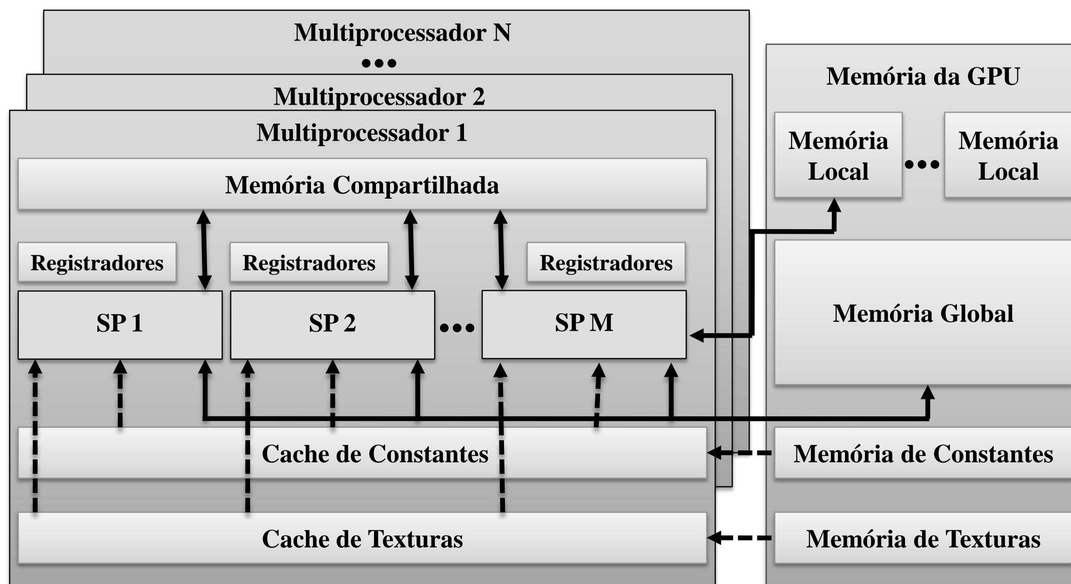

Figura 3.1: Modelo de hardware da arquitetura CUDA.

A unidade de escalonamento nos SMs é o **warp**: um grupo de 32 *threads* que executam a mesma instrução. Quando uma das *threads* diverge das outras em seu caminho de execução, é necessário executar serialmente cada caminho tomado, desabilitando o restante das *threads*, o que acarreta em redução no desempenho.

Também na Figura 3.1 é possível visualizar a hierarquia de memória e como ela se relaciona com os elementos de processamento. Cada SP tem acesso a registradores de 32 *bits*. Para núcleos em um mesmo multiprocessador, é possível acessar dados em uma memória compartilhada gerenciável de baixa latência, semelhante a um *scratch-pad*(RYO 2008a). Os SPs podem também acessar a memória global, que é maior porém possui uma latência superior em até 2 ordens de magnitude. Encontram-se também no SM memórias cache especiais para constantes e texturas, que são acessíveis somente para leitura.

3.1.2 Modelo de Programação

No modelo de programação é feita a distinção entre o que é executado na CPU, chamado de hospedeiro (*host*), e na GPU, chamda de dispositivo (*device*). Utilizando C para CUDA é possível definir funções chamadas de *kernels*, que serão chamadas na CPU e executadas em paralelo na GPU por até milhares de *threads*. O modelo de programação CUDA permite agrupar as *threads* em uma hierarquia a fim de permitir um maior controle sobre a granularidade dos problemas nela resolvidos e maior escalabilidade. Essa hierarquia é apresentada na Figura 3.2. As *threads* são agrupadas em blocos, que por sua vez são reunidos em grades. É possível indexar blocos e *threads* como elementos de até duas e três dimensões, respectivamente. Isto permite mapear mais facilmente algoritmos que trabalham com matrizes ou volumes. As *threads* em um mesmo bloco podem se comunicar através da memória compartilhada, localizada no SM, que possui um tempo de acesso próximo ao dos registradores. Além disso, é possível sincronizá-las com a primitiva *syncthreads()* a um custo próximo de apenas uma instrução.

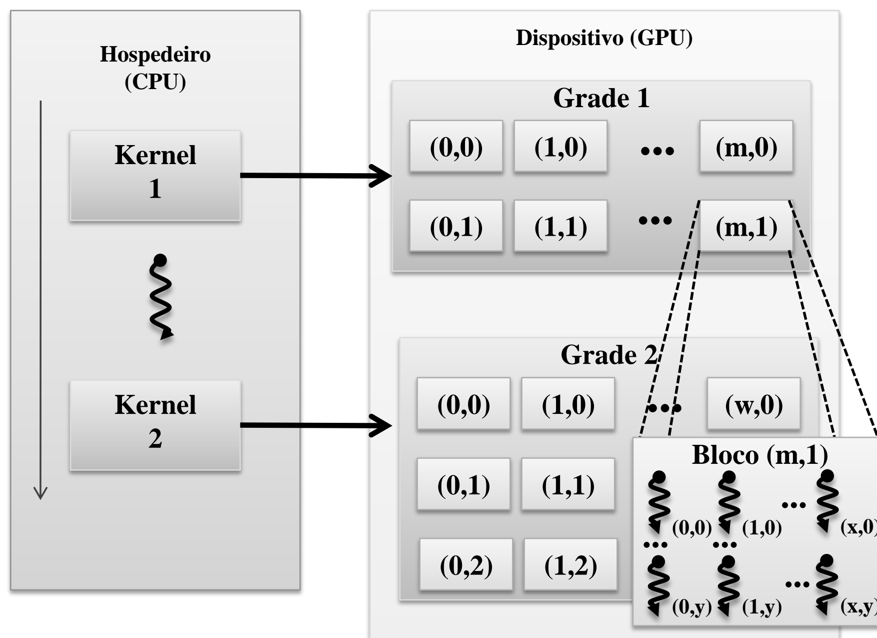

Figura 3.2: Hierarquia de *threads* no modelo de programação.

É possível ver também na Figura 3.2 a utilização de CUDA para computação heterogênea. A GPU é utilizada como um dispositivo de co-processamento, executando trechos com alto paralelismo, enquanto a CPU pode se encarregar de trechos sequenciais. Ao executar uma chamada de *kernel*, os blocos dentro da grade são alocados para os multiprocessadores vistos anteriormente na Figura 3.1, sendo que cada SM pode executar mais de um bloco simultaneamente.

A Figura 3.3 reune as informações anteriormente apresentadas, mostrando como é realizado o mapeamento dos modelos de *software*, apresentados ao desenvolvedor, para o *hardware*. As *threads* são executadas nos núcleos chamados de *Scalar Processors*, os blocos nos multiprocessadores, e uma grade é enviada para execução em uma GPU.

Ao programar em CUDA, é necessário copiar os dados da memória principal da CPU

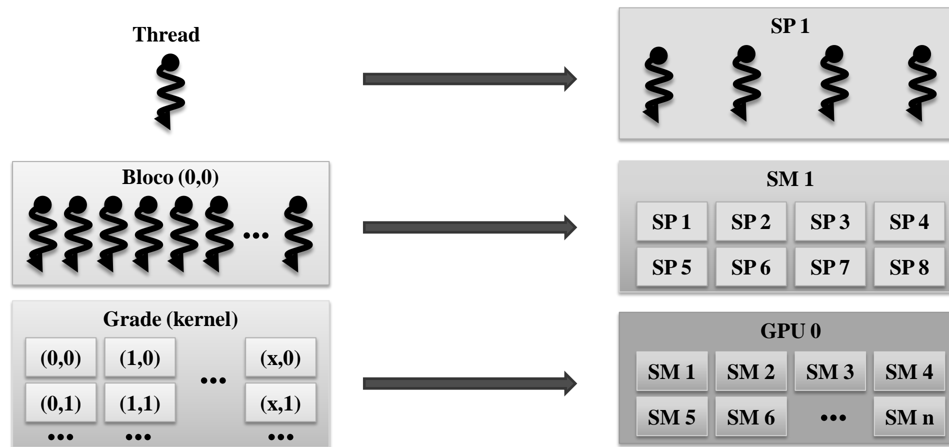

Figura 3.3: Mapeamento de threads para processadores da arquitetura CUDA.(PIL 2009)

para a memória da GPU explicitamente, com uma função semelhante à *memcpy()*. Isso restringe a taxa de transferência de dados ao limite do barramento PCI-Express, que pode ser até uma ordem de magnitude inferior ao barramento interno da memória da placa gráfica(VOL 2008). A Tabela 3.1 resume parte do que já foi mencionado anteriormente e apresenta outras informações relativas às memórias na GPU.

Tabela 3.1: Propriedades das Memórias nas Arquiteturas NVIDIA séries G80 e G200

Memória	**Localização**	**Tamanho**	**Latência**	**Somente Leitura**	**Escopo**
Global	GPU	até 1024MB	$400-600$ ciclos	não	global
Local	GPU	depende da global	$400-600$ ciclos	não	função
Registradores	SM	até 64KB por SM	≈ 0 ciclos	não	função
Compartilhada	SM	16KB por SM	≥ 4 ciclos	não	função
Constante	SM (cache)	64KB ao todo	$0-600$ ciclos	sim	global
Textura	SM (cache)	depende da global	$0-600$ ciclos	sim	global

O acesso a memórias que não se encontram no SM é geralmente custoso, sendo pelo menos uma ordem de magnitude superior. A memória de constantes e textura pode ser armazenada em *cache*, mas quando ocorrem faltas(*misses*), é necessário buscar os dados fora do SM com o mesmo custo de acesso que a memória global. A quantidade de memória global e o número de registradores disponíveis varia de acordo com o modelo da placa. A memória local é utilizada quando não há registradores suficientes para todas as *threads* executadas no multiprocessador e é necessário armazenar os dados na memória global.

3.1.3 Capacidade Computacional

As diferentes gerações da arquitetura CUDA são agrupadas por suas capacidades de computação - *Compute Capability*. Elas são definidas por um número de revisão maior e outro menor. As características descritas acima enquadram-se em maior parte nas placas

de capacidade computacional 1.x, isto é, seu número maior de revisão é 1. As revisões menores indicam pequenas melhorias como aumento no banco de registradores, ou novos padrões de acesso à memória, que serão importantes nos próximos capítulos do trabalho.

3.1.4 Arquitetura Fermi

A arquitetura Fermi apresenta diversas mudanças em seu projeto, sendo considerada uma nova revisão das capacidades computacionais (*Compute Capability* 2.0). Dentre suas maiores mudanças estão o aumento no número de núcleos nos multiprocessadores, o aumento no desempenho em operações de precisão dupla, inclusão de memória *cache* em 2 níveis e adição de ECC à memória(NVI 2009a). A Tabela 3.2 relaciona algumas de suas características, relacionando-a com sua antecessora (Série G200), citada anteriormente.

Tabela 3.2: Mudanças significativas na arquitetura Fermi

	Arquitetura	
Característica	G200	Fermi
Transistores	1.4 bilhões	3 bilhões
Núcleos (SPs)	240	512
Pico Teórico de Dupla Precisão	30 operações FMA/ciclo	256 operações FMA/ciclo
Pico Teórico de Precisão Simples	240 operações MAD/ciclo	512 operações FMA/ciclo
Suporta Padrão IEEE 754-2008 [2]	Não	Sim
Escalonadores de Warps por SM	1	2
Cache L1 (por SM)	–	Configurável 16K ou 48K
Cache L2 (SPs)	–	768K
Suporte a ECC	Não	Sim
Kernels Concorrentes	Não	Sim
Endereçamento Load/Store	32-bit	64-bit

O aumento na capacidade de computação para operações de precisão dupla, e o suporte ao padrão IEEE para operações de ponto flutuante representam mudanças significativas na arquitetura, visto que essas limitações restringiam a sua utilização para certos algoritmos (BRE 2008). Além disso, essa mudança permite uso extensivo da nova arquitetura para aplicações de alto desempenho (HPC), já comprovada pelo surgimento de um *cluster* heterogêneo com tal arquitetura em segundo lugar na lista dos 500 supercomputadores mais rápidos [3] atualmente. Outras mudanças arquiteturais significativas foram a adição de um escalonador de *warps*, a adição de memória *cache* e a possibilidade de execução de *kernels* em paralelo.

3.2 Intel Larrabee

A arquitetura Larrabee foi projetada pela empresa Intel como uma solução *many-core* para ser utilizada tanto em processamento gráfico quanto para computação de alto desem-

[3]http://www.top500.org

penho (*High-Performance Computing* - **HPC**). Utilizando a arquitetura x86, múltiplos núcleos, e provendo uma *cache* L2 coerente, ela visa oferecer um elevado desempenho na execução de aplicações paralelas com baixa complexidade para os desenvolvedores (SEI 2008).

3.2.1 Arquitetura

Os núcleos da arquitetura Larrabee são baseados na arquitetura Intel Pentium, isto é, sem execução fora de ordem. Eles foram modificados, sendo acrescidas unidades de processamento de vetores. Cada núcleo seu é capaz de processar 16 operações de ponto flutuante simultaneamente, de maneira similar, mas $4\times$ mais rápido que as instruções SSE nos processadores x86 convencionais(SEI 2008). Outras características incluidas foram: instruções que permitem controle explícito da cache, suporte a *multithreading* e a *pthreads* (POSIX *threads*).

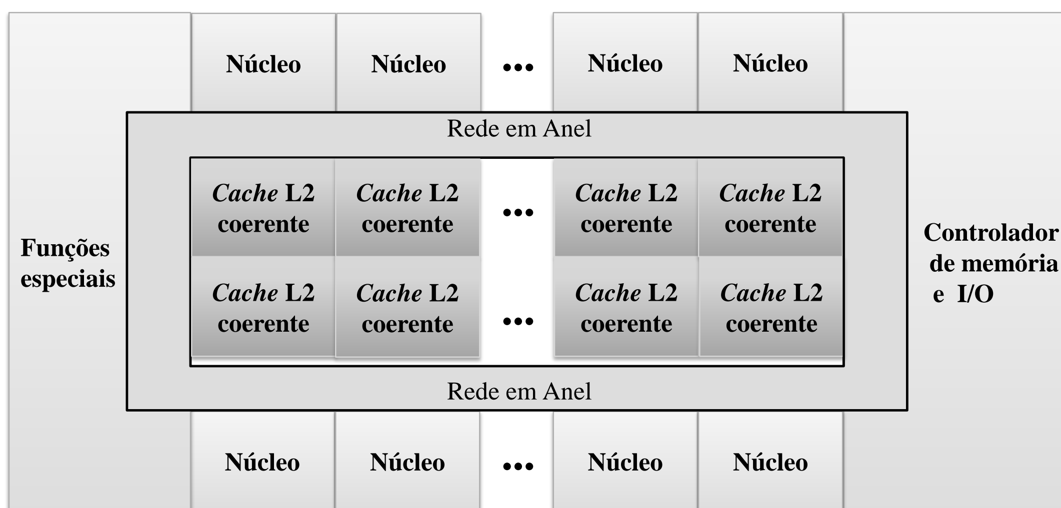

Figura 3.4: Organização da arquitetura Intel Larabee, adaptado de (SEI 2008).

A comunicação dos núcleos com os controladores de memória, entrada e saída é feita através de um rede de interconexão em anel de alta largura de banda, como ilustrado na Figura 3.4. A quantidade de núcleos varia de acordo com a implementação. Cada núcleo pode acessar seu próprio subconjunto de memória *cache* L2 coerente a fim de prover uma alta largura de banda e simplificar a sincronização e compartilhamento de dados. Diferentemente das GPUs convencionais, várias funções para manipulações gráficas não são implementadas em hardware, mas sim em software, a fim de manter a arquitetura simples e extensível.

3.2.2 Modelo de Programação

Um dos maiores atrativos da arquitetura Larrabee é sua compatibilidade parcial com o conjunto de instruções x86 para arquiteturas multi-core. Isto quer dizer que diversas aplicações não precisarão ser reescritas para serem nela executadas. A API da arquitetura foi extendida para permitir definir afinidade de *threads* com núcleos específicos. É possível também programar as unidades de processamento vetorial diretamente utilizando C++ ou *assembly*.

O compatilhamento dos dados é feito transparentemente através do seu modelo hierarquico de memórias *cache* coerentes. O controle para seções críticas é feito através de mecanismos já conhecidos como *locks* e semáforos. É possível também realizar controle explícito para cópias dos dados nos diferentes níveis de memória. Outra característica interessante no modelo de programação é o suporte a estruturas de dados irregulares, possível pois a arquitetura provê um suporte eficiente a operações *scatter-gather*. Exemplo de tais estruturas de dados são árvores de ponteiros, estruturas de dados espaciais ou matrizes esparsas grandes.

3.3 ATI Stream

O ambiente da AMD (Advanced Micro Design) para desenvolvimento em GPGPU é conhecido como **ATI Stream SDK**. Ele incorpora as placas da AMD compatíveis com a tecnologia ATI Stream[4] e o *framework* **OpenCL** (*Open Computing Language*) para computação em arquiteturas heterogêneas(MUN 2009). O *framework* OpenCL será abordado na seção seguinte em mais detalhes.

3.3.1 Arquitetura

As GPUs nessa arquitetura são denominadas de *Compute Devices*, ou seja dispositivos de computação. Elas possuem grupos de **unidades de computação** (*Compute Units*, antigamente chamados de *SIMD Engines*), que por sua vez contém diversos núcleos (*stream cores*), conforme é ilustrado na Figura 3.5. Os núcleos são responsáveis pela execução de *kernels* em fluxos de dados independentes (*streams*). Os menores elementos de computação nessa arquitetura são as ALUs dentro de cada núcleo, chamados de *processing elements*, ou elementos de processamento.

Cada núcleo (*stream core*) é organizado como um processador VLIW no qual até cinco operações escalares podem ser submetidas para execução(ATI 2010). Os elementos de processamento podem executar operações de ponto flutuante com precisão simples ou de inteiros. Um dos elementos é capaz de executar operações especiais como seno, cosseno e logaritmo. Para executar operações de precisão dupla, é necessário utilizar dois ou quatro elementos de processamento conectados para executar apenas uma única instrução, reduzindo deste modo o desempenho dessa arquitetura para tais operações em até duas ou quatro vezes. A quantidade de Unidades Computacionais varia de acordo com o dispositivo.

É possível distinguir 6 domínios de memória:

- Memória Privada (*private*) - de acesso individual para elementos de processamento (ALUs);
- Memória Local - de acesso individual para unidades de processamento (Cores), compartilhada entre suas ALUs;
- Memória Global - acessível por todos elementos;
- Memória de Constantes - acessível para escrita para CPU e somente para leitura pela GPU;

[4] http://ati.amd.com/technology/streamcomputing/

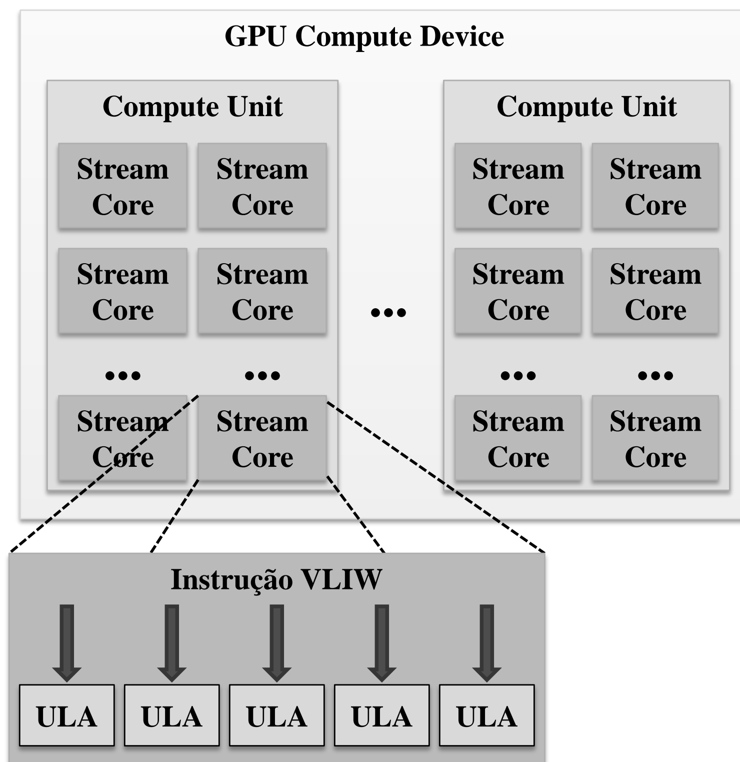

Figura 3.5: Organização da arquitetura ATI Stream.

- Memória do Hospedeiro - de acesso privativo pela CPU;

- Memória PCIe Compartilhada - região de memória do hospedeiro acessível para leitura e escrita tanto pela GPU quanto pela CPU. É utilizada para realizar transferências.

Essas memórias também são organizadas hierarquicamente, sendo o acesso cada vez mais rápido nas memórias mais próximas aos elementos de processamento. Isto é, acessar a memória privada e local é rápido enquanto que acessos à memória global e à memória PCIe compartilhada podem ser de até 1 e 2 ordens de magnitude mais demorados, respectivamente. A Figura 3.6 ilustra um típico fluxo de dados a partir do hospedeiro (*host*), até os elementos de processamento da GPU, na memória privada (*private*).

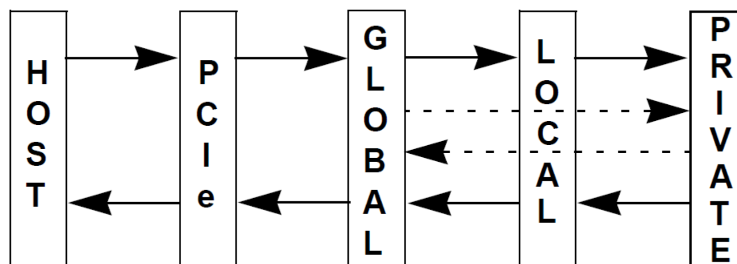

Figura 3.6: Fluxo da transferência de dados entre a memória entre hospedeiro e dispositivo. (ATI 2010)

3.4 OpenCL

O OpenCL[5] (*Open Computing Language*) é um padrão aberto criado com o objetivo de possibilitar o desenvolvimento de aplicações que executem em um ambiente de computação heterogêneo, isto é, com GPUs ou qualquer outro co-processador(MUN 2009). Nele é feita também a distinção entre o que é executado na CPU (hospedeiro) e fora dela, nos dispositivos. O hospedeiro define contextos de execução para as funções que executarão em paralelo nos dispositivos, chamadas de *kernels*.

3.4.1 Modelo de Execução

A sua principal característica para prover paralelismo tanto a nível de tarefa quanto a nível de dados está em como os *kernels* são executados. Quando um kernel é submetido para execução, é definido um **espaço de indexação** (*index space*) de até 3 dimensões, também chamado de *NDRange*.

Cada instância executada do kernel em diferentes elementos de processamento é chamada de **item de trabalho** (*work item*), e é identificada pelo seu ponto no espaço de indexação, que provê um ID único para ela. Todos os itens de trabalho executam o mesmo código, podendo seguir caminhos diferentes durante a execução e operar sobre diferentes dados.

Os itens de trabalho são reunidos em **grupos de trabalho** (*work groups*), que provêem uma decomposição de maior granularidade no domínio. Cada grupo possúi também um identificador único. Os itens de trabalho em um mesmo grupo são executados concorrentemente em elementos de processamento de uma unidade de computação, como ilustrado na Figura 3.7.

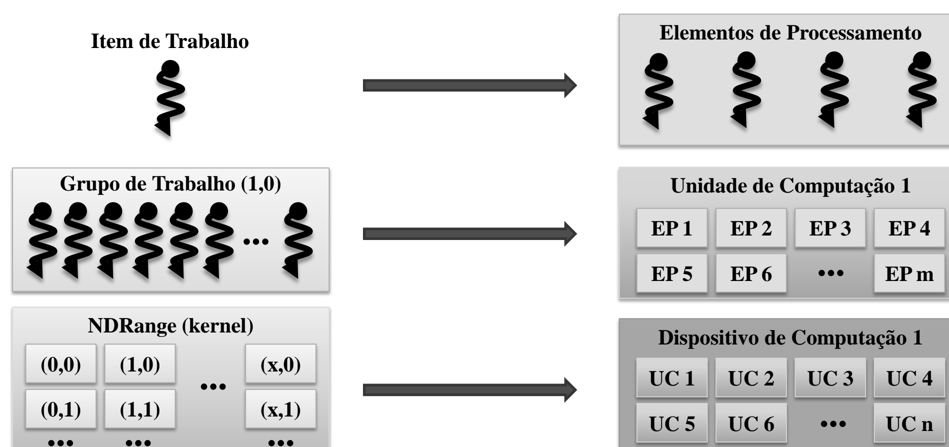

Figura 3.7: Mapeamento dos elementos de OpenCL para arquitetura ATI.

É possível utilizar OpenCL tanto na arquitetura CUDA quanto na arquitetura ATI Stream, sendo que ele é a alternativa adotada pelo segundo como interface padrão de programação.

[5] http://www.khronos.org/opencl/

3.5 Discussão

As três arquiteturas apresentadas nesse capítulo podem ser utilizadas não apenas para processamento dedicado de gráficos mas também como co-processadores a fim de obter um maior desempenho. Para isto elas oferecem uma interface simplificada de programação, com o intuito de abstrair a complexidade da arquitetura.

Tabela 3.3: Relação entre elementos de hardware da NVIDIA e ATI

Terminologia CUDA	**Terminologia ATI Stream-OpenCL**
SP / *CUDA Core*	*Stream Core*
Streaming Multiprocessor (SM)	*Compute Unit / SIMD Engine*
Device	*Compute Device*
Host	*Host*

É possível traçar facilmente paralelos entre as arquiteturas da NVIDIA e ATI, sendo que ambas evoluiram de processadores gráficos dedicados e atualmente podem também ser utilizadas para processamento de propósitos gerais. As Tabelas 3.3 e 3.4 apresentam os termos que cada uma delas utiliza para os elementos de *software* e de *hardware*, respectivamente.

Devido a suas semelhaças arquiteturais, é possível o surgimento de *frameworks* como OpenCL, que oferecem uma interface única para programação, utilizando as mesmas abstrações. É possível notar que as duas arquiteturas fazem um forte uso da divisão hierárquica para sub-dividir os problemas. Além disso, os níveis de memória e os padrões de acesso que serão descritos no próximo capítulo também podem ser mapeados para a arquitetura da ATI. Consequentemente, acredita-se que o trabalho aqui apresentado, desenvolvido na arquitetura CUDA, pode ser facilmente estendido para a arquitetura ATI, utilizando OpenCL.

Tabela 3.4: Relação entre elementos do modelo de programação da NVIDIA e ATI

Terminologia CUDA	**Terminologia ATI Stream-OpenCL**
thread	*Work-Item*
Bloco	*Work-Group*
Grade	*NDRange*
Memória Global	Memória Global
Memória de Constantes	Memória de Constantes
Memória Local	Memória Privada
Memória Compartilhada	Memória Local

4 AVALIAÇÃO DA HIERARQUIA DE MEMÓRIA

Neste capítulo serão descritos os testes que foram realizados no presente trabalho a fim de analisar o impacto das otimizações de memória em GPGPUs. É importante ressaltar que o modelo de computação nessa arquitetura necessita de um controle manual refinado do armazenamento dos dados. É preciso primeiramente copiar os dados da memória principal do computador para a memória global da GPU, sendo que essa transferência é limitada de acordo com o barramento PCIe. Em seguida, os dados são manipulados na placa gráfica, transferidos da memória global, que é maior e possui alta latência, para a memória compartilhada, que é geralmente pequena e de baixa latência. Na memória compartilhada os dados podem ser acessados pelas *threads* que ocupam o mesmo bloco ao custo semelhante de um acesso a registradores, se não houver conflito no acesso.

Os testes e otimizações apresentados neste capítulo focam na memória global da placa gráfica e na memória compartilhada, localizada no *chip*. Não será analisada a arquietura Fermi nem as memórias dedicadas para constantes e texturas. As seções seguintes apresentam otimizações que devem ser realizadas para melhor utilizar a hierarquia de memória, maximizando a taxa de transferência. A próxima seção descreve a metodologia de testes utilizada e a descrição do ambiente no qual os testes foram executados. Em seguida serão abordadas as otimizações nos acessos à memória global e à memória compartilhada, comparando seu desempenho com acessos não otimizados. Por fim, são apresentadas otimizações específicas para o problema utilizado em nossa análise, que é a transposição de matrizes.

4.1 Metodologia

Os testes foram realizados utilizando C para CUDA, e os *timers* de eventos contidos em sua API, que fornecem precisão de meio microssegundo (NVI 2009b). São utilizados dois casos de teste: cópia de matrizes e transposição de matrizes, ambos com endereços de origem e destino em posições diferentes da memória. Para simplificar foram utilizadas apenas matrizes quadradas, mas a generalização pode ser facilmente aplicada. As matrizes são compostas por palavras de 4 *bytes* (*floats*).

Será utilizada como métrica para o presente estudo a largura de banda obtida na transferência dos dados, em *gigabytes* por segundo (GB/s). Ela pode ser obtida multiplicando-se a quantidade de elementos da matriz pelo tamanho da palavra de cada elemento (em *bytes*), e por 2, pois realizaremos as operações de leitura e escrita. A fórmula abaixo

representa essa equação.

$$LarguradeBanda = \frac{N^2 \times 2 \times 4}{1024^3 \times t}$$

O tempo t foi obtido realizando chamadas ao *kernel* dentro de laços, executando no mínimo 20 iterações, e dividindo o tempo final obtido pela quantidade de iterações. Os resultados apresentados nas seções seguintes apresentam confiança de 95% e erro relativo de 5%. O ambiente no qual os testes foram executados é descrito em detalhes abaixo, na Tabela 4.1.

Tabela 4.1: Configuração da Plataforma de Testes

Configuração	
Processador	Intel Core 2 Duo E8500 $3,16$ GHz (Socket LGA775, FSB 1333MHz, 6MB Cache)
Memória	OCZ Reaper HPC Edition 4GB (2048MB $\times$ 2, DDR2 1066MHz)
Placa Mãe	MSI P7N SLI Platinum (chipset NVIDIA nForce 750i SLI, PCIe $\times$ 16)
Placa Gráfica	GeForce GTX 280 (GX-280N-ZDDU, 1GB 512-bit GDDR3)
Sistema Operacional	Ubuntu 9.04 32 bits (Kernel Linux 2.6.28-15-generic)
CUDA Driver	3.0
Compilador	nvcc versão 0.2.1221

A placa de vídeo utilizada para os testes possui 30 multiprocessadores, cada um contendo 8 SPs, resultando em 240 núcleos ao todo. Como sua capacidade computacional é 1.3, o número máximo de *threads* por bloco é 512, e o tamanho da memória compartilhada para cada SM é 16KB. O procedimento de testes será descrito mais detalhadamente dentro de cada subseção posterior.

4.2 Análise da Memória Global

Os acessos à memória global são realizados por *warps*, e são divididos em duas requisições de memória, uma para cada *half-warp*. Na arquitetura estudada o tamanho do *warp* é de 32 *threads*, e, consequentemente, o *half-warp* é composto por 16 *threads*. É possível realizar transações de 34, 64 e 128 bytes na memória global, sendo que os endereços devem estar alinhados a valores múltiplos desses tamanhos.

4.2.1 Acesso Coalescido

O acesso coalescido à memória consiste em realizar uma única transação na memória, recuperando um segmento para todas as *threads* de um *half-warp*. Isto pode ser feito

naturalmente quando todas as *threads* estão alinhadas à memória: elas acessam elementos em um mesmo segmento, e apenas uma transação é realizada como ilustrado na Figura4.1. Caso haja divergência, isto é, se algumas das *threads* não acessarem a memória, assim é realizada apenas uma transação.

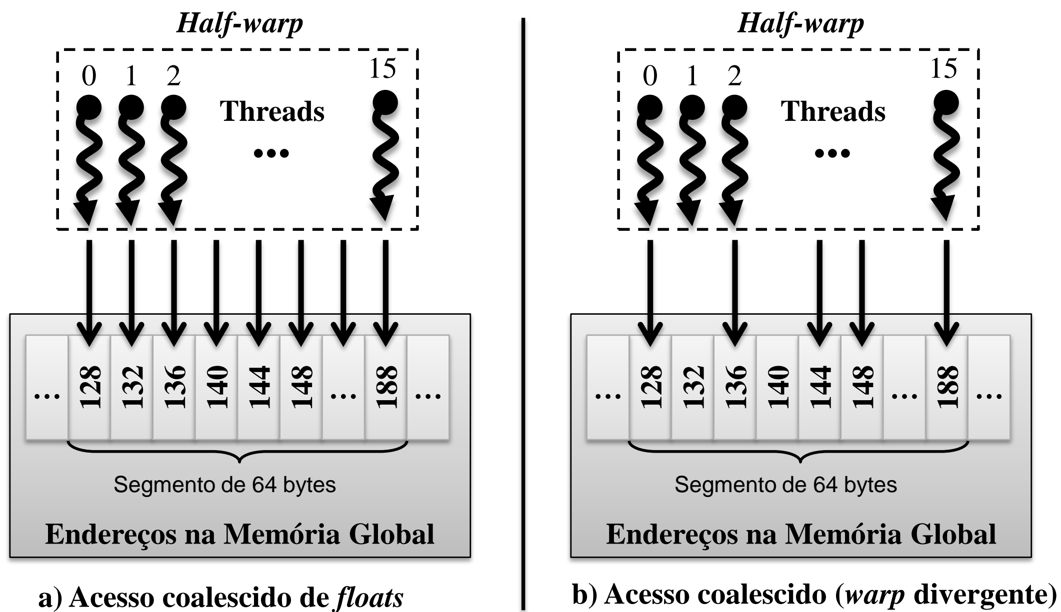

Figura 4.1: Acessos coalescidos na memória global.

Quanto maior a capacidade computacional do dispositivo (ver Seção 3.1.3), mais facilmente a memória pode ser coalescida. Isto é, nos dispositivos mais antigos (Compute Capability < 1.2) as restrições para coalescer a memória são as mais rigorosas. Se o acesso a memória for coalescido para capacidades computacionais antigas, também será nas mais novas.

Para que todas as threads de um *half-warp* possam realizar apenas uma ou duas transferências de modo coalescido as seguintes restrições devem ser obedecidas (para CC $<$ 1.2):

- As threads devem acessar:
  - Palavras de 4 bytes, resultando em uma transação de 64 bytes, ou
  - Palavras de 8 bytes, resultando em uma transação de 128 bytes, ou
  - Palavras de 16 bytes, resultando em duas transações de 128 bytes;
- Todas as palavras acessadas devem estar no mesmo segmento, de tamanho igual ao da transação realizada;
- As threads devem acessar as palavras sequencialmente, isto é, a enésima thread no *half-warp* deve acessar a enésima posição do segmento;

Se algum dos items requeridos acima não for cumprido, é feita uma transação de memória para cada *thread* no *half-warp*, reduzindo assim a vazão. A Figura 4.2 apresenta

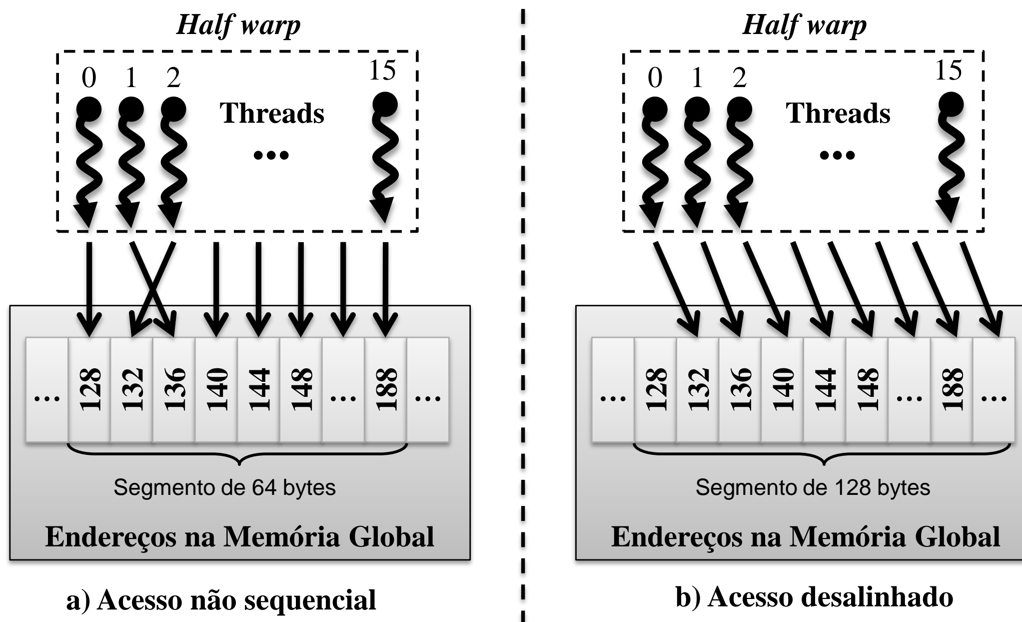

Figura 4.2: Acessos não alinhados na memória global.

casos em que não é possível coalescer o acesso para dispositivos com capacidade computacional menor que 1.2, sendo necessário realizar 16 transações ao todo.

Os dispositivos mais recentes, com uma capacidade computacional superior a 1.2 possuem menos restrições para coalescer os acessos a memória. É possível acessar qualquer palavra dentro do segmento, em qualquer ordem, incluindo acessos a uma mesma palavra, e uma única transação de memória para cada segmento endereçado pelo *half-warp* é realizada. O exemplo ilustrado na Figura 4.2 pode ser coalescido em tais dispositivos. Mais detalhadamente, é utilizado o seguinte algoritmo pra determinar as transações realizadas:

1. É localizado o segmento no qual se encontra o endereço acessado pela *thread* ativa de menor índice. O tamanho do segmento depende do tamanho da palavra lida:
   - Palavras de 1 byte acessam segmentos de 32 bytes,
   - palavras de 2 bytes, acessam segmentos de 64 bytes, e
   - palavras de 4, 8 ou 16 bytes, acessam segmentos de 128 bytes.
2. São buscadas todas as *threads* ativas com endereços no mesmo segmento.
3. É feita uma redução do tamanho da transação, caso possível:
   (a) Se a transação é de 128 bytes e apenas a metade superior ou inferior dos endereços é acessada, a transação é reduzida a 64 bytes,
   (b) Se a transação é de 64 bytes (originalmente ou após a redução anterior) e apenas a metade superior ou inferior dos endereços é acessada, a transação é reduzida a 32 bytes.
4. Executa a transação e marca as *threads* que acessaram o segmento como inativas.
5. Repete até todas as *threads* no *half-warp* concluirem seus acessos.

Abaixo, na Figura 4.3, é ilustrado um exemplo de como é executado esse algoritmo. É considerado um *half-warp* de tamanho 16, acessando palavras de 4 bytes (*floats*, por exemplo). Primeiramente a *thread* de índice 0 acessa o endereço 116. Como as palavras são de 4 bytes, o segmento é de 128 bytes e o endereço acessado encontra-se no primeiro segmento. As outras *threads* no mesmo segmento são as de índice 1 e 2. Em seguida, tenta-se realizar a redução da transação, que pode ser feita em apenas 32 bytes, que é a menor transação. A transação é concluida para esse segmento e os passos são executados para o restante das *threads* ativas. Por fim, são realizadas duas transações, uma de 32 bytes e outra de 64 bytes.

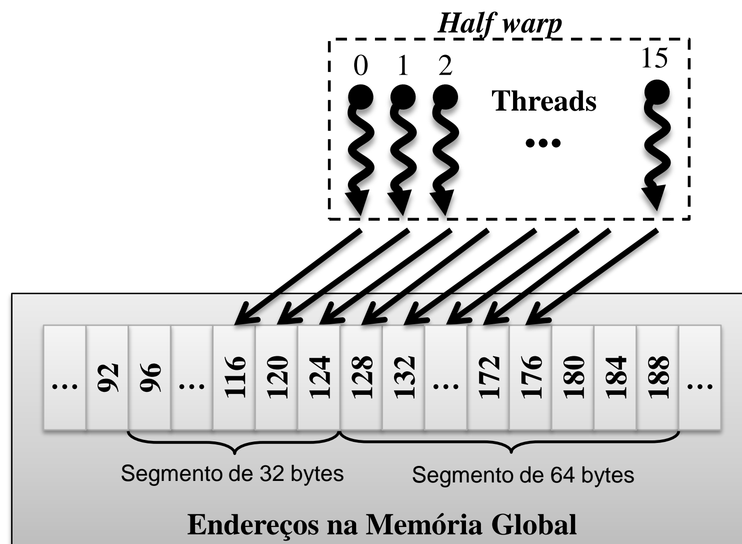

Figura 4.3: Acesso a memória global para dispositivos com capacidade computacional 1.2 ou superior.

4.2.2 Testes

Segundo a fabricante, é possível obter uma taxa de transferência superior em até uma ordem de magnitude utilizando acessos coalescidos. Os testes realizados nesta seção têm por objetivo analisar o impacto do acesso não coalescido ou desalinhado. Para isto são realizados seguintes tipos de acesso:

- Cópia direta de dados coalescidos;
- Cópia desalinhada;
- Cópia com espaçamentos entre os endereços.

Foi realizada a cópia de uma matriz de *floats* quadrada, de tamanho 2048×2048. Para realizar a cópia, foram criados blocos de tamanho 16×16, onde cada *thread* realiza a cópia de um elemento da matriz. A cópia desalinhada representa o deslocamente de apenas uma posição para cada elemento acessado, conforme representado anteriormente na Figura 4.2(b). A cópia com espaçamentos (*strides*) representa acessos que indexam os elementos a cada *n* espaços. Isto é, ao invés de acessar os elementos sequencialmente, eles são acessados em espaços determinados pelo tamanho do *stride*.

4.2.3 Resultados

Os resultados são apresentados na Figura 4.4. O limite teórico é definido pela largura e frequência do barramento. É importante ressaltar que a implementação inicial realizada neste teste não foi totalmente otimizada, sendo que não é este o foco do teste, mas sim apresentar a diferença entre o acesso coalescido e acessos não coalescidos.

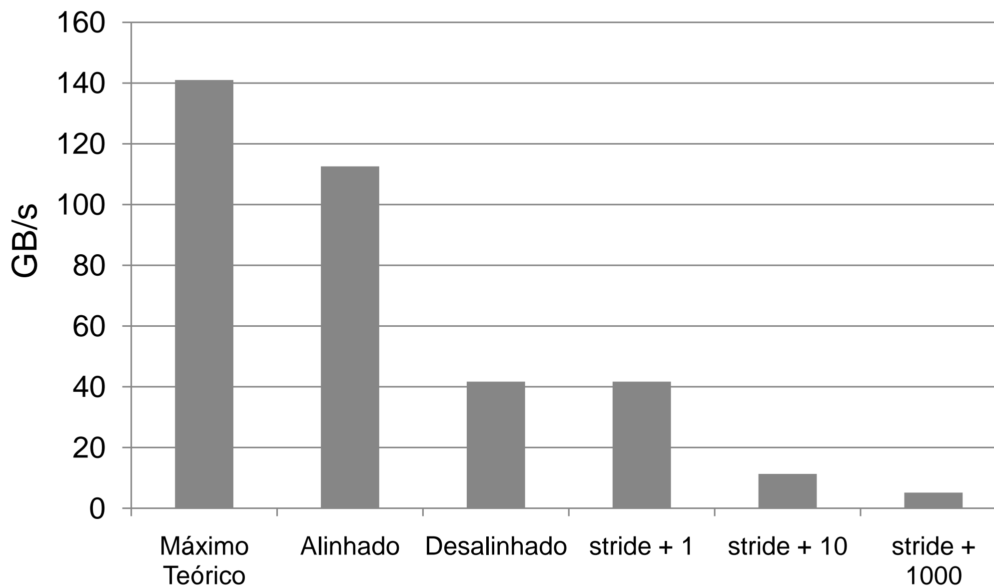

Figura 4.4: Cópia na memória global.

É possível ver que quanto mais desalinhados estão os dados maior é a queda no desempenho. Isto ocorre pois é necessário realizar mais transações na memória, sendo aproveitados cada vez menos dados em cada transação. O desempenho é reduzido a menos da metade ao realizar um deslocamento de apenas uma posição. Para as cópias com espaçamento, quanto maior o espaço entre cada elemento, maior a quantidade de transações necessárias, e, consequentemente, pior o desempenho final.

4.3 Análise da Memória Compartilhada

A memória compartilhada fica localizada em cada multiprocessador, dentro do chip. Consequentemente, seu acesso é muito mais rápido se comparado ao da memória local ou global, como já foi visto anteriormente. A fim de obter uma maior largura de banda, a memória compartilhada é divida em 16 módulos de mesmo tamanho chamados de **bancos**, que podem ser acessados simultaneamente. Deste modo, qualquer acesso composto por n endereços acessando n bancos distintos pode ser realizado simultaneamente, resultando em uma largura de banda teoricamente até 16 vezes maior que a de um único módulo. A Figura 4.5 ilustra dois exemplos de acessos sem conflitos a memória compartilhada para palavras de 32 bits.

Contudo, se dois endereços em uma requisição de memória se encontram no mesmo banco, ocorre um conflito no acesso, e é preciso serializá-lo. A requisição é então dividida em requisições separadas, sem conflito, reduzindo assim a largura de banda teórica

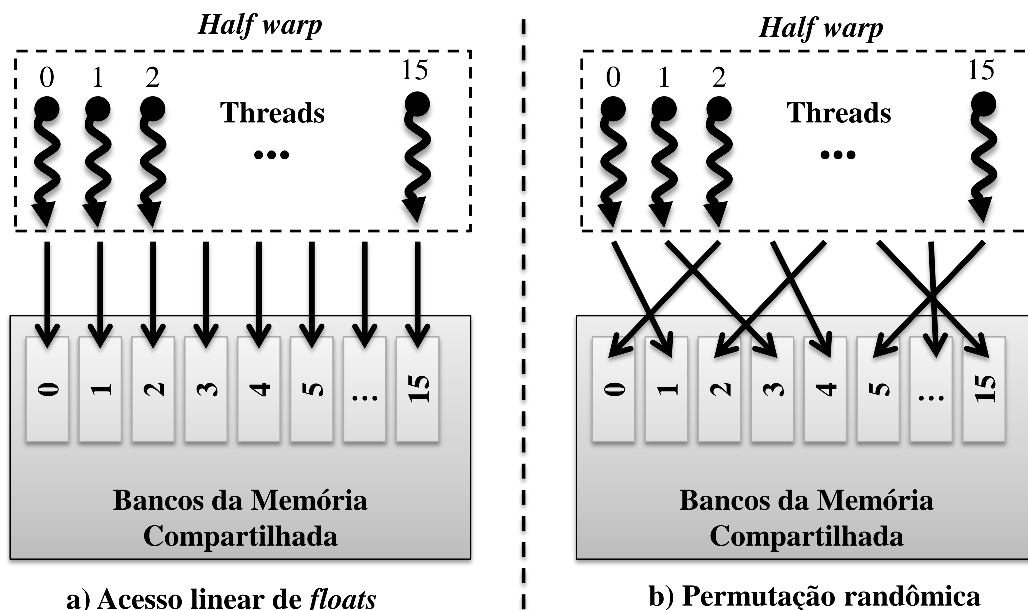

Figura 4.5: Acesso a memória compartilhada sem conflitos.

de modo proporcional a quantidade de conflitos. As subseções seguintes descrevem os conflitos em detalhes e apresentam testes que analisam seu impacto.

4.3.1 Conflitos de Banco

Os bancos da memória compartilhada são organizados de modo que acessos consecutivos a palavras de 32 bits são alocados para bancos consecutivos. Uma requisição para acessar a memória compartilhada feita dentro de um *warp* é dividida em duas, uma para cada *half-warp*, que são executadas independentemente. Deste modo, não ocorrem conflitos entre *threads* pertencentes a metades diferentes do *warp*. Caso múltiplas *threads* tentem escrever em uma mesma posição da memória compartilhada utilizando uma instrução não atômica, apenas uma *thread* em cada *half-warp* executará a operação, e a ordem de qual será executada por último é indefinida.

Um padrão comum de acesso a memória compartilhada, acessando palavras de 32 *bits*, é apresentado na Figura 4.6. Cada *thread* possui um número que a identifica(*tid*), e é utilizado para indexar um array dado um espaçamento **s**. No caso da imagem, esse espaçamento equivale a 2 e a cada 8 *thread* será acessado o mesmo banco, causando deste modo um conflito de dois acessos. Consequentemente, será preciso executar duas transações, uma para cada conflito. Deste modo, é possível ver que a *thread* (*tid*) e a *thread* (*tid* + *n*) acessarão o mesmo banco quando $s \times n$ for múltiplo da quantidade de bancos, isto é, 16 em nosso caso. Caso o espaçamento fosse de 4, 8 ou 16, ocorreriam, respectivamente, 4, 8 e 16 conflitos.

4.3.2 Testes

Os testes apresentados aqui analisam não apenas a memória compartilhada, mas também a memória global. É utilizado como caso de teste a transposição de matrizes, que é um exemplo particularmente interessante pois se a operação é realizada diretamente na

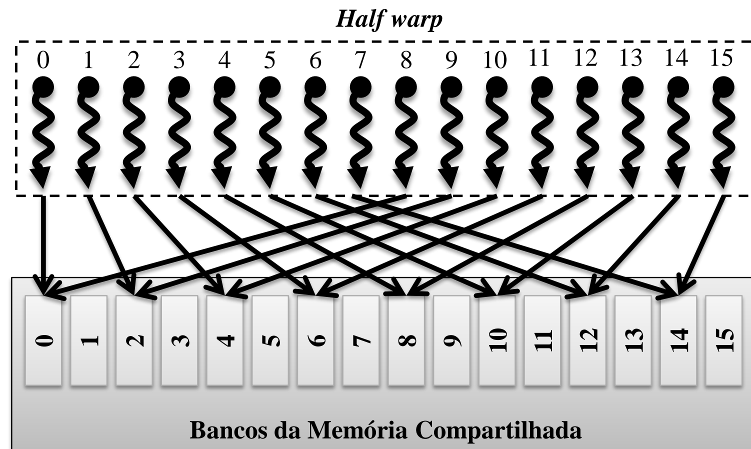

Figura 4.6: Acesso a memória compartilhada com conflitos.

memória global, a leitura ou a escrita deverão ocorrer de modo não coalescido, pois serão acessados endereços não contíguos.

Para solucionar esse problema, é utilizada a memória compartilhada. Primeiramente são lidos os dados de maneira coalescida e então escritos, transpostos, na memória compartilhada. Como a memória compartilhada é de baixa latência, e não sofre penalidades por acessos não coalescidos, é possível realizar essas operações com um custo muito baixo. É necessário então sincronizar as *threads* no mesmo bloco para garantir que todas realizaram a leitura. Os dados da memória compartilhada são então lidos e escritos de modo linear novamente na memória global, de modo coalescido.

A fim de paralelizar a operação, foram utilizados blocos de tamanho 16×16, isto é, com 256 *threads*. A memória compartilhada foi alocada em uma matriz de 16×16 palavras de 4 *bytes* e os acessos para transposição são feitos em colunas, isto é, com espaçamentos de 16 palavras, causando consequentemente 16 conflitos. Para resolver esses conflitos foi utilizada uma técnica chamada de *padding*, que consiste simplesmente em aumentar a quantidade de memória compartilhada em uma coluna. Com essa operação, a indexação da matriz passa a mapear as colunas com um elemento a mais, e deste modo, eliminando conflitos nos acessos por colunas.

4.3.3 Resultados

O gráfico abaixo apresenta os resultados obtidos, sendo que no eixo horizontal estão representados os diferentes tamanhos de matrizes utilizados, e no eixo vertical as taxas de transferência obtidas em GB/s.

Como esperado, o desempenho da cópia coalescida e da transposição de matrizes sem otimizações, isto é, sem acessos coalescidos, apresenta até uma ordem de magnitude de diferença para matrizes grandes. Ao utilizar a memória compartilhada para realizar leituras e escritas coalescidas na memória global, o desempenho aumentou em mais de 100%, comparado a transposição direta. Entretanto, o desempenho ainda está muito inferior à cópia coalescida, apresentando quedas acentuadas em matrizes múltiplas de 512 elementos. A próxima seção aborda este problema e a solução encontrada. A eliminação de conflitos na memória compartilhada apresentou em alguns casos desempenho superior em até 25%, sendo que seu efeito não pode ser percebido nitidamente nas quedas bruscas

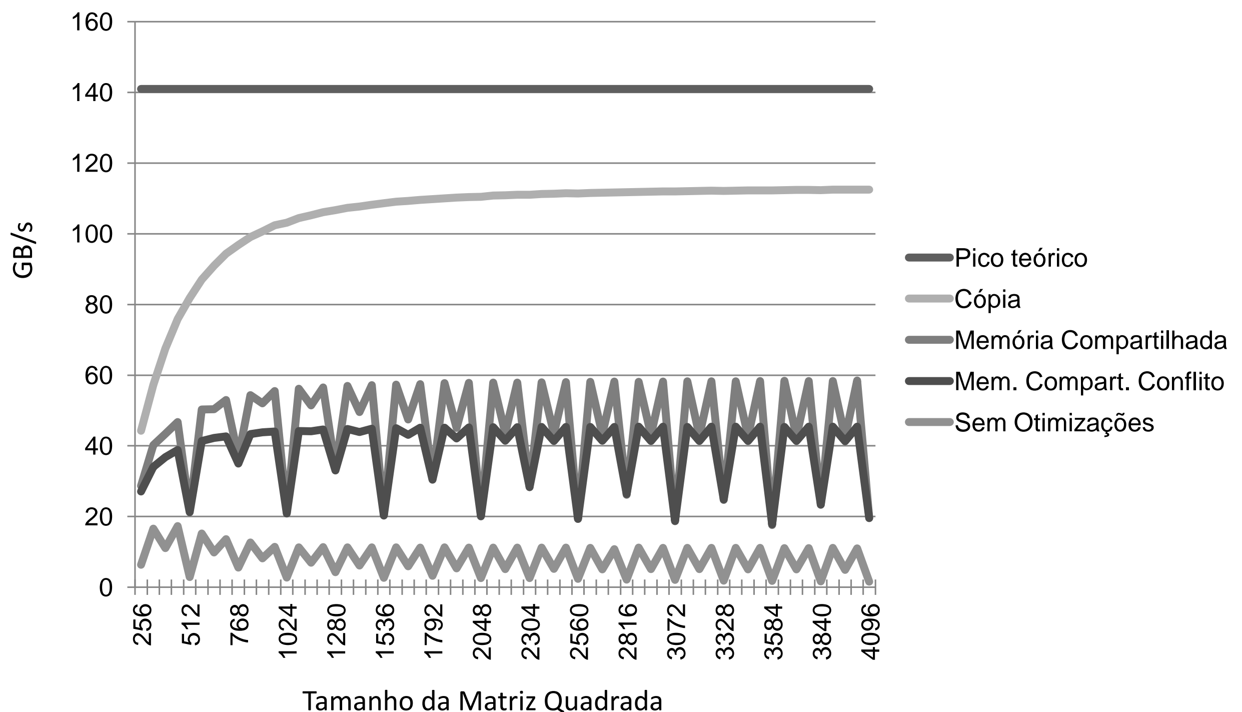

Figura 4.7: Largura de banda na transposição de matrizes.

de desempenho anteriormente mencionadas.

4.4 Particionamento da Memória Global

Assim como a memória compartilhada é dividida em bancos de 32 *bits* de largura, a memória global é dividida em partições de 256 *bytes* de largura. A quantidade de partições varia de acordo com o modelo da placa gráfica, sendo que a placa utilizada neste estudo (pertencente a série 200) possui 8 partições. A fim de otimizar o acesso à memória global, as *warps* ativas devem estar devidamente distribuidas nessas partições. Caso haja desbalanceamento em alguma partição, isto é, múltiplas *warps* requisitando acessos simultaneamente na mesma partição, esta partição pode saturar, fazendo com que as *warps* tenham que esperar até suas requisições serem atendidas, enquanto outras partições encontram-se livres.

Enquanto o acesso coalescido trata de acessos a memória global realizados dentro de um *half-warp*, o particionamento se preocupa com acessos entre *half-warps* ativos. Deste modo, é importante saber como os blocos são escalonados nos multiprocessadores a fim de entender melhor o comportamento dos *warps*.

Quando um *kernel* é executado, a ordem em que os blocos são enviados aos multiprocessadores é determinada pelo ID do bloco, que é definido como: *BlockIdx.x* + *gridDim.x* $\times$ *BlockIdx.y*, onde gridDim.x é o tamanho da grade na dimensão x. Este é um ordenamento por colunas (*row-major*) dos blocos na grade. Assim que a ocupação máxima for alcançada, blocos adicionais são enviados aos multiprocessadores conforme necessário. A ordem e a velocidade com que esses blocos serão concluídos não pode ser determinada. Deste modo, os blocos ativos são inicialmente contíguos, mas vão se tornando cada vez

menos conforme a execução do *kernel* avança.

Utilizando como exemplo uma matriz de 2048×2048 elementos dividida em elementos menores que serão processados pelos blocos, chamados de *tiles*, de tamanho 32×32, e analisando como estes elementos são mapeados para as partições da memória, é possível ver como ocorre tal problema (Figura 4.8).

0	1	2	3	4	5
64	65	66	67	68	69
128	129	130	...		

0	64	128			
1	65	129			
2	66	130			
3	67	...			
4	68				
5	69				

Figura 4.8: Saturação de partições na memória global.

Com oito partições de largura de 256 *bytes*, todos os dados em espaços de 2048 bytes (ou 512 *floats*) são mapeados para a mesma partição. Qualquer matriz de elementos de 4 *bytes* como *floats*, por exemplo, e um número de colunas múltiplo de 512, possuirá colunas com elementos mapeados a uma única partição. No exemplo acima, as duas primeiras colunas mapeiam elementos para a mesma partição. Combinando o meapeamento dos elementos para as partições e como os blocos são escalonados, é possível ver que blocos concorrentes acessarão as tiras (tiles) na matriz de entrada percorrendo as linhas, resultando em uma carga balanceada para as partições. Já na matriz de saída, os elementos são acessados por colunas, resultando no problema anteriormente encontrado.

Uma solução possível para o problema assemelha-se a utilizada com a memória compartilhada para evitar conflitos no banco, isto é, adicionar uma coluna à matriz de modo que os acessos passem a ser feitos sem conflitos. Isto é possível, e seu resultado pode ser visualizado no mesmo gráfico da Figura 4.7. Contudo, essa solução não pode ser adotada em todos os domínios de problema, e consequentemente faz-se necessária uma alternativa mais flexível. Tal alternativa é apresentada na subseção seguinte, juntamente com os resultados obtidos após sua aplicação.

4.4.1 Diagonalização de Blocos

O problema encontrado nos testes da seção anterior pode ser resolvido utilizando uma técnica chamada de diagonalização de blocos (RUE 2009). Ela consiste em alterar a maneira como os blocos são indexados, e, consequentemente, escalonados. Embora o programador não possa definir a ordem em que os blocos serão escalonados, que é definida pelo identificador do bloco, é possível definir uma outra interpretação para o identificador.

Como as threads e blocos possuem componentes x e y para identificá-los, geralmente é assumido que eles se encontram em um plano cartesiano, o que não precisa ser necessariamente verdade. Uma maneira de evitar a sobrecarga de partições na memória global é usando uma interpretação diagonal dos componentes x e y que identificam o bloco: O componente y representa diferentes cortes diagonais na matriz, e o elemento x indica a distância ao longo de cada diagonal. A Figura 4.9 ilustra ambas as interpretações dos

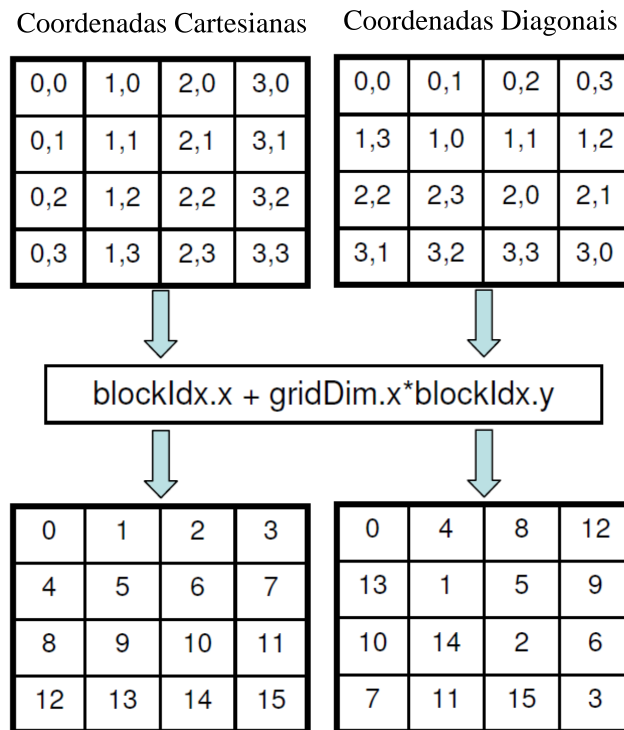

Figura 4.9: Indexação dos blocos na diagonal.

componentes do identificador de bloco utilizando uma matriz 4×4. Acima são mostradas as coordenadas com dois componentes, cartesiana e diagonal, e abaixo como elas são mapeadas para uma única dimensão.

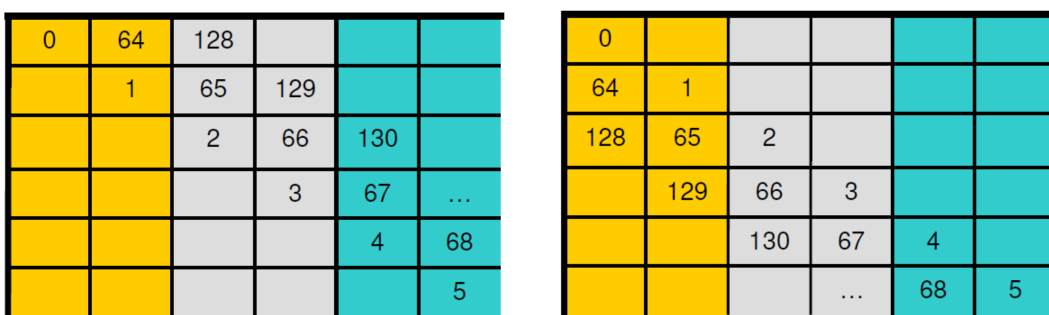

Figura 4.10: Diagonalização dos blocos.

Um dos principais atrativos desse mapeamento é a facilidade com que ele pode ser implementado, bastando recalcular os componentes x e y do bloco e substituí-los no trecho restante do código. Isto facilita não apenas a criação de novos kernels mas também a alteração de kernels já existentes. Analisando o exemplo anterior que mapeava os elementos de coordenadas cartesianas para as partições, utilizando agora coordenadas diagonais, é possível ver que não ocorre mais a saturação de uma única partição (Fig. 4.10).

4.4.2 Resultado

Alterando o kernel do experimento anterior, indexando as matrizes de maneira diagonal, foram obtidos os seguintes resultados, com e sem conflitos na memória compartilhada.

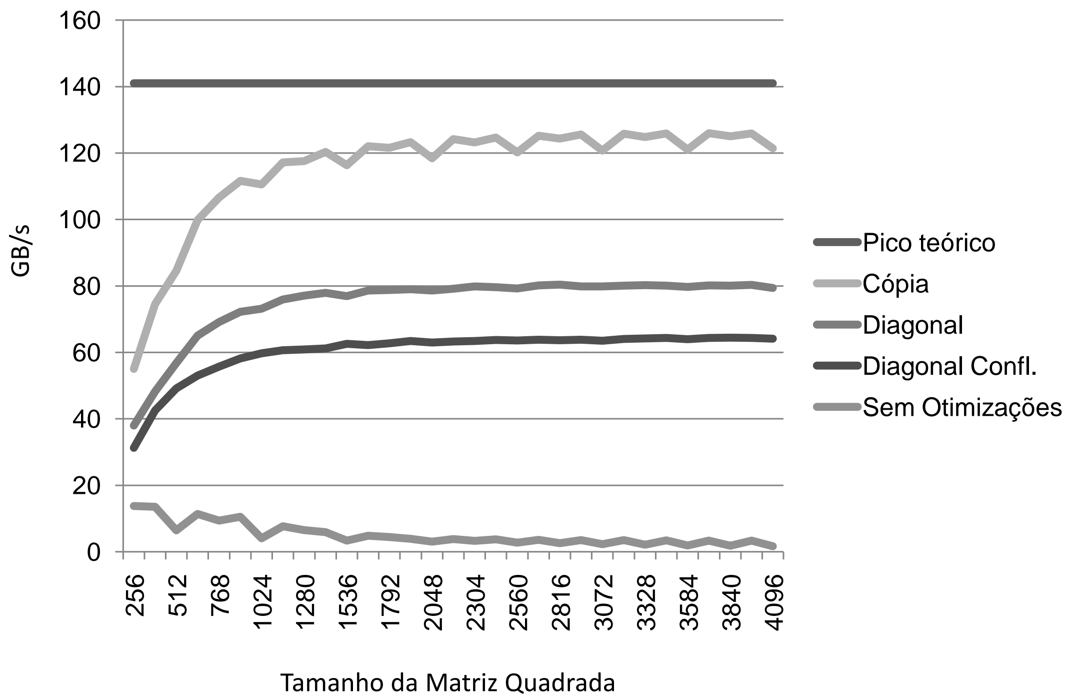

Figura 4.11: Cópia na memória global.

Ao alterar a maneira como os blocos são indexados no kernel, foi possível diminuir o agrupamento de elementos em uma única partição, obtendo assim um considerável aumento na largura de banda, além de uma taxa praticamente constante de transferência para problemas grandes de diferentes tamanhos. É importante ressaltar que diferentemente da quantidade de bancos de memória compartilhada, a quantidade de partições na memória global varia de acordo com os modelos existentes.

Agora que a transferência não apresenta mais a degradação no desempenho devido a saturação em partições na memória global, é mais visível a diferença causada com conflitos na memória compartilhada. No caso do exemplo utilizado, o conflito foi de 16 transações, isto é, uma para cada *thread*, e a queda no desempenho final foi em torno de 23%.

5 CONCLUSÕES

Neste trabalho foram apresentados diferentes otimizações nos acessos à memória global e à memória compartilhada na arquitetura CUDA utilizando como caso de estudo uma placa GTX 280. Através de *benchmarks* que realizaram a cópia e a transposição de matrizes entre essas memórias, foi possível analisar o impacto de cada otimização, de acordo com o momento em que ela foi aplicada.

Para a cópia direta de dados, percebeu-se até uma ordem de magnitude de diferença entre a cópia direta coalescida e a cópia de dados com grandes espaçamentos entre seus elementos. Isto ocorre pois mesmo nas placas gráficas mais recentes os acessos só podem ser coalescidos quando os dados ainda apresentam alguma localidade, isto é, quando se encontram em segmentos próximos. Devido à maior flexibilidade para coalescer os acessos nos modelos mais recentes da arquitetura, o impacto de acessos desalinhados e com pequenos espaçamentos foi uma redução de aproximadamente 60% na taxa de transferência. Como os modelos mais antigos não permitem o acesso coalescido para dados não alinhados, acredita-se que o impacto em sua transferência seja equivalente ao de um grande espaçamento nas placas mais recentes, pois em ambos casos é preciso realizar uma transferência para cada segmento lido e transferido.

Através do exemplo da transposição de matrizes, foi possível estudar o impacto de acessos não coalescidos na transferência de dados, e como é possível utilizar a memória compartilhada para coalescer os acessos na memória global. Observou-se uma diferença de até $7\times$ entre o acesso direto não coalescido, e o acesso coalescido utilizando a memória global. Foi constatado, no entanto, que mesmo para os acessos coalescidos da matriz transposta, seu desempenho foi muito inferior ao da cópia direta coalescida. A causa disso estava na má distribuição dos acessos a diferentes partições na memória global, um efeito também chamado de *partition camping*. Utilizando a técnica de diagonalização para indexar os blocos, os acessos às partições de memória foram distribuidos de maneira mais uniforme. Sem as penalidades do acesso a memória global, foi possível visualizar melhor o impacto de conflitos na memória compartilhada, representando uma redução de até 23% na taxa de transferência final obtida.

Deste modo, concluiu-se que as otimizações devem ser aplicadas em etapas, sendo primeiramente necessário otimizar o acesso à memória global, de modo a evitar acessos não coalescidos, possibilitando um aumento de desempenho na largura de banda de até $10\times$. Em seguida deve-se eliminar os conflitos no acesso aos bancos da memória compartilhada. Nem sempre as otimizações estão descritas nos guias fornecidos pelos fabricantes, como é o caso da solução para o problema de má distribuição nas partições de memória.

É importante ressaltar que os resultados obtidos neste estudo não representam os mais otimizados possíveis. Foram apenas utilizados alguns exemplos a fim de demonstrar o foco do estudo, ou seja, o impacto das otimizações de memória. Fatores como o tamanho da partição da matriz, ou quantidade de elementos processados por *thread* podem possibilitar um aumento ainda maior no desempenho final.

Acredita-se que o trabalho aqui desenvolvido pode ser estendido para a arquitetura ATI Stream devido às semelhanças arquiteturais anteriormente mencionadas, como o uso de hierarquias de memórias e de processadores. Como trabalhos futuros possíveis encontram-se estender os testes realizados em CUDA para OpenCL, que é uma linguagem portável e poderia ser utilizada futuramente na arquitetura ATI Stream. Pretende-se também analisar as diferenças nos acessos na nova arquitetura Fermi da NVIDIA, citada na Seção 3.1.4. Essa arquitetura além de permitir diferentes tipos de acessos coalescidos ainda oferece uma memória compartilhada de tamanho configurável e memórias caches em dois níveis.

REFERÊNCIAS

[ALV 2009] ALVES, M. A. Z. **Avaliação do compartilhamento das memórias cache no desempenho de arquiteturas multi-core**. 2009. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul.

[ASA 2006] ASANOVIC, K. et al. The landscape of parallel computing research: a view from berkeley. **Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report**, v.18, n.2006-183, p.19, 2006.

[ATI 2010] **Ati stream sdk opencl programming guide**. [S.l.]: Advanced Micro Devices, Inc., 2010.

[BRE 2008] BREITBART, J. **Case studies on gpu usage and data structure design**. 2008. Dissertação (Mestrado em Ciência da Computação) — Universität Kassel.

[BUC 2004] BUCK, I. et al. Brook for gpus: stream computing on graphics hardware. In: SIGGRAPH '04: ACM SIGGRAPH 2004 PAPERS, 2004, New York, NY, USA. **Anais...** ACM, 2004. p.777–786.

[BUC 2005] BUCK, I. **Stream computing on graphics hardware**. 2005. Tese (Doutorado em Ciência da Computação) — , Stanford, CA, USA. Adviser-Hanrahan, Pat.

[DAT 2008] DATTA, K. et al. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: SC '08: PROCEEDINGS OF THE 2008 ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2008, Piscataway, NJ, USA. **Anais...** IEEE Press, 2008. p.1–12.

[GOV 2006] GOVINDARAJU, N. K. et al. A memory model for scientific algorithms on graphics processors. **SC Conference**, Los Alamitos, CA, USA, v.0, p.6, 2006.

[GOV 2007] GOVINDARAJU, N. K.; MANOCHA, D. Cache-efficient numerical algorithms using graphics hardware. **Parallel Computing**, v.33, n.10-11, p.663–684, 2007.

[HAR 2002] HARRIS, M. J. et al. Physically-based visual simulation on graphics hardware. In: HWWS '02: PROCEEDINGS OF THE ACM SIGGRAPH/EUROGRAPHICS CONFERENCE ON GRAPHICS HARDWARE, 2002, Aire-la-Ville, Switzerland, Switzerland. **Anais...** Eurographics Association, 2002. p.109–118.

[HAR 2005] HARRIS, M. Mapping computational concepts to gpus. In: SIGGRAPH '05: ACM SIGGRAPH 2005 COURSES, 2005, New York, NY, USA. **Anais...** ACM, 2005. p.50.

[HE 2007] HE, B. et al. Efficient gather and scatter operations on graphics processors. In: SC '07: PROCEEDINGS OF THE 2007 ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2007, New York, NY, USA. **Anais...** ACM, 2007. p.1–12.

[HIM 2006] HIMAWAN, B.; VACHHARAJANI, M. Deconstructing hardware usage for general purpose computation on gpus. **Fifth Annual Workshop on Duplicating, Deconstructing, and Debunking (in conjunction with ISCA-33)**, 2006.

[KIR 2010] KIRK, D. B.; W. HWU, W. mei. **Programming massively parallel processors**: a hands-on approach. pub-ELSEVIER:adr: Elsevier, 2010. xviii + 258p.

[LI 2009] LI, Y.; DONGARRA, J.; TOMOV, S. **A note on auto-tuning GEMM for GPUs**. inst-UT-CS:adr: Department of Computer Science, University of Tennessee, Knoxville, 2009. LAPACK Working Note. (212).

[MUN 2009] MUNSHI, A. **The opencl specification**. [S.l.]: Khronos OpenCL Working Group, 2009.

[NVI 2009b] **Nvidia compute unified device architecture (cuda) c programming best practices guide**. [S.l.]: NVIDIA Corporation, 2009.

[NVI 2009] **Nvidia compute unified device architecture (cuda) programming guide 2.3**. [S.l.]: NVIDIA Corporation, 2009.

[NVI 2009a] **Nvidia's next generation cuda compute architecture**: fermi. [S.l.]: NVIDIA Corporation, 2009.

[OWE 2005] OWENS, J. D. et al. A survey of general-purpose computation on graphics hardware. In: EUROGRAPHICS 2005, STATE OF THE ART REPORTS, 2005. **Anais...** [S.l.: s.n.], 2005. p.21–51.

[PIL 2009] PILLA, L. L. **Análise de Desempenho da Arquitetura CUDA Utilizando os NAS Parallel Benchmarks**. 2009. 60p. Trabalho de Conclusão de Curso — Universidade Federal do Rio Grande do Sul.

[RUE 2009] RUETSCH, G.; MICIKEVICIUS, P. **Optimizing matrix transpose in cuda**. [S.l.]: NVIDIA Corporation, 2009.

[RYO 2008] RYOO, S. et al. Program optimization space pruning for a multithreaded gpu. In: CGO '08: PROCEEDINGS OF THE 6TH ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION, 2008, New York, NY, USA. **Anais...** ACM, 2008. p.195–204.

[RYO 2008a] RYOO, S. et al. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In: PPOPP '08: PROCEEDINGS OF THE 13TH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2008, New York, NY, USA. **Anais...** ACM, 2008. p.73–82.

[SEI 2008] SEILER, L. et al. Larrabee: a many-core x86 architecture for visual computing. **ACM Transactions on Graphics-TOG**, v.27, n.3, p.18–18, 2008.

[VOL 2008] VOLKOV, V.; DEMMEL, J. W. Benchmarking gpus to tune dense linear algebra. In: SC '08: PROCEEDINGS OF THE 2008 ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2008, Piscataway, NJ, USA. **Anais...** IEEE Press, 2008. p.1–11.

[VOL 2008a] VOLKOV, V.; DEMMEL, J. **Lu, qr and cholesky factorizations using vector capabilities of gpus**. [S.l.]: EECS Department, University of California, Berkeley, 2008. (UCB/EECS-2008-49).