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

LAÉRCIO LIMA PILLA

Análise de Desempenho da Arquitetura CUDA Utilizando os NAS Parallel Benchmarks

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, dezembro de 2009

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Pilla, Laércio Lima

Análise de Desempenho da Arquitetura CUDA Utilizando os NAS Parallel Benchmarks / Laércio Lima Pilla. – Porto Alegre: Graduação em Ciência da Computação da UFRGS, 2009.

60 f.: il.

Trabalho de Conclusão (bacharelado) – Universidade Federal do Rio Grande do Sul. Curso de Bacharelado em Ciência da Computação, Porto Alegre, BR–RS, 2009. Orientador: Philippe Olivier Alexandre Navaux.

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

“Não há riqueza como o conhecimento,
nem pobreza como a ignorância.”
— ALI IBN ABI TALIB

AGRADECIMENTOS

Agradeço à minha família por ter formado as minhas bases, aos meus professores por me ajudarem a crescer e às pessoas do Grupo de Processamento Paralelo e Distribuído por me mostrarem como utilizar meu conhecimento.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
DMA	*Direct Memory Access*
FFT	*Fast Fourier Transform*
Flops	*Floating Point Operations Per Second*
GPGPU	*General-Purpose computing on Graphics Processing Units*
GPU	*Graphics Processing Unit*
IMT	*Interleaved Multithreading*
Mops	Milhões de Operações por Segundo
NaN	*Not a Number*
NPB	NAS Parallel Benchmarks
NUCA	*Non-Uniform Cache Access*
PRAM	*Parallel Random Access Machine*
SIMD	*Single Instruction, Multiple Data*
SIMT	*Single Instruction, Multiple Threads*
SM	*Streaming Multiprocessor*
SMP	*Symmetric Multiprocessor*
SP	*Scalar Processor*
ULA	Unidade Lógica e Aritmética
VLIW	*Very Large Instruction Word*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Processadores gráficos vêm sendo utilizados como aceleradores paralelos para computações de propósito geral (GPGPU), não detidos mais apenas em aplicações gráficas. Isto acontece devido ao custo reduzido e grande potencial de desempenho paralelo dos processadores gráficos, alcançando Teraflops. CUDA (*Compute Unified Device Architecture*) é um exemplo de arquitetura com essas características. Diversas aplicações já foram portadas para CUDA nas áreas de dinâmica de fluídos, reconhecimento de fala, alinhamento de sequências, entre outras. Entretanto, não há uma definição clara de quais tipos de aplicações podem se aproveitar dos potenciais ganhos de desempenho que as GPGPUs trazem, visto que a arquitetura do hardware é do tipo SIMD (*Simple Instruction, Multiple Data*) e que existem restrições nos acessos à memória. Visando estudar esta questão, este trabalho apresenta uma análise de desempenho da arquitetura de placa gráfica CUDA guiada por modelos paralelos e benchmarks. Para isso, os benchmarks EP e FT dos NAS Parallel Benchmarks foram portados e otimizados para CUDA, mantendo o uso de operações de ponto flutuante de precisão dupla. Estes dois benchmarks fazem parte das categorias *MapReduce* e *Spectral Methods*, respectivamente, dentro da classificação Dwarf Mine. Uma análise de desempenho foi realizada, fazendo uma comparação entre os resultados obtidos pelas novas versões implementadas e as versões originais do código compiladas para execução de forma sequencial e paralela com OpenMP. Os resultados obtidos mostraram *speedups* de até 21 vezes para o benchmark EP e quase 3 vezes para o benchmark FT, quando comparadas as versões para CUDA com as versões com OpenMP. Estes resultados indicam uma compatibilidade entre a arquitetura CUDA e aplicações pertencentes às categorias *MapReduce* e *Spectral Methods*.

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

ABSTRACT

Graphic processors are being used not only for graphic applications but as parallel accelerators for general-purpose computations (GPGPU). This happens due to their reduced costs and performance potential, reaching Teraflops. CUDA (Compute Unified Device Architecture) is an example of architecture with this characteristics. Several applications in the areas of fluid dynamics, speech recognition, sequence alignment and others have already being ported to CUDA. However, there is not a clear definition of which kinds of applications can take profit of the potential gains of performance that the GPGPUs have, since they have a SIMD (Single Instruction, Multiple Data) architecture and that are restrictions to the memory access. Aiming to study that question, this work presents a performance analysis of the CUDA GPU architecure guided by parallel models and benchmarks. For this, the EP and FT benchmarks from the NAS Parallel Benchmarks were ported and optimized to CUDA, keeping the use of double precision floating-point operations. These two benchmarks are included in the MapReduce and Spectral Methods classes, respectively, from the Dwarf Mine classification. A performance analysis was made comparing the results obtained by the new implemented versions of the benchmarks and their original versions, compiled to execute in a sequential manner and in a parallel manner with OpenMP. The obtained results showed speedups up to 21 times for the EP benchmark and almost 3 times for the FT one, when comparing the CUDA versions to the versions with OpenMP. These results indicate a compatibility between the CUDA architecture and the applications belonging to the MapReduce and Spectral Methods classes.

Keywords: GPGPU, Parallel Computing, Performance Analysis.

1 INTRODUÇÃO

A indústria da computação segue uma mudança de sentido em direção à computação paralela, como consequência das limitações dos circuitos integrados (ASA 2006). A latência de memória praticamente inalterada durante as últimas décadas, o pouco paralelismo em nível de instrução que ainda pode ser explorado eficientemente e o máximo de dissipação de energia em chips resfriados a ar são limitantes que contribuíram para tal mudança. Ao invés de processadores monolíticos mais rápidos, a produção foca-se em múltiplos processadores por chip.

A Figura 1.1 ilustra o crescimento do desempenho de processadores entre 1978 e 2005 em relação ao processador VAX 11/780. Pode-se observar uma desaceleração no crescimento de desempenho a partir de 2002 para aproximadamente 20%, devido aos fatores citados anteriormente. Agora, o crescimento no desempenho de processadores baseia-se no uso de arquiteturas paralelas SMP (*Symmetric Multiprocessor*). Todavia, entre as alternativas atuais para aumento de desempenho, uma diferente plataforma que vem ganhando destaque envolve o uso de processadores gráficos como aceleradores paralelos para computações de propósito geral, ou **GPGPU** (*General-Purpose computing on Graphics Processing Units*).

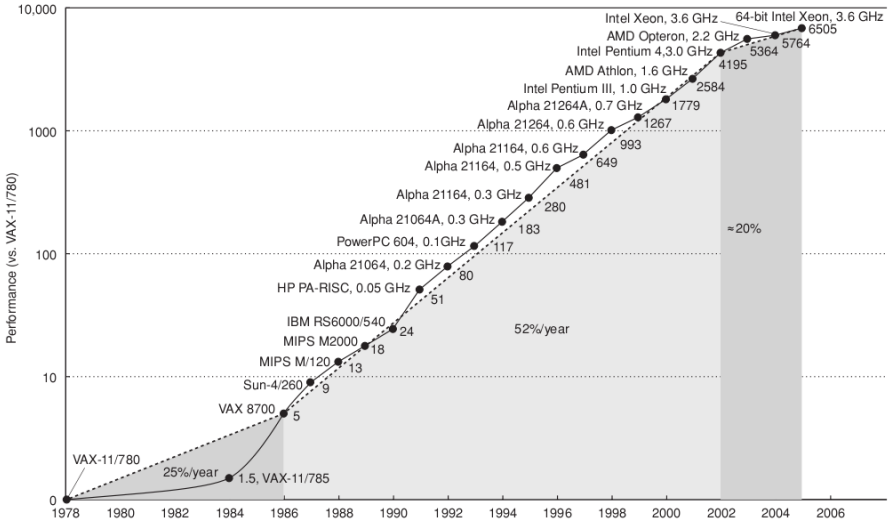

Figura 1.1: Crescimento do desempenho de processadores convencionais em relação ao processador VAX 11/780 (HEN 2006).

Unidades de Processamento Gráfico, ou **GPUs** (*Graphics Processing Units*), são hardwares dedicados a renderização de imagens, servindo como aceleradores neste processo. Segundo a Taxonomia de Flynn (DUN 90), processadores gráficos podem ser vistos como

máquinas **SIMD** (*Single Instruction, Multiple Data*), executando a mesma função paralelamente sobre grandes quantidades de dados, como pixels. A evolução das GPUs levou ao desenvolvimento de arquiteturas com centenas de processadores simples. Um exemplo disto é a arquitetura **CUDA** (*Compute Unified Device Architecture*) (NVI 2009). O desempenho obtido com o uso dos múltiplos processadores simples dessa arquitetura acaba sendo superior ao visto em processadores convencionais, como ilustrado na Figura 1.2. Na figura, o eixo vertical apresenta o pico de desempenho teórico em ponto flutuante de precisão simples e o eixo horizontal representa o tempo. As linhas superior e inferior representam o desempenho da arquitetura CUDA e de CPUs (*Central Processing Units*) da empresa Intel, respectivamente. Um alto desempenho aliado a baixos custo e consumo de energia tornam as GPGPUs, como CUDA, em plataformas interessantes. A introdução de facilidades de programação tornou possível a implementação de algoritmos de propósito geral em GPUs. Exemplos de algoritmos acelerados para CUDA incluem dinâmica de fluídos (TÖL 2007), reconhecimento de fala (CHO 2008), simulação de falhas (GUL 2008), alinhamento de sequências (MAN 2008; SCH 2007), criptografia (MAN 2007), previsão do tempo (MIC 2008), etc.

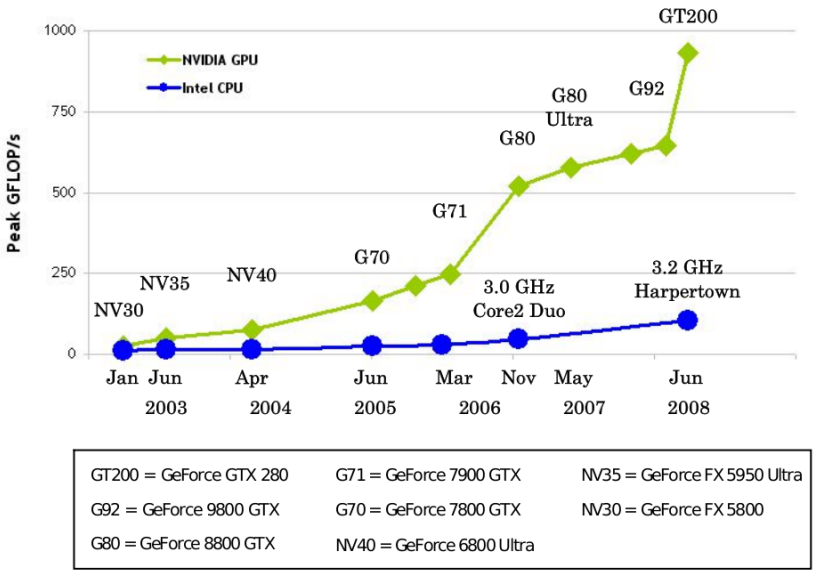

Figura 1.2: Comparativo entre GPUs e CPUs em número de operações de ponto flutuante de precisão simples por segundo. (NVI 2009)

Apesar de casos com aceleração de até duas ordens de grandeza, nem todos algoritmos podem obter ganhos com a migração para GPUs. Paralelismo massivo em nível de dados e alta intensidade aritmética são traços importantes encontrados nos algoritmos executados em CUDA. Além disso, características como a ausência de padrão previsível de acesso a memória e a falta de estrutura de dados mapeáveis facilmente para a hierarquia de memória da GPU agem contra o ganho de desempenho.

Como a plataforma de GPGPU é atrativa, tanto pelo seu potencial de desempenho como pelo seu preço, ela acaba chamando a atenção de consumidores nas áreas científica e comercial. Entretanto, a falta de conhecimento sobre quais aplicações podem se aproveitar do potencial desempenho das GPUs tende a ser prejudicial, podendo levar a aquisição de hardware que não pode ser aproveitado ou à escolha de uma diferente plataforma não tão qualificada para a aplicação específica.

Assim, o estudo do comportamento de aplicações executadas sobre placas gráficas mostra-se importante. Com isso, pode-se evitar o grande investimento de implementação com aplicações que não conseguem aproveitar das características desta plataforma. Porém, é inviável testar todos os algoritmos existentes. Uma alternativa possível está no uso de um conjunto de benchmarks representativos, como os **NAS Parallel Benchmarks** (NPB) (BAI 94), utilizados para a avaliação de sistemas paralelos. Assim, o estudo deve aproveitar-se das características similares entre os métodos algorítmicos para expandir seus resultados e evitar a implementação de algoritmos muito similares.

Este trabalho apresenta uma análise de desempenho de GPUs guiada por modelos paralelos e benchmarks, com o intuito de observar quais aplicações possuem potencial para ganho de desempenho nesta plataforma. Mais especificamente, **este trabalho aborda a implementação de dois NAS Parallel Benchmarks para CUDA e a verificação de seus desempenhos**.

Trabalhos relacionados são apresentados no Capítulo 2, estando entre eles a classificação de algoritmos paralelos utilizada neste trabalho, **Dwarf Mine**. O Capítulo 3 apresenta as características de processadores gráficos atuais e arquiteturas com similaridades, com foco na arquitetura CUDA. Os benchmarks paralelos do NAS são caracterizados no Capítulo 4. Neste capítulo também é detalhada a paralelização dos benchmarks EP e FT para CUDA. Comparativos do desempenho entre as versões originais e as para CUDA podem ser vistos no Capítulo 5. Por fim, o Capítulo 6 termina este trabalho, mostrando as suas principais contribuições e apresentando sugestões de trabalhos futuros.

2 TRABALHOS RELACIONADOS

Este capítulo descreve alguns trabalhos inseridos no contexto do presente documento. A Seção 2.1 apresenta dois modelos de programação paralela considerados pelo autor como não adequados para GPUs. A Seção 2.2 aborda a classificação Dwarf Mine, utilizada neste trabalho. A Seção 2.3 trata sobre trabalhos envolvendo a arquitetura CUDA e a paralelização de algoritmos para tal arquitetura. Por fim, trabalhos envolvendo comparação de desempenho entre diferentes sistemas são apresentados na Seção 2.4.

2.1 Modelos de Programação Paralela

Definir um modelo preciso e geral para computadores paralelos parece impossível, sendo a modelagem dos custos de comunicação a parte mais complexa (CAS 2008). O modelo **PRAM** (*Parallel Random Access Machine*) (CAS 2008) resolve este problema simplesmente ignorando tais custos, resultando assim em um modelo imperfeito quando comparado aos custos reais na execução de algoritmos, mas que torna possível a classificação de algoritmos e obtenção de resultados de complexidade.

O modelo PRAM compreende uma memória central compartilhada que pode ser acessada por diversas unidades de processamento. Estas unidades executam os algoritmos de forma síncrona. O modelo não limita o tamanho desta memória nem o número de unidades de processamento.

Harish e Narayanan apresentam a interface de programação CUDA como próxima ao modelo PRAM (HAR 2007). No artigo, eles mostram resultados sobre a implementação de algoritmos paralelos de grafos para CUDA, utilizando-os para grafos grandes (com milhões de vértices). Entre os algoritmos apresentados, está o de busca em largura (*Breadth First Search*).

O algoritmo se desenvolve em etapas. A cada etapa, os vértices de um conjunto estipulado na etapa anterior (um vértice inicial, no caso da primeira etapa) verificam as suas vizinhanças e marcam cada um de seus vizinhos ainda não visitados para executarem na próxima etapa. Cada um desses vizinhos tem seu custo (número de arestas para alcançar um vértice partindo do vértice inicial) armazenado como o custo do vértice que o alcança mais um. O algoritmo para quando não houver mais nenhum vértice a executar.

Porém, foi identificada uma condição de corrida no algoritmo, a qual leva vértices a executarem na etapa em que foram visitados, ou seja, de forma adiantada. Este problema seria solucionado com uma sincronização global. Entretanto, sincronizações globais somente são feitas no início e no fim de chamadas de kernel (funções que executam na GPU). A ausência de uma forma de sincronização global simples indica uma possível incompatibilidade entre a arquitetura CUDA e o modelo PRAM.

Em (HOU 2008), é apresentada a linguagem de programação BSGP (*Bulk-Synchronous*

GPU Programming). Ela busca aliar características do modelo **BSP** (*Bulk-Synchronous Parallel*) (RIG 2009) às GPUs. O modelo BSP é baseado na sucessão de superetapas, onde cada superetapa é composta por computações locais, comunicações entre processos e uma barreira de sincronização. Entretanto, o modelo BSP não trata de aspectos como regularidade das computações, padrões de comunicação e acesso a memória, entre outros, considerados importantes no presente trabalho.

2.2 Classificação Dwarf Mine

Em meados de 2005, diversos pesquisadores da UC Berkeley começaram a se reunir para discutir sobre o futuro da pesquisa na área de computação paralela, tendo em vista a mudança da predominância do paradigma sequencial para o paralelo guiado pelo desenvolvimento das arquiteturas multicore (ASA 2006). Este esforço resultou em um relatório técnico apresentando sete questões principais que deveriam ser tratadas nos próximos anos. Duas delas tratam diretamente sobre as aplicações paralelas: (I) Quais são as aplicações e (II) Quais são os *kernels* comuns entre as aplicações.

Para responder estas perguntas, eles organizaram uma classificação nomeada **Dwarf Mine** [1] (ASA 2006). Um *dwarf* representa um método algorítmico que captura padrões de computação e comunicação. Esta nomenclatura é inspirada no trabalho de Phil Colella, o qual identificou sete métodos numéricos que considerou importantes até a próxima década - nomeados *Seven Dwarfs*. Os pesquisadores de Berkeley estenderam esta lista, incluindo outros padrões encontrados em benchmarks e métodos de construção de algoritmos. As treze categorias de métodos algorítmicos podem ser vistas na Tabela 2.1. É importante ressaltar que uma aplicação poderia se encontrar em mais de uma categoria de problema (ASA 2006).

O presente trabalho é guiado por essa classificação, pois há um mapeamento claro entre os NAS Parallel Benchmarks e as categorias de métodos algorítmicos (ASA 2006). A classificação Dwarf Mine possibilitará a extensão dos resultados a serem obtidos com os benchmarks para categorias de algoritmos. Além disso, ela nos auxilia ao mostrar quais benchmarks se encontram na mesma categoria, podendo-se, assim, evitar implementações que não trarão novos resultados.

2.3 CUDA

No Capítulo 1, são citadas as aplicações de dinâmica de fluídos (TÖL 2007), reconhecimento de fala (CHO 2008), simulação de falhas (GUL 2008), alinhamento de sequências (MAN 2008; SCH 2007), criptografia (MAN 2007) e previsão do tempo (MIC 2008), paralelizadas para a arquitetura CUDA. Na grande maioria dos casos, o foco destas implementações está no ganho de desempenho para a aplicação e não na análise de desempenho da arquitetura em si.

Em (GOV 2008) é apresentada a implementação de uma biblioteca de FFT (*Fast Fourier Transform*) para CUDA com desempenho de duas a quatro vezes superior à biblioteca CUFFT, disponibilizada com CUDA. Porém, a biblioteca é implementada com precisão simples, o que impossibilita seu uso junto ao benchmark FT dos NPB.

Lee, Min e Eigenmann desenvolveram um tradutor de código em OpenMP para CUDA com otimizações (LEE 2009). Entre os exemplos de aplicações paralelizadas, estão os

[1] http://view.eecs.berkeley.edu/wiki/Dwarf_Mine

Tabela 2.1: Classificação Dwarf Mine

	Nome	**Características**
1	*Dense Linear Algebra*	Dados são representados como matrizes ou vetores densos.
2	*Sparse Linear Algebra*	Dados são esparsos (grande número de valores iguais a zero) e normalmente comprimidos, o que leva a acessos indexados.
3	*Spectral Methods*	Dados são representados no domínio frequência. Envolve padrões específicos de permutação de dados e uso de operações $multiply - add$.
4	*N-Body*	Baseado na interação entre diversos pontos discretos (como partículas).
5	*Structured Grids*	Representado por uma grade regular, onde os pontos são atualizados de forma conjunta.
6	*Unstructured Grids*	Representado por uma grade irregular, onde a conectividade entre os pontos deve ser explicitada. As vizinhanças são formadas baseadas em características da aplicação e dos dados.
7	*Monte Carlo / MapReduce*	Cálculos estatísticos baseados em repetições aleatórias. Os dados são usualmente independentes e há pouca ou nenhuma comunicação.
8	*Combinational Logic*	Cálculos simples (funções lógicas) sobre grandes quantidades de dados.
9	*Graph Traversal*	Baixa intensidade aritmética e grande número de indireções.
10	*Dynamic Programming*	Solução do problema baseada na solução de subproblemas com sobreposição. Envolve a comunicação dos resultados parciais independentes.
11	*Backtrack & Branch+Bound*	Divisão recursiva de problemas e composição de resultados parciais.
12	*Construct Graphical Models*	Grafos onde os vértices representam variáveis aleatórias e as arestas são dependências condicionais.
13	*Finite State Machines*	Sistemas cujo comportamento é definido por estados sensíveis a entradas e estados anteriores.

benchmarks EP e CG do NAS. Eles obtiveram ganhos de desempenho por volta de 300 vezes com o benchmark EP e 4 vezes com o CG, quando comparados as suas versões sequenciais. Entretanto, estes ganhos foram obtidos com o uso de ponto flutuante de precisão simples, devido a limitações da placa gráfica utilizada, enquanto os benchmarks possuem resultados que somente são considerados corretos com o uso de precisão dupla. Apesar do interesse em utilizar a plataforma desenvolvida pelos autores, não foi encontrada nenhuma versão do tradutor disponível na Internet. Todavia, as idéias de otimizações apresentadas neste artigo foram utilizadas no presente trabalho.

Em (RYO 2008), é apresentada a avaliação de uma placa gráfica com arquitetura CUDA através de um conjunto de aplicações paralelas e suas otimizações. Os autores ressaltam uma série de otimizações gerais para a arquitetura que obtiveram bons resultados. Entretanto, não é utilizada qualquer classificação para o conjunto de aplicações escolhida, sendo apenas reunidas aplicações que poderiam obter ganhos de desempenho com a arquitetura.

2.4 Comparação de Desempenho entre Sistemas Paralelos

Em (ROL 2009), é apresentado um conjunto de aplicações e benchmarks das áreas de processamento de alto desempenho e sistemas embarcados para a avaliação de arquiteturas multicore e seus sistemas de software. Entre dezenas de aplicações possíveis, foram escolhidas 17 delas. Entre os benchmarks selecionados, estão presentes cinco benchmarks do NAS (CG, EP, FT, IS e MG). Os autores consideram a classificação de algoritmos Dwarf Mine (ASA 2006) em seu trabalho, apresentando uma cobertura das treze diferentes categorias.

Marco Antonio Alves (ALV 2009) utilizou os NAS Parallel Benchmarks (em versão FORTRAN com OpenMP) para a comparação de desempenho entre diferentes configurações de compartilhamento de cache em processadores multicore. Ele escolheu este conjunto de benchmarks baseado na sua disponibilidade, diversidade, foco em processamento de alto desempenho, portabilidade e variabilidade do tamanho das cargas de trabalho. Entretanto, o autor não utilizou uma classificação de algoritmos paralelos para reduzir o número de benchmarks executados ou estender os resultados.

Em (JES 2009) é descrito um planejamento para uma comparação do desempenho entre FPGAs (*Field Programmable Gate Array*) e GPUs. Neste trabalho, a classificação de algoritmos Dwarf Mine é utilizada como ferramenta para a comparação entre as duas diferentes arquiteturas. O autor apresenta que seus algoritmos de interesse, estéreo denso e estéreo esparso, possuem diversas características similares, mas que ambos precisarão ser implementados e testados por não fazerem parte da mesma categoria na classificação.

3 ARQUITETURAS DE CO-PROCESSAMENTO

Em (GEL 89), a ideia de múltiplos núcleos em um único chip foi inicialmente apresentada. Olukotun defende tal abordagem em (OLU 96). Ele compara um processador monolítico complexo e um processador com múltiplos núcleos mais simples, ambos ocupando a mesma área. Seu trabalho apresenta ganhos de desempenho com aplicações paralelas executando em múltiplos núcleos e, utilizando apenas um dos núcleos, uma pequena queda no desempenho na execução de uma aplicação sequencial. Baseado nesta idéia inicial, diversas arquiteturas foram desenvolvidas, cada qual com características específicas - focadas em cargas de trabalho, programabilidade, custos, consumo de energia, reusabilidade de código, entre outros.

Este capítulo apresenta um estudo sobre as principais arquiteturas de co-processamento com múltiplos núcleos - GPUs e similares - e suas linguagens de programação. O foco é dado na biblioteca e arquitetura da empresa NVIDIA, nomeada CUDA (*Compute Unified Device Architecture*), a qual é utilizada neste trabalho na implementação e testes de perfis paralelos. Ao fim do capítulo, apresenta-se uma discussão sobre as similaridades e diferenças entre as arquiteturas apresentadas, com o intuito de ilustrar o quanto os resultados obtidos neste trabalho poderão ou não serem estendidos às diferentes arquiteturas.

3.1 CUDA

As placas gráficas da empresa NVIDIA vêm ganhando destaque no mercado. Isso acontece devido ao crescente desempenho - alcançando o patamar de TFlops (*Tera floating point operations per second*, ou trilhões de instruções de ponto flutuante por segundo) - e às facilidades de programação trazidas por **CUDA**[1] (NVI 2009).

CUDA é tanto uma extensão da linguagem de programação C que permite o desenvolvimento de programas que executam em parte nas GPUs, quanto o nome da arquitetura destas GPUs. Essa linguagem traz abstrações que auxiliam na programação para GPGPU. Por exemplo, não é necessário o mapeamento dos dados para vértices ou fragmentos, como acontecia anteriormente (HAR 2002; HIM 2006). Entretanto, a linguagem ainda se mantém próxima à arquitetura, sendo apenas compatível com placas gráficas da NVIDIA das últimas gerações (a partir da G80).

As principais características tanto da linguagem quanto da arquitetura são apresentadas nas subseções seguintes. Primeiramente, são tratadas questões sobre os processadores e a hierarquia de threads. Em sequência, a hierarquia de memória é apresentada. A subseção seguinte apresenta questões sobre a precisão e desempenho das placas gráficas. Por fim, detalhes sobre a futura geração de GPUs da arquitetura CUDA são apresentados.

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

3.1.1 Processadores

A arquitetura das placas gráficas compatíveis com CUDA é construída baseada em multiprocessadores chamados de *Streaming Multiprocessors*, ou **SMs**. O número de SMs varia entre as placas gráficas. A mudança nesse número somente pode acontecer pois os SMs são modulares e escaláveis. O aumento no desempenho de novas gerações de GPUs baseia-se principalmente - embora não unicamente - na adição destes multiprocessadores.

Um SM consiste em 8 núcleos. Estes núcleos são conhecidos como Processadores Escalares, ou **SPs** (*Scalar Processors*). Todos os núcleos em um multiprocessador executam a mesma instrução sobre suas threads, pois são controlados pelo mesma Unidade de Instruções. Cada SP executa suas threads em IMT (*Interleaved Multithreading*) (BRE 2008).

Existe um mapeamento direto de threads para processadores, como ilustrado na Figura 3.1. Um procedimento que executa na placa gráfica de forma paralela é chamado de **kernel**. Um kernel não pode conter recursão nem chamadas a outras funções. As threads que executam em um kernel são organizadas em blocos. Os blocos, por sua vez, são organizados em uma grade. Cada bloco é mapeado para um SM. Porém, um bloco pode ter mais threads do que um multiprocessador é capaz de executar paralelamente - o que será mostrado mais a frente como uma característica favorável ao desempenho. Além disso, o número de blocos independe do número de SMs da placa gráfica, não havendo necessidade de recompilação de código para GPUs mais novas. A sincronização por barreira é possível em nível de instrução, mas somente entre threads de um mesmo bloco. Como o paralelismo dos dados é tratado com a abstração de milhares de threads ao invés de vetores, esta organização vem sendo chamada de **SIMT** (*Single Instruction, Multiple Threads*) (NVI 2009).

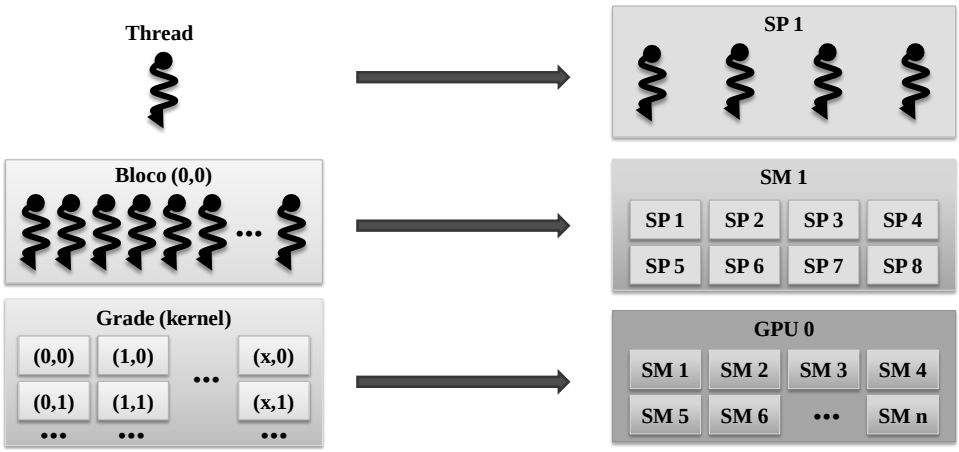

Figura 3.1: Mapeamento de threads para processadores da arquitetura CUDA.

As threads são organizadas de forma consecutiva em ***warps***. Cada *warp* contém 32 threads que executam em paralelo em um SM, 4 threads em pipeline em cada SP. Como todos os SPs em um multiprocessador são controlados pela mesma Unidade de Instruções, a divergência nos caminhos de execução entre as threads de um mesmo *warp* é tratada através da sequencialização destes caminhos. Com isso, quanto maior o número de caminhos divergentes entre estas threads, pior o desempenho.

Múltiplas placas gráficas podem ser utilizadas em paralelo em uma mesma máquina mantendo uma abstração de apenas um acelerador. Isso é possível através do modo SLI (*Scalable Link Interface*). Porém, caso seja desejado que cada GPU seja vista em separado

(para executar kernels diferentes em cada uma delas, por exemplo), é necessário desligar tal modo.

3.1.2 Memória

Uma placa gráfica possui sua própria memória, em separado da memória principal do computador. Os dados utilizados pela GPU devem ser escritos e, posteriormente, lidos através de chamadas a procedimentos (como $cudaMemcpy()$).

A hierarquia de memória da placa gráfica está intimamente ligada à hierarquia de threads e, consequentemente, aos processadores. Cada multiprocessador possui memória dos seguintes tipos: registradores de 32 bits, memória compartilhada, cache de constantes e cache de texturas.

Cada SP possui seus próprios registradores, os quais são concorridos pelas threads que executam nele. Todos os SPs em um mesmo multiprocessador acessam a mesma memória compartilhada, na qual podem ler ou escrever. As caches também são compartilhadas pelos SPs em um mesmo SM. Elas são memórias apenas de leitura que aceleram acessos às memórias de constantes e texturas, situadas externamente ao SM. Estas 4 memórias possuem custos de acesso pequeno porém são de tamanho limitado (na ordem de KBytes).

A Figura 3.2 ilustra a hierarquia de memória na placa gráfica. Além da memória nos multiprocessadores, a GPU possui uma memória de alta latência e grande tamanho. Nela estão contidas as memórias de constantes e texturas (as quais possuem cache nos SMs), além da memória global e local. Ambas servem para escrita e leitura. Entretanto, a memória local é individual a cada thread, enquanto a memória global é compartilhada por todas as threads em todos multiprocessadores.

É importante ressaltar que esta arquitetura não utiliza uma hierarquia de cache para acelerar o acesso à memória. Apesar de múltiplas fontes apresentarem tal informação (CHO 2008; HE 2007; TÖL 2007), o material disponibilizado pela empresa NVIDIA nega a existência dessa cache (NVI 2009a).

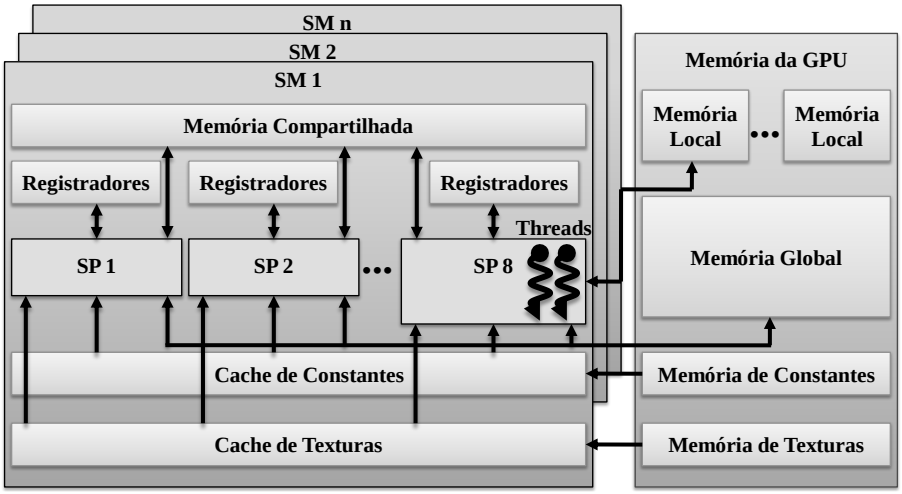

Figura 3.2: Hierarquia de memória da arquitetura CUDA.

A latência no acesso às memórias local e global envolve de 400 a 600 ciclos. Esse atraso pode ser escondido com o escalonamento de outras threads, as quais encontram-se em espera no SP. Desta forma, mantém-se os processadores sempre executando instruções sobre os dados, aumentando o desempenho da aplicação e a eficiência. Além disso, acessos a dados contíguos em memória global podem ser **coalescidos** (agregados) em leituras

de palavras de 384 bits, aumentando a banda passante.

3.1.3 Precisão e Desempenho

O alto desempenho obtido nas placas gráficas está ligado - não exclusivamente - a um decréscimo na precisão obtida nas computações. O limite teórico de desempenho apresentado em gráficos, como na Figura 1.2 no Capítulo 1, considera instruções de ponto flutuante de precisão simples. As placas gráficas da empresa NVIDIA não são complacentes com o Padrão 754 (GOL 91) definido pela IEEE para computações em ponto flutuante.

As GPUs executam sobre ponto flutuante de precisão simples não estendida. Um valor expresso dessa forma apresenta 32 bits, sendo 1 bit para a representação do sinal, 8 bits para o expoente e 23 bits para a mantissa. Além disso, as operações envolvem o uso de 1 bit escondido (ou dígito de guarda), utilizado para a manutenção de certo grau de precisão.

A não complacência com o padrão envolve a falta de suporte a números não normalizados, o que pode levar a erros relativos grandes em algoritmos com números normalizados próximos a fronteira de *underflow* (GOL 91). Além disso, NaNs (*Not a Number*) não são sinalizados e retornam o padrão hexadecimal $0x7FFFFFFF$. Além do retorno desse padrão, não há nenhum elemento de alerta e detecção de exceções. O modo de arredondamento não pode ser configurado dinamicamente. As operações de adição e multiplicação normalmente são combinadas em uma única instrução $multiply{-}add$, o que acaba por truncar o resultado intermediário da multiplicação. Por fim, as operações de divisão e raiz quadrada são implementadas de forma não complacente com o padrão IEEE. Essa reduzida precisão impossibilita o uso de GPUs para a computação de certos algoritmos (BRE 2008).

A possibilidade de processamento sobre dados em precisão dupla somente existe a partir de placas gráficas com *Compute Capability* 1.2 (NVI 2009). Porém, o limite teórico com precisão dupla é muito inferior à precisão simples (12 vezes menor). Por exemplo, na placa gráfica GTX 280, são alcançados teóricos $933,12$ GFlops para operações com precisão simples, mas apenas aproximados 78 GFlops são alcançados com precisão dupla (NVI 2008). Isso acontece devido ao diferente número de unidades aritméticas dedicadas as diferentes precisões, havendo apenas uma ULA (Unidade Lógica e Aritmética) de precisão dupla por multiprocessador.

O número máximo de operações sobre ponto flutuante de precisão simples é calculado considerando a instrução $multiply{-}add$, que executa 2 operações sobre os dados em cada SP. A partir da geração de placas gráficas GT200, consegue-se executar mais uma operação simultaneamente, utilizando uma unidade de funções especiais para a execução de uma multiplicação. Assim, alcançam-se 3 operações de ponto flutuante por ciclo por SP.

3.1.4 Arquitetura Fermi

Em um relatório recente, a empresa NVIDIA apresentou informações sobre a sua próxima geração de placas gráficas, nomeada **Fermi** (ou GT300) (NVI 2009a). Entre as principais diferenças para as versões atuais da arquitetura, estão a inclusão de cache em 2 níveis, aumento para 16 SMs por multiprocessador, uso de duas Unidades de Instrução por multiprocessador, aumento no desempenho de precisão dupla, adequação com o Padrão 754 para computações de ponto flutuante e possibilidade de execução de múltiplos kernels em paralelo. Tais alterações na arquitetura tendem a aumentar a usabilidade e eficiência das placas gráficas. Entretanto, ainda não está claro como isto influenciará o

desempenho destas GPUs como um todo.

3.2 Intel Larrabee

A Corporação Intel é a maior empresa de semicondutores do mundo. Ela é conhecida pela criação e desenvolvimento da série de processadores x86. A empresa comercializa processadores centrais com múltiplos núcleos desde 2005. Porém, seu produto de destaque na linha de GPUs, o **Intel Larrabee**[2] (SEI 2008), ainda está em desenvolvimento. As suas características principais envolvem a manutenção de uma memória cache compartilhada coerente - buscando simplificar o uso da memória ao programador - e a delegação de algumas funções tradicionais de placas gráficas para uma camada de software.

3.2.1 Processadores

A arquitetura do Larrabee envolve múltiplos núcleos de processadores x86 simples, sem execução fora de ordem, baseados no Intel Pentium. A cada núcleo foi adicionada uma unidade de processamento de vetores, blocos para filtragem de texturas, suporte a multithreading (4 threads competem por 2 pipelines), extensões para 64 bits, além do aumento da cache L1 (utilizada quase como um banco de registradores, devido a sua baixa latência) e suporte a instruções de controle de cache. O número de núcleos não foi definido ainda, sendo simulados de 8 a 48 núcleos (SEI 2008).

Algumas funções de placas gráficas não são implementadas diretamente na arquitetura, mas sim delegadas para implementação em software. Essa escolha foi feita com interesse em trazer maior simplicidade à arquitetura e extensibilidade, disponibilizando a possibilidade de implementação de novas e diferenciadas funções para processamento gráfico através de uma linguagem de alto nível.

3.2.2 Memória e comunicação

Os núcleos possuem caches L1 privadas, caches L2 compartilhadas em NUCA (*Non-Uniform Cache Access*) e uma memória global. Eles se comunicam através de uma rede de interconexão com banda de 64 bytes que passa pela cache L2, mantendo assim a coerência da mesma. A Figura 3.3 ilustra tal rede. Uma rede em anel com comunicação em ambos sentidos é utilizada para até 16 núcleos. No caso de um número maior de núcleos, diversos anéis são utilizados. Os dados são passados em uma direção em ciclos pares e na outra em ciclos impares.

Para reduzir a complexidade ligada ao acesso à memória, a leitura de dados para a unidade aritmética de vetores dispensa a necessidade de armazenamento de dados de forma contígua. Assim, dados de 16 posições de memória diferentes podem ser agregados em um vetor com apenas uma instrução e vice-versa. Além disso, máscaras de bits podem ser utilizadas para informar quais partes de um vetor devem ser considerados na execução de uma instrução. Isto possibilita leituras e escritas de dados empacotadas e o mapeamento de pequenas construções condicionais para vetores (SEI 2008).

[2] http://software.intel.com/en-us/articles/larrabee/

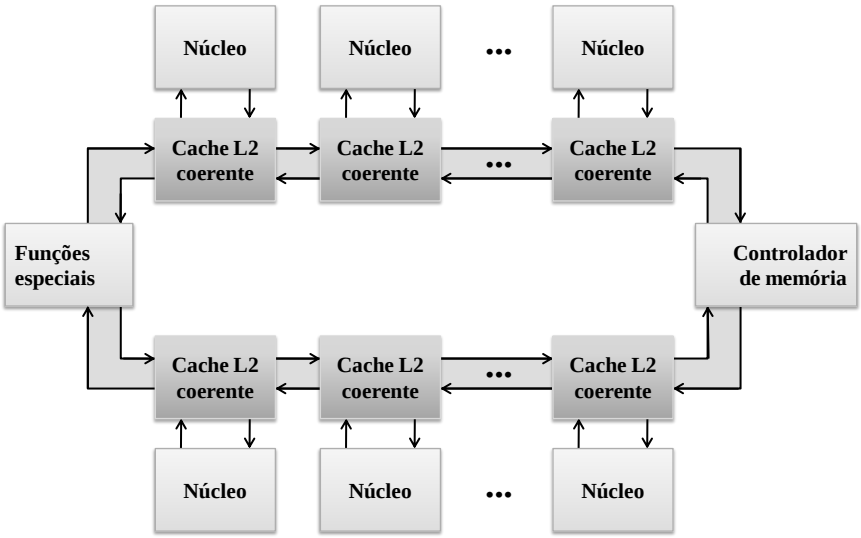

Figura 3.3: Diagrama da arquitetura do processador Larrabee.

3.3 ATI Stream

O nome **ATI Stream**[3] representa um ambiente de programação para placas gráficas desenvolvido pela empresa Advanced Micro Devices (AMD) (ATI 2009). Ele envolve tanto a arquitetura das placas gráficas quanto componentes de software. Entretanto, diferentemente de sua concorrente NVIDIA, as bibliotecas de programação para as placas gráficas ATI não tiveram tanto sucesso. A empresa atualmente aposta na adoção do framework OpenCL[4] (*Open Computing Language*), desenvolvido no final do ano 2008 (MUN 2009). Tal framework busca padronizar o desenvolvimento de aplicações para arquiteturas heterogêneas.

3.3.1 Processadores

A arquitetura ATI Stream possui uma hierarquia de processamento ilustrada na Figura 3.4. As placas gráficas seguem um certo padrão, tendo suas principais diferenças no número de processadores chamados **SIMD Engines**. Cada SIMD Engine contém múltiplos núcleos simples, todos controlados pela mesma Unidade de Instrução. Cada núcleo possui cinco ULAs e executa instruções VLIW (*Very Large Instruction Word*). Assim, até cinco operações escalares podem ser emitidas paralelamente em cada núcleo. Além disso, operações de ponto flutuante de precisão dupla utilizam quatro ULAs. Isto faz com que o desempenho máximo de precisão dupla seja cinco vezes menor do que o desempenho máximo para precisão simples.

Todas as threads em um mesmo SIMD Engine executam a mesma instrução. Múltiplas threads são executadas em IMT para esconder a latência de acesso à memória (YAN 2007). Assim, quatro diferentes threads podem disparar cada qual uma instrução VLIW em um tempo de quatro ciclos. O conjunto destas múltiplas threads em um SIMD Engine é chamado de ***wavefront***. O número de threads em um *wavefront* depende da placa gráfica utilizada (ATI 2009).

Divergências entre as threads em um mesmo SIMD Engine (por exemplo, diferentes

[3]http://www.amd.com/stream
[4]http://www.khronos.org/opencl/

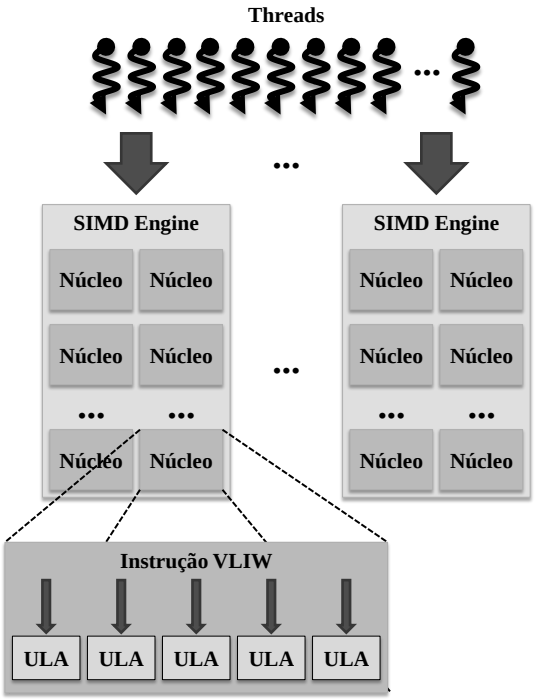

Figura 3.4: Organização da arquitetura ATI Stream.

escolhas tomadas em uma instrução if) são resolvidas sequencializando os diferentes caminhos. Além disso, os SIMD Engines despacham suas instruções de forma independente entre si.

3.3.2 Memória

A memória é organizada em três principais domínios:

- Memória do hospedeiro - de acesso privativo pela CPU;
- Memória do dispositivo - de acesso privativo pela GPU;
- Memória PCIe compartilhada - trecho de memória do hospedeiro disponibilizada para transferências entre a CPU e a GPU. Ambos podem alterá-la.

Em um acesso à memória do dispositivo pelas threads, uma linha de memória é carregada e armazenada em cache. Nos documentos referentes à arquitetura não são especificados detalhes sobre hierarquia de cache nem tamanho desta memória.

Os acessos à memória do dispositivo tendem a ser uma ordem de grandeza mais rápidos do que o acesso à memória compartilhada. Além disso, acessos aos registradores tendem a ser mais rápidos do que leituras na cache. Threads que acessam a memória compartilhada são bloqueadas enquanto suas requisições são processadas. Escritas simultâneas a até oito segmentos de memória são permitidas pela arquitetura, mas apenas uma vez por invocação de kernel.

3.4 Cell Broadband Engine Architecture

A arquitetura dos processadores **Cell** [5] é fruto de uma aliança entre as empresas Sony, Toshiba e IBM (GSC 2006). O projeto, iniciado em 2000, buscava uma arquitetura para ambientes de processamento intensivo de dados, objetivando um aumento em desempenho em uma ordem de magnitude sobre os computadores domésticos a serem produzidos em 2005. Entre as metas do projeto estavam a redução da área por núcleo, o aumento do número de núcleos por chip e um aumento no desempenho por área.

Os processadores Cell foram utilizados na sua versão primária no PlayStation 3, focados no processamento de imagens e jogos complexos. A segunda versão, PowerXCell 8i, é utilizada em lâminas em sistemas de alto desempenho, como o RoadRunner (BAR 2008). Os sistemas híbridos que utilizam o PowerXCell 8i estão entre os sistemas computacionais de maior desempenho e maior desempenho por watt.

3.4.1 Processadores

O desenho de núcleos mais simples leva a uma melhora na frequência de relógio e eficiência no consumo de energia (CRA 2008). Assim, o processador Cell utiliza-se de 8 núcleos simples SIMD chamados *Synergistic Processor Elements* (**SPE**), ou Elementos de Processamento Sinérgicos. Estes núcleos são especializados para processamento vetorial. Cada SPE possui 128 registradores de 4 palavras (16 bytes) de tamanho. Não há circuitos dedicados para operações escalares, executando-se todas as operações sobre uma unidade aritmética vetorial. No caso de computações escalares, os dados lidos da memória são, primeiramente, alinhados nos registradores. Em seguida, a operação é executada sobre todo o registrador. Por fim, o resultado pode sofrer um deslocamento na posição dentro do registrador.

Os 8 SPEs são controlados por outro núcleo, chamado *Power Processor Element* (**PPE**), ou Elemento de Processamento Power, pois é um núcleo baseado na arquitetura Power da IBM. Esse núcleo de 64 bits também pode ser visto como um núcleo hospedeiro, delegando tarefas computacionalmente complexas aos SPEs.

O PPE suporta 2 threads simultaneamente, enquanto cada SPE suporta apenas 1 thread. Porém, um SPE possui 2 pipelines: o par, dedicado à aritmética de ponto flutuante e ponto fixo; e o impar, que implementa as outras funções, como leitura, escrita e aritmética sobre números inteiros. Com cada SPE executando a uma frequência de $3,2$ GHz, um processador Cell da primeira geração alcança um pico de $204,8$ GFlops teóricos na execução de ponto flutuante precisão simples. Porém, como as operações de precisão dupla não conseguem aproveitar inteiramente o pipeline, o desempenho máximo em precisão dupla é de $14,6$ GFlops (BAR 2008). Essa questão foi resolvida com o processador PowerXCell 8i, que alcança até $102,4$ GFlops em precisão dupla. As operações de ponto flutuante são compatíveis com o padrão IEEE, menos o tratamento de exceções e pela troca de alguns valores padrão, ambos em precisão simples (GSC 2006).

Além dos componentes da arquitetura Cell mencionados anteriormente, há ainda uma unidade lógica para testes, monitoramento e depuração (PHA 2005).

3.4.2 Memória e comunicação

Os núcleos são interligados através de um barramento chamado *Element Interconnect Bus* (**EIB**), ou Barramento de Interconexão de Elementos. Além dos núcleos, estão ligados ao EIB um controlador de acesso a memória e um controlador de entrada e saída.

[5] https://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine

Esse barramento é formado por 4 anéis de 16 bytes, utilizados para a transferência de dados, 2 em cada direção. Além disso, tem-se um caminho de comandos em estrela. O EIB trabalha com a metade da frequência de relógio dos SPE (PHA 2005).

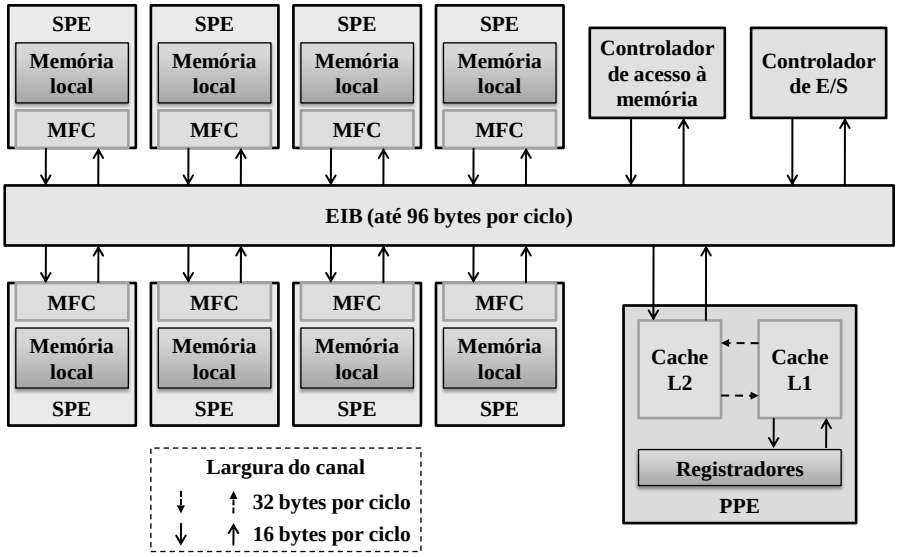

Figura 3.5: Diagrama do processador Cell com foco na hierarquia de memória e comunicação.

O PPE possui cache em 2 níveis e acessa a memória principal diretamente, através do EIB. Já os SPEs não possuem memória cache por não ser interessante em um ambiente com aplicações com fluxos de dados, onde não se tem um bom índice de reuso dos dados (CRA 2008). Além disso, um SPE não acessa a memória principal diretamente. Cada SPE possui um controlador de memória chamado *Memory Flow Controller* (**MFC**), como ilustrado na Figura 3.5. O MFC provê transferências de dados coerente para a memória local do SPE, em blocos de até 16 KBytes, através de pedidos de DMA (*Direct Memory Access*) (GSC 2007). A transferência de dados acontece em paralelo ao processamento no núcleo. Para esconder a latência no acesso à memória principal utilizasse uma técnica chamada *multi-buffering* - enquanto o SPE executa sobre um conjunto de dados em sua memória local, o MFC copia os próximos dados.

3.4.3 Programação

O uso de processadores Cell em sistemas híbridos está baseado em 2 componentes principais para programação:

- *Data Communication and Synchronization* (**DaCS**) - interface para transferência de dados ponto a ponto para múltiplas plataformas;
- *Accelerated Library Framework* (**ALF**) - facilitador da divisão de trabalho no sistema. Visualiza a plataforma como um hospedeiro e um conjunto de aceleradores.

Esses componentes buscam apresentar o sistema de forma opaca ao invés de transparente, ou seja, trazer facilidades de programação mas sem esconder as características da arquitetura, disponibilizando formas de melhorar o desempenho na presente plataforma (CRA 2008). Através destas diferentes bibliotecas, é possível desenvolver programas segundo diferentes modelos de aceleração. Entre as formas principais está o pipeline

de funções, onde cada SPE executa algumas funções sobre um conjunto de dados e passa os resultados para o próximo SPE. Além desta, trabalha-se com um modelo de *Remote Procedure Call* (RPC), onde um SPE recebe uma requisição e devolve o resultado da computação. Por fim, também pode-se manter cada SPE funcionando de forma autônoma (GSC 2007). Um programa normalmente começa sua execução como uma thread no PPE e posteriormente dispara mais threads no PPE ou em SPEs, controlado as threads de forma baseada no modelo de pthreads.

Além de programas desenvolvidos para a arquitetura Cell, os processadores podem executar programas desenvolvidos para a arquitetura Power e *spulets*, programas que executam virtualizados em SPEs.

3.5 Discussão

Nesta seção são resumidas características das diferentes arquiteturas apresentadas anteriormente. A Tabela 3.1 apresenta tais informações. É importante ressaltar que as empresas apresentam suas arquiteturas com nomenclaturas diferentes. Assim, a organização em tópicos da tabela é baseada em conceitos mais gerais.

As quatro arquiteturas apresentadas neste capítulo foram desenvolvidas tanto para processamento gráfico quanto para processamento de aplicações de propósito geral. Além disso, estas arquiteturas são acessíveis ao programador de forma opaca, apresentando sempre alguma forma de controle de memória, buscando equilibrar a facilidade de programação com o desempenho.

As arquiteturas dos processadores Cell e Larrabee são mais próximas das CPUs multicore, tendo menor número de threads e podendo trabalhar com paralelismo de tarefas. Já as arquiteturas CUDA e ATI Stream utilizam-se de milhares de threads entrelaçadas processando dados independentes para obter desempenho. Enquanto as arquiteturas Larrabee e ATI Stream utilizam cache, CUDA apresenta caches pequenas e somente de leitura e o processador Cell utiliza apenas DMA nos seus SPEs. Este fato torna o último menos indicado para o paralelismo de dados, pois ele não possui uma memória compartilhada simples.

Considerando as arquiteturas estudadas, a que mais se aproxima da arquitetura CUDA - utilizada neste trabalho - é a arquitetura ATI Stream. Ambas utilizam o entrelaçamento de threads para esconder latências de memória, apresentam organizações hierárquicas de processamento, contém apenas uma Unidade de Instruções para cada conjunto de núcleos e conseguem agregar acessos a memória principal da placa gráfica. Entretanto, CUDA possui uma hierarquia de memória complexa (o que envolve memórias compartilhas dentro de cada SM e caches apenas para algumas memórias específicas e limitadas), enquanto a arquitetura ATI Stream utiliza apenas caches. Além disso, CUDA possui formas claras de comunicação por memória compartilhada e sincronização entre grupos de threads, enquanto tais características não foram encontradas na segunda arquitetura.

Analisando as características das arquiteturas apresentadas, chega-se à conclusão de que os resultados obtidos com a arquitetura CUDA não poderão ser estendidos para as arquiteturas Larrabee e Cell, devido às suas grandes diferenças. Acredita-se que tais resultados possam ser estendidos parcialmente para a arquitetura ATI Stream, faltando apenas um método de mapeamento das formas de comunicação entre threads vistas em CUDA para esta arquitetura.

Tabela 3.1: Comparativo entre arquiteturas de GPUs

Característica	**CUDA**	**Larrabee**	**ATI Stream**	**Cell**
Número de núcleos:	>100	>10	>100	<10
Organização dos núcleos:	hierárquica (multiprocessadores com 8 núcleos)	independentes	hierárquica (SIMD Engine com múltiplos núcleos)	hierárquica (processador principal controla processadores secundários)
Comunicação entre núcleos:	memória principal entre SMs e memória compartilhada dentro do SM	via cache L2	memória	DMA
Número de threads em paralelo por núcleo:	4 em pipeline	2	4 em pipeline com instruções VLIW	1
Heterogeneidade dos núcleos:	homogêneos	homogêneos	homogêneos	heterogêneos
Organização de memória:	hierárquica	NUCA	hierárquica	hierárquica
Cache:	parcial	total	parcial	sem cache
Conjunto de instruções:	próprio	derivado do x86 com extensões	VLIW próprio	próprio
Instruções vetoriais:	não (SIMT)	sim	não (SIMT)	sim
Granularidade de paralelismo:	dados	dados e tarefas	dados	dados e tarefas
Possibilidade de executar um SO independentemente:	não	não	não	sim

4 IMPLEMENTAÇÃO DOS NPB PARA CUDA

Os **NAS Parallel Benchmarks** (NPB)[1] (BAI 95) foram originalmente desenvolvidos para a avaliação de desempenho de supercomputadores altamente paralelos. Os NPB são baseados em aplicações e núcleos de computação paralelos de dinâmica de fluídos. Com o passar dos anos, versões foram desenvolvidas com bibliotecas como OpenMP (JIN 99) e MPI (BAI 95), assim como versões multinível para ambientes híbridos. Com isso, os benchmarks passaram a ser utilizados também para a avaliação de outros sistemas paralelos. Este capítulo apresenta este conjunto de benchmarks e a implementação de dois deles, **EP e FT**, para CUDA. A versão dos benchmarks utilizada neste trabalho é a 2.3 implementada na linguagem C com OpenMP.

A criação deste conjunto de benchmarks é justificada pelo NAS (*NASA Advanced Supercomputing Division*, anteriormente *Numerical Aerodynamic Simulation Program*) na incapacidade de outros benchmarks da época (LINPACK, por exemplo) avaliarem o desempenho de máquinas altamente paralelas (BAI 94). A escolha de um conjunto de benchmarks, ao invés de uma única aplicação, deve-se ao fato que a migração de uma grande aplicação para uma nova arquitetura paralela é trabalhosa e não justificável para apenas a obtenção de resultados de desempenho. Além disso, benchmarks devem ser genéricos o bastante para não favorecer algum aspecto de arquitetura específico nem necessitarem de muitas otimizações para diferentes arquiteturas (BAI 95). Por fim, seus requisitos devem ser moldáveis o bastante para se adaptar a características de novos sistemas. Assim, foram desenvolvidos e disponibilizados originalmente 8 benchmarks e 3 classes de problemas de diferentes tamanhos para estes. As versões mais atuais incluem outros 3 benchmarks e 3 novas classes de problema. Entre as regras para a implementação dos NAS Parallel Benchmarks, é especificado que **todas as operações de ponto flutuante devem ser executadas com precisão dupla** (palavras de 64 bits). Uma listagem dos benchmarks é apresentada na Tabela 4.1.

As vantagens do conjunto de benchmarks do NAS incluem sua disponibilidade (sendo gratuito), diversidade (apresentando múltiplos benchmarks e classes de problemas), foco em processamento de alto desempenho e portabilidade (ALV 2009). Além disso, há um mapeamento claro entre seus benchmarks e a classificação Dwarf Mine (ASA 2006).

Outra característica interessante dos benchmarks é que eles já incluem os códigos de verificação de resultados, de captura de tempo de execução e cálculo de desempenho em milhões de operações por segundo (Mops). Tais verificações e medições são feitos de forma similar para cada uma das seis classes de problema dos benchmarks. As diferentes classes são listadas abaixo:

- S - Sample: Instância mais simples de problema, utilizada apenas para verificação

[1] http://www.nas.nasa.gov/Resources/Software/npb_changes.html

Tabela 4.1: Benchmarks Paralelos do NAS

Sigla	**Nome**	**Característica principal**
BT	*Block Tridiagonal solver*	Aplicação de dinâmica de fluídos que resolve equações tridimensionais com matrizes densas.
CG	*Conjugate Gradient*	Utiliza estruturas de dados de grades computacionais desestruturadas. Usada para testes de comunicações irregulares.
DC	*Data Cube Operator*	Utilizado para verificar o desempenho de sistemas com hierarquias de memória complexas. Presente a partir da versão 3.1 em forma serial e 3.2 com OpenMP.
DT	*Data Traffic*	Apenas presente em versão MPI. Presente a partir da versão 3.2.
EP	*Embarrassingly Parallel*	Usado para estimar o limite de desempenho atingível com ponto flutuante.
FT	*Fast Fourier Transform*	Soluciona equações diferenciais parciais de 3 dimensões, presente em diversas aplicações.
IS	*Integer Sort*	Utilizado para verificação do desempenho de comunicações e cálculos de inteiros.
LU	*Lower-Upper solver*	Similar a BT, porém utiliza métodos diferentes para solução.
MG	*MultiGrid*	Usado para testes na comunicação de dados estruturados.
SP	*Scalar Pentadiagonal solver*	Similar a BT, porém trabalha com estruturas de dados de grades estruturadas.
UA	*Unstructured Adaptive*	Usado para verificar o desempenho do sistema sob acessos de memória irregulares e dinâmicos. Presente a partir da versão 3.1.

do funcionamento do benchmark na máquina. Não possui propósito de medição de desempenho.

- W - Workstations: Instância pequena, visando computadores com restrições no tamanho da memória principal. Também escolhido para uso em simulações, devido ao seu balanço entre tempo de execução, escalabilidade e variação (ALV 2009).

- A, B, C e D - Maiores instâncias, em ordem crescente. Apenas as classes A e B existiam originalmente. Como os benchmarks foram desenvolvidos para serem extensíveis, novas classes foram adicionadas quando as anteriores pararam de representar os limites dos sistemas atuais.

As implementações dos benchmarks disponibilizadas pelo NAS estão escritas em FORTRAN. Para este trabalho, foi utilizada uma versão na linguagem C disponibilizada pelo Projeto de Compilador Omni (F2C)[2] da Universidade de Tsukuba, Japão. Nas próximas subseções são apresentados maiores detalhes sobre os benchmarks EP e FT e suas implementações para CUDA[3].

4.1 Embarrassingly Parallel

O benchmark EP (*Embarrassingly Parallel*) implementa a geração de pares de números aleatórios baseados em desvios gaussianos independentes. Este problema é típico em diversas aplicações de simulação de *Monte Carlo*. O benchmark encontra-se incluso na categoria 7 (*MapReduce*) da classificação Dwarf Mine (ASA 2006). Os números aleatórios podem ser calculados de forma completamente independente. Além disso, os requisitos de memória para este problema são reduzidos, pois espaços de memória podem ser reutilizados em uma implementação com iterações. Este benchmark foi escolhido como o primeiro a ser paralelizado por estimar o limite de desempenho com ponto flutuante que pode ser obtido com uma arquitetura (JIN 99).

A versão OpenMP do benchmark é caracterizada por um grande laço paralelo, onde o índice da iteração é utilizado para a geração de diferentes números aleatórios. A Figura 4.1 representa a organização do benchmark em um fluxograma. Internamente ao laço, estão presentes cálculos em laços não paralelizáveis, os quais envolvem a geração de números aleatórios e a verificação da adequação desses números segundo função de aceitação. São utilizadas memórias locais a cada thread, havendo apenas uma redução por soma dos valores locais para dois valores escalares (sx e sy) e um vetor ($q[]$). Este benchmark e sua categoria, *MapReduce*, apresentam alta intensidade aritmética, regularidade nas computações e nos acessos a memória, independência de dados e comunicação quase ausente.

4.1.1 Implementação para CUDA

O primeiro passo para a transformação do código em OpenMP para CUDA foi a alteração do grande laço paralelo em um kernel, que passa a executar na GPU. Com isso, o índice da iteração acaba sendo substituído pelo identificador da thread.

As variáveis foram divididas em duas categorias: temporárias e permanentes. As variáveis temporárias são utilizadas nas computações parciais do kernel. Já as variáveis

[2]http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/

[3]Implementações para CUDA disponíveis para download em http://hpcgpu.codeplex.com/

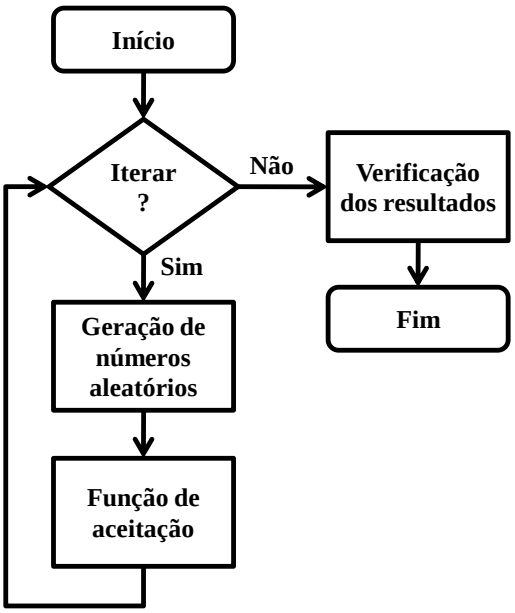

Figura 4.1: Representação do benchmark EP em fluxograma simplificado.

permanentes são utilizadas para manter na GPU dados que são utilizados por mais de um kernel, assim como para a comunicação de dados entre CPU e GPU. As variáveis temporárias são alocadas internamente ao kernel e as permanentes são agregadas em vetores e alocadas na CPU através de chamadas à biblioteca CUDA ($cudaMalloc()$). Porém, decidiu-se também alocar permanentemente os vetores temporários que seriam utilizados pelas threads. Essa escolha busca evitar o uso da memória local (por thread), o que pode levar a uma redução do desempenho por não poder acessada de forma coalescida.

Os vetores locais (por thread) foram agregados em um grande vetor de vetores, ou matriz contígua. Como a alocação em memória da GPU é feita em vetores, a transformação de uma posição bidimensional para uma unidimensional é calculada manualmente. Este tipo de estrutura de dados será referenciada simplesmente como *matriz* a partir deste momento. A Figura 4.2(a) ilustra o acesso a este tipo de matriz, onde T, M e tid representam o número de threads, a quantidade de memória para cada thread e o identificador de cada thread, respectivamente.

O acesso às matrizes em linhas, como representado na Figura 4.2(a), apesar de ter uma implementação mais simples, acaba sendo prejudicial ao desempenho da aplicação. Isto acontece devido ao fato de que os acessos à memória global, onde estão armazenadas as matrizes, acabam sendo feitos individualmente (não coalescidos), subutilizando a largura de banda do acesso à memória. Este problema foi resolvido com a alteração para acesso às matrizes em colunas, coalescendo (agregando) todos os acessos à memória. Esta idéia é apresentada em (LEE 2009). A Figura 4.2(b) ilustra tal padrão de acesso implementado.

Os valores das variáveis sx, sy e $q[]$ são obtidos através de uma redução de soma dos valores obtidos em cada thread. Para organizar esta redução, os valores por thread foram armazenados na memória global da GPU em forma de 2 vetores e 1 matriz. Com isso, foi possível implementar um kernel de redução. A necessidade de um segundo kernel está ligada ao fato de que sincronizações globais entre as threads somente acontecem no início e no fim da execução dos kernels (NVI 2009).

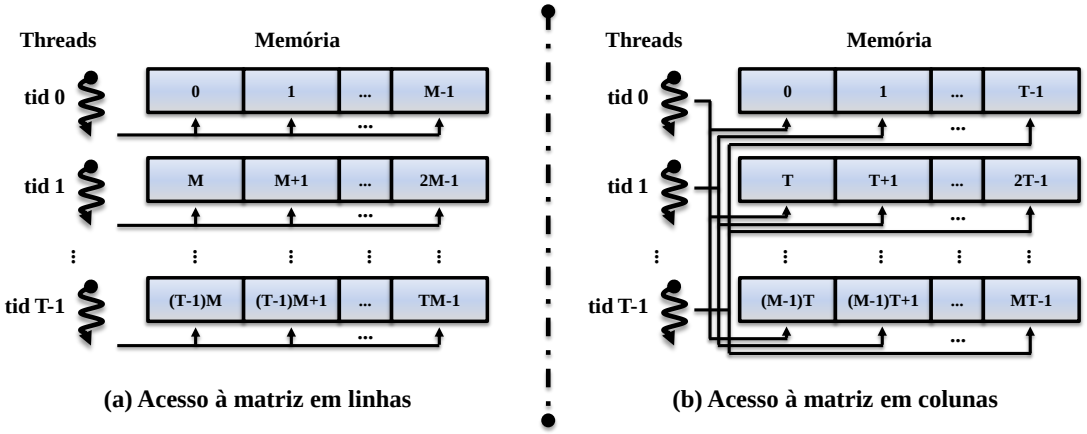

Figura 4.2: Acesso a vetores agregados (matriz) pelas threads em GPU.

O kernel de redução é executado por 3 blocos de threads. Para cada bloco, é destinada a redução de um dos valores. As funções de redução utilizam-se de memória compartilhada para a comunicação entre threads de um mesmo bloco. O destino de cada função de redução para um bloco de threads está ligada a três características de CUDA, listadas abaixo.

- Comunicação através de memória compartilhada: os resultados parciais da redução podem ser comunicados entre threads de um mesmo bloco de forma rápida;
- Sincronização entre threads: através da primitiva $\_\_syncthreads()$, é possível sincronizar as threads em um mesmo bloco. Com isso, tem-se a garantia que os valores comunicados estão corretos;
- Serialização de divergências: os problemas com queda de desempenho devido a divergências nos caminhos de execução das threads somente acontecem entre threads de um mesmo bloco. Assim, com a separação de um bloco para cada função de redução, é possível emular a execução de 3 diferentes kernels sem perda de desempenho.

Como cada thread possui seus próprios requisitos de memória (aproximados 16KB), o paralelismo na execução do kernel está limitada pelo tamanho da memória principal da GPU. Para resolver instâncias maiores do benchmark que sofrem com essa limitação, foi adicionado ao kernel principal um laço que itera sobre todo ele, similar ao laço paralelo original em OpenMP, ilustrado na Figura 4.1. Com isso, caso a instância seja maior do que a memória comportaria, acaba-se por reutilizar a memória de cada thread, fazendo com que elas calculem números aleatórios utilizando novos índices.

4.2 Fast Fourier Transform

O benchmark FT (*Fast Fourier Transform*) implementa a Transformada rápida de Fourier (FFT), método presente em diversas aplicações científicas. Este benchmark encontra-se incluso na categoria 3 (*Spectral Methods*) da classificação Dwarf Mine (ver Seção 2.2). O algoritmo é executado em fases síncronas, envolvendo cálculos e permutações de dados em matrizes tridimensionais de números complexos. Este problema possui sérios requisitos de memória, diferentemente do benchmark EP. Este benchmark foi escolhido devido a sua representatividade.

A Figura 4.3 representa a organização do benchmark em um fluxograma. A versão OpenMP do benchmark é caracterizada pela iteração de uma sequência de funções, não havendo paralelismo entre elas. Entretanto, cada função possui seu próprio paralelismo interno. Os cálculos nas diferentes dimensões apresentam diferentes padrões de acesso à memória. São utilizadas memórias locais a cada thread para a execução dos trechos paralelos e matrizes em memória global para comunicação dos resultados parciais. O próprio benchmark possui uma função de *checksum* para a verificação dos resultados entre cada iteração. Este benchmark e sua categoria, *Spectral Methods*, apresentam uma intensidade aritmética reduzida, regularidade dentro de cada iteração, dependência de dados e comunicação de dados utilizando diferentes padrões.

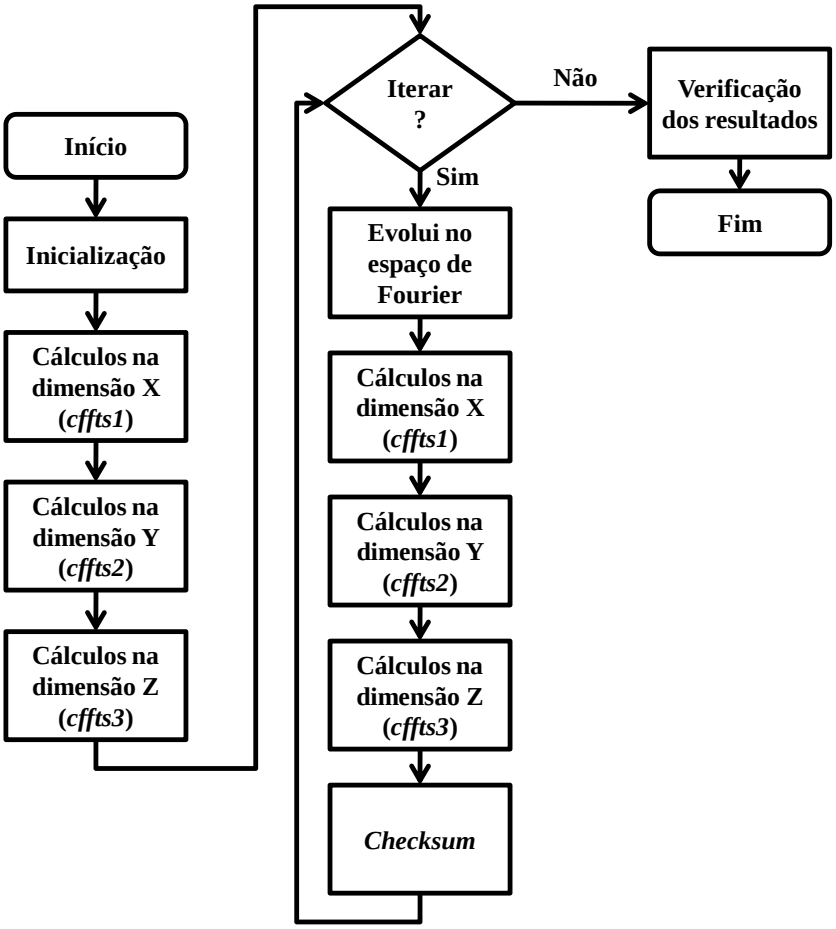

Figura 4.3: Representação do benchmark FT em fluxograma simplificado.

4.2.1 Implementação para CUDA

O framework de CUDA possui uma biblioteca de FFT, chamada CUFFT (NVI 2009). Para a utilização desta biblioteca, seria necessário armazenar a matriz principal de números complexos através de chamadas a funções de alocação da CUFFT. Porém, esta biblioteca não foi utilizada pois a função de *checksum* do benchmark precisa acessar valores armazenados nessa matriz.

A primeira alteração do código OpenMP para CUDA foi na separação das matrizes e vetores de números complexos - representados como dois números em ponto flutuante de precisão dupla - em duas: uma para a parte real e a outra para a parte imaginária. Esta

Tabela 4.2: Parâmetros de funções do benchmark FT.

Função	**Dimensão**	**Paralelismo**	**Equação de Nova Posição**
$cfffts1()$	NX	$NY \times NZ$	$pos(x,y,z) = z + y \times NZ + x \times NZ \times NY$
$cfffts2()$	NY	$NX \times NZ$	$pos(x,y,z) = x + z \times NX + y \times NX \times NZ$
$cfffts3()$	NZ	$NX \times NY$	$pos(x,y,z) = x + y \times NX + z \times NX \times NY$

separação foi necessária para possibilitar acessos coalescidos à memória global da GPU. Além disso, as matrizes tridimensionais foram linearizadas, para poderem ser alocadas na GPU. A Equação 4.1 apresenta uma mesma posição na matriz segundo as duas diferentes formatações, onde NX, NY e NZ representam o tamanho das 3 dimensões e x, y e z representam valores para cada uma das dimensões ($0 \leq x \leq NX$, $0 \leq y \leq NY$, $0 \leq z \leq NZ$).

$$matriz\_3D[z][y][x] = matriz\_linearizada[x + y \times NX + z \times NX \times NY] \quad (4.1)$$

Variáveis locais e globais foram ajustadas conforme necessidades da GPU. Algumas variáveis que ficavam em memória global mas não eram alteradas durante a execução do benchmark foram transformadas em constantes. Esta alteração buscou reduzir o número de acessos desnecessários à memória global da GPU, assim como transferências entre as memórias da CPU e da GPU. Além disso, vetores locais às threads foram transformados em vetores agregados (ou *matrizes*, conforme definido na Subseção 4.1.1).

Inicializações sequenciais foram mantidas na CPU. É importante ressaltar que essas inicializações estão inclusas no tempo medido pelo benchmark. Apesar do interesse de armazenar alguns vetores na memória de texturas da GPU que somente são inicializados e alterados em CPU, isto não foi possível pois texturas não podem armazenar valores em precisão dupla.

A função de *checksum* foi dividida em duas partes. A primeira executa em GPU. Ela faz computações em paralelo e a redução dos resultados parciais. A segunda parte trata de apresentar na saída padrão os valores obtidos. Assim, a transferência de todos os valores obtidos no *checksum* é feita de uma só vez. Esses valores são utilizados posteriormente pela função de verificação de corretude dos resultados do benchmark.

A parte computacionalmente mais intensa do código envolve 3 funções, $cfffts1()$, $cfffts2()$ e $cfffts3()$, as quais executam sobre dados nas dimensões X, Y e Z, respectivamente. Internamente, tais funções chamam uma mesma função, $cfftz()$, a qual executa os cálculos sobre os diferentes dados. Os laços internos de cada uma das 3 funções foram colapsados e, posteriormente, transformados em chamadas para a execução paralela do kernel $cfftz()$. A matriz a ser passada para o kernel é rotacionada por outra função na GPU para que os acessos à memória possam ser coalescidos. Após a execução do kernel, os dados são rotacionados de forma inversa, para manter a formatação original da matriz. A Tabela 4.2 apresenta informações referentes às 3 funções, como as dimensões em que elas executam, o número de tarefas que podem ser executadas concorrentemente e as equações para o cálculo da nova posição dos dados (para acessar a memória global de forma coalescida). Na tabela, NX, NY e NZ representam o tamanho das três dimensões da matriz.

5 RESULTADOS

Este capítulo apresenta resultados experimentais obtidos na execução dos benchmarks EP e FT nas suas versões originais (sequencial e OpenMP) e paralelizadas para CUDA. A Seção 5.1 apresenta a metodologia utilizada nos experimentos, a Seção 5.2 mostra resultados referentes ao benchmark EP e a Seção 5.3 trata sobre os resultados obtidos na execução do benchmark FT.

5.1 Metodologia

Os NPB possuem métodos próprios de captura de tempo e cálculo de desempenho. As unidades apresentadas nos resultados são tempo e milhões de operações por segundo (Mops), onde o tipo de operação está ligada ao próprio benchmark. Para a métrica tempo, quanto menor seu valor, melhor o resultado. Já para a métrica Mops, o contrário é verdadeiro. Os resultados apresentados neste capítulo representam os valores médios obtidos na execução dos benchmarks com uma confiança de 95% e erro relativo de 10% (utilizando a distribuição t de Student). As diferentes versões dos benchmarks foram executados um mínimo de 6 vezes, com uma execução inicial extra descartada. O ambiente de execução dos experimentos é apresentado abaixo. As execuções com OpenMP utilizam 2 threads, uma em cada núcleo.

- Processador: Intel Core 2 Duo E8500 $3,16$ GHz.
- Memória: 4 GB DDR2 1066 MHz.
- Placa Gráfica: GeForce GTX 280 com 1 GB de memória (*Compute Capability* 1.3).
- Sistema Operacional: Ubuntu 9.04 32 bits (Kernel Linux 2.6.28-15-generic).
- CUDA Driver: 2.3 (190.18 Beta)
- Compiladores:
  - CUDA: nvcc versão $0.2.1221$.
  - OpenMP e Sequencial: gcc versão $4.3.1$.
- Opções de compilação:
  - CUDA: -O3 -arch sm_13.
  - OpenMP: -O3 -fopenmp
  - Sequencial: -O3

As comparações de desempenho consideram o mesmo ambiente executando apenas uma thread, duas threads (usando OpenMP) e múltiplas threads em GPU. No último caso, as computações não acontecem de forma concorrente entre CPU e GPU. A métrica de ganho de desempenho, ou *speedup*, e sua notação utilizada neste trabalho são definidos na Equação 5.1, onde $t(X)$ e $t(Y)$ representam os tempos médios de execução de diferentes versões do benchmark. Esta forma de comparação - utilizando o mesmo ambiente - foi escolhida pelo autor, não existindo ainda um consenso entre as diversas referências sobre como comparar o desempenho de CPUs e GPUs.

$$X \rightarrow Y \iff \frac{t(X)}{t(Y)} \tag{5.1}$$

Os métodos de captura de tempo dos benchmarks consideram apenas as suas regiões paralelas. O benchmark FT é uma exceção, pois também conta com a medição dos custos de inicialização sequenciais. As implementações dos benchmarks para CUDA incluem também os tempos de alocação de memória na GPU e transferência de dados. Esta escolha foi feita baseada devido aos sobrecustos ligados à comunicação entre CPU e GPU, os quais podem ser significativos no tempo total de execução de uma aplicação.

5.2 Embarrassingly Parallel

O benchmark EP foi executado para as instâncias de problema W, A e B, cujas quantidades de números aleatórios gerados são 2^{26}, 2^{29} e 2^{31}, respectivamente. A geração de números aleatórios é a unidade de operação considerada no cálculo de Mops (milhões de operações por segundo) do benchmark. Maiores instâncias do benchmark não são apresentadas devido a uma limitação do driver de CUDA em placas gráficas não dedicadas (utilizadas também com fins gráficos e não somente para cálculos), a qual cancela a execução de um kernel caso ele ultrapasse um limite de tempo.

A Figura 5.1 apresenta os tempos de execução do benchmark EP. O eixo horizontal representa as diferentes instâncias. O eixo vertical mostra os tempos medidos em segundos. As colunas representam tempos médios de execução para diferentes versões do benchmark. As interrupções nas colunas significam que seus valores estão acima do comportado pelo gráfico. Os resultados obtidos podem ser melhor compreendidos com a Figura 5.2, que apresenta o ganho comparativo de desempenho para as diferentes versões do benchmark EP. Aqui, o eixo vertical representa o *speedup* comparando duas implementações do benchmark.

Na Figura 5.2, pode-se ver um ganho de desempenho de $2\times$ comparando as versões sequencial e com OpenMP. Como o benchmark é, conforme seu nome indica, embaraçosamente paralelo, é esperado um ganho de desempenho linear ao número de unidades trabalhando em paralelo.

Considerando os ganhos de desempenho comparando as versões com OpenMP e para CUDA, tem-se *speedups* de $15\times$, $20\times$ e $21\times$ para as instâncias W, A e B, respectivamente. A maior diferença entre os ganhos para as instâncias W e A é devida a subutilização da GPU pela menor instância, não havendo nela paralelismo o bastante para aproveitar todos os recursos da GPU. Já para a instância B, a variação de desempenho é menor. Isto acontece porque, para este tamanho de problema, a memória da GPU limita a quantidade de threads que podem executar concorrentemente. Assim, para resolver o problema, é necessário executar iterações com as threads. A quantidade de computação por thread acaba crescendo, mas os custos de alocação de memória e das reduções manteve-se o mesmo

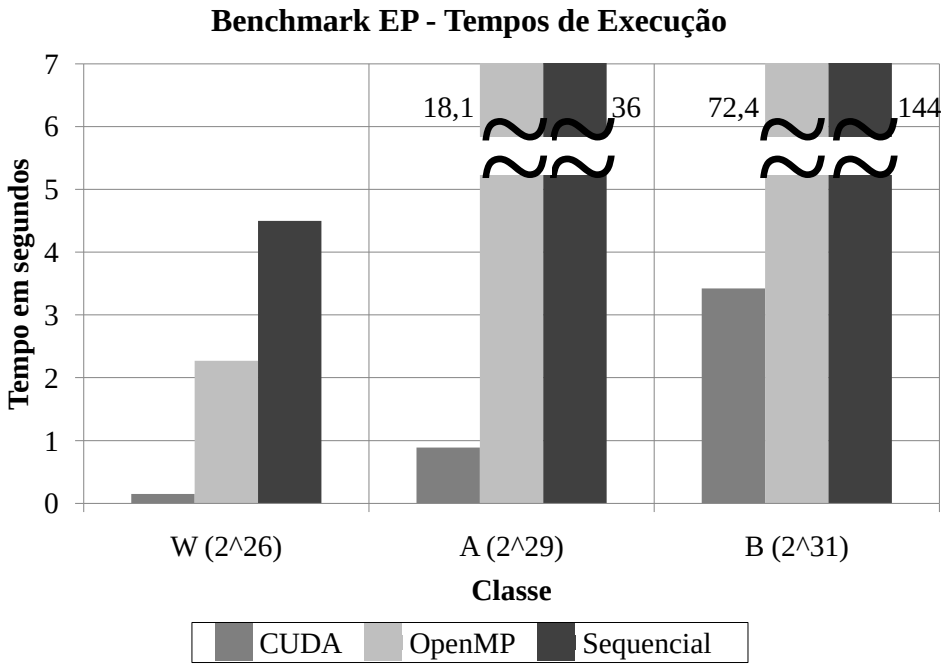

Figura 5.1: Tempos de execução para diferentes classes e implementações do benchmark EP.

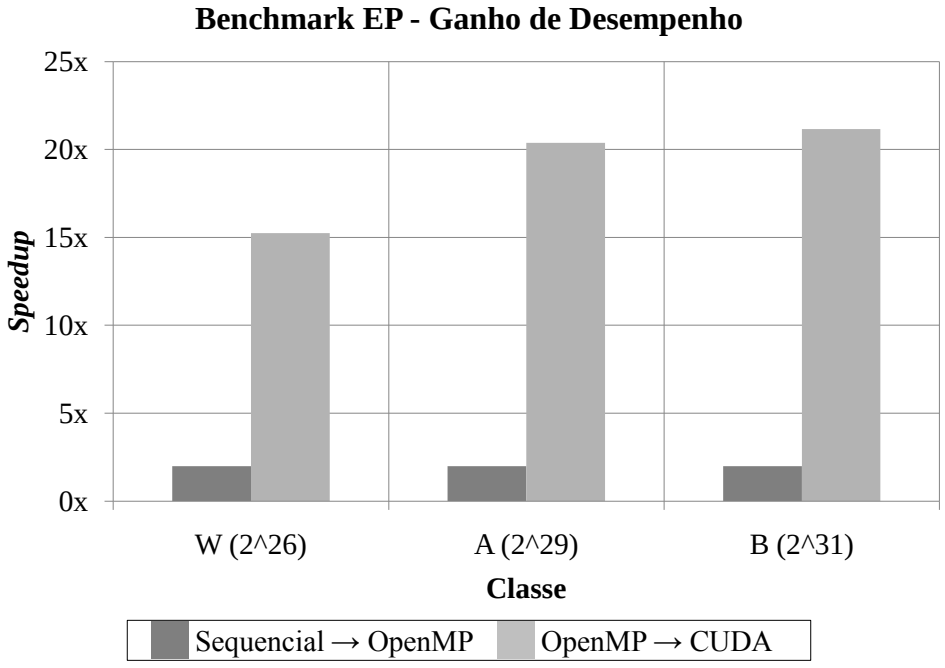

Figura 5.2: Ganho de desempenho para diferentes classes e implementações do benchmark EP.

que para a instância A. Com isso, os sobrecustos de transferência entre as memórias da CPU e da GPU acabam sendo um pouco mais absorvidos pelo desempenho da execução do kernel.

A Figura 5.3 apresenta a quantidade de números aleatórios gerados por segundo para as diferentes instâncias e implementações do benchmark. O eixo vertical representa esta quantidade. Estes valores mostraram-se constantes para as versões sequencial e com OpenMP, significando que os recursos de CPU são utilizados ao máximo mesmo na menor instância. Já para a versão para CUDA, estes valores acabam crescendo com o tamanho da instância do problema, aproveitando cada vez mais os recursos da placa gráfica e absorvendo os seus sobrecustos.

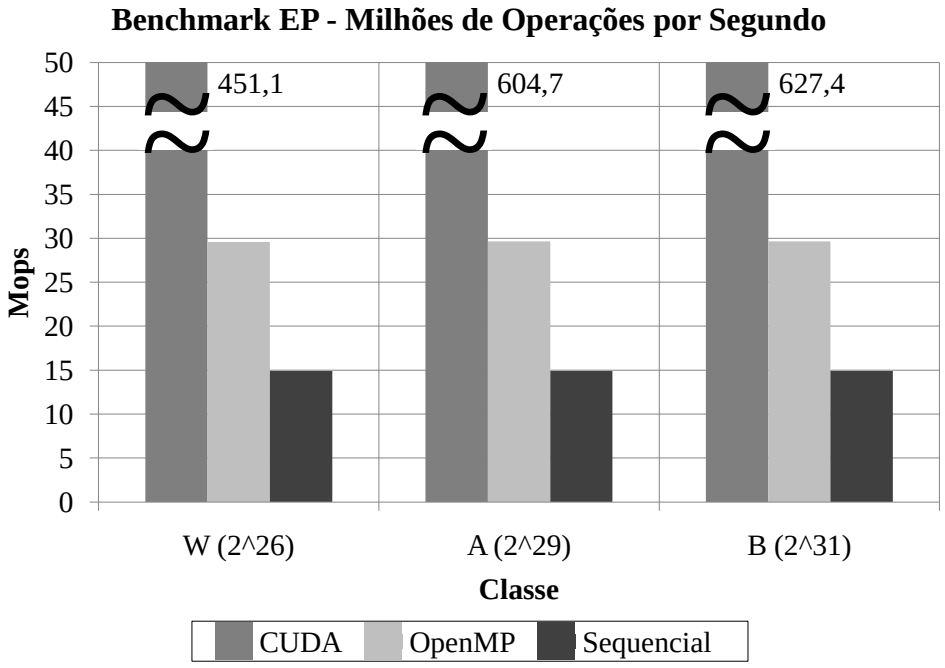

Figura 5.3: Número de operações por segundo para diferentes classes e implementações do benchmark EP.

5.3 Fast Fourier Transform

O benchmark FT foi executado para as instâncias de problema W e A, onde o tamanho das dimensões do problema ($NX \times NY \times NZ$) são $128 \times 128 \times 32$ e $256 \times 256 \times 128$, respectivamente. O número de instruções de ponto flutuante de precisão dupla é a unidade de operação considerada no cálculo de Mops do benchmark. Maiores instâncias do benchmark não foram executadas devido a limitações de memória da GPU (a GPU possui 1 GB de memória e a instância B do benchmark ocupa aproximadamente $1,4$ GB quando executada na CPU).

Para este benchmark, são apresentadas duas versões de código para CUDA: a versão 1 (**v1**), sem acessos coalescidos a memória; e a versão 2 (**v2**), com acessos coalescidos. A comparação entre estas duas versões visa examinar se os custos de permutação dos dados da matriz na versão 2 são menores do que os custos de execução sem leituras de dados coalescidas na versão 1. Esta informação é importante, tendo em vista que os padrões de permutação de dados são características marcantes deste benchmark e de sua categoria (*Structured Grids*).

A Figura 5.4 apresenta os tempos de execução do benchmark FT. O eixo horizontal

representa as diferentes instâncias. O eixo vertical mostra os tempos medidos em segundos. As colunas representam tempos médios de execução para diferentes versões do benchmark. As interrupções nas colunas significam que seus valores estão acima do comportado pelo gráfico. Na figura, pode-se perceber que os menores tempos foram obtidos com a versão 2 do código para CUDA. Os resultados obtidos podem ser melhor compreendidos com a Figura 5.5, que apresenta o ganho comparativo de desempenho para as diferentes versões do benchmark FT. Aqui, o eixo vertical representa o *speedup* comparando duas implementações do benchmark.

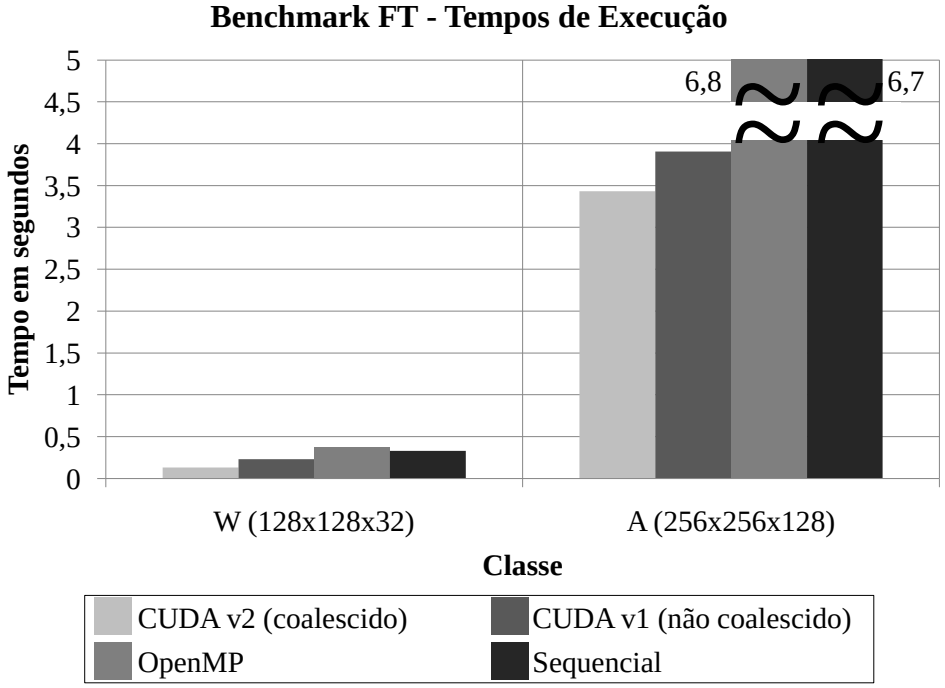

Figura 5.4: Tempos de execução para diferentes classes e implementações do benchmark FT.

Na Figura 5.5, pode-se ver uma perda de desempenho ao se utilizar OpenMP quando comparado a versão sequencial do benchmark. Isto ilustra que os custos de comunicação de dados acabam sendo maiores do que os ganhos ao executar duas threads em paralelo.

Ainda na figura, é visível um *speedup* de quase $3\times$ com a versão 2 do código para CUDA quando comparada a versão OpenMP para a instância W. Já para a instância A, o *speedup* é reduzido para aproximadas $2\times$, enquanto que o ganho de desempenho cresce junto com o tamanho das instâncias quando consideramos a versão 1 para CUDA.

A Figura 5.6 apresenta a quantidade de operações de ponto flutuante com precisão dupla executados por segundo para as diferentes instâncias e implementações do benchmark. O eixo vertical representa esta quantidade. Estes valores mostraram-se crescentes para as implementações com OpenMP e para CUDA na versão 1, e decrescentes para as outras implementações.

A diferença de comportamento para as duas diferentes versões para CUDA exigiu um estudo mais detalhado de seus tempos de execução. Para isso, foram capturados os tempos de execução das três funções que possuem implementações diferentes entre as duas versões do benchmark: $cfffts1()$, $cfffts2()$ e $cfffts3()$. Os valores obtidos para as diferentes instâncias do benchmark para a versão 1 (sem acesso coalescido) são apresentadas

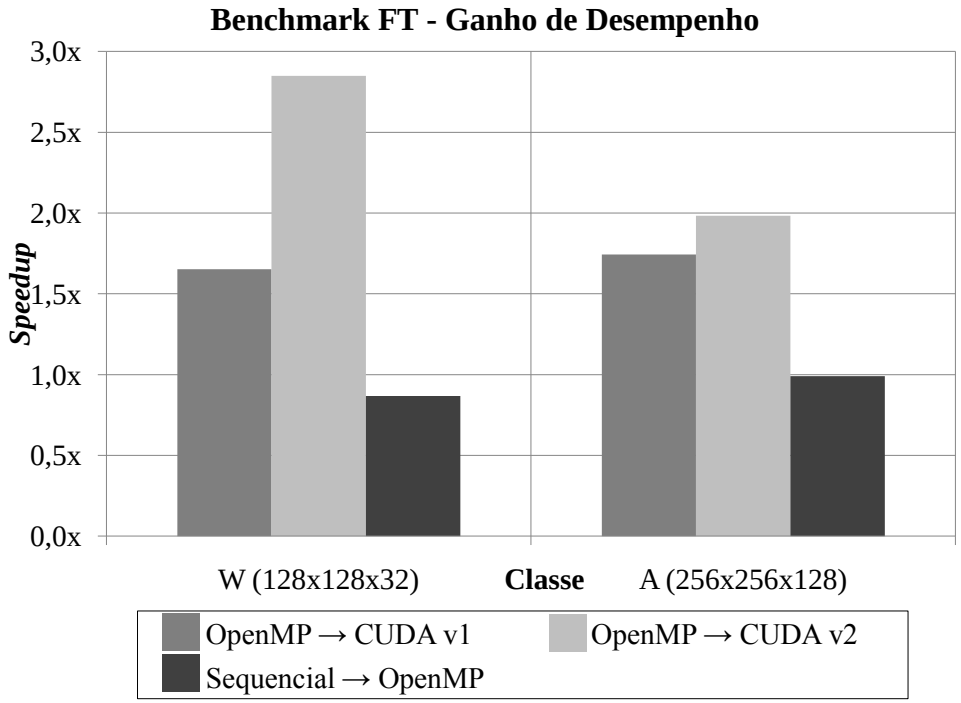

Figura 5.5: Ganho de desempenho para diferentes classes e implementações do benchmark FT.

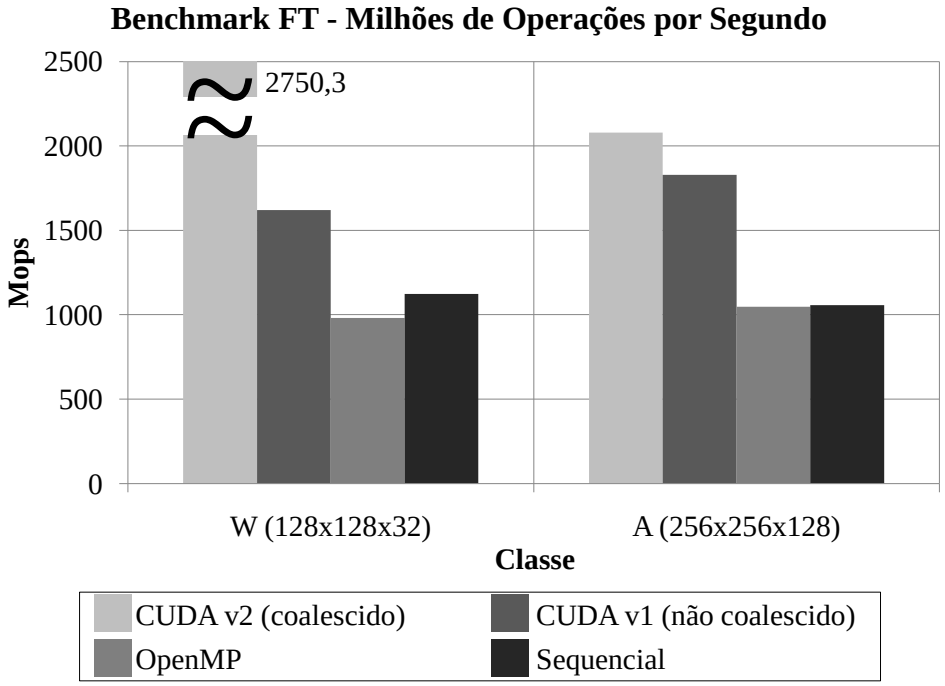

Figura 5.6: Número de operações por segundo para diferentes classes e implementações do benchmark FT.

na Tabela 5.1. Os mesmos valores obtidos para a versão 2 (com acesso coalescido) são apresentados na Tabela 5.2. Para a versão 1, pode-se ver que os tempos entre as funções são similares entre si. Já para a versão 2, tem-se tempos menores do que para a versão 1,

Tabela 5.1: Tempo de execução em segundos para diferentes funções do benchmark FT em sua versão para CUDA sem acesso coalescido.

Função	**Tempo para instância** W	**Tempo para a instância** A
$cfffts1()$	$0,04826$ s	$0,88730$ s
$cfffts2()$	$0,05426$ s	$1,13169$ s
$cfffts3()$	$0,03665$ s	$1,04860$ s

Tabela 5.2: Tempo de execução em segundos para diferentes funções do benchmark FT em sua versão para CUDA com acesso coalescido.

Função	**Tempo para instância** W	**Tempo para a instância** A
$cfffts1()$	$0,02716$ s	$2,23851$ s
$cfffts2()$	$0,01048$ s	$0,19403$ s
$cfffts3()$	$0,00791$ s	$0,15424$ s

com exceção do tempo para a função $cfffts1()$ na instância A. Este tempo apresenta-se aproximadamente $11\times$ e $15\times$ maior do que os tempos das funções $cfffts2()$ e $cfffts3()$, respectivamente, para a mesma instância.

Uma análise mais profunda dos tempos obtidos para a função $cfffts1()$ mostra que, em média, foram gastos $2,08974$ segundos na rotação da matriz para a instância A. Esse valor representa 93% do tempo de execução da função e 63% do tempo total da execução do benchmark. Esse sobrecusto está ligado ao fato de que a rotação da matriz é feita de forma não coalescida. O sobrecusto não é tão aparente na execução da instância W pois, neste caso, a matriz é menor, havendo um número inferior de acessos a memória. Além disso, o sobrecusto de permutação dos dados de forma não coalescida não aparece na execução da mesma função na versão 1 do código para CUDA porque nela não se faz necessária a rotação da matriz. Com isso, pode-se concluir que, para instâncias grandes do problema, os custos de permutação da matriz de forma não coalescida não compensam os ganhos ao executar computações com acessos coalescidos.

6 CONCLUSÕES

Este trabalho foi desenvolvido com o intuito de observar quais tipos de aplicações possuem potencial para ganho de desempenho em GPGPUs. Ele apresentou uma análise de desempenho da arquitetura CUDA utilizando os benchmark EP e FT dos NAS Parallel Benchmarks, os quais possuem um claro mapeamento à categorias da classificação Dwarf Mine. Para tal, foi necessário portar e otimizar ambos benchmarks para CUDA. Além disso, foi feita a comparação entre o desempenho obtido pelas novas versões e as versões originais do código, compiladas para execução de forma sequencial e paralela com OpenMP.

Obteve-se ganhos de até $21\times$ no uso da GPU para o benchmark EP - integrante da categoria *MapReduce* - quando comparado ao uso dos dois núcleos do processador com OpenMP. Isso foi possível graças à regularidade das computações e acessos à memória, além da independência dos dados. Quanto maior a instância do problema, melhores foram os resultados, pois os custos de transferência entre as memórias da CPU e da GPU são absorvidos pelo tempo de execução do kernel.

Para o benchmark FT - integrante da categoria *Spectral Methods* - obteve-se ganhos de quase $3\times$ comparados a versão com OpenMP. Diferentemente do benchmark EP, os ganhos não aumentaram com o crescimento das instâncias de teste. Isto aconteceu porque os sobrecustos de transposição da matriz, feita de forma não coalescida, acabaram tornando-se mais significativos com o aumento da matriz. Com isto, o ganho com o acesso aos dados nas computações de forma coalescida perdeu representatividade. Assim, para o caso de grandes matrizes, a melhor solução encontrada foi a de evitar a permutação dos dados e acabar acessando a matriz de forma não coalescida.

Estes resultados indicam uma **compatibilidade entre os algoritmos pertencentes as categorias *MapReduce* e *Spectral Methods*, e a arquitetura CUDA**. Com isso, implementações futuras de algoritmos pertencentes a estas duas categorias podem ser guiadas pelo presente trabalho, tendo uma base do possível desempenho a ser obtido nessa arquitetura. Porém, estes resultados somente são possíveis com o conhecimento e uso de detalhes da arquitetura CUDA, como padrões de acesso coalescido, limitações de memória e precisão, entre outros.

O uso da classificação Dwarf Mine foi importante para evitar a implementação no trabalho de algoritmos de uma mesma categoria e para a extensão dos resultados. Já o uso dos NAS Parallel Benchmarks mostrou-se positivo pois eles contam com códigos de verificação e medição de desempenho. Além disso, eles possuem um mapeamento claro com a classificação utilizada no trabalho e tem sua própria representatividade no meio científico.

Acredita-se que os resultados obtidos com este trabalho possam ser estendidos parcialmente para a arquitetura ATI Stream devido às similaridades encontradas, como apre-

sentado no Capítulo 3, faltando apenas um método de mapeamento das formas de comunicação entre threads vistas em CUDA para esta arquitetura. Isso representa uma possível portabilidade de desempenho entre essas arquiteturas, o que pode vir a ajudar desenvolvedores de código com a biblioteca OpenCL - a qual traz portabilidade de plataforma.

Trabalhos futuros incluem a implementação e verificação de outros benchmarks e algoritmos representativos de categorias da classificação Dwarf Mine em placas gráficas com arquitetura CUDA. Além disso, tendo a classificação e algoritmos de interesse, considera-se futuramente a implementação dos mesmo benchmarks para outras arquiteturas e sistemas, possibilitando a comparação com a arquitetura CUDA.

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 2009] **Ati stream computing - technical overview**. [S.l.]: Advanced Computing Devices, Inc., 2009.

[BAI 94] BAILEY, D. et al. The nas parallel benchmarks. **NASA Ames Research Center, RNR Technical Report RNR-94-007**, p.94–007, 1994.

[BAI 95] BAILEY, D. et al. **The nas parallel benchmarks 2.0**. Moffett Field, CA: Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[BAR 2008] BARKER, K. J. et al. Entering the petaflop era: the architecture and performance of roadrunner. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2008., 2008, Piscataway, NJ, USA. **Proceedings...** IEEE Press, 2008. p.1–11.

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

[CAS 2008] CASANOVA, H.; LEGRAND, A.; ROBERT, Y. **Parallel algorithms**. [S.l.]: Chapman & Hall/CRC, 2008.

[CHO 2008] CHONG, J. et al. Data-parallel large vocabulary continuous speech recognition on graphics processors. In: ANNUAL WORKSHOP ON EMERGING APPLICATIONS AND MANY CORE ARCHITECTURE (EAMA), 1., 2008. **Proceedings...** [S.l.: s.n.], 2008. p.23–35.

[CRA 2008] CRAWFORD, C. H. et al. Accelerating computing with the cell broadband engine processor. In: COMPUTING FRONTIERS, 2008., 2008, New York, NY, USA. **Proceedings...** ACM, 2008. p.3–12.

[DUN 90] DUNCAN, R. A survey of parallel computer architectures. **Computer**, Los Alamitos, CA, USA, v.23, n.2, p.5–16, 1990.

[GEL 89] GELSINGER, P. P. et al. Microprocessors circa 2000. **Spectrum, IEEE**, v.26, n.10, p.43–47, Oct 1989.

[GOL 91] GOLDBERG, D. What every computer scientist should know about floating-point arithmetic. **ACM Comput. Surv.**, New York, NY, USA, v.23, n.1, p.5–48, 1991.

[GOV 2008] GOVINDARAJU, N. K. et al. High performance discrete fourier transforms on graphics processors. In: SC '08: PROCEEDINGS OF THE 2008 ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2008, Piscataway, NJ, USA. **Anais...** IEEE Press, 2008. p.1–12.

[GSC 2006] GSCHWIND, M. et al. Synergistic processing in cell's multicore architecture. **Micro, IEEE**, v.26, n.2, p.10–24, March-April 2006.

[GSC 2007] GSCHWIND, M. et al. An open source environment for cell broadband engine system software. **COMPUTER**, p.37–47, 2007.

[GUL 2008] GULATI, K.; KHATRI, S. P. Towards acceleration of fault simulation using graphics processing units. In: DAC '08: PROCEEDINGS OF THE 45TH ANNUAL CONFERENCE ON DESIGN AUTOMATION, 2008, New York, NY, USA. **Anais...** ACM, 2008. p.822–827.

[HAR 2007] HARISH, P.; NARAYANAN, P. J. Accelerating large graph algorithms on the gpu using cuda. **Lecture Notes in Computer Science**, v.4873, p.197, 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.

[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.

[HEN 2006] HENNESSY, J. L.; PATTERSON, D. A. **Computer architecture, fourth edition**: a quantitative approach. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[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.

[HOU 2008] HOU, Q.; ZHOU, K.; GUO, B. Bsgp: bulk-synchronous gpu programming. In: SIGGRAPH '08: ACM SIGGRAPH 2008 PAPERS, 2008, New York, NY, USA. **Anais...** ACM, 2008. p.1–12.

[JES 2009] JESSEN, J. B. **Comparing fpga and gpu performance for both sparse and dense stereo reconstruction**. [S.l.]: The Maersk Mc-Kinney Moller Institute, Faculty of Engineering, University of Southern Denmark, 2009.

[JIN 99] JIN, H.; FRUMKIN, M.; YAN, J. The openmp implementation of nas parallel benchmarks and its performance. **NASA Ames Research Center," Technical Report NAS-99-011**, 1999.

[LEE 2009] LEE, S.; MIN, S.-J.; EIGENMANN, R. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In: PPOPP '09: PROCEEDINGS OF THE 14TH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2009. **Anais...** ACM, 2009. p.101–110.

[MAN 2008] MANAVSKI, S. A.; VALLE, G. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. **BMC Bioinformatics**, v.9, n.2, p.–10, 2008.

[MAN 2007] MANAVSKI, S. A. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. **Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on**, p.65–68, Nov. 2007.

[MIC 2008] MICHALAKES, J.; VACHHARAJANI, M. Gpu acceleration of numerical weather prediction. **Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on**, p.1–7, 2008.

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

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

[NVI 2008] **Nvidia geforce gtx 200 gpu architectural overview**. [S.l.]: NVIDIA Corporation, 2008.

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

[OLU 96] OLUKOTUN, K. et al. The case for a single-chip multiprocessor. **SIGOPS Operating Systems Review**, v.30, n.5, p.2–11, 1996.

[PHA 2005] PHAM, D. et al. The design and implementation of a first-generation cell processor. **Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International**, p.184–592, Feb. 2005.

[RIG 2009] RIGHI, R. d. R. **MigBSP: a new approach for processes rescheduling managemen on bulk synchronous parallel applications**. 2009. Tese (Doutorado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul.

[ROL 2009] ROLLS, D. et al. **Benchmark and application selection for the evaluation of the microgrid architecture and its tool chain**. [S.l.]: Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs, 2009.

[RYO 2008] 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.

[SCH 2007] SCHATZ, M. et al. High-throughput sequence alignment using graphics processing units. **BMC Bioinformatics**, v.8, n.1, 2007.

[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.

[TÖL 2007] TÖLKE, J. Implementation of a lattice boltzmann kernel using the compute unified device architecture developed by nvidia. **Computing and Visualization in Science**, 2007.

[YAN 2007] YANG, J.; GOODMAN, J. Symmetric key cryptography on modern graphics hardware. In: ASIACRYPT, 2007. **Anais...** [S.l.: s.n.], 2007. p.249–264.