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

EDUARDO DIAS CAMARATTA

Implementação em GPU da Transformada de Legendre

Trabalho de Graduação.

Prof. Dr. Philippe Olivier Alexandre Navaux
Orientador

Porto Alegre, dezembro de 2009.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ALU	*Arithmetic Logic Unit*
API	*Application Programming Interface*
BLAS	*Basic Linear Algebra Subprograms*
CPTEC	Centro de Previsão de Tempo e Estudos Climáticos
CPU	*Central Processing Unit*
CUBLAS	CUDA *Basic Linear Algebra Subprograms*
CUDA	*Compute Unified Device Architecture*
GPU	*Graphics Processing Unit*
ILP	*Instruction Level Parallelism*
INPE	Instituto Nacional de Pesquisas Espaciais
MPI	*Message Passing Interface*
PCIe	*Peripheral Component Interconnect Express*
SFU	*Super Function Unit* ou *Special Function Unit*
SM	*Streaming Multiprocessor*
SIMD	*Single Instruction Multiple Data*
SIMT	*Single Instruction Multiple Thread*
TPC	*Thread Processor Cluster*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Ao se aproximar dos limites físicos para a produção de chips, a indústria de hardware necessita enfrentar uma série de problemas para tentar manter os ganhos de desempenho vistos ao longo das últimas décadas. Assim, a existência de aplicações reais para as quais ganho em desempenho é fundamental motiva a busca por soluções que utilizem os recursos existentes de forma diferente ou mais eficiente. Neste contexto, este trabalho apresenta uma implementação para uma arquitetura não tradicional de um algoritmo de grande importância para aplicações altamente demandante de recursos, os modelos meteorológicos. O algoritmo escolhido é a Transformada de Legendre, e a arquitetura é a GPU. Para desenvolvimento da solução, foi utilizado o framework responsável por popularizar a utilização destes processadores em processamento de alto desempenho, o CUDA.

Palavras-Chave: Transformada de Legendre, GPU, CUDA, modelos meteorológicos, computação de alto desempenho.

Legendre Transform Implementation on GPU

ABSTRACT

While approaching the physical limits for chips production, the hardware industry needs to face a series of problems for trying to maintain the gains in performance seen over the past decades. Thus, the existence of real applications for which the gain in performance is fundamental motivates the search for solutions that use the existent resources in a different or more efficiently manner. In this context, this work presents an implementation for a nontraditional architecture of a very important algorithm for applications which highly demand resources, the weather prediction models. The chosen algorithm is the Legendre Transform, and the architecture is the GPU. For developing the solution, it was used the CUDA framework, which was responsible for the popularization of using such processors for high-performance computing.

Keywords: Legendre Transform, GPU, CUDA, weather prediction models, high-performance computing.

1 INTRODUÇÃO

Nos últimos anos a indústria de processadores enfrenta dificuldades em manter os ganhos de desempenho das últimas décadas, devido a constante aproximação dos limites físicos no desenvolvimento de circuitos. Problemas como consumo e confiabilidade, antes comuns apenas em projetos de circuitos embarcados, estão agora entre as principais preocupações no processo de construção de processadores de propósito geral. A exploração do paralelismo de dados e tarefas, com o emprego de múltiplos núcleos de processamento, foi a alternativa encontrada para contornar as dificuldades com aumento do ILP (*Instruction Level Parallelism*) e da frequência.

Complementando a situação, certas aplicações são naturalmente mais demandantes de recursos de *hardware* do que outras. Modelos de previsão meteorológica, conhecidos por executarem nos supercomputadores mais rápidos do mundo, beneficiam-se quando o ambiente no qual executam é mais veloz, pois assim é possível realizar mais cálculos sobre modelos maiores e, consequentemente, obter mais resultados em menos tempo e maior precisão.

O impacto dos problemas mencionados aumentará enquanto não houver um salto tecnológico que introduza novos limites para os componentes de *hardware*. A partir disso, propor soluções que utilizem os recursos existentes de forma mais eficiente, ou de formas diferentes, visando ganho de desempenho independentemente do aumento da velocidade do *hardware*, é uma necessidade.

Neste contexto, a utilização de GPUs (*Graphics Processing Units*) para aceleração de processamento de propósito geral ganhou força. Estes processadores, que inicialmente implementavam um rígido *pipeline* gráfico, evoluíram gradualmente para um arquitetura programável de grande flexibilidade, devido à exigência permanente por gráficos mais realistas. Inicialmente, o paralelismo massivo inerente de aplicações gráficas sobrecarregava os processadores de propósito geral. As GPUs foram criadas para obter desempenho através da exploração deste paralelismo, com a capacidade de executar operações iguais, simultaneamente, sobre uma quantidade de dados muito maior do que uma CPU (*Central Processing Unit*). Todavia, o *pipeline* gráfico rígido implementado pelas primeiras GPUs não era suficiente para obtenção do realismo desejado. Assim surgiram as GPUs programáveis através de conjuntos de operações sobre vértices e pixels, os *shaders*.

A possibilidade de programação, e a estrutura naturalmente matricial dos dados gráficos possibilitou a utilização das GPUs para processamento de propósito geral, inicialmente restrito ao emprego de texturas para leitura e escrita de dados inteiros, e da utilização de APIs (*Application Programming Interface*) gráficas para a realização das operações, geralmente de álgebra linear. Com a unificação dos tipos de *shaders* e a introdução de operações de ponto flutuante, necessários para certos cálculos gráficos

com maior necessidade de precisão, a empresa NVIDIA criou o primeiro *framework* que permitiu a utilização de GPUs para processamento de propósito geral sem a obrigatoriedade da utilização de APIs gráficas direta ou indiretamente, o CUDA.

O CUDA permite que o desenvolvedor explore o paralelismo de dados de uma maneira amigável, com a construção de um programa que será executado sobre uma grande quantidade de dados, em paralelo. Mas diferentemente das APIs gráficas, o CUDA expõe a arquitetura da GPU como um processador de propósito geral de múltiplos núcleos, com uma hierarquia de memória específica. Esta visão diferente das GPUs, os ganhos substanciais de desempenho obtidos em algumas aplicações desenvolvidas sobre o CUDA, e a constante evolução do *framework*, popularizaram a utilização dos processadores gráficos em processamento de propósito geral. E assim como no emprego de processadores de múltiplos núcleos, a exploração do paralelismo é fundamental para sustentar os ganhos de performance.

Motivado pelo cenário descrito, o objetivo deste trabalho é apresentar uma implementação de um algoritmo altamente demandante de recursos de *hardware*, a Transformada de Legendre, para uma arquitetura não convencional, a GPU. Esta transformada é parte fundamental de modelos de previsão meteorológica. E composta com a Transformada de Fourier constitui a Transformada Harmônica Esférica, uma ferramenta essencial para a transformação dos dados de entrada destes modelos para um domínio mais adequado para a aplicação de cálculos e algoritmos específicos, que facilitem a simulação dos eventos meteorológicos.

Este trabalho está estruturado da seguinte forma: os três primeiros capítulos contextualizam o leitor, detalhando o problema, sua visão teórica e o cenário no qual está inserido no capítulo 2; a arquitetura alvo no capítulo 3; e o ambiente de desenvolvimento no capítulo 4. O capítulo 5 detalha a implementação base utilizada, a solução desenvolvida, a metodologia de avaliação e um trabalho relacionado. O capítulo 6 apresenta os resultados deste trabalho e, por fim, o capítulo 7 apresenta as conclusões e possíveis trabalhos futuros.

2 TRANSFORMADA DE LEGENDRE

A Transformada de Legendre (ARNOLD, 1989), nomeada em homenagem ao matemático francês Adrien-Marie Legendre, é uma ferramenta matemática comumente utilizada na Mecânica Clássica, Mecânica Estatística e Termodinâmica. E assim como outras transformadas, ela é empregada para expressar a informação de uma função de uma maneira mais conveniente, ou mais formalmente: para mover a representação de um campo entre espaços (ZIA et al., 2008). Aplica-se uma transformada quando dados estão representados de uma maneira diferente da mais adequada para a situação na qual são necessários. No contexto da realização de operações sobre tais dados, em determinada representação um cálculo pode ser complexo e ineficiente. Em uma representação diferente ele pode ser simples e eficiente.

O objetivo da Transformada de Legendre é a projeção de dados de grade em uma esfera, como ilustrado na Figura 2.1. Aplicada nesta direção, a transformação recebe o nome de Transformada Direta de Legendre. A reconstrução da grade a partir dos dados projetados na esfera é chamada de Transformada Inversa de Legendre. Em ambas a realização da transformação é feita com a utilização dos Polinômios Associados de Legendre, uma subclasse importante das Funções Associadas de Legendre, soluções para a equação diferencial de Legendre.

Figura 2.1: Representação da Transformada de Legendre.

2.1 Aspectos Matemáticos

Considerando um sistema de coordenadas esféricas:

$$(\lambda, \theta) \in [-\pi, \pi] \times [-\pi/2, \pi/2],$$

A Transformada de Legendre é representada pela seguinte equação (PEIXOTO et al, 2009):

$$\Upsilon_n^m(\lambda, \theta) = e^{-im\lambda} P_n^m(\sin\theta)$$

Os Polinômios de Legendre de ordem m e grau n podem ser expressos pela fórmula de Rodrigues (onde $\mu = \sin\theta$):

$$P_n^m(\mu) = \frac{1}{\sqrt{2}} \frac{(1-\mu^2)^{|m|/2}}{2^n n!} \frac{d^{n+|m|}(1-\mu^2)}{d\mu^{n+|m|}}$$

Eles são soluções do problema de Sturm-Liouville, limitados nos extremos do intervalo, e são calculados através de relações de recorrência:

$$P_{n+1}^m = \frac{1}{\epsilon_{n+1}^m} \mu P_n^m - \frac{\epsilon_n^m}{\epsilon_{n+1}^m} P_{n-1}^m$$

Onde,

$$\epsilon_n^m = \left(\frac{n^2 - m^2}{4n^2 - 1} \right)^{1/2}$$

Ainda, as seguintes relações, obtidas diretamente da fórmula de Rodrigues, são importantes na implementação de modelos globais de previsão meteorológica:

$$P_n^m(\mu) = P_n^{-m}(\mu)$$
$$P_n^m(\mu) = 0, \qquad n < |m|$$

E, dependendo de *n* + *m*, a relação que determina que os polinômios são funções pares e ímpares, fato importante para economia computacional:

$$P_n^m(-\mu) = (-1)^{n+|m|} P_n^m(\mu)$$

2.2 Transformada Harmônica Esférica

Transformada Harmônica Esférica é o nome da composição da Transformada de Legendre com a Transformada de Fourier. Com ela é possível transformar dados representados em uma grade no domínio escalar para uma esfera no domínio frequência. Para isso, aplica-se a Transformada de Fourier para fazer a mudança entre os domínios e, sobre o resultado, aplica-se a Transformada de Legendre, para projetá-lo na esfera, como ilustrado na Figura 2.2. A conversão da grade escalar para a esfera no domínio

frequência é a fase chamada de Análise, ou simplesmente Transformada Harmônica Esférica Direta. A transformação na direção contrária é a fase chamada de Síntese, ou Transformada Harmônica Esférica Inversa.

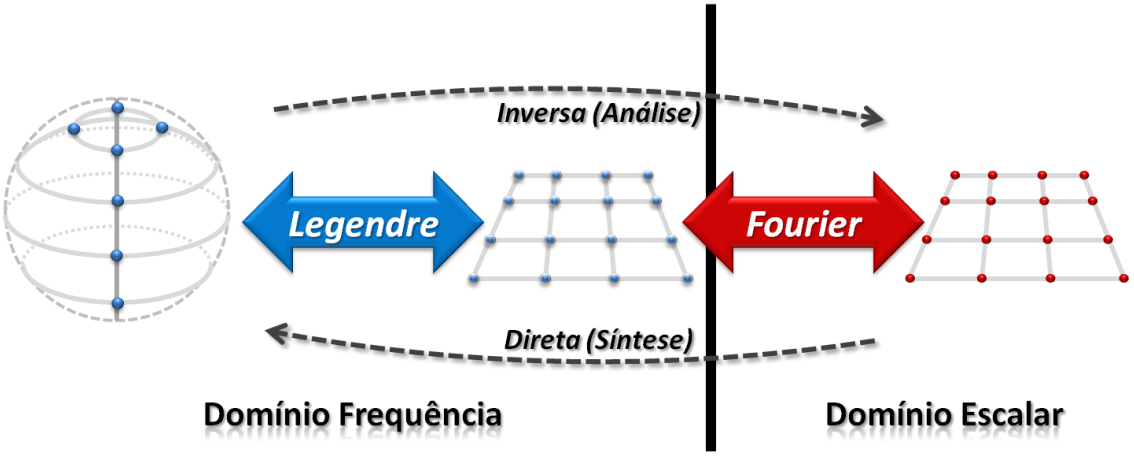

Figura 2.2: Representação da Transformada Harmônica Esférica.

Esta transformada é parte fundamental de modelos de previsão meteorológica de circulação global, pois é possível considerar a esfera como uma representação do planeta, e os dados no domínio frequência as medições de parâmetros meteorológicos. Com a transformada os dados são movidos para a representação mais adequada para a aplicação das equações meteorológicas.

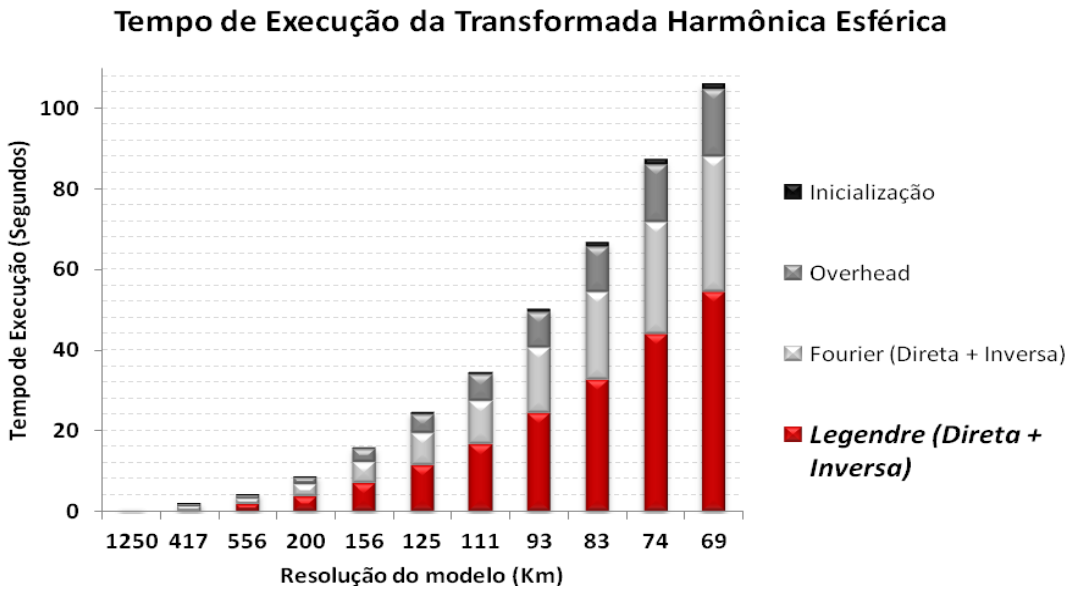

Figura 2.3: Tempo de execução[1] da Transformada Harmônica Esférica.

[1] Tempo de execução total do *benchmark* da Transformada Harmônica Esférica do CPTEC/INPE em um único núcleo de um Core 2 Duo E8500, variando a resolução do modelo de 1250 Km (T10) a 59 Km (T190), com 28 camadas verticais.

A importância da Transformada de Legendre na Transformada Harmônica Esférica é evidenciada após a análise do tempo de execução do seu algoritmo sequencial, mostrado na Figura 2.3. As transformadas Direta e Inversa de Legendre respondem pela maior parcela do tempo de computação. A Transformada Harmônica Esférica, por sua vez, é responsável por uma parcela considerável do tempo de execução dos modelos globais de previsão (ALVES, 2009).

Na prática, a importância de ambas as transformadas é ainda maior, pelo fato de executarem em modelos com resoluções maiores, diversas vezes ao dia. No modelo de circulação global do CPTEC/INPE (Centro de Previsão de Tempo e Estudos Climáticos/Instituto Nacional de Pesquisas Espaciais), por exemplo, o tempo é avançado em passos discretos denominados *timesteps*. Para cada *timestep*, executa-se uma Transformada Harmônica Esférica Inversa e uma Direta. Para 24 horas de previsão meteorológica, com um *timestep* de 1200 segundos (valor típico), as transformadas são executadas 144 vezes. Como o modelo prevê o tempo para dez dias, são 1440 execuções. Assim, uma parcela importante de tempo de processamento e de recursos de *hardware* é gasta com estas transformações.

3 PROCESSADORES GRÁFICOS

Atualmente o mercado de GPUs está divido entre três grandes fabricantes: Intel, AMD e NVIDIA. A AMD entrou no mercado ao adquirir a ATI, empresa que por vários anos foi concorrente principal da NVIDIA. A Intel, apesar de estar no mercado de aceleradores gráficos, não é concorrente direta de AMD e NVIDIA no segmento de GPUs de alta performance, por focar-se em *chips* de custo reduzido e performance limitada, integrados a placas-mãe. Apesar de tanto AMD quanto NVIDIA fabricarem placas gráficas com GPUs de alta performance e possuírem soluções para processamento de propósito geral, este trabalho irá se concentrar nas GPUs da NVIDIA, dado que estas possuem atualmente o ambiente mais amigável e maduro para desenvolvimento desta classe de aplicações. Então, a descrição da arquitetura-alvo realizada neste capítulo irá se basear na GPU de mais alta performance fabricada atualmente pela NVIDIA, a GT200.

As primeiras aplicações gráficas não executavam em uma arquitetura alvo distinta em relação ao restante dos *softwares*. A CPU era a responsável pela execução de todas as instruções necessárias para a exibição dos resultados. No entanto, rapidamente estas aplicações evoluíram a ponto de sobrecarregar a CPU. A partir da dependência criada pelos benefícios trazidos pelas aplicações gráficas, e da grande quantidade de recursos que elas necessitavam, foram criados *chips* aceleradores, as GPUs, com intuito de melhorar a performance específica desta classe de aplicações.

Os primeiros aceleradores eram baseados em arquiteturas que implementavam um rígido *pipeline* gráfico, aproveitando-se do paralelismo das operações sobre os *pixels*, componentes básicos das imagens. Logo, criou-se a necessidade de ir além dos resultados obtidos com os algoritmos básicos de rasterização (PURCELL, 2005), e passou-se a utilizar conjuntos de operações denominados *shaders* (PHARR, 2004), responsáveis por uma melhora substancial na qualidade dos gráficos gerados. Para a execução dos *shaders*, as GPUs incorporaram um *pipeline* programável, através de instruções semelhantes à de uma CPU.

O fato dos *shaders* serem na prática algoritmos de propósito geral que utilizam vértices e *pixels* como entrada e saída – essencialmente valores numéricos, e do crescente aumento de capacidade das GPUs em executarem milhares destas operações em paralelo, despertou a atenção para a utilização destas arquiteturas não somente como aceleradoras gráficas. Os *shaders* passaram então a ser utilizados para a escrita de algoritmos de propósito geral limitados, que faziam suas entradas e saídas em texturas (matrizes de pixels, consequentemente matrizes de valores numéricos). Neste contexto surgiram as primeiras linguagens que buscavam abstrair as limitações desta forma de utilização, e as GPUs com a capacidade de executar tais algoritmos passaram a ser chamadas de GPGPUs (*General Purpose GPUs*).

3.1 Arquitetura

As atuais GPUs são processadores *manycore*, altamente paralelos, com suporte a *multithread*, e com uma grande largura de banda. A grande diferença entre GPUs e CPUs está na área do *chip* dedicada a *cache*, controle e ULAs. Enquanto nas CPUs a maior parte da área é dedicada a controle e *cache*, nas GPUs estas áreas são pequenas em comparação com a parcela ocupada pelas ULAs, como mostrado na Figura 3.1. É devido a enorme quantidade de ULAs e a capacidade de utilização de todas em paralelo, que uma GPU alcança uma performance aritmética máxima pelo menos uma ordem de magnitude superior a uma CPU. Deve-se considerar também que a área total dos atuais *chips* aceleradores gráficos é substancialmente maior do que *chips* de CPUs - o GT200, da NVIDIA, tem cerca de 1,4 bilhões de transistores, ocupando uma área de 583,2 mm² (Figura 3.2). O consumo de energia também é grande, sendo que uma placa de alta performance equipada com um GT200 pode consumir até 236W.

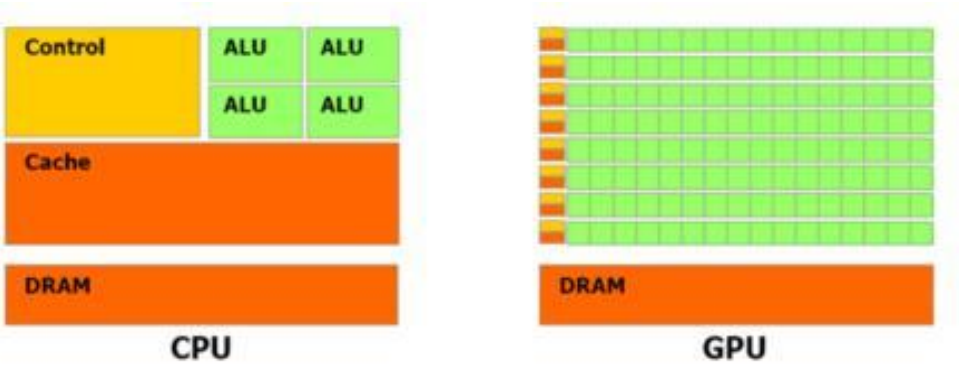

Figura 3.1: Área proporcional dos componentes básicos em um chip de uma CPU comparativamente ao chip de uma GPU (NVIDIA, 2009b).

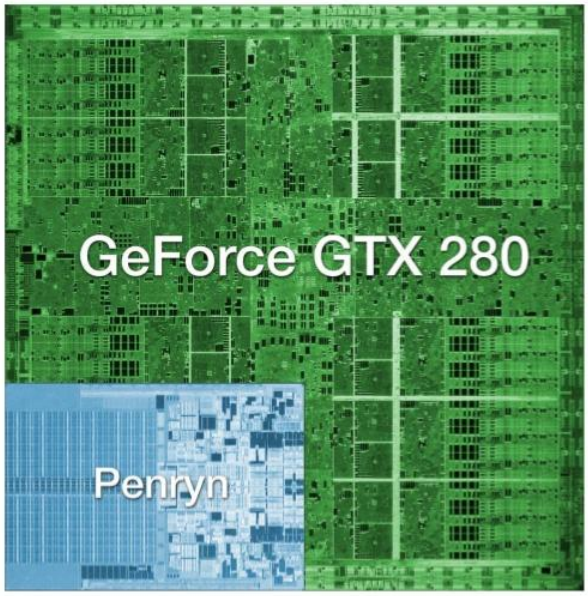

Figura 3.2: Área do *chip* de uma GPU (GeForce GTX 280, *chip* GT200) e de uma CPU (Penryn, da Intel).

Como mostrado na Figura 3.3, a arquitetura da GT200 é composta por 10 TPCs (*Thread Processor Cluster*s) - chamados de *Texture Processor Clusters* em GPUs anteriores. Cada TPC possui três SMs (*Streaming Multiprocessors)* e uma memória dedicada ao armazenamento das texturas utilizadas em processamento gráfico ("TEX", na Figura 3.3). Um SM, por sua vez, possui uma *cache* de 8 KB para instruções, uma *cache* de 8KB para dados, uma unidade para busca e distribuição de instruções e uma memória compartilhada de 16 KB entre oito *Streaming Processors* e duas SFUs (*Super Function Units* ou *Special Function Units)* – que executam operações como seno, cosseno e exponenciação; além de um arquivo de registradores de 64 KB, compartilhado entre todas as unidades de execução. Esta memória compartilhada não é uma *cache*, de forma que o programador tem controle sobre que dados que nela estarão. O custo para acesso à memória compartilhada entre as SPs é o mesmo de acesso aos registradores, desde que não haja conflito de acesso a bancos. Um SM ainda possui uma unidade para execução de instruções de precisão dupla.

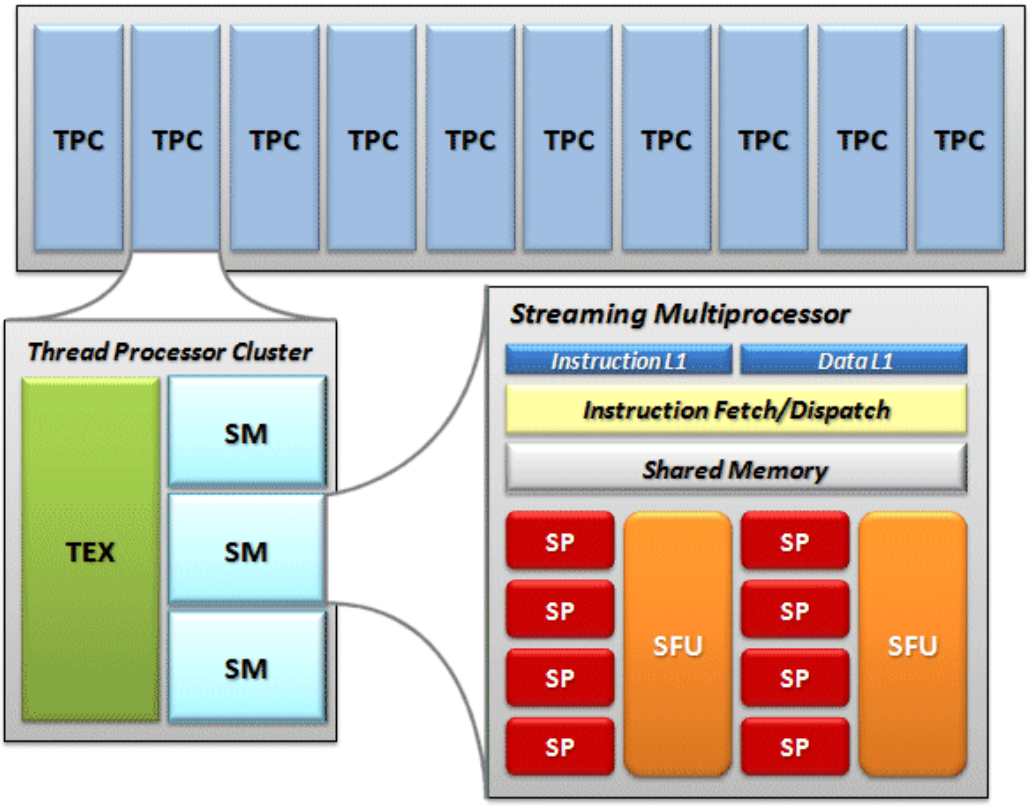

Figura 3.3: Diagrama da arquitetura da GPU GT200.

Os SMs gerenciam *warps*, que são grupos de 32 *threads*. Cada SM pode gerenciar até 32 *warps* ao mesmo tempo. Então, em um dado instante o GT200 pode gerenciar até 30.720 *threads*. Em cada SM, as instruções são executadas em uma maneira semelhante à SIMD. A cada ciclo, a mesma instrução é executada para todas as *threads* escalonadas. Este comportamento é chamado pela NVIDIA de SIMT (*Single Instruction Multiple Thread*). Ainda, os SPs e as SFUs tem uma frequência duas vezes maior do que o SM no qual se encontram. As *threads* não tem custo de escalonamento, mas se

elas seguirem caminhos diferentes em decorrência de *branches* com resultados distintos, elas não serão mais executadas totalmente em paralelo, executando alternadamente os caminho distintos tomados.

3.2 Desempenho

Com a popularização dos computadores como ferramenta de entretenimento, a necessidade por aumento de resolução das imagens, aumento da sua complexidade e qualidade (refletidos em *shaders* mais complexos), as GPUs tiveram sua performance aumentada em um ritmo superior ao das CPUs, como ilustrado na Figura 3.4.

Além disso, dada a natureza extremamente paralela dos dados que processam e algoritmos que executam, enquanto os fabricantes de CPU se empenhavam em aumentar frequência dos seus processadores, o objetivo dos engenheiros responsáveis pela criação de novas GPUs era aumentar o paralelismo a cada nova versão da arquitetura. Com o aumento de paralelismo, o ritmo de ganho de desempenho foi sustentado, e atualmente a performance aritmética máxima teórica de uma GPU é mais de uma ordem de magnitude superior a uma CPU.

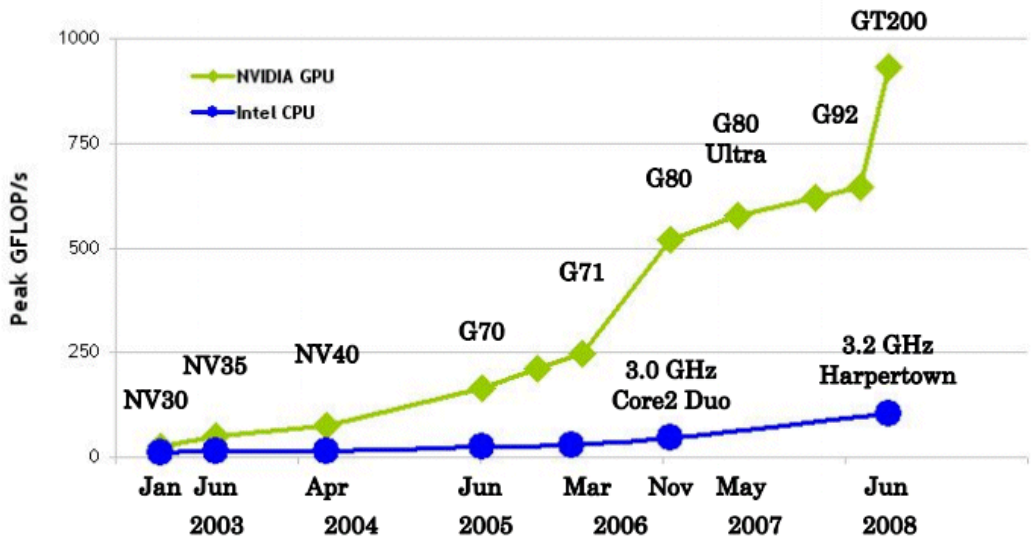

Figura 3.4: Evolução da performance aritmética máxima de GPUs e CPUs entre 2003 e 2008 (NVIDIA, 2009b).

Entretanto, algumas considerações devem ser feitas em relação a estes números. Primeiramente, as GPUs foram concebidas para processamento de apenas *pixels* e vértices. E geralmente este tipo de dado não necessita de grande precisão, considerando que pequenos erros nos cálculos não acarretarão mudanças significativas nas imagens processadas. Desta forma, a performance alcançada pelas GPUs é sustentada apenas pela execução de instruções com aritmética de precisão simples. Aplicações com cálculos de precisão dupla conseguem aproveitar menos de 10% da performance obtida com precisão simples, pela quantidade reduzida de ALUs (*Arithmetic Logic Units*) de 64 bits, e impossibilidade de utilização das ALUs de 32 bits em conjunto.

Aplicações que forem baseadas em comunicação também terão performance limitada se executadas em uma GPU. No contexto de aplicações gráficas nas quais elas foram concebidas, não há razão para a existência de comunicação entre *pixels* e vértices. Então, existe apenas uma pequena quantidade de memória de baixa latência compartilhada entre as ALUs de um núcleo, que não é utilizada quando a GPU está fazendo processamento gráfico. Mas para um núcleo de um grupo comunicar-se com um núcleo de outro grupo, a única maneira disponível é a utilização da memória externa ao *chip* e, consequentemente, mais lenta.

A quantidade de *cache* disponível por núcleo também é reduzida, de modo a permitir uma maior área para as unidades aritméticas. Uma aplicação que dependa de uma quantidade de dados maior do que o possível de ser colocado na *cache* também sofrerá penalizações severas, dado que o acesso à memória externa ao *chip* tem latência muito superior a o que ocorre com uma CPU. Apesar da grande largura de banda (Figura 3.5), ela é compartilhada entre um grande número de núcleos. Se a quantidade de dados necessários pela aplicação exceder também a quantidade de memória da placa gráfica, a aplicação novamente sofrerá penalizações, dado que ficará limitada pela largura de banda do barramento PCIe (que é de aproximadamente 6,4 GBytes/s), e dificilmente conseguirá uma quantidade de dados suficientemente grande para alimentar as centenas de ALUs.

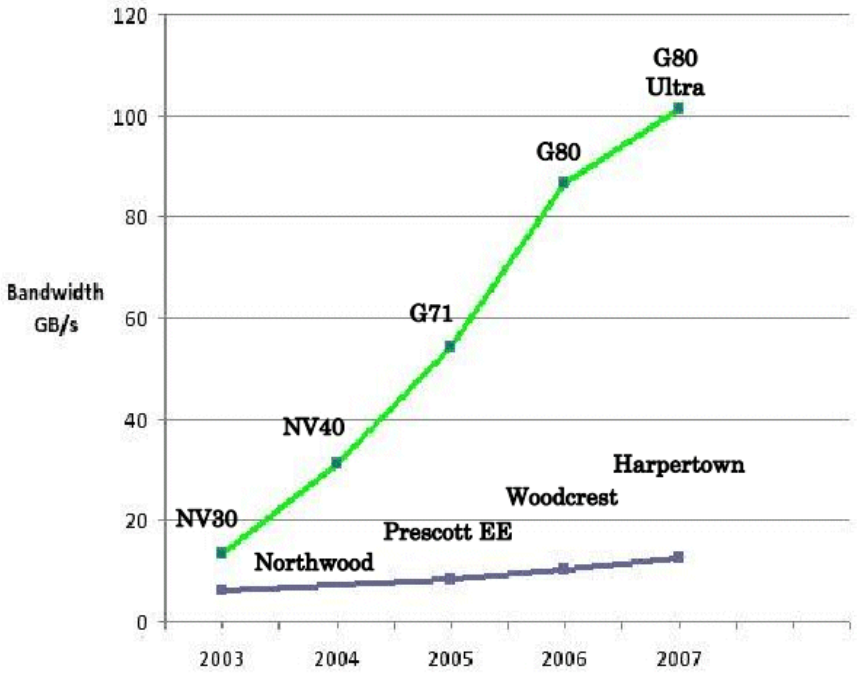

Figura 3.5: Evolução da largura de banda de GPUs e CPUs, entre 2003 e 2007 (NVIDIA, 2009b).

Esta arquitetura especial faz com que exista apenas um grupo de aplicações que conseguem tirar proveito dos recursos oferecidos. Um *software* desenvolvido para uma CPU específica não pode simplesmente ser executado em uma GPU sem nenhuma conversão. A simples tradução direta das instruções viabilizaria a execução, mas obter uma performance aceitável apenas desta maneira seria impossível.

Uma aplicação que consegue tirar bom proveito do desempenho disponibilizado por uma GPU é, basicamente, uma aplicação semelhante a um *shader* gráfico, ou seja, composta por uma série de cálculos aritméticos simples, altamente paralelizáveis, com pouca dependência, pouca comunicação e que não depende da existência de uma grande *cache*. Aplicações com muitas instruções de *branch*, dependentes de *cache* grande e veloz, de baixa latência ou que sejam pouco paralelizáveis, subutilizarão os recursos de tal forma que sua execução em GPUs não é lucrativa. A atual geração de GPUs também tem poucos recursos para execução de instruções de precisão dupla, de forma que o desempenho atingido por softwares que dependem exclusivamente de grande precisão também será restrito.

4 FRAMEWORK CUDA

Com o advento das primeiras GPUs programáveis, a possibilidade de obtenção da performance que elas ofereciam tornou-se um atrativo para o surgimento das primeiras implementações de algoritmos de propósito geral destinados a estas arquiteturas. Entretanto, os primeiros desenvolvedores de algoritmos que executavam em GPUs precisavam conhecer as APIs gráficas existentes e a forma de utilizá-las, pois elas eram a única forma possível de exploração destes recursos. Mas, para um desenvolvedor não acostumado com as terminologias utilizadas na área gráfica, não só era difícil aprender estas tecnologias, como também era complexo utilizá-las, dadas as restrições em relação ao emprego de primitivas gráficas para implementação de código de propósito geral.

Com intuito de encapsular as APIs gráficas, e de expor a GPU apenas como um processador aritmético, surgiram as primeiras linguagens e bibliotecas de programação de propósito geral para GPUs, como a BrookGPU, um conjunto de extensões para a linguagem C. Apesar do grande *overhead* causado pelas APIs gráficas, e das incompatibilidades decorrentes de atualizações de *drivers* realizadas pelos fabricantes de placas gráficas, esta linguagem foi importante por trazer a atenção para a execução de algoritmos de propósito geral nas GPUs.

Os desenvolvedores da BrookGPU tornaram-se parte do time de desenvolvedores da NVIDIA, e passaram a utilizar uma nova estratégia. Trabalhando com o fabricante das GPUs, não era mais necessário apoiar-se em APIs gráficas feitas por terceiros. E de posse dos detalhes arquiteturais das GPUs, foi possível desenvolver um *driver* especificamente para exploração destes processadores para execução de algoritmos de propósito geral, bem como uma camada de abstração de mais alto-nível, e uma série de bibliotecas para facilitar a utilização do novo *hardware* disponível.

Assim nasceu o CUDA (*Computer Unified Device Architecure*) (NVIDIA, 2009b), o primeiro *framework* oficial de um fabricante de GPUs para a execução de algoritmos de propósito geral nestes processadores, composto por um conjunto de extensões para a linguagem C, um compilador específico, um *driver*, uma API, bibliotecas matemáticas aceleradas por GPU, ferramentas para *profiling* e *debugging*, além de extensa documentação. O sucesso do CUDA é também apoiado no *hardware*, dado que a abstração via *software* tornou-se possível a partir da arquitetura escalável criada pela NVIDIA a partir da GPU G80 (NVIDIA, 2009c), duas gerações anteriores a GPU GT200, descrita neste trabalho. A partir desta geração as modificações realizadas na arquitetura fundamentaram-se apenas em ampliação no número de componentes da arquitetura, como número de núcleos, ampliação de *caches*, aumento do número de *threads* gerenciadas e quantidade de registradores por núcleo.

4.1 Hierarquia de *Software*

O CUDA é fundamentado em uma hierarquia de *software* mostrada na Figura 4.1, definida em três camadas:

- Um *driver*, a primeira camada de abstração do *hardware*.
- Uma API denominada CUDA *Runtime Library*, que executa sobre a API do *driver*.
- Um conjunto de bibliotecas matemáticas que utilizam a *Runtime Library*.

O *driver*, por ser a camada de mais baixo nível de acesso ao *hardware*, é também a camada que oferece a maior performance, flexibilidade e controle. No entanto, a programação com esta API é complexa. Esta complexidade é abstraída por uma API de mais alto nível, a CUDA *Runtime Library*. A chamada de uma única função nesta API é equivalente a chamada de várias funções da API do *driver*.

Por fim, existe um conjunto de bibliotecas matemáticas implementadas utilizando a *Runtime Library*: a CUBLAS (NVIDIA, 2009a), uma série de funções para cálculos de álgebra linear nas GPUs; e a CUFFT, que implementa a transformada de Fourier.

Uma aplicação desenvolvida em CUDA poderá utilizar-se diretamente da API do *driver*, ou optar pela utilização da *Runtime Library*, dado que elas são mutuamente exclusivas. Não é possível misturar em um único código chamadas de ambas APIs. Este trabalho se concentrará na descrição dos recursos oferecidos por pela *Runtime Library*, dado que ela implementa praticamente todos os recursos disponíveis na API do *driver*, e de forma mais simples.

As aplicações CUDA que executam sobre a hierarquia descrita podem ser desenvolvidas utilizando-se de uma série de extensões sobre a linguagem C, ou com APIs alternativas, como o OpenCL e a Microsoft Direct Compute. É também possível desenvolver aplicações CUDA sobre a linguagem Fortran, através das extensões propostas em (FLAGON, 2007). Contudo, no momento da concepção deste trabalho, dos três ambientes desenvolvidos, as extensões para a linguagem C originais do CUDA constituíam a plataforma mais madura, estável e completa e, por este motivo, foram utilizadas neste trabalho.

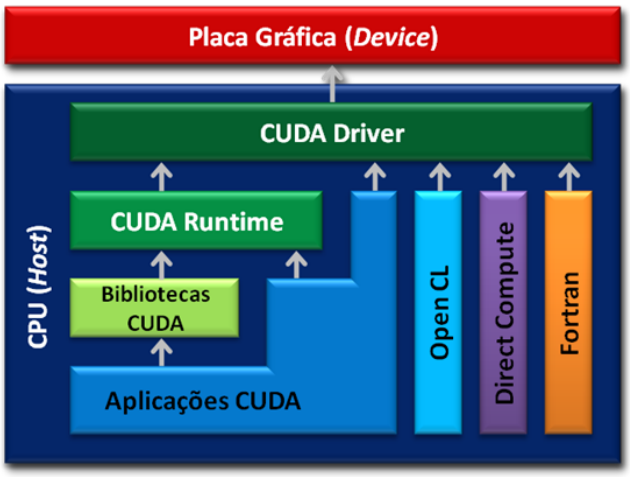

Figura 4.1: Hierarquia de Software CUDA.

4.2 CUDA Threads

As *threads* são o componente básico de um programa CUDA, entretanto, neste contexto a terminologia é utilizada de uma forma diferente das *threads* de CPUs. Ao contrário das CPUs, a troca de contexto entre *threads* na GPU é feita com custo zero, por elas serem executadas em grupos de *threads* iguais, e por apenas uma mesma instrução de todas *threads* do grupo executar por vez no SM. As *threads* CUDA também não podem ser recursivas, não podem fazer chamadas de sistema, não recebem número variável de argumentos, não podem criar variáveis estáticas e suas funções não podem ter endereço calculado.

O CUDA também assume que as *threads* executarão em um dispositivo (*device*) diferente do hospedeiro (*host*) no qual elas foram chamadas. Os dispositivos CUDA são as placas de vídeo compatíveis com o *driver*, com a arquitetura mostrada na Figura 4.2. Nesta figura, a "Memória Compartilhada" corresponde a *Shared Memory* descrita na seção 3.1, e a "Memória Global" a memória da placa gráfica. O hospedeiro é a máquina na qual o código C padrão, que faz as chamadas CUDA, está executando. A comunicação entre ambos é possível através da utilização de funções disponíveis na API utilizada. Este modelo está ilustrado na Figura 4.3.

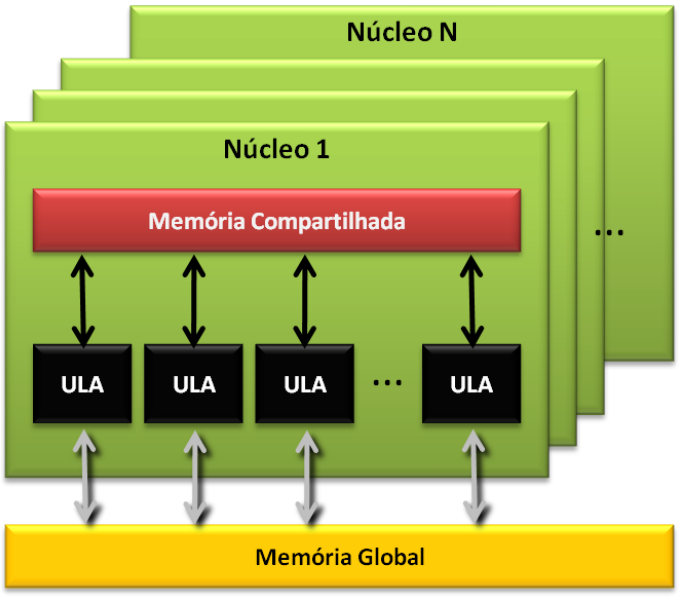

Figura 4.2: Arquitetura de um dispositivo CUDA.

As *threads* executam em bloco, e são identificadas pelo *ThreadIdx*, um vetor de três componentes que permite localizá-las unidimensionalmente, bidimensionalmente ou tridimensionalmente dentro deste. Esta forma de identificação traz facilidade ao se utilizar matrizes nos algoritmos. O código de uma *thread* é uma simples função em C, chamada neste contexto de *kernel*. *Threads* de um bloco podem compartilhar variáveis, que estarão situadas na *Shared Memory*. *Threads* são disparadas em um bloco de blocos, denominado *grid*. *Threads* de um bloco são sincronizadas através da chamada a função *__syncthreads*(), que funciona como uma barreira, sendo que todas as *threads*

devem esperar até que uma possa prosseguir. É fundamental que os blocos de *threads* sejam desenvolvidos de maneira que seja possível executá-los em qualquer ordem, paralelamente ou sequencialmente, de modo a garantir a boa utilização dos recursos e código escalável, sendo que o dimensionamento dos *grids* e blocos será feito de acordo com o tamanho dos dados e não com o número de recursos disponíveis no *hardware*.

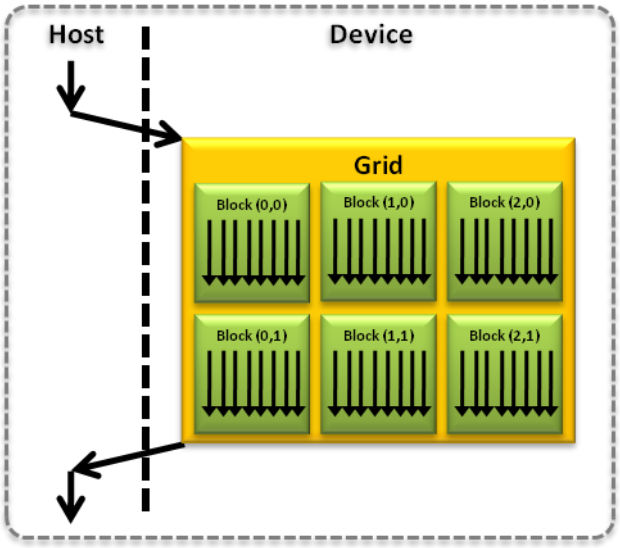

Figura 4.3: Modelo de execução CUDA.

4.3 Hierarquia de Memória e Estrutura de um Programa CUDA

Uma boa compreensão da hierarquia de memória disponível na GPU, mostrada na Figura 4.4, e acessível através do CUDA é fundamental para obtenção de bom desempenho destas arquiteturas. O *host* comunica-se com o *device* através da memória global do dispositivo, pelo barramento PCIe. O programa alocará e desalocará memória no *device*, e manterá a comunicação entre *device* e *host* através de chamadas a *Runtime Library*. *Threads* além dos seus registradores também possuem um espaço exclusivo de memória. *Threads* de um bloco comunicam-se entre si através de uma memória compartilhada dentro do SM no qual executarão ou através da memória global de alta latência do dispositivo. *Threads* ainda podem ler dados de uma memória de constantes e de uma memória de texturas, que funcionam como *caches read-only*.

A partir desta hierarquia é possível estabelecer a estrutura e comportamento de um programa CUDA. Ele será basicamente uma série de funções C que executarão e alocarão memória no *host* e no *device*, transmitirão dados para o *device*, lançarão execução de *kernels* via *grids* de blocos de *threads* no *device* - que se comunicarão via memória compartilhada ou memória global, e utilizarão dos seus registradores e da memória exclusiva para resolução do problema - e finalmente o *host* esperará pelo término da execução dos *kernels* no *device*, antes de ler os resultados da memória global.

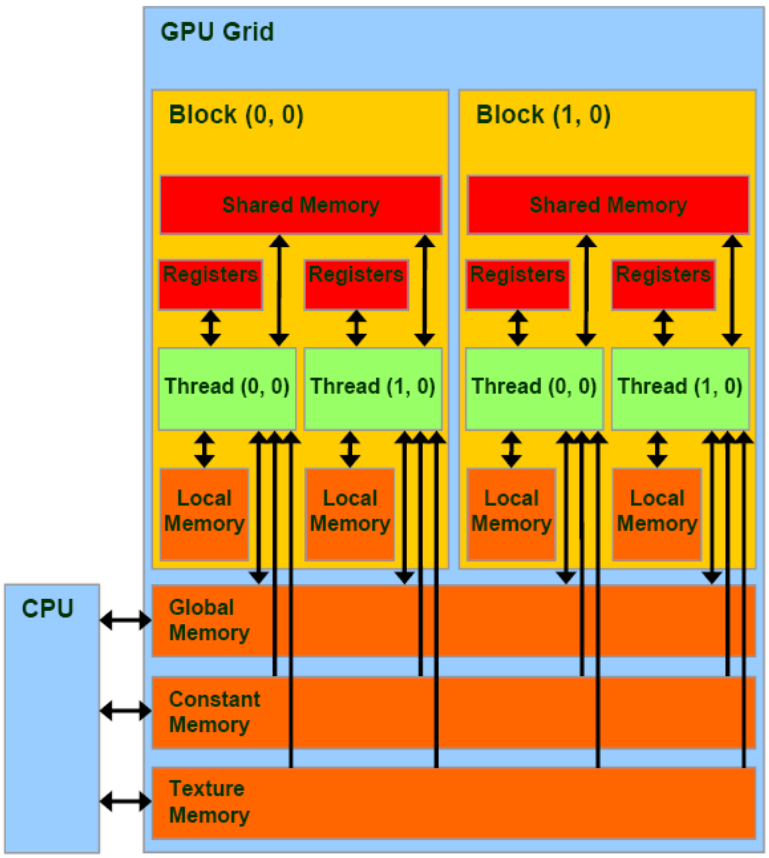

Figura 4.4: Hierarquia de Memória CUDA (NVIDIA, 2009b).

5 A IMPLEMENTAÇÃO EM GPU

Motivado pelo cenário descrito no capítulo 1, o objetivo deste trabalho é, desde sua concepção, a implementação de um algoritmo altamente demandante de recursos computacionais em uma arquitetura não convencional. Faz também parte do objetivo a escolha de uma aplicação com uso real, não sintética ou puramente acadêmica.

A parceria entre o Grupo de Processamento Paralelo e Distribuído da UFRGS com o CPTEC/INPE tornou evidente para o grupo a grande quantidade de recursos que é necessária para a execução dos atuais modelos meteorológicos, e o quão importante é a realização de estudos sobre estes modelos, de forma a prepará-los para as próximas arquiteturas, e melhorar sua eficiência nas arquiteturas existentes. Nestes modelos, conforme citado na seção 2.2, identificou-se a Transformada de Legendre como responsável por uma parcela importante no desempenho, e por este motivo ela foi escolhida como alvo deste trabalho.

Durante a concepção do trabalho os processadores gráficos popularizaram-se na aceleração de aplicações de propósito geral. Da mesma forma, o CUDA consolidou-se como *framework* para programação destas arquiteturas. Devido a substancial quantidade de operações aritméticas paralelizáveis existentes na Transformada de Legendre, e pelos motivos descritos nos capítulos 1, 3 e 4, as GPUs foram escolhidas como a arquitetura a ser explorada, e o CUDA como ambiente de desenvolvimento.

A Transformada de Legendre não é executada de forma isolada nestes modelos, fazendo parte da Transformada Harmônica Esférica, discutida na seção 2.2. Assim, apesar de focar-se na implementação da Transformada de Legendre, este trabalho foi realizado sobre uma implementação da Transformada Harmônica Esférica, detalhada na Seção 5.1. Para a análise de desempenho considerou-se não só a computação da Transformada de Legendre, mas também a Transformada Harmônica Esférica como um todo, de forma a verificar o ganho obtido com as modificações implementadas, e o impacto delas no desempenho total do algoritmo original.

Precisão é um requisito fundamental em modelos de previsão climática. Entretanto, os processadores gráficos existentes durante o desenvolvimento deste trabalho possuíam performance de pico em precisão dupla em torno de uma ordem de magnitude inferior a performance na execução de operações aritméticas em precisão simples. Por este motivo, também foram desenvolvidas modificações no código base da Transformada Harmônica Esférica para que ele executasse em precisão simples, com intuito de observar o impacto do aumento dos recursos de *hardware* no desempenho do algoritmo. Consequentemente, a Transformada de Legendre também foi modificada da mesma maneira.

As próximas seções deste capítulo vão detalhar o código base utilizado, os aspectos de corretude e precisão, o ambiente de desenvolvimento, a solução desenvolvida, a metodologia de avaliação, e um trabalho relacionado, desenvolvido paralelamente a este.

5.1 Implementação Base

O trabalho foi desenvolvido utilizando como base a implementação da Transformada Harmônica Esférica do CPTEC/INPE. O código é escrito em Fortran 90, utiliza as bibliotecas MPI (*Message Passing Interface*) e OpenMP, e tem cerca de 15 mil linhas, organizados em 14 módulos. Destes destaca-se o módulo *Transform*, que contém as rotinas principais para realização da Síntese e a Análise, e que inicializam os Polinômios de Legendre. É neste módulo que foram feitas a maioria das modificações desenvolvidas neste trabalho.

O código base é um *benchmark* da Transformada Harmônica Esférica, com a Transformada de Fourier implementada da maneira tradicional, e a Transformada de Legendre da forma matricial. Nesta forma, simplificadamente, os coeficientes espectrais são multiplicados pelos Polinômios de Legendre (Figura 5.1). Resumidamente, o código do *benchmark* realiza os seguintes passos:

Transformada Harmônica Esférica

1: **Cálculo dos Polinômios de Legendre**. Eles são utilizados em todas as transformações subsequentes.

2: Execução da **Transformada Harmônica Esférica**:

- **a:** **Inicialização** dos **campos espectrais** com uma constante **α**.
- **b:** Execução da **Transformada Inversa de Legendre** sobre os campos espectrais.
- **c:** Execução da **Transformada Inversa de Fourier** sobre o resultado do passo **2.b**.
- **d:** Execução da **Transformada Direta de Fourier** sobre o resultado do passo **2.c**.
- **e:** Execução da **Transformada Direta de Legendre** sobre o resultado do passo **2.d**.

3: **Verificação de corretude**: os campos espectrais resultantes do passo **2.e** são comparados com a constante **α**. Se transformadas está implementada corretamente, ou seja, se no resultado todos os campos espectrais possuem novamente o valor da constante **α** com a qual foram inicializados no passo **2.a**, o *benchmark* segue a execução. Caso contrário, ele encerra com erro.

4: A **Transformada Harmônica Esférica** é chamada mais **dez vezes**, para medição de desempenho.

5: **Exibição dos resultados** da medição.

Apesar da forma matricial de Legendre, a transformada não possui uma implementação simples. Os principais motivos que contribuem para isso são a grade quadrática, as comunicações em MPI e as otimizações com intuito de ganho de

desempenho. Estes motivos também são responsáveis por subutilização dos recursos na implementação para GPU.

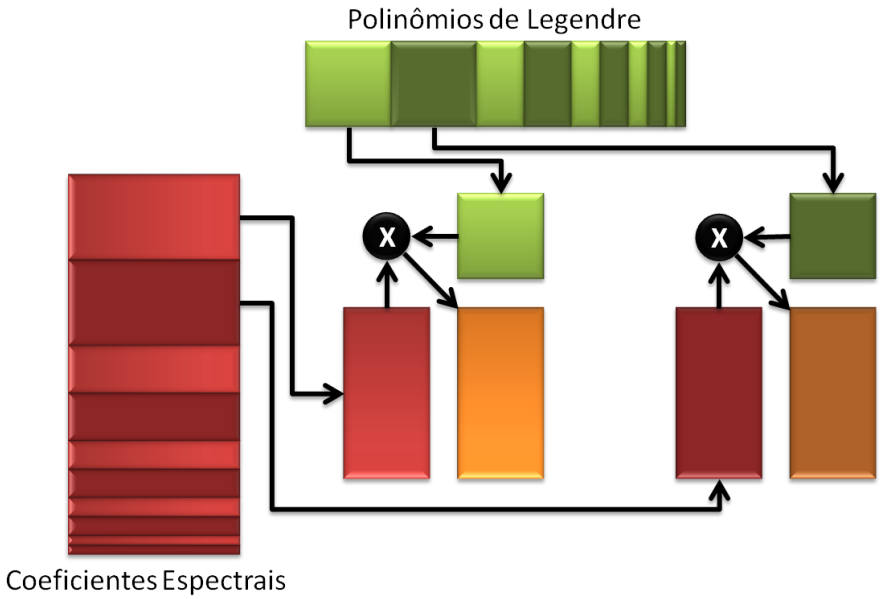

Figura 5.1: Representação da Transformada de Legendre na forma matricial.

Por causa da grade quadrática não é feita uma única multiplicação matricial na Transformada de Legendre, mas uma série de multiplicações, e a cada nova iteração do *loop* de multiplicações o tamanho das matrizes diminui. Este é um ponto importante para a análise posterior do desempenho da solução que utiliza a GPU, pois apesar de não alterar a proporção *flops*:*word*, parâmetro de grande relevância para a performance de uma aplicação portada para processadores gráficos, aumenta a quantidade de transferências entre CPU e placa gráfica, e subutiliza os mecanismos de *cache* e *pipeline*. Além disso, o *overhead* do *driver* CUDA e latência do barramento PCIe, desprezíveis quando se realiza uma única chamada ou transferência CUDA sobre uma grande quantidade de dados, passa a ser importante no tempo total de execução.

As comunicações em MPI presentes depois da Transformada de Legendre Inversa e antes da Transformada de Legendre Direta são responsáveis também por um aumento das transferências de dados entre CPU e placa gráfica. Se tais comunicações não existissem, e em um escopo de implementação completa da Transformada Harmônica Esférica em GPU, colocar a Transformada de Fourier também em GPU cortaria pela metade a quantidade de transferências.

Por fim, as otimizações de desempenho para CPU são responsáveis por um gasto extra da banda do barramento PCIe no momento em que passa-se a utilizar GPUs. As matrizes de coeficientes espectrais são agrupadas em uma única grande matriz, mas cada matriz agrupada possui linhas ou colunas de valores inúteis para o resultado, mas úteis para evitar conflito de acesso a bancos de memória, e melhorar a performance no acesso a memória quando são executadas em CPU.

5.1.1 MPI e OpenMP na Implementação Base

Apesar de o código utilizar as bibliotecas MPI e OpenMP, não é objetivo deste trabalho comparar a performance de GPUs com aplicações que as utilizam, portanto, elas foram desabilitadas durante o desenvolvimento e nas medições de desempenho. Todavia, o código base utiliza MPI porque os modelos do qual faz parte executam, na prática, em *clusters* com centenas de processadores. A partir disso, as modificações desenvolvidas sobre o algoritmo foram feitas com a preocupação em não interferir no comportamento da aplicação quando o MPI fosse habilitado. Para todas as comunicações, tomou-se o cuidado em manter coerentes os *buffers* utilizados, alimentados com valores atualizados.

Em um ambiente no qual seja habilitado e utilizado o MPI, o código modificado continuará executando corretamente, explorando o paralelismo disponível, da mesma forma que antes das modificações. Nos processadores que estiverem associados a placas gráficas, as mudanças realizadas serão aproveitadas, e parte da computação que seria feita em CPU será realizada na GPU. Neste caso, mesmo que o processador gráfico não acelere a computação, a CPU terá sido desonerada daquele cálculo, podendo executar outra tarefa. Em nodos sem placas gráficas, a aplicação se comportará da mesma maneira que a anterior as modificações propostas por este trabalho. Em *clusters* heterogêneos, com alguns processadores associados a placas gráficas com suporte a CUDA e outros não, as modificações introduzirão desbalanceamento de carga, apesar de produzirem o resultado igual a implementação original. Entretanto, este problema não faz parte do escopo deste trabalho.

Em relação ao OpenMP, ele é utilizado no código base em um trecho que foi modificado para a utilização das GPUs, a Transformada de Legendre. Neste caso, como o código que executava em CPU com OpenMP foi alterado para utilizar GPU, se a aplicação modificada executar em um processador de múltiplos núcleos, ela fará uso apenas de um deles. Como este trabalho propõe-se a comparar a performance do código que executa em um único núcleo de uma CPU com a implementação alterada que executa em um único núcleo e que utiliza a GPU para aceleração, esta modificação não resulta em qualquer diferença para os testes realizados. Em um escopo maior, no qual a aplicação alterada seja executada em uma máquina com múltiplos núcleos associados a uma ou mais GPUs, adaptações deverão ser feitas de forma a utilizar todos os recursos de processamento disponíveis. Entretanto, estas alterações estão fora do escopo deste trabalho.

5.2 Corretude e Precisão

Como citado no passo três do algoritmo descrito na seção 5.1, existe uma fase de verificação de corretude da computação realizada. Esta fase é importante para garantir que as modificações feitas no código original não interfiram na qualidade dos resultados.

Esta verificação é possível pela estrutura matemática do algoritmo implementado. Como descrito no capítulo 2, as transformadas possuem duas direções opostas, uma com o objetivo de mover os campos para um espaço diferente, e outra que restaura os campos ao espaço original. Na teoria, a composição da transformada inversa sobre a direta deve resultar em um valor exatamente igual a entrada. A partir disso, é possível verificar se as transformadas foram implementadas e executadas corretamente com a execução da inversa sobre a direta, ou vice-versa, restaurando o valor colocado como

entrada. Entretanto, como os processadores trabalham apenas valores com representação discreta de precisão finita, uma comparação entre o valor esperado e resultado obtido geralmente resultará em diferença, pelo erro de representação, e todos os erros de arredondamento e truncagem acumulados nas operações realizadas. Assim, o comparador do código utilizado leva em consideração não a igualdade absoluta dos valores, mas a quantidade de bits iguais que eles possuem na mantissa, do mais significativo para o menos significativo. Neste código são necessários pelo menos 20 bits iguais, seguindo a regra descrita, para que um valor seja considerado igual a outro, em precisão dupla (valor representado em 64 bits nas arquiteturas no qual o trabalho foi desenvolvido).

Este trabalho realizou também a adaptação do código para execução em precisão simples (32 bits). Neste caso, todos os problemas de precisão descritos acima se tornam ainda maiores. Nesta implementação, a verificação de corretude também é executada, mas ao contrário das implementações em precisão dupla, foi modificada para não acusar erro no caso em que menos de 20 bits mais significativos forem iguais entre os valores comparados. O resultado da verificação de corretude, apesar de não interpretado de forma binária, correto ou incorreto, é verificado ao final das execuções do código em precisão simples. A verificação de corretude e precisão é ainda mais importante no caso das implementações em processadores gráficos, por causa das diferenças de representação de valores de ponto flutuante, em comparação com as CPUs.

Além disso, a utilização dos valores sintéticos na inicialização dos campos espectrais não gera erros na medição do desempenho em comparação à utilização de valores reais, pois os resultados obtidos a partir destas entradas são alcançados via execução de operações elementares, em quantidade proporcional ao tamanho da entrada, e não aos seus valores.

5.3 A Solução

Após a escolha e breve análise do problema, o trabalho iniciou-se com o estudo de arquiteturas alvo. De todas as arquiteturas consideradas, a GPU foi escolhida pelos motivos apresentados no capítulo 1, neste capítulo, e pelas características mostradas no capítulo 3. Após a definição da arquitetura, passou-se ao estudo dela e da implementação do problema escolhido, descrita na seção 5.1 obtida com o CPTEC/INPE.

5.3.1 Instrumentação

Apesar de a implementação original possuir seu próprio módulo para medição de tempo, foi desenvolvida uma nova biblioteca para este fim, mais flexível, portável em relação aos diferentes ambientes no qual o trabalho de implementação foi desenvolvido, Microsoft Windows e Unix, e com a geração de relatórios mais detalhados. Com esta biblioteca, foram inseridos dezenas de novos *timers* na aplicação base, com intuito de analisar não só a contribuição das grandes fases já medidas pelos *timers* anteriores, mas também de mensurar subfases, e o impacto de operações como alocação e inicialização de memória no desempenho total do algoritmo. A partir destas modificações, os perfis de execução gerados com a nova biblioteca de medição mudaram em relação aos resultados do módulo original. Basicamente, tempo que antes era contabilizado como computação de transformadas foi identificado como proveniente de alocações e inicializações de memória, passando a ser classificado como *overhead*.

Com estas modificações também foi identificado que a maior parte da fase de inicialização é dedicada ao cálculo dos Polinômios de Legendre, detalhados na seção 2.1. Este resultado é importante ao se comparar os resultados obtidos no trabalho relacionado descrito na seção 5.5 com os resultados deste trabalho, mostrados no capítulo 6.

Com a análise do desempenho do código original feita, e com a confirmação da computação da Transformada de Legendre como a responsável pela maior fatia de tempo de execução da Transformada Harmônica Esférica (Figura 2.3), partiu-se para a implementação em GPU através do *framework* CUDA.

5.3.2 Primeira Implementação

A partir do conhecimento adquirido principalmente com os manuais sobre o *framework* e arquitetura (NVIDIA, 2009b), a primeira solução desenvolvida foi a escrita de *kernels* CUDA para multiplicações dos coeficientes espectrais projetados na esfera pelos Polinômios de Legendre, na transformada inversa, e para a multiplicação dos coeficientes espectrais de grade pelos mesmos polinômios, na transformada direta.

Como mencionado no capítulo 4, existem extensões que permitem a escrita de códigos CUDA diretamente a partir do Fortran, linguagem da implementação base. Mas tais extensões (FLAGON, 2007) não implementavam a especificação CUDA completamente no momento em que a implementação foi feita. Assim, optou-se por implementar os *kernels* na linguagem nativa para as extensões do *framework*, C, e utilizar a interface ISO-C Binding da linguagem Fortran, que permite a interoperabilidade entre as duas linguagens, para a comunicação entre o algoritmo base e a solução desenvolvida.

Mas, mesmo sendo possível a comunicação entre Fortran e C, algumas diferenças fundamentais entre as duas linguagens incorreram na necessidade de implementação de conversões adicionais nos dados transferidos. Estas diferenças se devem a como Fortran gerencia o tipo matriz de forma distinta em relação a C. Enquanto no C a forma de armazenamento de matrizes é *row-major*, Fortran utiliza *column-major*. Outra diferença reside na possibilidade de passagem de uma secção da matriz como parâmetro na linguagem Fortran, e a inexistência de uma característica igual em C. Ambas as diferenças foram resolvidas na implementação em linguagem C com aritmética de ponteiros.

A primeira implementação não resultou em ganho de performance, fazendo com que o tempo de computação da transformada ficasse superior ao do algoritmo original. Precipitadamente e erroneamente, a perda de desempenho foi atribuída ao *overhead* causado pelas conversões adicionais feitas no código C para compatibilidade com o gerenciamento do tipo matriz em Fortran. Após a medição detalhada do tempo de execução, e constatação de que os *kernels* CUDA implementados eram os principais responsáveis pela perda de performance, partiu-se para um estudo aprofundado da arquitetura alvo, com objetivo de identificar porque a implementação desenvolvida havia subutilizado os recursos da GPU, e de propor uma solução eficiente, que efetivamente acelerasse a implementação original.

5.3.3 Implementação Atual

O estudo da arquitetura baseou-se principalmente no trabalho de Volkov (2008). Ele executou uma série de microbenchmarks sobre a arquitetura das GPUs da NVIDIA,

com intuito de identificar gargalos, latências, *throughputs*, e reconstruir a hierarquia de memória a partir destes resultados (Figura 5.2), para então escrever rotinas eficientes para álgebra linear densa, baseadas nas informações obtidas. Com isso, as versões de Volkov para as rotinas GEMM e SYRK da biblioteca BLAS (*Basic Linear Algebra Subprograms*) sobre GPU tiveram desempenho melhor do que as desenvolvidas pela empresa NVIDIA, fabricante da arquitetura e do *framework*. As rotinas de Volkov foram então incorporadas no CUBLAS (NVIDIA, 2009a), a partir da versão 2.0. Os resultados de Volkov são importantes não só pelo ganho de desempenho nas novas rotinas da biblioteca CUBLAS, mas por trazer uma diferente visão da arquitetura. Volkov constatou, por exemplo, a existência de *caches* L1 e L2, *overheads* no acesso a memória compartilhada e latência de *pipeline*, e atingiu 98% do pico de performance das GPUs através de técnicas comuns na programação de processadores vetoriais. Poucos dos dados inferidos por Volkov são divulgados pelo fabricante, apesar de seu conhecimento contribuir positivamente na escrita de *kernels* que utilizem as GPUs no seu potencial máximo.

Com novas informações sobre a arquitetura, diante da percebida dificuldade na construção de *kernels* eficientes para resolução de álgebra linear densa na GPU, e do fato de as rotinas de Volkov terem sido incorporadas no CUBLAS, ao invés da escrita de *kernels* CUDA próprios para a multiplicação pelos Polinômios de Legendre, uma nova implementação foi feita utilizando as novas rotinas do CUBLAS, especialmente o DGEMM (multiplicação de matrizes de precisão dupla).

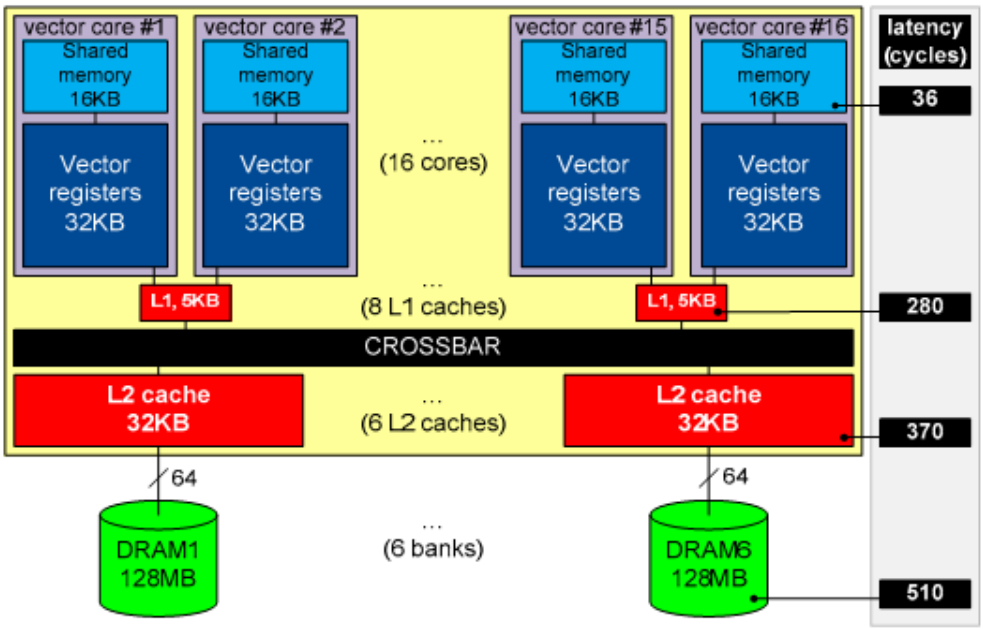

Figura 5.2: Arquitetura da GPU NVIDIA G80, construída a partir da execução de *microbenchmarks* (VOLKOV, 2008).

Simplificadamente, na Transformada de Legendre, a cada iteração para multiplicação de uma secção da matriz de coeficientes espectrais pela matriz com os Polinômios de Legendre, ao invés da utilização de *kernels* próprios em CUDA chamados a partir da ISO-C Binding para realização das operações, utiliza-se a rotina DGEMM do CUBLAS. Além da simplicidade da implementação – pois o CUBLAS

utiliza a forma *column-major* no trato de matrizes, como o Fortran - esta modificação resultou em ganho de desempenho comparativamente com o código original que executa exclusivamente em CPU. Os resultados são apresentados no capítulo 6.

Entretanto, os resultados obtidos não estão no patamar dos usados no *marketing* que os fabricantes de GPUs fazem da sua plataforma para computação de alto desempenho. Isso deve-se a estrutura do algoritmo base, discutida na seção 5.1, especificamente pelo número de transferências de memória entre RAM e placa gráfica, causadas pela grade quadrática, e pela grande quantidade de dados transferidos.

Como a aceleração relativamente baixa da implementação feita está relacionada a transferências de memória, foi desenvolvida uma nova implementação, com a utilização de *Page-Locked Memory*, um recurso disponível na biblioteca básica do CUDA. A *Page-Locked Memory* consiste em um conjunto de páginas da memória RAM que não muda de endereço físico e não sofre *swap*. Logo, com uma porção de memória física exclusiva, os tempos de transferência entre o dispositivo e a memória RAM são diminuídos. A implementação que usa este recurso foi feita novamente com a utilização do ISO-C Binding, para mapeamento das funções de alocação de memória *Page-Locked* do CUDA escritas originalmente para linguagem C na linguagem Fortran. Os resultados do impacto da utilização de Page-Locked Memory na implementação que utiliza GPU são mostrados no capítulo 6.

5.3.4 Implementação em Precisão Simples

Como a performance máxima teórica das GPUs atuais em precisão dupla é quase uma ordem de magnitude inferior ao desempenho máximo teórico em precisão simples, e frente ao iminente lançamento de atualizações arquiteturais (NVIDIA, 2009d), é importante observar o impacto que o aumento de recursos de *hardware*, essencialmente mais unidades de execução de precisão dupla, traz na implementação.

Mas diante dos esforços envolvidos na simulação de tal configuração, optou-se por reimplementar o algoritmo original em precisão simples. Conforme descrito na seção 5.2, os requisitos de precisão não permitem que o algoritmo forneça resultados corretos caso escrito desta maneira. No entanto, os resultados ficam próximos do correto, e o acesso a mais unidades de execução, e a possibilidade de observação de tal parâmetro no desempenho do algoritmo justificam a implementação.

Desta forma, o algoritmo original e as implementações feitas para GPU foram modificadas para executar em precisão simples. Os resultados destas modificações são apresentados no capítulo 6.

5.4 Metodologia de Avaliação

Para avaliação do desempenho optou-se por considerar o tempo de computação da Transformada Direta e Inversa de Legendre somadas e o tempo total de execução da Transformada Harmônica Esférica, compreendendo a fase de inicialização, as transformadas direta e inversa de Legendre e Fourier, e os *overheads*. Tanto para Legendre quanto para a transformada completa, são somados os tempos das 11 rodadas do *benchmark* descrito na seção 5.1. E todos os resultados apresentados neste trabalho representam a média de 10 execuções destas 11 rodadas. A verificação de corretude é habilitada e realizada nos testes em precisão dupla, e acontece entre a primeira e segunda rodada. Nos testes em precisão simples a verificação de corretude também é

feita, mas não aborta a execução em caso de detecção de diferença acima do *threshold* configurado. Para os testes em precisão simples, verificou-se erro já no algarismo mais significativo. No entanto, este erro nunca ultrapassou uma unidade. Verificou-se também que o teste de corretude não influi na performance da primeira rodada, nem nas rodadas posteriores. Além disso, o tempo de verificação não é computado nos resultados.

Os testes foram executados no ambiente detalhado na Tabela 5.1 e na Tabela 5.2, que respectivamente exibem a configuração de *software* e *hardware*. Conforme descrito na seção 5.1.1, o algoritmo não é *multi-threaded*, e teve o OpenMP e MPI desabilitados, portanto, executa em somente um dos núcleos do processador *dual-core* desta máquina. A placa gráfica utilizada, apesar de ser de alta performance, é desenvolvida para aceleração de gráficos, e todas as implementações testadas aceleradas por GPU vão dividi-la com o processamento de vídeo do sistema operacional. Este fator não modifica o desempenho dos testes de forma relevante, mas deve ser levado em consideração.

Tabela 5.1: Configuração de *software* da máquina de teste.

Componente	*Versão*
Sistema Operacional	Ubuntu 9.04 - 32bit
Compilador C	GCC 4.3.3
Compilador Fortran	GFortran 4.3.3
CUDA	2.3, *driver* 190.18

Tabela 5.2: Configuração de *hardware* da máquina de teste.

Componente	*Modelo*
Processador	Intel Core 2 Duo E8500 (Dois núcleos com frequência de 3,16GHz, socket LGA 775, FSB 1333MHz, 6MB Cache).
Memória	OCZ Reaper HPC Edition 4GB (2048MB x2, DDR2 1066 MHz PC2-8500 Kit).
Placa Mãe	MSI P7N SLI Platinum (*socket* LGA 775, *chipset* NVIDIA nForce 750i SLI).
Disco Rígido	Seagate 250GB ST3250410AS (7200RPM, 16MB *Cache*, SATA 3.0Gb/s).
Placa Gráfica	XFX GTX 280 XXX Edition GX-280N-ZDDU (670MHz, 1GB@2500MHz)

As medições foram realizadas com a variação do tamanho da entrada. Isso é feito através da modificação da truncagem do modelo, representada pela letra ‘T’. A partir da truncagem vertical é possível obter a resolução, conforme a Tabela 5.3 e a Tabela 5.4. O número de camadas verticais é mantido estável em 28. Este valor foi escolhido por ser típico nas execuções reais. Valores maiores ou menores para o número de camadas verticais influem linearmente no tempo de execução e na quantidade de dados

processados. Assim, aumentar ou diminuir este valor apenas faz com que seja possível executar modelos com truncagem menor ou maior, respectivamente, considerando os limites de memória.

Para a truncagem, o valor mínimo escolhido foi 10, por ser um dos menores valores que não gera erro na execução do modelo, e pela pequena quantidade de dados processados. O valor máximo de truncagem depende do tamanho dos dados, da quantidade máxima de memória RAM disponível para a aplicação, e do número de camadas verticais. Devido as limitações da arquitetura x86, no máximo 3 GB de memória são endereçados pelo sistema operacional. O número de camadas verticais foi fixado em 28, conforme descrito no parágrafo anterior. A partir disso, nos testes em precisão simples foi possível executar um modelo com truncagem até 260, e nos testes em precisão dupla até 190. Entre o valor mínimo e máximo de truncagem foram feitas nove medições, aumentando a truncagem em 25 a cada nova medição em precisão simples e, consequentemente, aumentando em 18 em precisão dupla. Esta quantidade de medições foi considerada suficiente para a identificação de eventuais máximos ou mínimos locais.

Tabela 5.3: Resoluções dos testes realizados em precisão simples.

Truncagem	*T10*	*T35*	*T60*	*T85*	*T110*	*T135*	*T160*	*T185*	*T210*	*T235*	*T260*
Resolução (Km)	1250	370	208	156	111	92	80	69	52	55	50

Tabela 5.4: Resoluções dos testes realizados em precisão dupla.

Truncagem	*T10*	*T28*	*T46*	*T64*	*T82*	*T100*	*T118*	*T136*	*T154*	*T172*	*T190*
Resolução (Km)	1250	416	277	200	156	125	111	92	83	74	69

Além das implementações mencionadas nas seções anteriores: utilizando GPU para aceleração, em precisão simples e dupla, com e sem *Page-Locked Memory*; também foi testado o impacto da utilização de *Page-Locked Memory* no algoritmo base, em precisão simples e dupla. O objetivo é expor o *overhead* de alocação e inicialização deste tipo de memória.

E, finalizando, a partir da identificação da performance inesperadamente ruim nas implementações de CPU em precisão simples, também foi testado, para todas as implementações, o efeito das otimizações matemáticas e do conjunto de instruções SIMD, ativadas através das *flags* de compilação "*ffast-math"*, "*mfpmath=sse,387"* e "*msse2"*. Além disso, todas as implementações foram compiladas com a *flag* de otimização O3. A Tabela 5.5 mostra todas as 16 implementações contempladas nos testes. Nesta tabela, "CPU" refere-se ao código base, "CPU+GPU" ao código base acelerado por GPU na Transformada de Legendre, "SSE" indica compilação com as *flags* matemáticas e SIMD e "PL" indica utilização de *Page-Locked Memory*.

Tabela 5.5: Implementações testadas

CPU Double	CPU Single	CPU+GPU Double	CPU+GPU Single
CPU Double SSE	CPU Single SSE	CPU+GPU Double SSE	CPU+GPU Single SSE
CPU Double PL	CPU Single PL	CPU+GPU Double PL	CPU+GPU Single PL
CPU Double PL SSE	CPU Single PL SSE	CPU+GPU Double PL SSE	CPU+GPU Single PL SSE

5.5 Trabalho Correlato

Soman (2009) implementou a Transformada Harmônica Esférica com o intuito de acelera-la com a utilização de processadores gráficos. Após analisar o desempenho da implementação de Drake et al. (2008) escrita em MATLAB, Soman transferiu trechos da computação para a GPU, utilizando o *framework* CUDA. Os trechos implementados são parte do cálculo dos Polinômios de Legendre, da fase de Síntese, e da fase de Análise.

Na implementação feita por Soman, parte do código executa em CPU, e parte em GPU. No cálculo dos Polinômios de Legendre, em GPU foi feita a multiplicação do polinômio pelo fator de normalização. Na Análise, a Transformada Inversa de Legendre e, na Síntese, a Transformada Direta de Legendre. As Transformadas de Fourier são feitas em CPU.

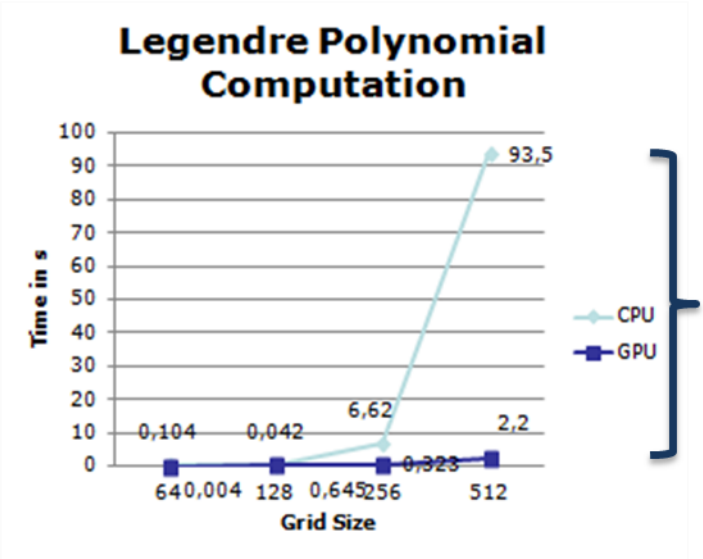

Figura 5.3: Tempo de execução da computação dos Polinômios de Legendre (SOMAN, 2009).

Soman comparou a execução do código sequencial original escrito totalmente em MATLAB com o código com trechos escritos em CUDA em uma máquina com um processador Intel Core 2 Quad de 2,5 GHz, 2,5 GiB de memória RAM, e uma placa gráfica NVIDIA 8800 GT. Na computação dos Polinômios de Legendre, Soman obteve até 42 vezes de ganho no tempo de execução[2] para o tamanho máximo testado (Figura 5.3). Na Síntese e na Análise, os trechos em GPU causaram perda de desempenho para os tamanhos de grade testados, como mostrado na Figura 5.4 e na Figura 5.5. O ganho no cálculo dos polinômios é atribuído à exploração do paralelismo de dados. A perda na fase de Análise é atribuída ao fato de a Transformada de Fourier utilizar valores complexos, que aumentam a quantidade de dados transferidos entre CPU e placa gráfica, em comparação aos valores reais. Este mesmo motivo é usado para justificar a perda na fase de Síntese.

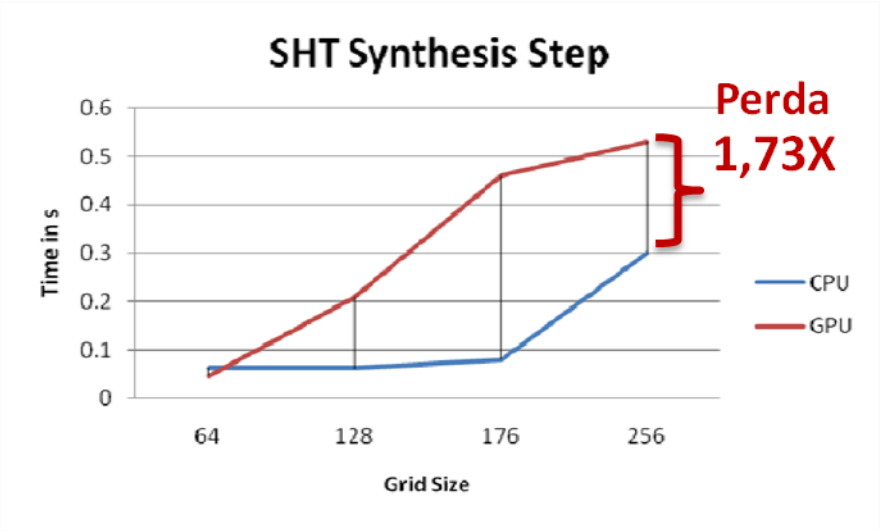

Figura 5.4: Tempo de execução da fase de Síntese (SOMAN, 2009).

Os tempos de execução da fase de Síntese e Análise do algoritmo sequencial utilizado por Soman são semelhantes ao do código utilizado neste trabalho. Entretanto, o tempo de execução da computação dos Polinômios de Legendre de Soman é mais de duas ordens de magnitude maior. Este tempo é, inclusive, maior que o tempo total de execução da Transformada Harmônica Esférica Direta e Inversa na versão sequencial utilizada neste trabalho, considerando entradas de tamanho equivalente. A versão em GPU escrita por Soman do cálculo dos Polinômios de Legendre é cerca de seis vezes mais lenta do que versão sequencial utilizada neste trabalho.

[2] Em cada gráfico apresentado por Soman é representado o desempenho de uma implementação nomeada como "GPU". Ela refere-se ao código base executando em CPU com porções modificadas para executarem em GPU, e não a um código que executa totalmente na GPU.

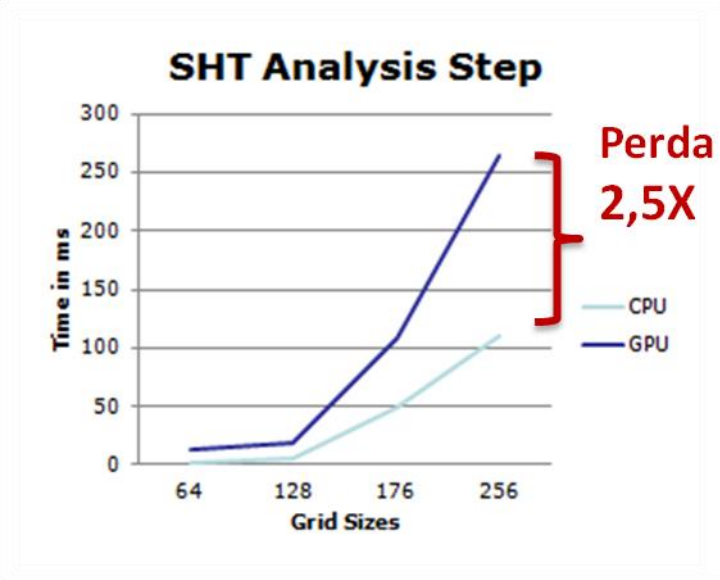

Figura 5.5: Tempo de execução da fase de Análise (SOMAN, 2009).

6 RESULTADOS

Este capítulo apresenta os resultados das implementações detalhadas na seção 5.3, seguindo a metodologia de avaliação descrita na seção 5.4, e está organizado da seguinte forma: na seção 6.1 são apresentados os resultados das implementações em CPU, com o objetivo de identificar a mais veloz no cálculo dos Polinômios de Legendre e no total da Transformada Harmônica Esférica, em precisão simples e precisão dupla, para posterior comparação com as implementações aceleradas por GPU. Nas seções 6.3 e 6.2 são apresentados os resultados referentes à Transformada de Legendre acelerada por GPU conforme detalhamento da seção 5.3.3. Nestas mesmas seções é feita a comparação do algoritmo mais rápido no cálculo da transformada executando exclusivamente em CPU com a mais rápida acelerada por GPU. Por fim, nas seções 6.5 e 6.4 o mesmo é feito, mas considerando a Transformada Harmônica Esférica.

Nos gráficos que apresentam comparações entre diferentes configurações de implementações que executam em uma única arquitetura, a aplicação mais veloz tem sua legenda circulada.

6.1 Implementação Base

6.1.1 Precisão Dupla

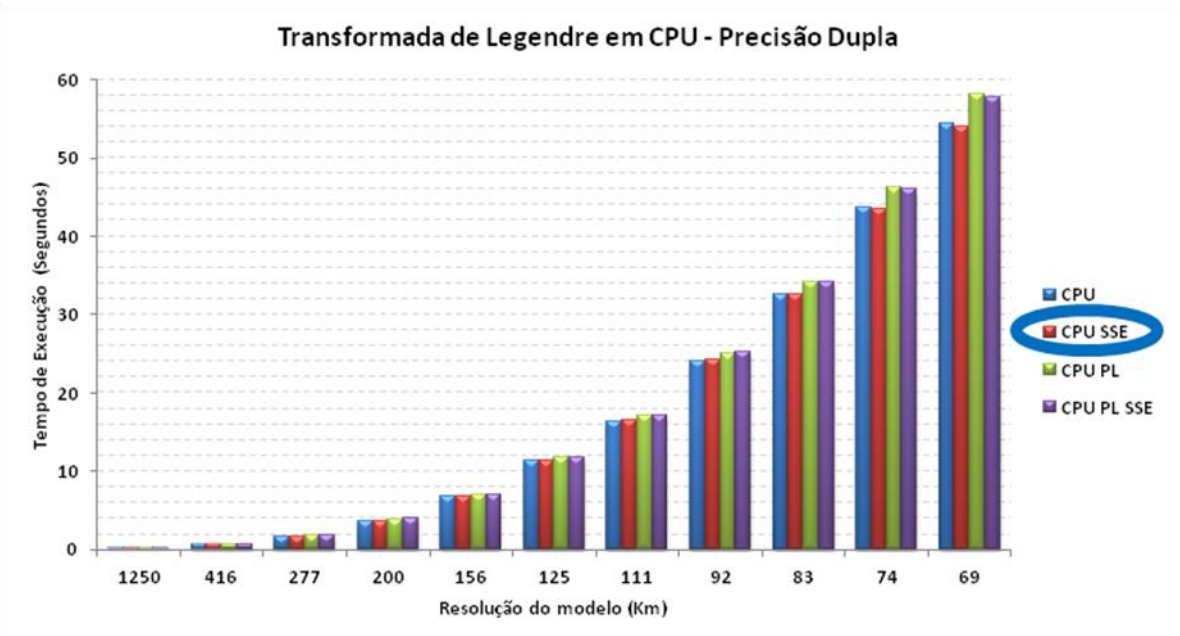

Figura 6.1: Transformada de Legendre em CPU - Precisão Dupla.

A Figura 6.1 e a Figura 6.2 mostram o tempo de execução de todas as implementações de CPU de precisão dupla, no cálculo da Transformada de Legendre e no total da Transformada Harmônica Esférica, respectivamente. Na Transformada de Legendre, a configuração mais rápida em CPU é a que faz uso das otimizações matemáticas e das instruções SIMD, como esperado. A utilização de *Page-Locked Memory* por parte destas implementações causou uma perda de 7% considerando os modelos com alta resolução. Nos modelos de baixa resolução, houve até 8 vezes de perda.

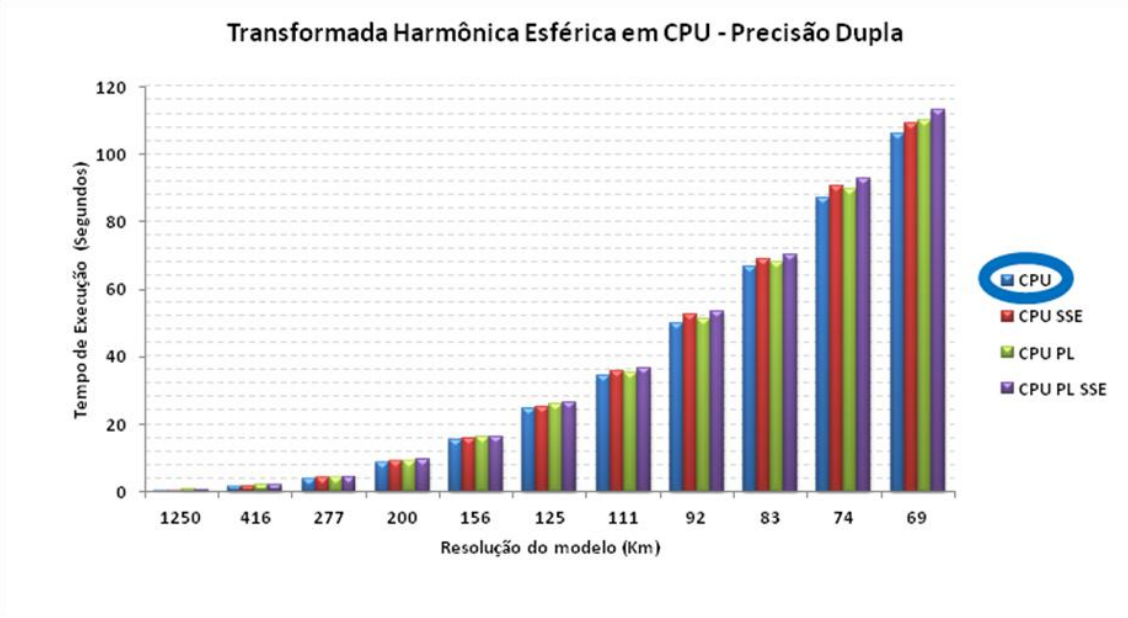

Figura 6.2: Transformada Harmônica Esférica em CPU - Precisão Dupla.

Agora, considerando a Transformada Harmônica Esférica, e diferentemente do esperado, a versão mais rápida é a compilada apenas com as otimizações básicas. No código completo da transformada as *flags* de otimizações extras causaram perda, ainda que pequena, ao invés de aceleração, provavelmente por falhas nos mecanismos de otimização do compilador. A utilização da *Page-Locked Memory* surtiu o efeito esperado.

6.1.2 Precisão Simples

A Figura 6.3 e a Figura 6.4 mostram o tempo de execução das implementações de CPU em precisão simples, no cálculo da Transformada de Legendre e no total da Transformada Harmônica Esférica, respectivamente. Assim como em precisão dupla, na Transformada de Legendre, a configuração mais rápida em CPU é a compilada com as *flags* de otimizações matemáticas e SIMD. Desta vez, a utilização de *Page-Locked Memory* causou perdas consideráveis. Na menor resolução esta perda foi de 2,25 vezes. No maior, chegou a 75%. Anomalamente, o desempenho de precisão simples no geral ficou abaixo do esperado, em comparação com precisão dupla. Conclui-se que, com o

código utilizado, o compilador gerou código mais eficiente para precisão dupla do que para precisão simples. Os resultados da Transformada Harmônica Esférica são semelhantes a Legendre, excetuando as proporções de tempo de execução.

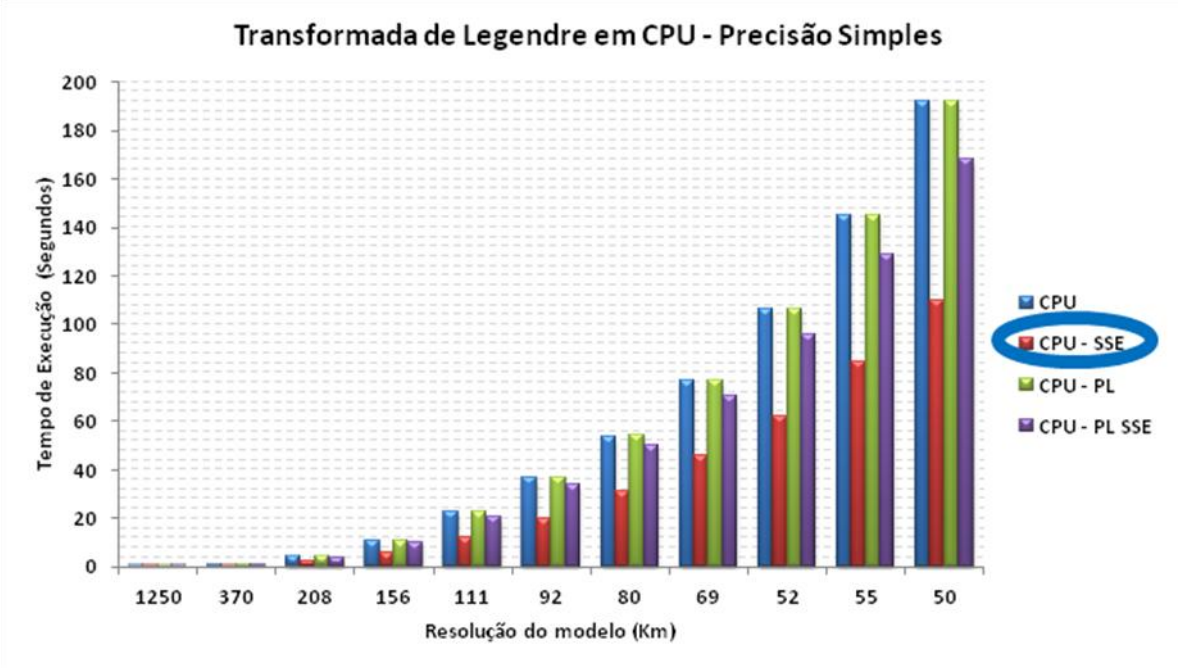

Figura 6.3: Transformada de Legendre em CPU - Precisão Simples.

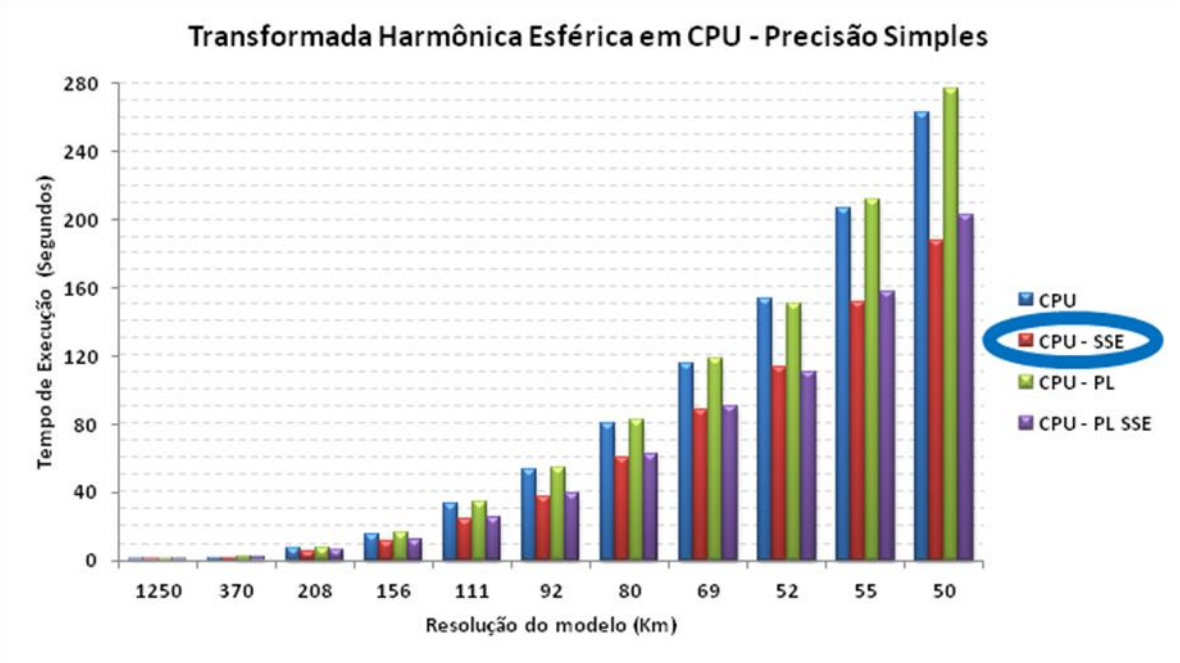

Figura 6.4: Transformada Harmônica Esférica em CPU - Precisão Simples.

6.2 Transformada de Legendre em Precisão Dupla

A Figura 6.5 mostra o tempo de execução do cálculo da Transformada de Legendre de todas as implementações aceleradas por GPU em precisão dupla. A versão mais veloz é a compilada com *flags* de otimização matemática e SIMD que faz uso da *Page-Locked Memory*, como se esperava. Todavia, estas otimizações fizeram uma diferença menor do que considerando as implementações que executam apenas em CPU, chegando algumas vezes até a causarem uma pequena perda, ao invés de ganho.

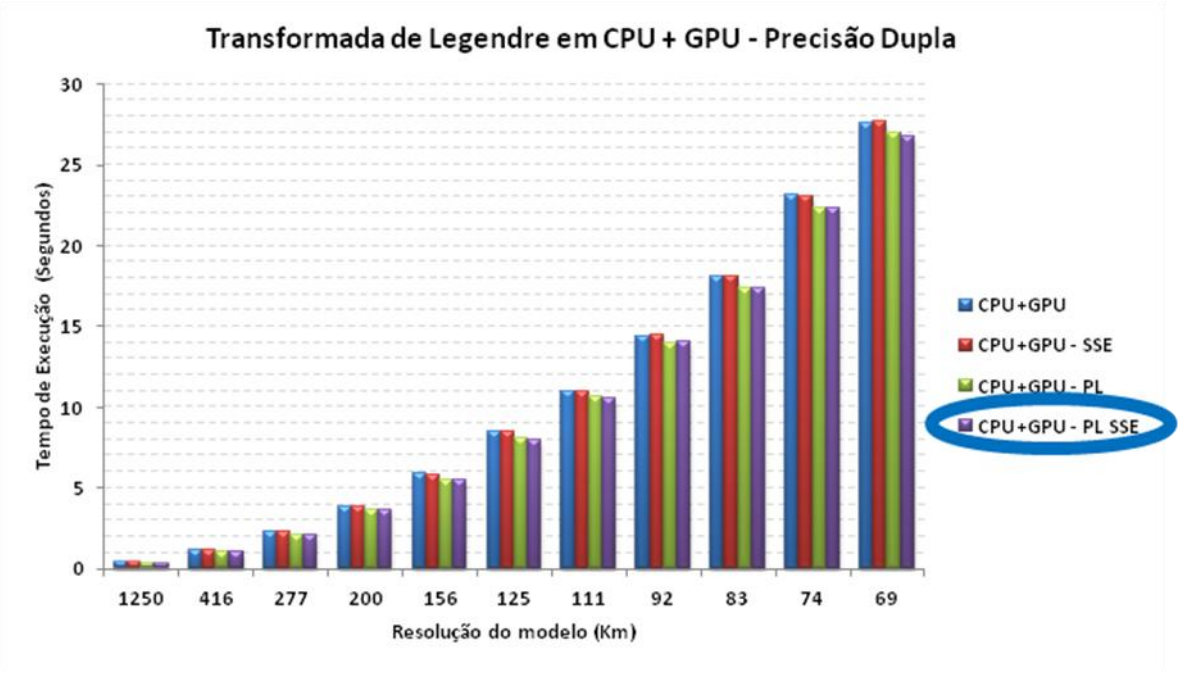

Figura 6.5: Transformada de Legendre em CPU+GPU - Precisão Dupla.

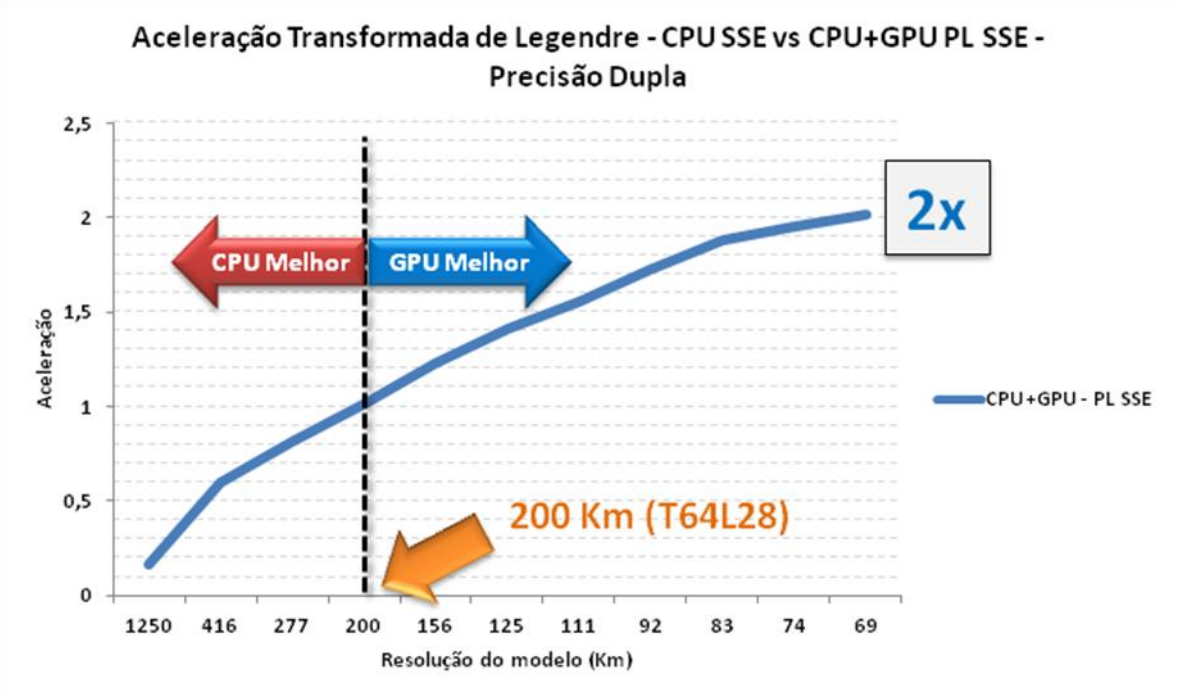

Figura 6.6: Transformada de Legendre: CPU vs GPU - Precisão Dupla.

Na Figura 6.6 são comparadas a implementação mais veloz em CPU e a implementação mais rápida com aceleração de GPU. Para o cálculo da Transformada de Legendre em precisão dupla, a GPU chegou a ser duas vezes mais rápida. Até T64, no entanto, a GPU causa perda de desempenho. Para a menor resolução, esta perda é de seis vezes. Isso é justificado pelas latências de chamadas, transferências, e pelos tempos de transferências. Estes são superados pelo aumento de velocidade no tempo de cálculo quando a GPU é alimentada com dados suficientes. Para resoluções pequenas, não é lucrativo perder tempo com latências e transferências de memória para processar dados que a CPU também processaria rapidamente.

6.3 Transformada de Legendre em Precisão Simples

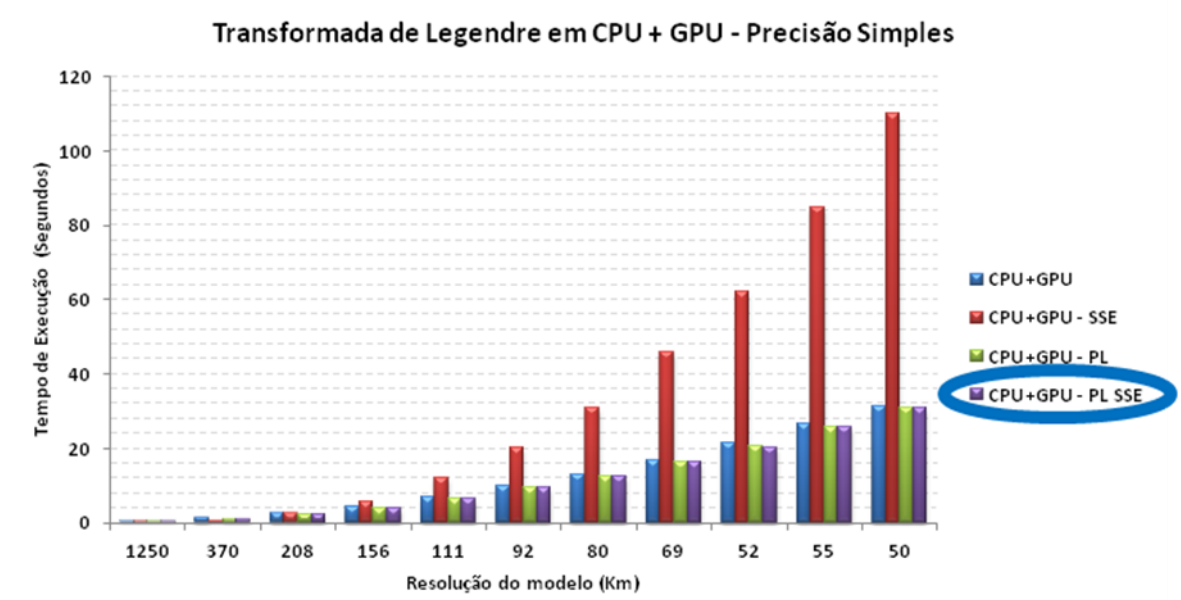

Figura 6.7: Transformada de Legendre em CPU+GPU - Precisão Simples.

Na Figura 6.7 são representados os tempos de execução do cálculo da Transformada de Legendre de todas as implementações aceleradas por GPU. Como esperado, a versão mais rápida é a compilada com *flags* de otimização extra e que faz uso da *Page-Locked Memory*. Neste teste o compilador gerou código extremamente ineficiente para a aplicação com as *flags* extras de otimização que não faz uso de *Page-Locked Memory*.

A Figura 6.8 mostra a comparação da mais rápida implementação em CPU com a mais rápida acelerada por GPU, na execução da Transformada de Legendre em Precisão Simples. A utilização de GPU resultou em ganhos de até 3,6 vezes e, semelhantemente ao código em precisão dupla, a CPU é mais lenta a partir de T60.

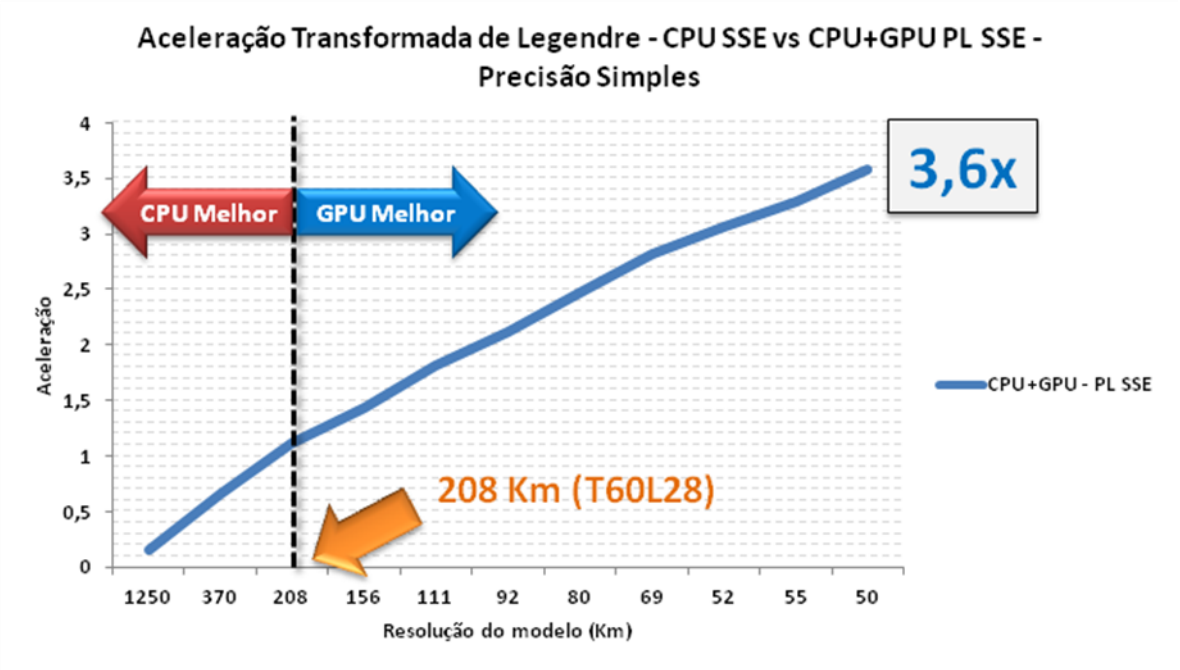

Figura 6.8: Transformada de Legendre: CPU vs GPU - Precisão Simples.

6.4 Transformada Harmônica Esférica em Precisão Dupla

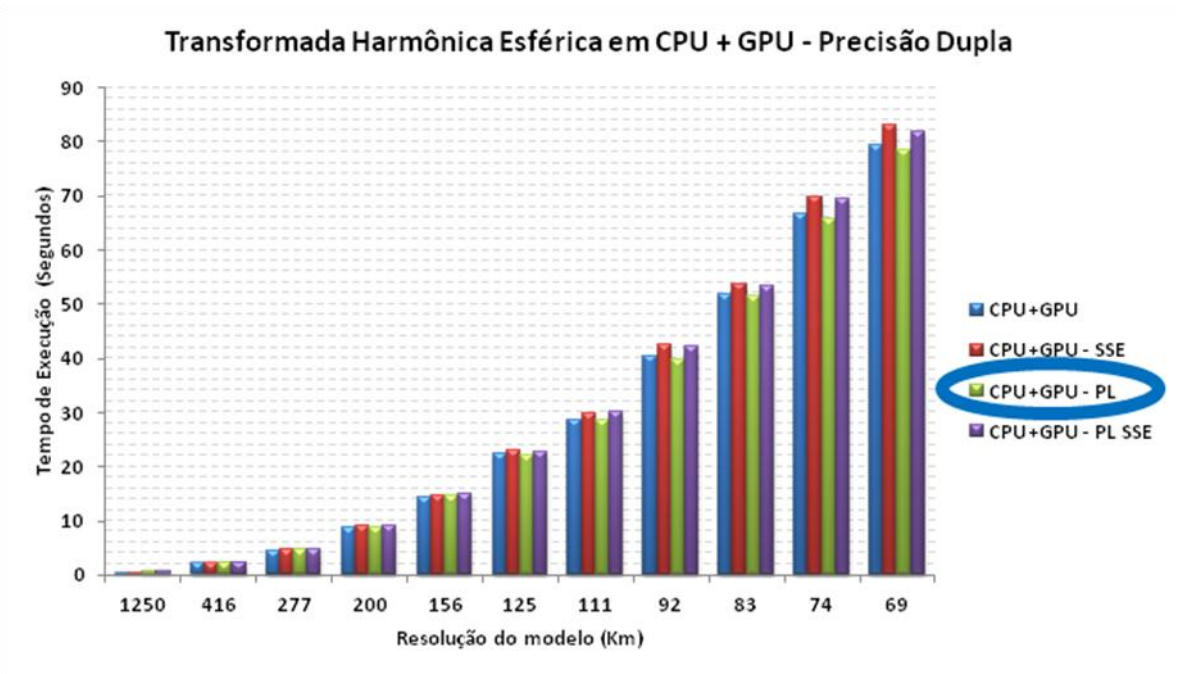

Figura 6.9: Transformada Harmônica Esférica em CPU+GPU - Precisão Dupla.

Na Figura 6.10, são comparadas a mais rápida implementação de CPU e a mais rápida acelerada por GPU, considerando-se a execução da Transformada Harmônica Esférica em precisão dupla. A aplicação com Legendre acelerada por GPU resulta em

até 25% de ganho no tempo total e, assim como na execução da Transformada de Legendre, a CPU passa a ser mais lenta a partir de T64.

A Figura 6.9 mostra a comparação de todas as implementações em precisão dupla aceleradas por GPU na execução da Transformada Harmônica Esférica completa. A mais veloz é a que faz uso de *Page-Locked Memory*, compilada sem as *flags* extras de otimização.

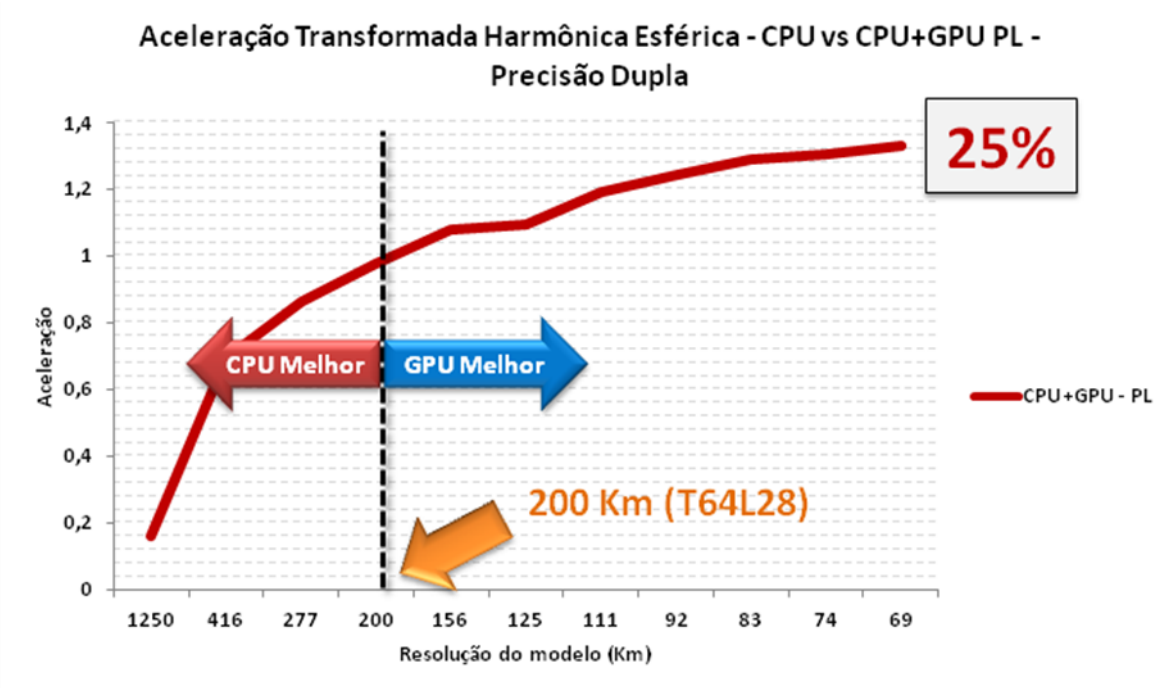

Figura 6.10: Transformada Harmônica Esférica: CPU vs GPU - Precisão Dupla.

6.5 Transformada Harmônica Esférica em Precisão Simples

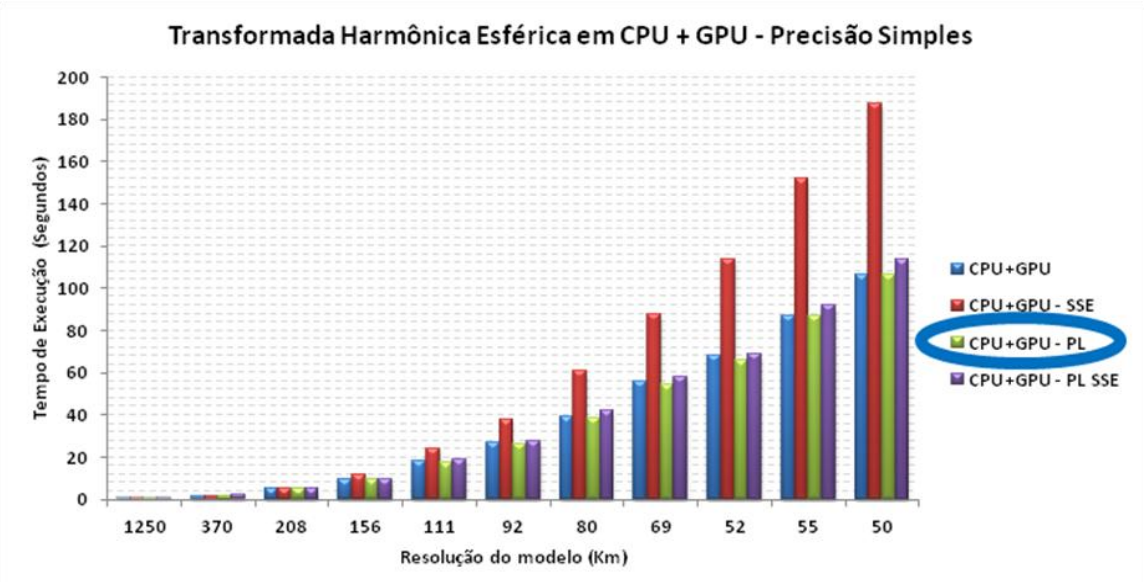

Figura 6.11: Transformada Harmônica Esférica em CPU+GPU - Precisão Simples.

A comparação de todas as implementações aceleradas por GPU em precisão simples na execução da Transformada Harmônica Esférica é mostrada na Figura 6.11. A configuração mais veloz é a que faz uso de Page-Locked Memory, sem *flags* de otimização extra.

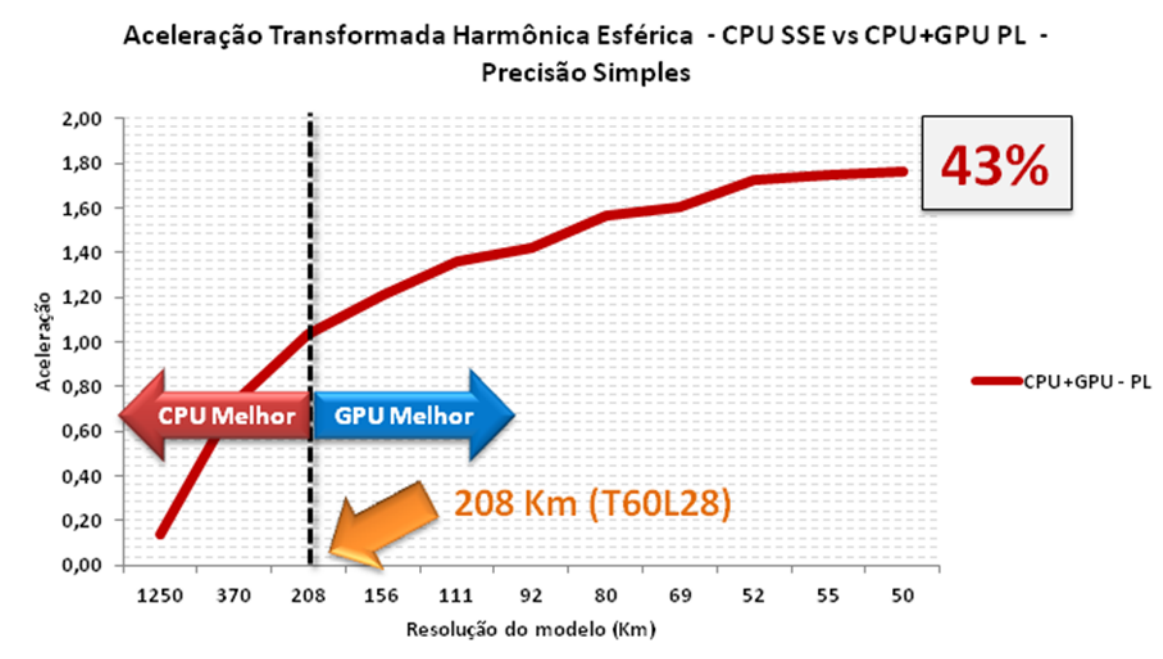

Figura 6.12: Transformada Harmônica Esférica: CPU vs GPU - Precisão Simples.

Na Figura 6.12 a mais rápida implementação de CPU é comparada com a mais veloz acelerada por GPU na execução da Transformada Harmônica Esférica em precisão simples. A GPU proporciona até 43% de ganho e, como nos outros testes, a CPU é mais lenta a partir de T60.

7 CONCLUSÃO

Este trabalho apresentou uma implementação em GPU da Transformada de Legendre, algoritmo fundamental em modelos de previsão meteorológica de circulação global, e altamente demandante de recursos de *hardware*. Nestes modelos, ela é composta com a Transformada de Fourier, constituindo a Transformada Harmônica Esférica.

A implementação em GPU acelerou a computação em até duas vezes para a máxima resolução testada em precisão dupla, causando uma melhora de até 25% na Transformada Harmônica Esférica. Em precisão simples, Legendre ficou até 3,6 vezes mais rápida, resultando em 43% de ganho no tempo total. Em todos os cenários a implementação desenvolvida foi mais eficiente que o trabalho correlato.

Os ganhos não são maiores devido as limitações das atuais GPUs na execução de código em precisão dupla, e a características do algoritmo, como a grade quadrática, a necessidade de mais transferências entre placa gráfica e memória RAM devido as comunicações em MPI, e a grande quantidade de dados envolvidos nestas transferências. Ainda assim, a GPU mostrou-se mais veloz que a CPU para resoluções maiores que 200 Km. E apesar da pouca aceleração em comparação com certas aplicações, com esta implementação a CPU é desonerada durante a execução da Transformada de Legendre, podendo executar outras tarefas.

A partir disso, trabalhos podem ser desenvolvidos para utilizar o potencial do crescente número de núcleos dos processadores de propósito geral em conjunto com o processamento das GPUs. E a GPU pode ser aproveitada para a execução de mais tarefas, como a Transformada de Fourier, que já possui diversas implementações para GPU com performance melhor do que os algoritmos de CPU. Nesta linha, também podem ser utilizadas múltiplas GPUs, para execução de modelos com resolução ainda maior.

Novas arquiteturas (NVIDIA, 2009d), (SEILER, 2008) e novas versões do *framework* CUDA podem possibilitar otimizações relativamente simples, como execução simultânea de múltiplas chamadas CUDA e sobreposição de processamento e transferências de memória, algo que atualmente não é possível, pois o CUBLAS é síncrono. Além disso, o aumento de unidades de execução de precisão dupla nestas arquiteturas pode beneficiar consideravelmente a implementação desenvolvida, como foi visto pelos resultados com precisão simples.

Por fim, a transformada pode ser reescrita em precisão simples, utilizando a técnica de refinamentos em precisão dupla quando necessário. Como mostrado, esta reescrita, independente de novas versões de *hardware* ou *software*, já resultaria em ganho de desempenho sobre a atual implementação.

REFERÊNCIAS

(ALVES, 2009) ALVES, M. A.; SOUTO, R. P., NAVAUX, P. O. A. **Avaliação de Escalabilidade do Modelo Atmosférico Global em Arquitetura Multi-Core**. Anais – 9ª Escola Regional de Alto Desempenho. SBC/UCS/UFSM/UFRGS/UPF, 2009.

(ARNOLD, 1989) ARNOLD, V. I. **Mathematical Methods of Classical Mechanics**. 2nd ed. New York: Springer, 1989.

(DRAKE, 2008) DRAKE, J. B.; WORLEY, P.; D'AZEVEDO, E. Algorithm 888: Spherical harmonic transform algorithms. **ACM Transactions on Mathematical Software**, New York, v.35, n.3, Article 23. October 2008.

(FLAGON, 2009) FLAGON: Fortran 9x Library for GPU Numerics. 2007. Disponível em <http://sourceforge.net/apps/trac/flagon/wiki>. Acesso em dez. 2009.

(NVIDIA, 2009a) NVIDIA CUBLAS Library Version 2.3. Santa Clara, California. June, 2009. Disponível em <http://developer.download.nvidia.com/compute/cuda/2_3/sdk/cudasdk_2.3_linux.run>. Acesso em dez. 2009.

(NVIDIA, 2009b) NVIDIA CUDA Compute Device Unified Architecture: Programming Guide Version 2.3.1. Santa Clara, California. August, 2009. Disponível em <http://developer.download.nvidia.com/compute/cuda/2_3/sdk/cudasdk_2.3_linux.run>. Acesso em dez. 2009.

(NVIDIA, 2009c) NVIDIA GeForce 8800 GPU Architecture Overview. Santa Clara, California. November, 2006. Disponível em <http://www.nvidia.com/object/IO_37100.html>. Acesso em dez. 2009.

(NVIDIA, 2009d) NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Version 1.1. 2009. Disponível em <http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>. Acesso em dez. 2009.

(PEIXOTO, 2009) PEIXOTO, P. da S.; BARROS, S. R. M. **Métodos Espectrais para Modelos Meteorológicos - Uma Introdução**. Disponível em<http://www.ime.usp.br/~pedrosp/MetEspecMeteo.pdf>. Acesso em dez. 2009.

(PHARR, 2005) PHARR, M.; FERNANDO, R. **GPU Gems 2**: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Boston: Addison-Wesley, 2004.

(PURCELL, 2005) PURCELL, T. J. et al. **Ray Tracing on Programmable Graphics Hardware**. International Conference on Computer Graphics and Interactive Techniques. Article No. 268. ACM, New York, NY, USA, 2005.

(SEILER, 2008) SEILER, L. et al. Larrabee: a many-core x86 architecture for visual computing. **ACM Transactions on Graphics**. New York, v.23, n.3. August 2008.

(SOMAN, 2009) SOMAN, V. **Accelerating Spherical Harmonic Transforms On The NVIDIA GPU**. 2009. ECE 734 VLSI Array Processors for Digital Signal Processing - Spring 2009 Class Projects. May 4, 2009. Disponível em < http://homepages.cae.wisc.edu/~ece734/project/s09/soman_rpt.pdf>. Acesso em dez. 2009.

(VOLKOV, 2008) VOLKOV, V.; DEMMEL, J. W. **Benchmarking GPUs to Tune Dense Linear Algebra**. SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. Austin, Texas: IEEE Press: 2008.

(ZIA, 2009) ZIA, R. K. P.; REDISH, E. F.; MCKAY, S. R. Making Sense of the Legendre Transform. **American Journal of Physics**, [S.l.], v.77, n.7, p. 614-622. July 2009.