MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

EXPERIMENTOS EM GPU PARA O MODELO DE PREVISÃO AMBIENTAL CCATT-BRAMS

RELATÓRIO FINAL DE PROJETO DE INICIAÇÃO CIENTÍFICA
(PIBIC/CNPq/INPE)

Leandro dos Santos Lessa (FATEC, Bolsista PIBIC/CNPq)
E-mail: leandroicinpe@gmail.com

Dr. Haroldo Fraga de Campos Velho (LAC/INPE, Orientador)
E-mail: haroldo@lac.inpe.br

Dra. Renata Sampaio da Rocha Ruiz (LAC/INPE, Orientadora)
E-mail: renata@lac.inpe.br

INPE
São José dos Campos
2013

Dados Internacionais de Catalogação na Publicação

Cutter

Sobrenome, Prenome(s) Completos do(s) Autor(es).
Título da publicação / Nome Completo do Autor(es). - São José dos Campos: INPE, ano da publicação.
i + 0p. ; (aa/bb/cc/dd-TDI)

Grau (Mestrado ou Doutorado em Nome do Curso) - Instituto Nacional de Pesquisas Espaciais, São José dos Campos, ano de defesa.
Orientador: Nome completo do orientador(es).

1. Assunto. 2. Assunto. 3. Assunto. 4. Assunto. 5. Assunto. I. Título.

CDU

Copyright AAAA do MCT/INPE. Nenhuma parte desta publicação pode ser reproduzida, armazenada em um sistema de recuperação, ou transmitida sob qualquer forma ou por qualquer meio, eletrônico, mecânico, fotográfico, reprográfico, de microfilmagem ou outros, sem a permissão escrita do INPE, com exceção de qualquer material fornecido especificamente no propósito de ser entrado e executado num sistema computacional, para o uso exclusivo do leitor da obra.

Copyright AAAA by MCT/INPE. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, microfilming or otherwise, without written permission from the INPE, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use of the reader of the work.

AGRADECIMENTOS

Agradeço a contribuição e o apoio do CNPq e INPE (Instituto Nacional de Pesquisas Espaciais) por acreditar e contribuir com esse projeto. Agradeço os meus orientadores Dr. Haroldo Fraga de Campos Velho e Dr. Renata Sampaio da Rocha Ruiz, pela paciência e dedicação para comigo, creio que foram pessoas chaves para o meu desenvolvimento profissional e pessoal. Serei enternamente grato.

RESUMO

Modelos de previsão numérica de tempo são programas que requerem uso de computação intensiva devido a alta complexidade dos fenômenos e dos algoritmos envolvidos. Uma estratégia para melhorar o desempenho desses algoritmos é o uso da computação heterogênea que combina CPU com aceleradores (dispositivos de *hardware* utilizados para realizar parte do processamento). Entre os dispositivos de *hardware*, ganha cada vez mais destaque as placas gráficas por meio GP-GPU (*General Purpose Graphical Processing Unit*). O presente trabalho de inciação científica consiste na investigação de estratégias para o desenvolvimento em GPU, usando o padrão CUDA, da rotina de parametrização da Turbulência de Smagorinsky (proposta em 1963), implementada no modelo de previsão BRAMS (*Brazilian Regional Atmospheric Modeling System*) e do modelo de previsão ambiental CCATT-BRAMS(Coupled-Chemical Aerosol and Tracer Transport model to the Brazilian Regional Atmospheric System) desenvolvido e mantido pelo CPTEC/INPE. Esta pesquisa inicia-se com o aprendizado do processo de instalação, configuração e execução do modelo. Em seguida, é realizada uma análise da rotina original, identificando os trechos a serem paralelizados. Após essa análise, foi feita a codificação da rotina em GPU. O desempenho da implementação foi avaliado por meio do cálculo de speedup, considerando duas resoluções horizontais para o BRAMS: 40 km e 20 km. A comparação entre os tempos obtidos com a rotina original (em Fortran) e a rotina em CUDA, mostraram um ganho significativo de desempenho, com *speed-up* de 8 até 17 vezes mais rápido que a implementação serial.

EXPERIMENTS IN GPU FOR FORECAST MODEL ENVIRONMENTAL CCATT-BRAMS

ABSTRACT

Numerical weather prediction models are codes requiring the use of the intensive computation. The simulation describes complex phenomena with very high number of arithmetic operations. A strategy to improve the performance in these models is the use of heterogeneous computing, combining CPU with accelerators. One hardware component for the latter purpose is the General Purpose of Graphics Processing Unit (GP-GPU). The present study investigates the turbulence parameterization routine in the model BRAMS (Brazilian Regional Atmospheric System), a regional meteorological code developed and supported by the CPTEC (Centro de Previsão de Tempo e Estudos Climáticos) of the INPE (Instituto Nacional de Pesquisas Espaciais). Several steps were needed to carry out the development: (i) BRAMS: installing, configuring, and running, (ii) analysis of the original Fortran routine, for identifying the component to be parallelized, (iii) turbulence GPU codification, using CUDA programming environment. Tests were executed considering two horizontal model resolutions on a South American region: 40 km, and 20 km. The performance showed a *speed-up* from 8 up to 17 times faster for GPU-code than serial one-core CPU.

LISTA DE FIGURAS

Pág.

Figura 2.1 - Crescimento exponencial do poder computacional 6
Figura 2.2 - Operações em ponto flutuante por segundo em GPU (NVIDIA) e CPU 9
Figura 2.3 - Largura de banda memória (memory bandwidth) CPU e GPU.................... 9
Figura 2.4 - Arquitetura CPU x GPU. 11
Figura 2.5 - Execução do programa em CUDA. 12
Figura 2.6 - Arquitetura Fermi. 14
Figura 2.7 - Ilustração do esquema da transferência de dados entre CPU e GPU.......... 15
Figura 2.8 - Esquema hierárquico de memória da GPU. 16
Figura 2.9 - Hierarquia de *threads*, blocos e *threads*. 17
Figura 2.10 - Relacionamento em nível de memória entre GPU e CPU. 18
Figura 2.11 – Sintaxe chamada *Kernel*.. 19
Figura 2.12 - Trecho de código com a parametrização de Smagorinsky. 23
Figura 2.13 - Trecho de código com a parametrização de Smagorinsky em GPU......... 24
Figura 3.1 - Lendo arquivos de entrada. 26
Figura 3.2 - Sintaxe cudaMalloc(). 27
Figura 3.3 - Transferência Assíncrona dos dados para GPU. 28
Figura 3.5 - Função Kernel. 30
Figura 3.6 - Sintaxe cudamemcpy(). 30
Figura 3.7 – Fluxograma de funcionamento do programa 31
Figura 4.1 – Topografia da região da América do Sul utilizada nos testes 34
Figura 4.2 – Gráfico dos tempos resolução 40km 35
Figura 4.3 – Gráficos do tempo resolução de 20km 36
Figura 4.4 – *Speed-up* com e sem transferência de dados, considerando resolução de 40km 38
Figura 4.5 – *Speed-up* com e sem transferência de dados considerando resolução de 20km 39

LISTA DE TABELAS

Pág.

Tabela 1 - Níveis e funções de cada memória da GPU. 16
Tabela 2 - Capacidade de computação da arquitetura fermi............................ 29
Tabela 3 - Resultado para resolução de 40 km.. 35
Tabela 4 - Resultados para resolução 20km.. 36
Tabela 5 - Tempos para resolução de 40 km (em ms).................................... 37
Tabela 6 - Tempos para resolução de 20 km (em ms)..................................... 37

LISTA DE SIGLAS E ABREVIATURAS

BRAMS	*Brazilian Regional Atmospheric Modeling System*
CCATT-BRAMS	*Coupled-Chemical Aerosol and Tracer Transport model to the Brazilian Regional Atmospheric System*
CNPq	Conselho Nacional de Pesquisa científico e tecnológico
CPTEC	Centro de Previsão de Tempo e Estudos Climáticos
CPU	*Central Processing Unit*
CSU	*Colorado State University*
CUDA	*Compute Unified Device Acceleration*
FLOPS	*Floating Point Operations per Second*
GB	*Giga Byte*
GDDRS	*Graphic Double Data Rate*
GPU	*Graphic Processing Unit*
HDF	*Hierarchical Data Format*
HPC	*High Performance Computing*
INPE	Instituto Nacional de Pesquisas Espaciais
KB	*Kilo Byte*
MIMD	*Multiple Instruction Multiple Data*
MISD	*Multiple Instruction Single Data*
MPI	*Message Passing Interface*
RAM	*Random Access Memory*
RAMS	*Regional Atmospheric Modeling System*
SIMD	*Single Instruction Single Data*
SIMT	*Single Instruction, Multiple Thread*
SISD	*Single Instruction Single Data*
SMs	*Streaming Multiprocessors*

SUMÁRIO

Pág.

1 INTRODUÇÃO ... 1

1.1.Objetivo ... 3

1.1.1.Objetivo geral ... 3

1.1.2.Objetivos Específicos ... 3

1.2.Metodologia ... 3

2 COMPUTAÇÃO DE ALTO DESEMPENHO COM USO DE GPU ... 6

2.1.Abordagem em GPU ... 8

2.2.CUDA-NVIDIA ... 11

2.2.1.Arquitetura Fermi ... 13

2.2.2.Hierarquia de memória ... 14

2.2.3.Principais funções em CUDA ... 18

2.3.Parametrização da turbulência ... 20

3 SIMULAÇÃO DA PARAMETRIZAÇÃO DE SMAGORINSKY COM GPU/CUDA ... 25

3.1.Preparação do ambiente de desenvolvimento ... 25

3.2.Implementação da parametrização em CUDA ... 25

3.3.Recursos computacional ... 31

4 RESULTADOS ... 33

4.1.Parametrização ... 33

4.2.Região utilizada nos testes ... 33

4.3.Desempenho em CUDA ... 34

4.3.1.Resolução de 40km ... 35

4.3.2.Resolução de 20 km ... 36

4.4.Análise dos resultados ... 37

5 CONSIDERAÇÕES FINAIS ... 41

5.1.Contribuições e conclusões ... 41

5.2.Trabalho futuros ... 42

REFERÊNCIAS BIBLIOGRÁFICAS ... 43

1 INTRODUÇÃO

Modelos de previsão numérica de tempo têm evoluído constantemente nas últimas décadas, propiciando melhoras significativas na precisão dos resultados. Nesses modelos, é possível identificar pelo menos 4 módulos: a dinâmica (integração numérica das equações de Navier-Stokes), a física (rotinas de simulação de processos físicos: turbulência, transferência radiativa, dinâmica de nuvens e precipitação – em geral a representação desses processos é chamada de *parametrização*), dados geofísicos (topografia, cobertura, umidade e tipo de solo; e modelo de superfície) e assimilação de dados (procedimento executado para identificar a condição inicial, combinando dados do modelo com dados observacionais – ver: KALNAY, 2003).

Devido a alta complexidade dos fenômenos e algoritmos envolvidos nesse tipo de modelagem, essas aplicações requerem o uso de computação intensiva. A computação paralela (vários processadores – CPU: *Central Processing Unit* – interligados) foi um marco importante para o ganho de desempenho de modelos computacionais de intensivo processamento numérico. Atualmente, um novo cenário está se estabelecendo com o emprego de aceleradores, que são dispositivos de *hardware* utilizados para realizar parte do processamento. Entre os dispositivos de *hardware*, ganha cada vez mais destaque a *Graphical Processing Unit* (GPU), por meio da GP-GPU (*General Purpose Graphical Processing Unit*) (LUEBKE, 2006). O processamento combinado de CPU com outros dispositivos de *hardware*, denominados aceleradores caracteriza a *Computação Heterogênea*.

Nesse contexto, elegeu-se para esta pesquisa de Iniciação Científica investigar formas de utilização da recente tecnologia de placas gráficas para melhorar o desempenho computacional do modelo de previsão numérica de tempo BRAMS (*Brazilian Regional Atmospheric Modeling System*). O módulo escolhido foi o módulo de turbulência. No Brasil, o Centro de Previsão de Tempo e Estudos Climáticos (CPTEC) do Instituto Nacional de Pesquisas

Espaciais (INPE) é um dos órgãos responsáveis pela pesquisa, desenvolvimento e operacionalização do modelo de previsão numérica de tempo BRAMS e do modelo CCATT-BRAMS (*Coupled-Chemical Aerosol and Tracer Transport model to the Brazilian Regional Atmospheric System*), que é o modelo de previsão ambiental e inclui química atmosférica.

O modelo BRAMS do CPTEC/INPE é utilizado desde 1994 (Panetta et al. 2007). Esse modelo teve sua origem no RAMS (*Regional Atmospheric Modeling System*), modelo numérico desenvolvido no Departamento de Ciências Atmosféricas da Colorado State University (CSU), nos Estados Unidos da América. O modelo RAMS originou-se de um modelo 3D de simulação de brisa do mar, associado à dinâmica de nuvens e evoluiu para um código computacional sofisticado capaz de realizar simulações meteorológicas de várias escalas espaciais e temporais (PIELKE et al., 1992). Adaptações para a meteorologia tropical e melhorias no código resultaram na versão atual do BRAMS. A nova versão (BRAMS 4.2) apresenta melhor desempenho no processamento serial e um melhor esquema de paralelismo (melhor vetorização do código e comunicação entre o processador mestre e os processadores escravos), além do aprimoramento da portabilidade e qualidade do *software* (BRAMS, 2013).

O modelo CCATT-BRAMS foi criado com o acoplamento do sistema de monitoramento ambiental CCATT e do modelo BRAMS. É um modelo de transporte *on-line*, totalmente consistente com situações dinâmicas atmosféricas, no qual pretende-se mensurar a dinâmica de poluentes quimicamente ativos ou não, gases e/ou material particulado, presentes na atmosfera e disponibilizar previsões sobre a qualidade do ar, estimando o balanço enérgico atmosférico (FREITAS et al., 2007).

É importante destacar que a pesquisa desenvolvida neste trabalho de Iniciação Científica está vinculada ao projeto "Atmosfera Massiva II - Escalabilidade de Modelos Atmosféricos para Arquiteturas Heterogêneas com 10k Cores", que

conta com o apoio do CNPq. Um dos objetivos específicos dessa pesquisa é a capacitação em programação em ambientes de computação heterogênea. Os objetivos gerais e específicos desta pesquisa são apresentados nas sessões seguintes.

1.1. Objetivo

1.1.1. Objetivo geral

Executar testes de análise de desempenho, avaliando o ambiente de programação CUDA-NVIDIA, para rotinas da parametrização de Smagorinsky do modelo BRAMS e CCATT-BRAMS do CPTEC/INPE.

1.1.2. Objetivos Específicos

a) Capacitação na instalação e execução do modelo de previsão BRAMS e CCATT-BRAMS.

b) Capacitação na instalação do ambiente de programação CUDA.

c) Capacitação em programação em ambientes de computação heterogênea: processamento híbrido CPU+GPU, com codificação em CUDA.

1.2. Metodologia

Na codificação para o acelerador GP-GPU (*General Purpose Graphical Processing Unit*) da parametrização do modelo CCATT-BRAMS, há várias estratégias a serem pesquisadas, pois estes dispositivos apresentam várias possibilidades de endereçamento de memória e distribuição de tarefas. Para isso foi necessário um extenso estudo sobre a computação de alto desempenho e revisão bibliográfica dos principais conceitos da programação em CUDA e de sua arquitetura. Uma análise detalhada da rotina original (codificada em Fortran) foi realizada, visando identificar as regiões a serem

paralelizadas e os parâmetros de entrada/saída. De posse desse conhecimento partiu-se para a codificação da rotina em CUDA.

Por fim, a análise de desempenho foi realizada por meio do cálculo de *speed-up*, que é definido como sendo a razão entre o tempo gasto com a implementação serial e a implementação paralela (no caso, em GPU). O ambiente de programação utilizado foi o CUDA-NVIDIA.

Este relatório de iniciação científica está estruturado em 5 capítulos:

- No Capítulo 2 é apresentado a revisão bibliográfica, englobando uma breve descrição dos conceitos da computação de alto desempenho, do modelo CUDA de programação paralela e da parametrização de Smagorinsky, alvo de nosso estudo.

- O Capítulo 3 apresenta o desenvolvimento do trabalho, descrevendo os principais passos e procedimentos envolvidos na implementação, bem como, os recursos computacionais utilizados.

- Os resultados obtidos mediante a análise de medidas de desempenho via *speed-up* são discutidos no Capítulo 4.

- Finalmente, no Capítulo 5 são apresentados as conclusões e sugestões de trabalhos futuros.

2 COMPUTAÇÃO DE ALTO DESEMPENHO COM USO DE GPU

Há uma demanda crescente de capacidade cada vez maior de processamento das arquiteturas de *hardware* e *software* vigentes, o que estimula o contínuo desenvolvimento da computação. *Hardware* e sistemas de *software* foram aprimorados com novas arquiteturas e técnicas de computação paralela, heterogênea e híbrida, afim de impulsionar o processamentos das aplicações industriais e cientificas. Contudo esses sistemas computacionais continuam em expansão, como observado na Figura 2.1, que ilustra o crescimento do desempenho computacional nas últimas décadas e uma projeção até 2020 em relação às operações em ponto flutuante.

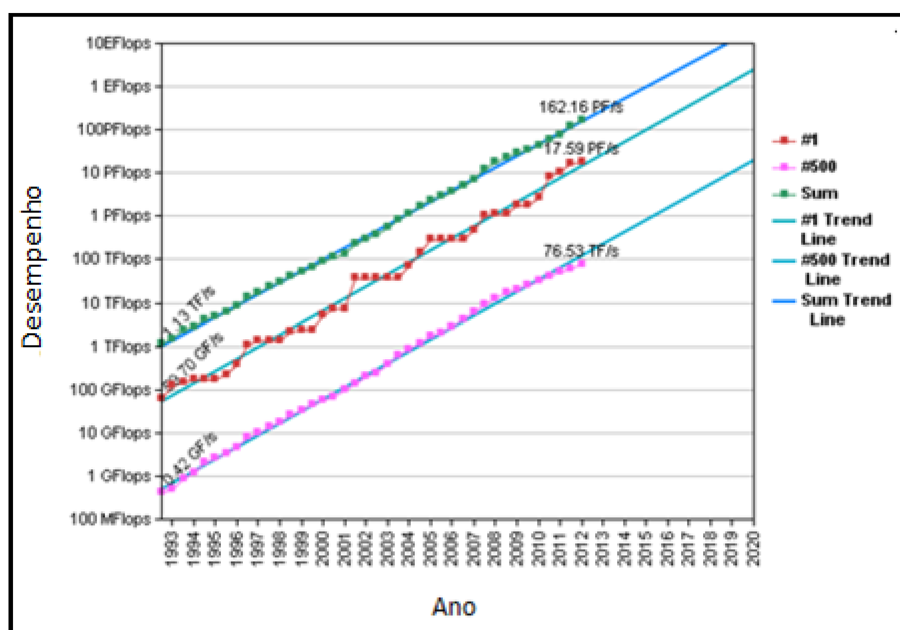

Figura 2.1 - Crescimento exponencial do poder computacional
(Disponível em: www.top500.org/statistics/perfdevel/ – acessado em: 04 jun. 2013.)

Durante cerca de 30 anos, um dos métodos mais importante para melhorar o desempenho dos dispositivos computacionais tem sido o aumento da velocidade processamento (frequência ou *clock*). Nos computadores pessoais, no decorrer da década de 80, o *clock* de processamento era em torno de 1 MHz. Atualmente, na maioria dos processadores as velocidades estão entre 1

GHz e 4 GHz, tendo uma grande diferença em comparação aos primeiros *clocks*. Sanders e Kandrot (2010) destacam que, como nos supercomputadores, o aumento de desempenho dos computadores pessoais estava focado no aumento da quantidade de processadores e da velocidade de processamento. Porém, o aumento da frequência de chamadas ao processador causa aquecimento progressivo. Deste modo, outras alternativas foram avaliadas para aumentar o poder de processamento, de modo a evitar o problema do superaquecimento. No ano de 2005 surgiram os primeiros computadores pessoais com a tecnologia de processadores de dois núcleos: *Dual Core*. Na sequência, vieram processadores com mais núcleos como os *quad-core*, *six-core*, *eight-core* e hoje há processadores com até 16-cores.

A computação de alto desempenho (HPC: *High Performance Computing*) utiliza o processamento paralelo em sistemas como supercomputadores ou *clusters* para executar algoritmos ou modelos numéricos que requerem alto poder computacional. Basicamente, o processamento paralelo pode ser entendido como o uso de vários processadores para resolver um dado problema, com o objetivo de obter um melhor desempenho, reduzindo o tempo total de execução (EL-REWINI E ABD-EL-BARR, 2005).

Um conceito importante relacionado a computação de alto desempenho é o conceito de classificação da arquitetura dos computadores paralelos. Nesse contexto, a classificação mais popular e uma das primeiras, é a taxonomia de Flynn, desenvolvida em 1966.

Basicamente, existem dois tipo de fluxos dentro do processador: o fluxo de instruções e o fluxo de dados, logo essa taxonomia considera o número de instruções executadas em paralelo em relação ao fluxo de dados para os quais essas instruções são submetidas (FLYNN, 1996). Um bom desempenho no fluxo de instruções depende da capacidade de processamento, enquanto que no fluxo de dados a comunicação entre a memória e o processador tem que

ser rápida. Segundo Flynn (1966), a arquitetura de computadores pode ser classificada em quatro diferentes modelos:

- **Single Instruction Single Data (SISD):** Caracteriza-se como um computador que executa um único fluxo de instrução sobre um único fluxo de dados. Pode ser comparado as CPU sequenciais que seguem o modelo de Von Neumann, onde um único fluxo de instruções passa pela unidade de controle e depois pelo processador, e um único fluxo de dados ocorre entre o processador e a memória.

- **Instruction Multiple Data (SIMD):** Consiste em um modelo de computação paralela que também está fundamentado no modelo de Von Neumann, porém o mesmo fluxo de instruções é executado em diferentes processadores e sobre diferentes fluxos de dados.

- **Multiple Instruction Single Data (MISD):** Consiste em diversas unidades de controle (*Control Unit*) onde passam os fluxos com múltiplas instruções a serem processadas por diferentes processadores de forma simultânea, porém sobre o mesmo fluxo de dados.

- **Multiple Instruction Multiple Data (MIMD):** Presente em máquinas paralelas, que executam fluxos independentes e separados de instruções sobre diferentes fluxos de dados. Essa arquitetura pode ser usada tanto para computadores de memória compartilhada, quanto para memória distribuída.

2.1. Abordagem em GPU

A grande demanda por melhores definições gráficas 3D e em tempo real (*Realtime*) das placas de processamento gráfico (GPUs) aumentou significativamente o desempenho na paralelização, no uso de múltiplos *threads*

e no número de *cores*, levando ao desenvolvimento de placas extremamente potentes. Asanovic et Al. (2006) ressalta que, com o desenvolvimento dos setores de jogos, as placas gráficas têm experimentado um desenvolvimento extraordinário, podendo executar mais de 1 trilhão de operações em ponto flutuante por segundo (Teraflops). As Figuras 2.2 e 2.3 mostram esse crescimento, comparando a evolução histórica das CPUs em relação as GPUs, em termo de Teraflops e da memória de largura de banda (*Memory Bandwidth*).

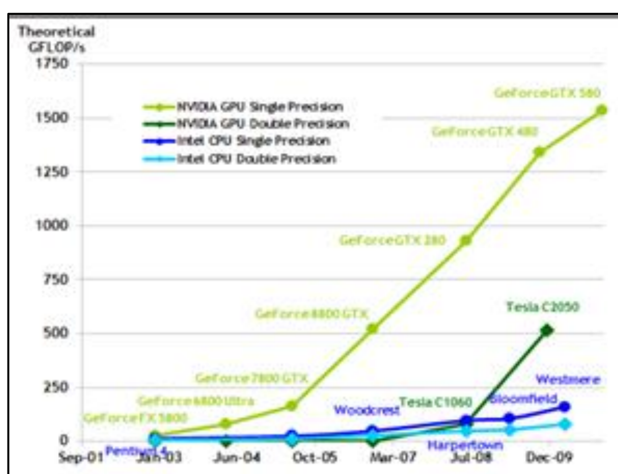

Figura 2.2 - Operações em ponto flutuante por segundo em GPU (NVIDIA) e CPU (INTEL). Fonte: NVIDIA CUDA (2011)

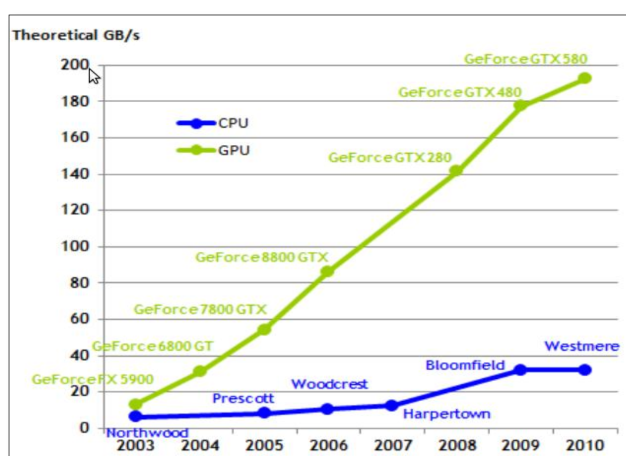

Figura 2.3 - Largura de banda memória (memory bandwidth) CPU e GPU Fonte: NVIDIA CUDA (2011)

Por meio das figuras, pode-se observar que, em relação às operações em ponto flutuante, as GPUs têm alcançado um desempenho muito superior ao desempenho das CPUs. Isso se deve ao fato de que as GPUs são projetadas para renderização de imagens, sendo o alto paralelismo e a computação intensiva suas especialidades.

Recentemente, tendo ainda como objetivo o aumento da velocidade de processamento, foi feita a combinação das CPUs com outros dispositivos de *hardware*, denominados aceleradores, entre os quais destaca-se a combinação CPU + GPUs. Descobriu-se a possibilidade do uso das mesmas em aplicações da computação científica. Dentro desse contexto surgiu a GPU de propósito geral - GPGPU (*General Purpose Computing on Graphics Processing Units*) (LUEBKE, 2006).

Na Figura 2.4, pode-se observar que uma diferença fundamental entre a GPU e a CPU: a GPU possui muito mais transistores dedicados ao processamento de dados do que memória *cache* e registradores. Hoje, as CPUs apresentam vários (multi) cores, com grande quantidade de memória *cache* e registradores, porém com baixo número de *cores*, enquanto as GPUs apresentam muitos (*many*) *cores*, mas com pouco valor de memória *cache* e registradores. Assim, a tecnologia das Placas Gráficas são bastante úteis para sistemas que necessitem de grande número de cálculos matemáticos e alto grau de paralelismo, problemas normalmente encontrados em aplicações científicas (NVIDIA CUDA, 2011).

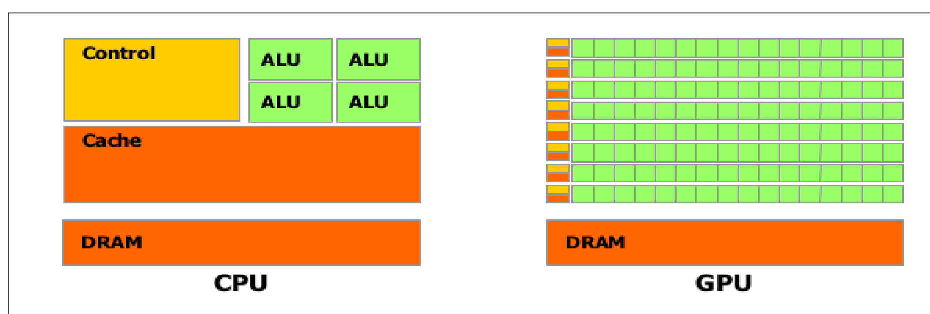

Figura 2.4 - Arquitetura CPU x GPU.
Fonte: NVIDIA CUDA (2011)

2.2. CUDA-NVIDIA

A NVIDIA desenvolveu um modelo de programação baseado em sua arquitetura, o modelo CUDA, que também facilita a programação em GPU. CUDA é uma plataforma de computação paralela e um modelo de programação que habilita as GPUs NVIDIA para executar programas escritos em linguagem de programação C, C++, Fortran, entre outros. CUDA permite aumentos significativos de desempenho computacional ao aproveitar a potência da unidade de processamento gráfico. Em 2006, a placa de vídeo GeForce 8800 GTX foi a primeira habilitada para programação em CUDA, contendo diversos componentes para melhorar o processamento gráfico e computacional (SANDERS E KANDROT, 2010).

O objetivo dessa interface de programação é criar aplicações de propósito geral em GPUs, utilizando a computação híbrida ou heterogênea: GPUs + CPUs. Na computação híbrida, a GPU atua como um co-processador, executando a parte da computação.

Um programa desenvolvido em CUDA trabalha com rotinas ou *kernels* paralelos, em que cada *kernel* executa em paralelo através de um conjunto de *threads*. Um conjunto de *threads* é denominado blocos de *threads*, que por sua vez são organizados em uma *grid* de blocos. A GPU instancia um *kernel* sobre

uma *grid* de blocos de *threads* paralelos. Cada *thread* dentro de um bloco, possui um identificador (*thread* ID), registradores, memória privada por *thread* e dados de entrada e saída. Um bloco de *threads* é um conjunto de *threads* executando concorrentemente e que podem cooperar entre si por meio de barreiras de sincronização e memória compartilhada. O bloco também possui um identificador (bloco ID) dentro de sua *grid*. Uma *grid* é um conjunto de blocos de *threads* que executam o mesmo *kernel*, lê a entrada a partir da memória global, escreve os resultados na memória global e sincroniza entre as chamadas dependentes do *kernel* (CUDA FERMI, 2009). Na Figura 2.5, tem-se um esquema dessa metodologia de paralelização.

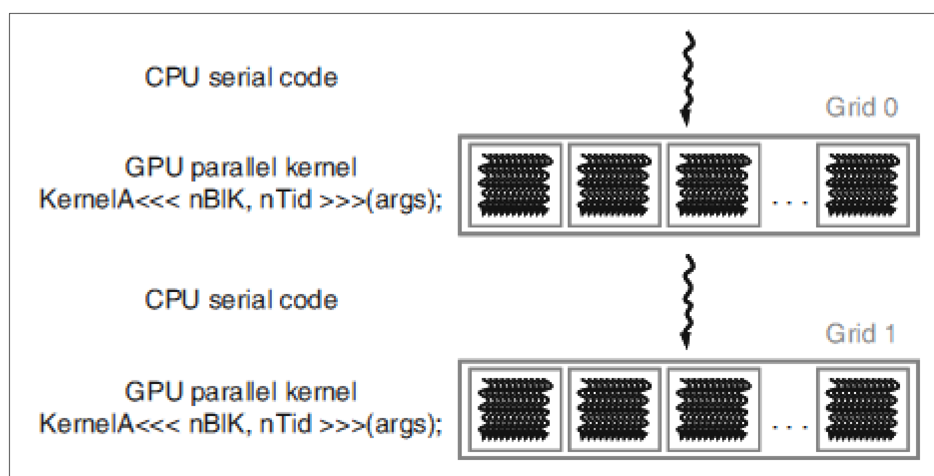

Figura 2.5 - Execução do programa em CUDA.
Fonte: KIRK e HWU (2010)

Na arquitetura CUDA, quando um programa na CPU inicia um *kernel*, os blocos dentro da *grid* são enumerados e distribuídos para os diversos multiprocessadores. A execução é feita de forma concorrente, sendo que a cada término de execução, novos blocos são atribuídos ao multiprocessador que é liberado. Tudo isso é possível, pois, a arquitetura é construída com o uso de vetores escalonados de multiprocessadores ou *Streaming multiprocessors* (SM). Esses multiprocessadores tem a capacidade de executar centenas de *threads* de forma concorrente, devido a necessidade de gerenciar essa grande

quantidade de *threads*, a NVIDIA desenvolveu a arquitetura SIMT (*Single-Instruction, Multiple-Thread*). Nesse modelo, múltiplos threads podem executar uma única instrução aumentando o nível do paralelismo.

2.2.1. Arquitetura Fermi

A arquitetura desenvolvida pela NVIDIA tem evoluído constantemente, principalmente com os lançamentos das arquiteturas Fermi (NVIDIA FERMI, 2009) e Kepler (NVIDIA KEPLER, 2012).

A seguir, serão apresentadas as principais características da arquitetura Fermi, como ilustrado na Figura 2.6. Essa arquitetura foi utilizada neste trabalho:

- Possui 16 Streaming *Multiprocessors* (SMs) com 32 *cores* cada um, tendo um total de 512 CUDA *cores*;

- Cada SM contém uma partição laranja (escalonador e distribuidor), uma partição verde (unidades de execução) e uma partição azul claro (registradores e *cache* L1);

- Possui 6 posições de memória de 64 Bits;

- Tem capacidade de até 6 GigaByte (GB) de *Graphic Double Data Rate version* 5 (GDDR5), que é um tipo de memória *Dynamic random-access memory* (DRAM) para placas gráficas de alto desempenho para aplicações de computadores que exigem alta largura de banda;

- Possui dois níveis de memória *cache* (L1 e L2), sendo que a memória *cache* L1 pode ser combinada com a memória compartilhada da seguinte forma: 48 KB *cache* L1 e 16 KB memória compartilhada ou 16 KB *cache* L1 e 48 KB memória compartilhada;

- Possui *cache* L2 com 768 KB compartilhada entre todos os *threads*;

- Possuiu o escalonador *GigaThread engine* que gerencia centenas de *threads* e permite a execução de *Kernel* concorrentes.

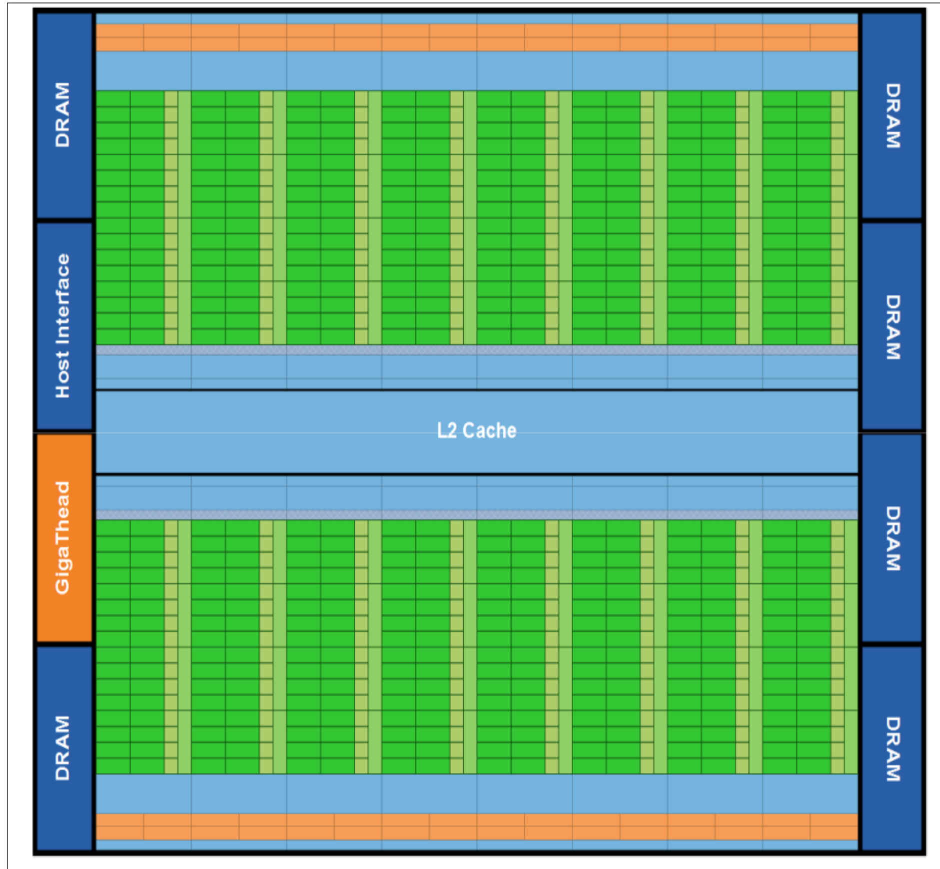

Figura 2.6 - Arquitetura Fermi.
Fonte: NVIDIA Fermi (2008)

2.2.2. Hierarquia de memória

Na programação híbrida, a troca de informação entre a CPU e a GPU é feita através de um barramento PCI-express, que é o responsável pela transferência de dados da memória do *host* (CPU) para a memória do dispositivo (GPU), e vice-versa. Isso ocorre porque não há compartilhamento de memória: a GPU possui sua própria memória RAM. Logo, para executar um *kernel* na GPU é necessário alocar memória na GPU e transferir esses dados da CPU para a memória alocada. A Figura 2.7 ilustra esse processo. No primeiro passo, os

dados necessários à resolução do problema são transferidos da memória da CPU para a memória da GPU. Em seguida, o problema é dividido em tarefas, que são distribuídas em diversos núcleos de processamento da GPU e os cálculos são efetuados. Após a finalização dos cálculos, os resultados são transferidos para a CPU e o programa continua.

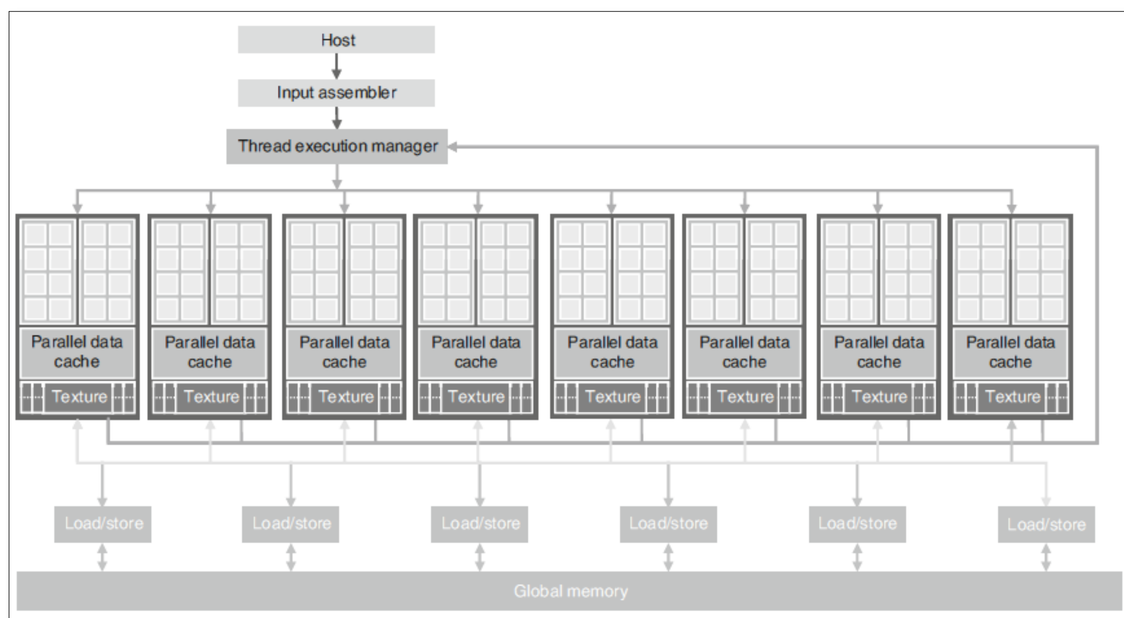

Figura 2.7 - Ilustração do esquema da transferência de dados entre CPU e GPU.
Fonte: KIRK e HWU(2010)

Nas placas gráficas, existem diferentes endereçamentos de memória, sendo eles: os registradores, a memória local, a memória compartilhada, a memória global, a memória constante e a memória de textura. Cada uma tem capacidades e formas de uso diferentes e estão distribuídas em diferentes níveis dentro da placa gráfica, assim como visto na Figura 2.8. A Tabela 1 apresenta um resumo das permissões de leitura e escrita disponíveis nos diferentes tipos de memória (NVIDIA CUDA, 2011).

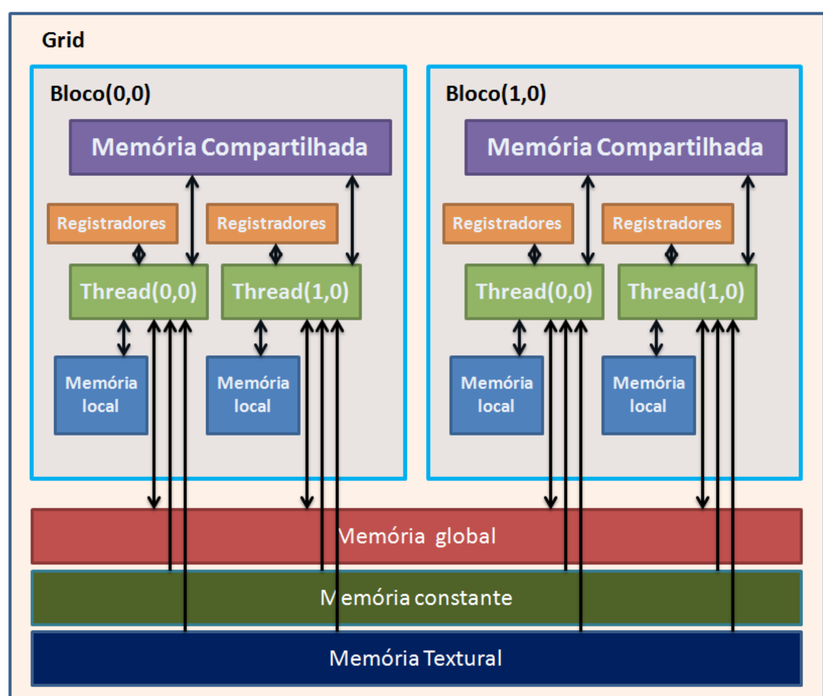

Figura 2.8 - Esquema hierárquico de memória da GPU.

Fonte: adaptado NVIDIA CUDA (2011)

Tabela 1 - Níveis e funções de cada memória da GPU.

Memória	Escopo	Função
Registradores	Thread	Leitura-Escrita
Memória Local	Thread	Leitura-Escrita
Memória Compartilhada	Bloco	Leitura-Escrita
Memória Global	Grade	Leitura-Escrita
Memória Constante	Grade	Leitura
Memória Textura	Grade	Leitura

Fonte: CUDA NVIDIA (2011)

Neste contexto, cada tipo de memória tem diferentes velocidades e capacidades de armazenamento, por exemplo, os registradores ficam mais próximos das *threads* e dessa forma as trocas de informações são muito mais rápidas do que com a memória local. Contudo, a memória local tem uma capacidade de armazenamento maior que a dos registradores (NVIDIA CUDA, 2011).

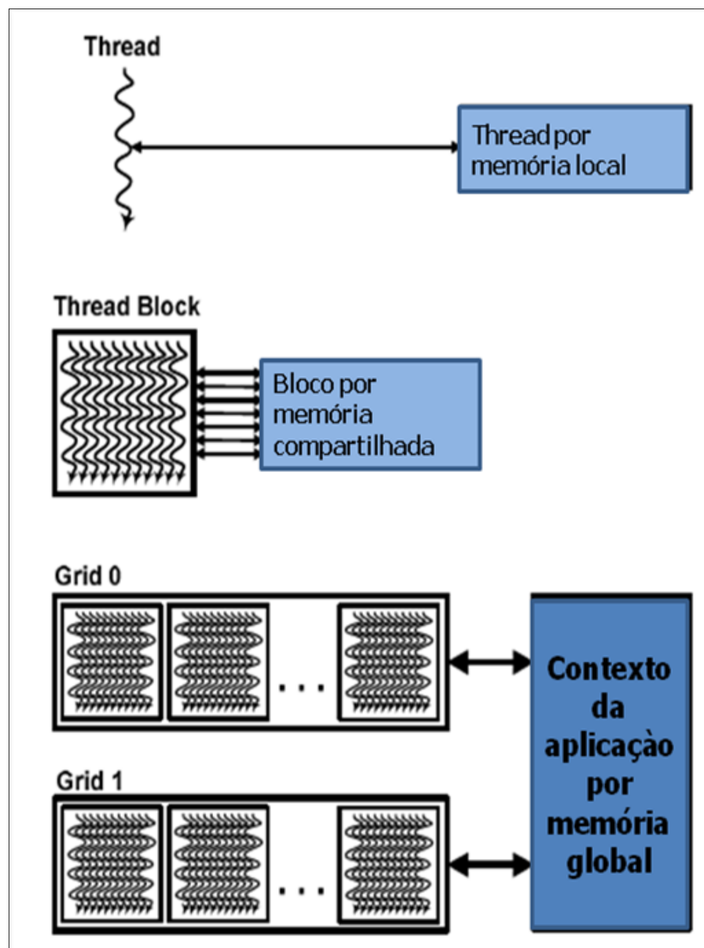

Figura 2.9 - Hierarquia de *threads*, blocos e *threads.*

Fonte: Adaptado de NVIDIA Fermi (2008)

Um esquema do funcionamento hierárquico da memória é mostrado na Figura 2.9: o *thread* tem os seus dados de execução armazenados em memória local que são visíveis apenas para aquele *thread*. Os blocos têm seus dados armazenados em sua própria memória compartilhada, tornando visível a informação para todos os *threads* presentes naquele bloco. As *grids*, por sua vez, armazenam seus dados na memória global, permitindo o acesso para todos os *threads* (NVIDIA CUDA, 2011).

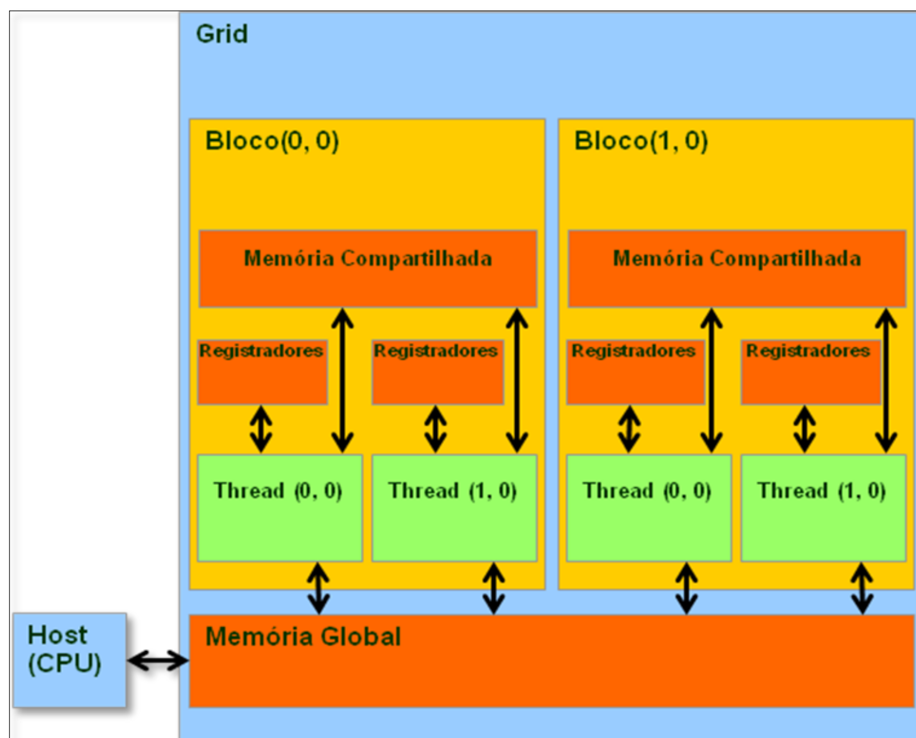

Figura 2.10 - Relacionamento em nível de memória entre GPU e CPU.
Fonte: Adaptado de NVIDIA Fermi (2008)

2.2.3. Principais funções em CUDA

A biblioteca CUDA apresenta uma função chamada *kernel*, cuja palavra chave é __global__. Sua utilização permite especificar a parte do código que será executado pelos *threads* durante a fase paralela (KIRK E HWU, 2010).

Na linguagem CUDA, cada *thread* apresenta um identificador único, o *thread* ID, que pode ser acessado no *kernel* através do uso da estrutura threadIdx. Para fazer a chamada à função __global__ e definir o número de *threads* é necessário o uso de uma nova sintaxe <<<...>>>, usada na chamada da função *kernel*, como no exemplo na figura 2.11.

```
1
2   (...)
3  void main(){
4
5   (...)
6
7   int num_thread_por_blocos;
8   int num_blocos_por_grid;
9
10  f_kernel<<<thread_por_blocos,num_blocos_por_grid>>>(param_1, param_2, ... )
11
12  (...)
```

Figura 2.11 – Sintaxe chamada *Kernel*.

Os blocos de *threads*, por sua vez, são identificados dentro da *grid* por meio de um identificador blockId que é definido pela estrutura blockIdx. Na arquitetura Fermi o número de *threads* por bloco é limitado a 1024 e o número de *threads* por multiprocessador é limitado a 1536. Porém, um *kernel* pode ser executado por vários blocos de *threads*, de modo que o número de *threads* total é igual ao número de *threads* por bloco, multiplicado pelo número de blocos.

As estruturas definidas para especificar a dimensão das *grids*, blocos e dos identificadores de blocos e *threads* são (NVIDIA CUDA, 2011):

- **gridDim**: fornece a dimensão da grid.
- **blockIdx**: fornece o índice do bloco dentro da *grid*.
- **blockDim**: fornece a dimensão do bloco.
- **threadIdx**: fornece o índice de *threads* dentro do bloco.
- **warpSize**: fornece o número de *threads* no *warp*.

Em CUDA, tem-se qualificadores de função, ou seja, palavras chaves que determinam onde será executada determinado processo, CPU ou GPU, sendo eles:

- **__device__**: especifica que a função será executada na GPU e somente poderá ser invocada a partir da mesma.
- **__global__**: especifica um *kernel*, que será executado no *device* e somente poderá ser invocado a partir da CPU. Obrigatoriamente retorna *void* e deve ter uma configuração de execução (números de *threads* e blocos) quando for chamada.
- **__host__**: especifica uma função que somente será executada e invocada a partir do host ou CPU.

Por padrão, quando uma função não tiver nenhum especificador ela será considerada com o especificador __global__.

Além dos qualificadores de função descritos acima, existem também os qualificadores de variáveis que determinam em que tipo de memória da GPU a variável será alocada são eles:

- **__device__**: especifica uma variável que reside na GPU, acessível a todos os threads e possui o mesmo tempo de vida da aplicação.
- **__constant__**: especifica uma variável que reside na memória constante e possui o mesmo tempo de vida da aplicação. É acessível por todos os threads de uma *grid* e pelo *host*, através da biblioteca *runtime*.
- **__shared__**: especifica uma variável que reside na memória de um bloco tendo o mesmo tempo de vida e somente acessível pelos *threads* que o compõem.

2.3. Parametrização da turbulência

O estudo sistemático da turbulência começou com Osborn Reynolds no século XIX. Foi ele quem introduziu em 1883 o parâmetro adimensional conhecido

como número de Reynolds, que permite estimar a ocorrência da turbulência. Em 1895, introduziu a chamada Decomposição de Reynolds, em que o escoamento turbulento é dividido em um fluxo médio (similar ao escoamento laminar) e uma flutuação aleatória (REYNOLDS, 1895). A incorporação dessa abordagem nas equações da dinâmica da atmosfera resulta em um sistema de equações que contém novas variáveis, conhecidas como tensores de Reynolds, os quais representam a turbulência. Porém, esse sistema de equações (também denominado equações turbulentas) constitui um sistema aberto, ou seja, possui um número de incógnitas maior do que o número de equações, problema conhecido na literatura como problema de fechamento da turbulência.

Para solucionar o problema de fechamento da turbulência, seria necessário um conjunto infinito de equações. A alternativa é representar os fluxos de Reynolds por grandezas já conhecidas. O procedimento é chamado de parametrização. para obter soluções aproximadas. Para a dinâmica da atmosfera, algumas parametrizações desenvolvidas são: parametrização de Smagorinsky (1963) – baseada na discretização espacial; parametrização de Mellor e Yamada (1974), que utiliza um prognóstico da energia cinética turbulenta; e parametrização de Taylor (Degrazia, et al., 2000) – baseada numa descrição estatística da turbulência.

O presente estudo irá focar na parametrização de Smagorinsky.

Na formulação de Smagorinsky, os fluxos turbulentos são parametrizados utilizando a teoria do fluxo-gradiente conhecida como Teoria K. Nessa parametrização os fluxos turbulentos, ou tensor de Reynolds, de quantidade de movimento são expressos por:

$$\overline{u_i' u_j'} = -K_{m_{ij}} (\overleftrightarrow{D})_j, \tag{2.3.1}$$

onde $K_{m_{ij}}$ é o coeficiente de difusividade turbulenta para o momento i na direção j e

$$(D)_j = \frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i} \tag{2.3.2}$$

Considerando as modificações propostas por Lilly (LILLY, 1962) e por Hill (HILL, 1974), o coeficiente de difusividade turbulenta na direção vertical pode ser parametrizado da seguinte maneira:

$$K_{mv} = (cs_z \Delta z)^2 [|D_v| + H(N)] f(R_i) \tag{2.3.3}$$

onde cs_z é um coeficiente de ajuste pré-calibrado, Δz é o espaçamento de grade computational na direção vertical e o termo $|D_v|$, referente a magnitude do tensor deformação na direção vertical, é dado por:

$$|D_v| = \left[\left(\frac{\partial \bar{u}}{\partial z} \right)^2 + \left(\frac{\partial \bar{v}}{\partial z} \right)^2 \right]^{\frac{1}{2}} \tag{2.3.4}$$

A contribuição da convecção na produção de turbulência $H(N)$ é dada por:

$$H(N) = \sqrt{max[0, -N^2]}, \tag{2.3.5}$$

onde o termo $f(Ri)$ é dado por:

$$f(Ri) = \sqrt{max\left[0,1 - \frac{K_{hv}}{K_{mv}} Ri\right]}, \tag{2.3.6}$$

aqui, a razão $\frac{K_{hv}}{K_{mv}}$ entre o coeficiente de difusividade de calor e momento é especificado pelo usuário e Ri é o número Richardson gradiente. Na Figura 2.12, tem-se um trecho do código em Fortran da rotina *mxdefm()* que implementa essa parametrização. Na implementação em GPU, o laço aninhado que realiza o cálculo nas direções x, y, e z será eliminado e o cálculo será feito de forma concorrente pelos blocos de threads. Na figura 2.13, mostra um trecho da implementação em GPU.

```
1  .....
2
3   elseif (idiffk(ngrid) .eq. 2) then
4      do j = ja,jz
5         do i = ia,iz
6            c1 = rtgt(i,j) * rtgt(i,j)
7            c2 = 1.0 / (dxt(i,j) * dxt(i,j))
8            c3 = csx2 * c2
9            akm = akmin(ngrid) * 0.075 * c2 ** (0.666667)
10           c4 = vonk * vonk * c1
11           do k = lpw(i,j),m1-1
12              ! old csz*dz len  scr1(k,i,j) = dn0(k,i,j) * c1 * vctr2(k)
13
14              ! asymptotic vertical scale length from bjorn with modifications:
15              ! c3 is (csx * dx)^2, c1*vctr2(k) is (csz * dz)^2, sq300 is the square
16              ! of 300 meters (used as a limit for horizontal grid spacing influence
17              ! on vertical scale length), ambda is (asymptotic_vertical_length_scale)^2,
18              ! and vkz2 is (vonk * height_above_surface)^2.
19
20              ambda = max(c1 * vctr2(k),min(sq300,c3))
21              vkz2 = c4 * zt(k) * zt(k)
22              scr1(k,i,j) = dn0(k,i,j) * vkz2 / (vkz2 / ambda + 1)  &
23
24              * (sqrt(vt3di(k,i,j))  &
25                   + enfl * sqrt(max(0.,-vt3dj(k,i,j))))*min(rchmax  &
26                   ,sqrt(max(0.,(1.-zkhkm(ngrid)*vt3dk(k,i,j)))))
27
28              scr2(k,i,j) = dn0(k,i,j)  &
29                   * max(akm,c3*sqrt(vt3dh(k,i,j)))
30              vt3dh(k,i,j) = scr1(k,i,j) * zkhkm(ngrid)
31
32           enddo
33        enddo
34     enddo
35     !cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

Figura 2.12 - Trecho de código com a parametrização de Smagorinsky.

```
1 ...
2 __global__ void turb2(int M, float *d_zm, float *vt3dj, float *vt3di, float *vt3dh, float *vt3dk,
3   float *rtgt,float vonk, float akmin, float csx, float *dxt, float *vctr2, float sq300, float *zt,
4   float *scr1,float *scr2, float *dn0, float enfl, float rchmax)
5   {
6   float tmp1,tmp2,tmp3,tmp4,c1,c2,c3,c4,vkz2;
7   float rmin = -100;
8   float rmax,csx2,akm,ambda;
9   float zkhkm = 3.0;
10  rmax = 1.0/zkhkm;
11  csx2 = csx * csx;
12  int tid = threadIdx.x + blockDim.x * blockIdx.x ;
13  int k = 30,cl;
14
15  if (tid < M)
16    {
17    tmp1 = fmaxf(vt3di[tid],1.0e-15);
18    tmp2 = vt3dj[tid]/tmp1;
19    tmp3 = fminf(tmp2,rmax);
20    vt3dk[tid] = fmaxf(tmp3,rmin);
21    c1 = rtgt[tid]*rtgt[tid];
22    c2 = 1.0/(dxt[tid]*dxt[tid]);
23    c3 = csx2*c2;
24    akm = akmin*0.075 * powf(c2,0.666667);
25    c4 = vonk*vonk*c1;
26    cl = int(tid%k);
27    tmp4 = fminf(sq300,c3);
28    ambda = fmaxf(c1*vctr2[cl],tmp4);
29    vkz2 = c4 * zt[cl] * zt[cl];
30
31    scr1[tid] = dn0[tid] * vkz2 / (vkz2 / ambda + 1) * (sqrtf(vt3di[tid]) + enfl *
32    sqrtf(fmaxf(0.0,-vt3dj[tid])))*fminf(rchmax,sqrtf(fmaxf(0.0,(1.0-zkhkm*vt3dk[tid]))));
33    scr2[tid] = dn0[tid]*fmaxf(akm,c3*sqrtf(vt3dh[tid]));
34    vt3dh[tid] = scr1[tid]*zkhkm;
35    }
36  }
37 ...
```

Figura 2.13 - Trecho de código com a parametrização de Smagorinsky em GPU.

3 SIMULAÇÃO DA PARAMETRIZAÇÃO DE SMAGORINSKY COM GPU/CUDA

Neste capítulo, será abordado como foi realizada a implementação da parametrização de Smagorinsky em CUDA, descrevendo de forma resumida, os passos usados na instalação e preparação do ambiente de desenvolvimento e o esquema geral do programa desenvolvido em C/CUDA.

3.1. Preparação do ambiente de desenvolvimento

Os principais passos envolvidos no preparo do ambiente de trabalho são listados a seguir:

- Familiarização com o processo de instalação, configuração e execução do modelo de previsão numérica BRAMS. Foi utilizada a versão 4.2, disponível para download em http://brams.cptec.inpe.br. Para a instalação e compilação do modelo é necessário os compiladores C e Fortran (intel: icc, ifortran ou GNU: g95 e gcc), biblioteca MPI (*Messege Passing Interface*) e biblioteca HDF (*Hierarchical Data Format*).

- Configuração do arquivo RAMSIN que contém os parâmetros de entrada usados em uma determinada simulação.

- Preparação do ambiente CUDA, onde são necessários os seguintes requisitos: Possuir uma placa GPU com suporte a tecnologia CUDA, sistema operacional com o compilador gcc/g++ e o pacote CUDA *Toolkit* disponível em https://developer.nvidia.com/cuda-downloads.

3.2. Implementação da parametrização em CUDA

Com o ambiente preparado, o próximo passo foi analisar a rotina de parametrização de Smagorinsky para identificar trechos a serem paralelizados em GPU. Vale lembrar que um desafio em computação híbrida é a

transferência de dados entre os dispositivos: uma vez que a GPU possui sua própria memória, os dados primeiramente devem ser enviados da CPU para a GPU por meio de um barramento PCI-Express e em sequência se realiza o processamento. Após essa análise, optou-se por implementar de forma isolada toda a rotina de Smagorinsky em CUDA. O primeiro passo foi identificar as variáveis de entrada e salvá-las em um arquivo. Esse arquivo será fornecido como entrada para a nova rotina em GPU.

Na implementação da rotina, inicialmente é feito a alocação dinâmica de memória na CPU e a leitura dos dados. A leitura é feita por meio do comando fopen(), associado a uma variável do tipo FILE, que é utilizada pela linguagem C para leitura e escrita de arquivos. Para sua utilização, necessita-se a importação da biblioteca <stdio.h>. Após a leitura do arquivo, é realizada a escrita dos dados nos vetores previamente alocados na memória da CPU. Finalizada a leitura, fecham-se os arquivos usando o comando fclose(). Esse processo está ilustrado na Figura 3.1.

```
1   #include<stdio.h>
2
3   (...)
4
5   //DECLARANDO OS ARQUIVOS DE ENTRADA
6   FILE *fp1;
7
8   //LENDO ARQUIVO DE ENTRADA
9   fp1 = fopen("dim1.dat","r");
10
11  //ADICIONANDO VALORES LIDOS EM VETORES
12   for (k = 0 ; k < M ; k++){
13       fscanf (fp1, "%f %f",
14               &h_zm[k],&h_zt[k]);
15      }
16
17  //FECHANDO ARQUIVO
18  fclose(fp1);
```

Figura 3.1 - Lendo arquivos de entrada.

Para a transferência de dados, existem 2 tipos de memórias possíveis, a memória não pinada e a memória pinada. Essa última possibilita a transferência de dados de forma mais rápida, por meio da cópia assíncrona. Seu uso é obtido por meio das funções cudaHostAlloc() e cudaMallocHost() e cudaFreeHost(). No caso de cópia não assíncrona, utiliza-se a função cudaMalloc(), que é bastante similar a função malloc() em C. A função cudaMalloc() possui dois parâmetros: um ponteiro para a memória alocada no dispositivo e o tamanho da alocação requerida, em *bytes*. Após terminar os cálculos, libera-se o espaço alocado utilizando a função cudaFree(), conforme a Figura 3.2.

```
1   #include "cuda.h"
2
3   (...)
4
5   //DECLARANDO VARIAVEIS
6
7   float *dev_vt3dh,*dev_scr1;
8
9   //ALOCANDO MEMÓRIA NA GPU
10
11  cudaMalloc((void**)&dev_vt3dh,sizeof(float)*M);
12  cudaMalloc((void**)&dev_scr1,sizeof(float)*M);
13
14  (...)
15
16  // LIBERANDO MEMÓRIA
17  cudaFree(dev_vt3dh);
18  cudaFree(dev_scr1);
```

Figura 3.2 - Sintaxe cudaMalloc().

Após a alocação de memória na GPU é realizada a transferência dos parâmetros (dados) que serão utilizados no cálculo. Portanto, para transferir estes dados para o *device* usa-se função cudaMemcpyAsync(). Essa função é capaz de enviar dados de forma assíncrona, ou seja, não é necessário que todos os *threads* finalizem suas tarefas para que outros threads, já finalizados,

continuem processando os dados. Para o uso deste método, são recebidos como parâmetro: um ponteiro para memória alocada no dispositivo, tamanho da posição requerida (em *byte*) e uma estrutura opaca do tipo cudaStream_t presente na biblioteca <"cuda_runtime_api.h">. Esse processo é ilustrado na figura 3.3.

```
1   #include "cuda.h"
2
3   (...)
4
5   //DECLARANDO VARIAVEIS
6
7   float *dev_vt3dh,*dev_scr1;
8
9   //ALOCANDO MEMÓRIA NA GPU
10
11  cudaMalloc((void**)&dev_vt3dh,sizeof(float)*M);
12  cudaMalloc((void**)&dev_scr1,sizeof(float)*M);
13
14  (...)
15
16  // LIBERANDO MEMÓRIA
17  cudaFree(dev_vt3dh);
18  cudaFree(dev_scr1);
```

Figura 3.3 - Transferência Assíncrona dos dados para GPU.

Ao fim das alocações e transferências dos dados, a GPU está pronta para executar a função *kernel*. Esta função é responsável pelo escalonamento do número de blocos por *grid* e do número de *threads* por bloco. Contudo, para construir essa organização dos threads é necessário uma estratégia, que é dependente da arquitetura de *hardware* disponível.

Na tabela 2, é mostrado as capacidades de computacionais relacionadas a arquitetura Fermi. Pode-se observar que cada multiprocessador têm um limite máximo de 1536 *threads*, podem ter até 8 blocos de *threads*, cada bloco com no máximo 1024 *threads*.

Tabela 2 - Capacidade de computação da arquitetura Fermi.

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K

Fonte: NVIDIA GTX 680.

A Figura 3.4 apresenta a codificação de uma parte paralela da rotina de turbulência. Na linha 4, usa-se a palavra chave "__global__" que indica que essa função é um kernel e pode ser chamada da CPU para gerar a grade de *threads* que será executada na GPU.

É valida a observação que todos os *threads* executam o mesmo código, dessa forma é necessário utilizar identificadores que permitam a esses *threads* atuarem nas partes que lhe forem designadas. As palavras chaves, visualizadas na linha 7 da Figura 3.4, plermitem que um *thread* acesse os registradores do *hardware* em tempo real (*runtime*), fornecendo as coordenadas de identificação para *threads* específicos e relacionando-os com a grade e o bloco de *threads*.

```
#include "cuda.h"
(...)
// FUNCÃO KERNEL
__global__ void turb1 (int M,float csx,float akmin,
float *dxt,float *dn0, float *vt3dh,float *scr2){

  int tid = threadIdx.x + blockDim.x * blockIdx.x ;
  float c1,c2,c3,akm;
  float extra2d,csx2,p,p2;
  csx2 = csx*csx;

  if(tid < M)
  {
   c2 = 1.0/(dxt[tid]*dxt[tid]);
   c3 = c2*csx2;
   akm = abs(akmin)* 0.075*powf(c2,0.666667);
   scr2[tid]=dn0[tid]*fmaxf(akm,c3*sqrt(vt3dh[tid]));
  }
}
void main(){
//ESTRATÉGIA
  int ThreadsPerBlock = 512;
  int BlocksPerGrid = (N + ThreadsPerBlock - 1) / ThreadsPerBlock;

//DECLARANDO, ENVIANDO DADOS PARA GPU
  (...)
  turb1<<<BlocksPerGrid,ThreadsPerBlock>>> (N,csx,akmin,d_dxt,d_dn0,d_vt3dh,d_scr2);

//RECEBENDO DADOS DA GPU E LIBERANDO MEMÓRIA
  (...)
```

Figura 3.5 - Função Kernel.

Uma vez que os cálculos são finalizados, os resultados precisam ser copiados de volta para a CPU. Para a transferência de dados da GPU para a CPU, pode-se utilizar a função cudaMemcpy(). Essa função requer 4 parâmetros: Ponteiro para o destino, ponteiro para a origem, número de *bytes* copiados e o tipo de transferência (*host* para *device*, *device* para *device*, *device* para *host* e *host* para *host*). Na Figura 3.6, tem-se um exemplo de sua sintaxe.

```

//RECEBENDO DADOS A CPU (DEVICE/HOST)

cudaMemcpy(h_vt3di,dev_vt3di,M*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(h_scr1,dev_scr1,M*sizeof(float),cudaMemcpyDeviceToHost);

```

Figura 3.6 - Sintaxe cudamemcpy().

Finalmente, terminado todos esses passos, a memória alocada na GPU e CPU são liberadas. A Figura 3.7 ilustrada um fluxograma da rotina implementada, seguindo a lógica de programação da arquitetura CUDA.

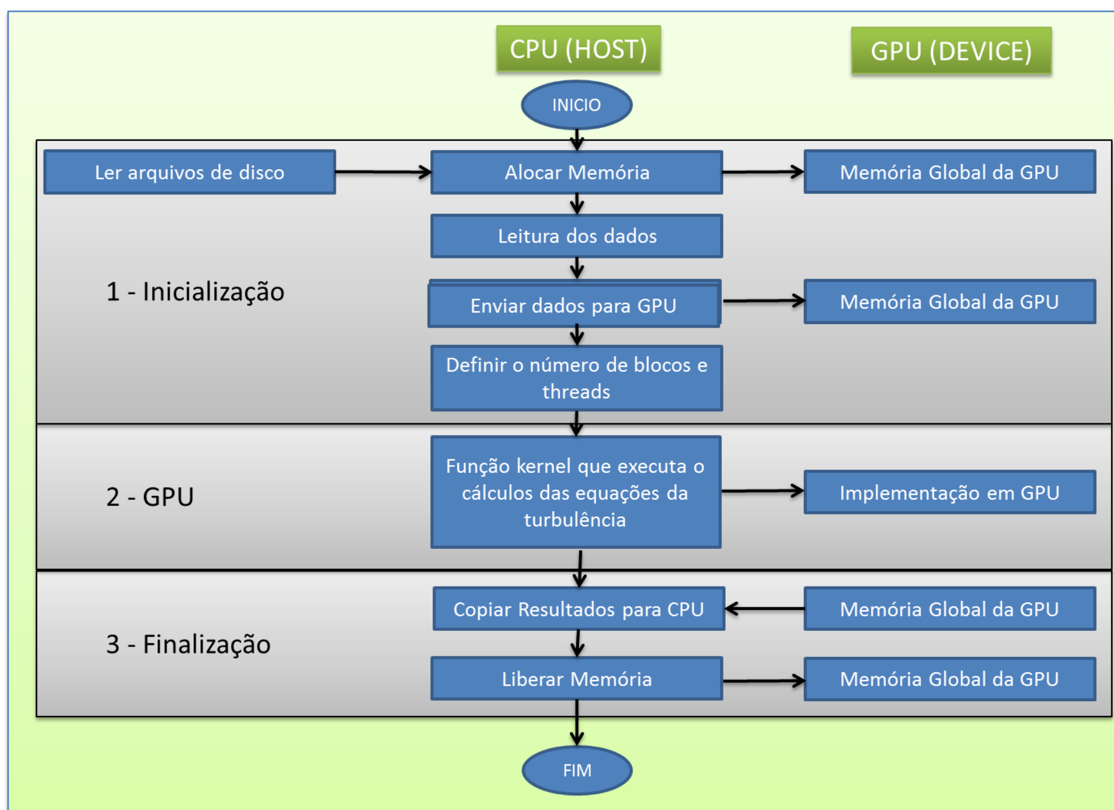

Figura 3.7 – Fluxograma de funcionamento do programa

3.3. Recursos computacional

- Máquina Jaburu: processador Intel Core I7 950 3.07 GHz, Quad-Core LGA 1366, 12GB de RAM 1333MHz, 2 TB de *hard disk* e uma Nvidia GeForce GTX 580 1.5GB GDDR5. Recurso proveniente do LAC/INPE.

- Sistema Operacional Linux 64 bits: distribuição Ubuntu 10.04.4 LTS.

- NVIDIA (R) *Cuda compiler driver, Cuda compilation tools*, release 4.2.

4 RESULTADOS

Este capítulo apresenta os resultados alcançados com a paralelização em CUDA da rotina de parametrização de Smagorinsky para turbulência. O ganho de desempenho é avaliado por meio do cálculo do *speed-up*, em relação à implementação serial e os resultados são apresentados em gráficos e tabelas. O *speed-up* é uma métrica de desempenho comumente utilizada em sistemas paralelos. Formalmente é definido como sendo a razão entre o tempo gasto na execução com 1 único processador e o tempo gasto na execução com vários processadores, ou seja, $\boldsymbol{S} = \frac{\boldsymbol{T_S}}{\boldsymbol{T_P}}$.

4.1. Parametrização

Parametrização são esquemas de simulação dos processos físicos que descrevem turbulência, radiação, dinâmica de nuvens e precipitação. Exemplos de parametrizações da turbulência são:

- Parametrização de Smagorinsky (1963): O fluxos turbulentos são parametrizados utilizando a teoria do fluxo-gradiente conhecida como Teoria K, com uso de parâmetros de discretização espacial.
- Parametrização de Mellor e Yamada (1974): fechamento de ordem 2,5, onde a difusão vertical é parametrizada com base no prognóstico de energia cinética turbulenta.
- Parametrização da teoria Taylor: Descrição estatística da turbulência.

4.2. Região utilizada nos testes

Para a realização dos testes, foi utilizada uma região da América do Sul mostrada na Figura 4.1. Duas resoluções horizontais foram empregadas para discretizar o domínio: 40 km, resultando em uma grade com 173.280 pontos e outra com 20 km de resolução, resultando em uma grade de 715.008 pontos.

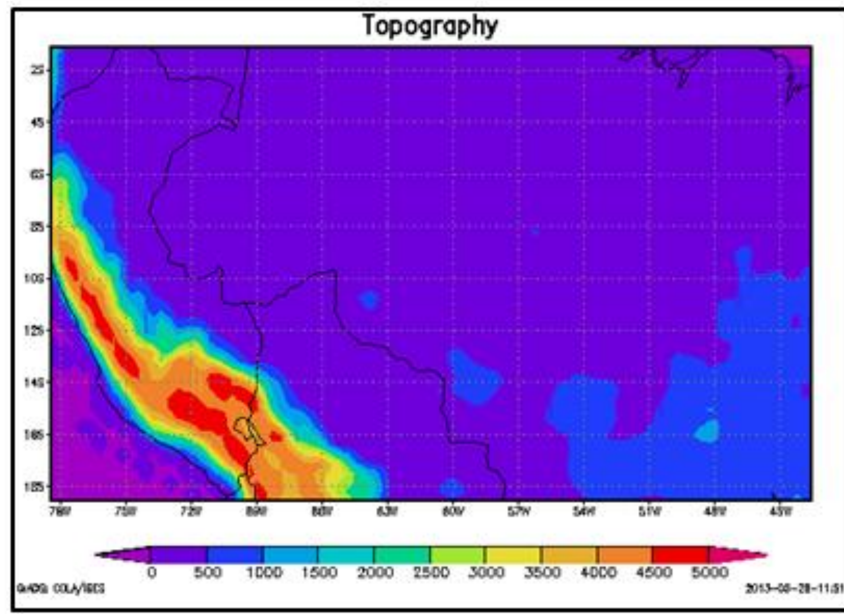

Figura 4.1 – Topografia da região da América do Sul utilizada nos testes

4.3. Desempenho em CUDA

As medidas de desempenho foram obtidas com uma rodada de 2880 iterações, ou *timesteps*. Os testes foram realizados em diferentes tipos de versões da parametrização, seguindo a implementação original de Smagorinsky, conforme o valor especificado no parâmetro IDIFFK. Os identificadores definidos pelo parâmetro IDIFFK controlam o tipo de parametrização a ser usado para calcular os coeficientes de difusão horizontal e vertical. Os valores permitidos para esse parâmetro são 1,2,3 e 4. O valor 1 e 2 é apropriado para um modelo de grade em que o espaçamento da grade horizontal é maior que o espaçamento vertical. Os valores 3 e 4 são usualmente indicados para grades que possuem o mesmo espaçamento tanto na horizontal quanto na vertical. Neste trabalho os testes foram realizados considerando os valores de IDIFFK = 1, 2 e 3.

A seguir serão apresentados e analisados os desempenhos alcançados com a implementação em CUDA utilizando a região em duas resoluções distintas, conforme mencionado acima.

4.3.1. Resolução de 40km

A Tabela 3 mostra a comparação de desempenho entre as implementações em serial e em CUDA. Nela, podemos verificar na primeira coluna os tipos de versões de parametrizações utilizadas, na segunda e terceira colunas os tempos de processamento em serial e em paralelo, respectivamente, e na quarta coluna o *speed-up*. Como se pode constatar, os tempos em serial variaram de 7.321,33 até 27.067,88 milissegundos e houve uma melhora de desempenho entre 8,81 e 12,54 vezes em relação ao tempo serial.

Tabela 3 – Resultado para resolução de 40 km

IDFFK	Tempo Serial (ms)	Tempo Paralelo (ms)	*Speed-up*
1	7.321,33	830,96	8,81
2	27.027,02	2.337,41	11,56
3	27.058,88	2.157,06	12,54

Para facilitar a visualização e identificar ainda mais a diferença entre os tempos das implementações foi elaborada a Figura 4.2. O eixo das abscissas apresenta os identificadores (IDIFFK), por sua vez, o eixo das ordenadas mostra os tempos de processamento em milissegundos.

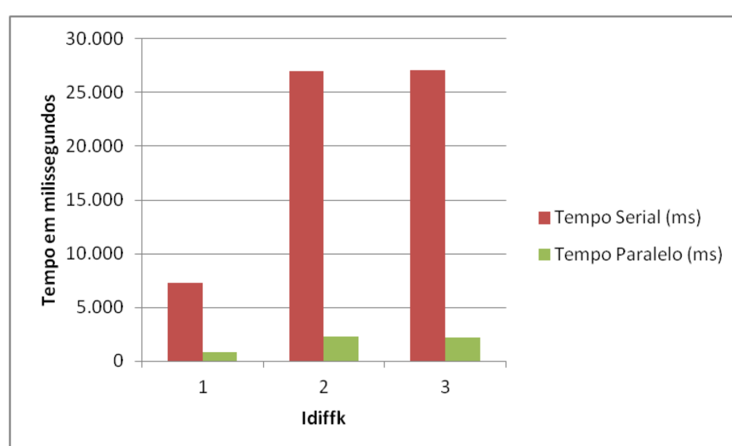

Figura 4.2 – Gráfico dos tempos resolução 40km

4.3.2. Resolução de 20 km

A comparação de desempenho entre as implementações em serial e em CUDA para resolução de 20 km é mostrada na Tabela 4. Como na tabela acima, ela apresenta na primeira coluna os tipos de versões de parametrizações utilizados, na segunda e terceira, respectivamente, os tempos de processamento em serial e em paralelo, e na quarta coluna o *speed-up*. Nesse caso, os tempos em serial variaram de 30.501,83 até 146.375,46 milissegundos e houve uma melhora de desempenho entre 9,37 e 17,21 vezes em relação ao tempo serial.

Tabela 4 – Resultados para resolução 20km

IDFFK	Tempo Serial (ms)	Tempo Paralelo (ms)	*Speed-up*
1	30.501,83	3.254	9,37
2	146.673,15	9.192,56	15,96
3	146.375,46	8.504,89	17,21

Por meio da Figura 4.3, pode se observar de forma gráfica a diferença entre os tempos das implementações.

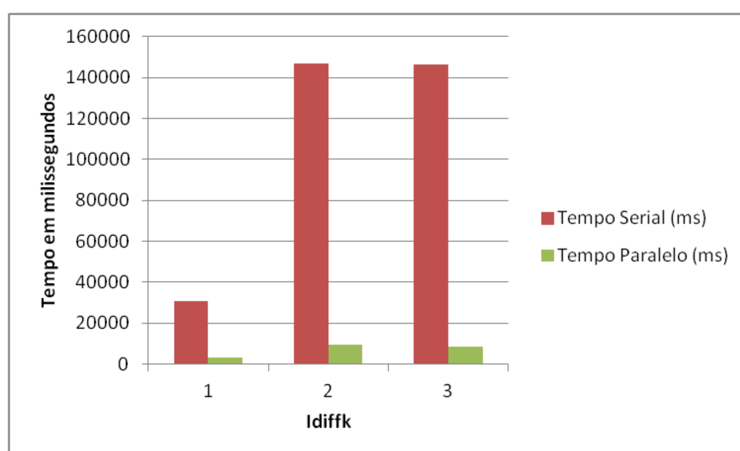

Figura 4.3 – Gráficos do tempo resolução de 20km

4.4. Análise dos resultados

Para avaliação de desempenho da implementação paralela utilizando placas gráficas (*many-core*), é feita com análises de alocação de memória, utilização de *threads* e a transferência de dados entre a CPU e a GPU, além dos resultados de *speed-up*.

As Tabelas 5 e 6 apresentam, detalhadamente, os tempos gastos pela implementação paralela. Nela, são evidenciados os tempos gastos com a transferência de dados entre a CPU e a GPU. Verifica-se que o maior consumo de tempo está, justamente, no envio de dados. Assim, essa característica é encarada como algo limitante à busca por desempenho nas aplicações em GPU.

Tabela 5 – Tempos para resolução de 40 km (em ms)

GRID 95x48x38	IDFFK = 1	IDFFK = 2	IDFFK = 3
Alocação memória CPU	3,98	3,98	3,98
Alocação memória GPU	1,20	1,20	1,20
Envio CPU para GPU	341,07	682,27	681,99
Kernel (Cálculo na GPU)	146,60	297,91	117,96
Envio GPU para CPU	337,28	1351,22	1351,10
Liberação memória	0,83	0,83	0,83
Total	830,96	2337,41	2157,06

Tabela 6 – Tempos para resolução de 20 km (em ms)

GRID 194x100x40	IDFFK = 1	IDFFK = 2	IDFFK = 3
Alocação memória CPU	8,99	8,99	8,99
Alocação memória GPU	1,21	1,21	1,21
Envio CPU para GPU	1384,07	2768,70	2767,79
Kernel (Cálculo na GPU)	539,89	1137,74	453,39
Envio GPU para CPU	1318,77	5274,85	5272,44
Liberação memória	1,07	1.07	1.07
Total	3254	9192,56	8504,89

Por fim, para constatar a eficiência da implementação em CUDA foi realizado o cálculo do *speed-up* em dois momentos: (a) considerando o tempo total, ou seja, de alocação e liberação de memória, transferência de dados e o cálculo

na GPU (*kernel*); (b) considerando somente o cálculo na GPU (*kernel)*. As Figuras 4.4 e 4.5 apresentam uma comparação de desempenho nos dois momentos citados, considerando as resoluções utilizadas. Verifica-se um desempenho muito superior considerando apenas o *kernel*. O tempo de alocação e liberação de memória não influi significativamente no desempenho pois essas operações somente são realizadas uma vez no início e fim do programa. A transferência de dados, por sua vez, é determinante no desempenho dessa aplicação, pois algumas variáveis que são atualizadas a cada passo de iteração, precisam ser enviadas da CPU para a GPU e vice-versa.

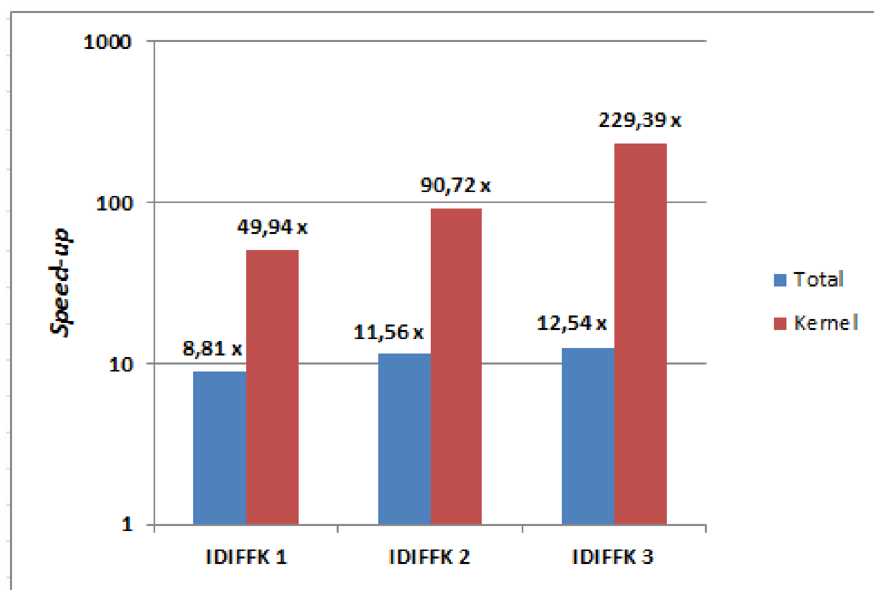

Figura 4.4 – *Speed-up* com e sem transferência de dados, considerando resolução de 40km

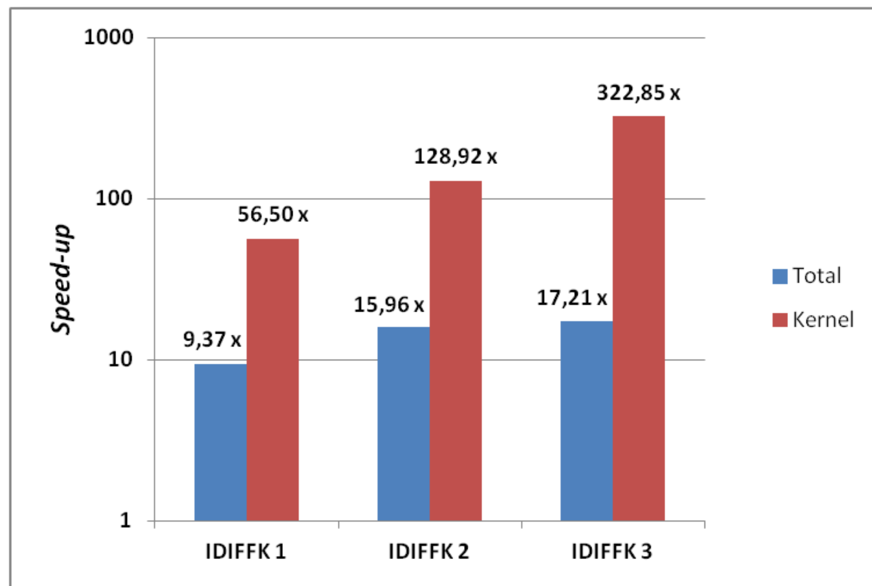

Figura 4.5 – *Speed-up* com e sem transferência de dados considerando resolução de 20km

5 CONSIDERAÇÕES FINAIS

Abaixo são descritas as contribuições deste estudo e destacam-se também possíveis trabalhos futuros relacionados.

5.1. Contribuições e conclusões

Foi desenvolvida uma versão em GPU da rotina de parametrização da turbulência denominada parametrização de Smagorinsky. A implementação do código foi realizada utilizando a linguagem de programação C e a plataforma CUDA. O desempenho da implementação foi avaliado por meio do cálculo de *speed-up*, considerando duas resoluções: 40 km e 20 km. A comparação entre os tempos obtidos com a rotina original (em Fortran) e a rotina em CUDA, mostram um ganho significativo de desempenho (até 17,21 vezes para a resolução de 20 km e até 12,5 vezes para a resolução de 40 km). Por meio da análise do tempo gasto em todo o processamento da versão CUDA, pode-se notar que o custo de transferência de dados é muito alto e tem papel chave no desempenho dessa aplicação.

Os resultados obtidos nesta pesquisa mostram que o uso de máquinas de arquitetura híbrida é uma boa alternativa para reduzir o tempo computacional desta aplicação. Os testes realizados em CUDA foram promissores para rotinas do modelo de previsão BRAMS e o modelo ambiental CCATT-BRAMS.

O uso da tecnologia de GP-GPU ainda não é algo simples. Foi necessário um longo período de aprendizado para entendimento dos ambientes de desenvolvimento CUDA e do modelo de previsão BRAMS. A metodologia adotada para a rotina de Smagorinsky. Existem outras rotinas de turbulência e outras parametrizações físicas a serem exploradas pela computação híbrida. Contudo, devido à dependência de fluxo, nem todos os processos podem ser paralelizados

A Computação Híbrida mostrou-se uma eficiente abordagem a ser explorada e estudada. Por fim, o estudo sobre esse novo paradigma, incluindo o

paralelismo de dados e a arquitetura CUDA contribuíram significativamente para o meu desenvolvimento acadêmico e profissional.

5.2. Trabalho futuros

As contribuições alcançadas com este trabalho não encerram as pesquisas relacionadas com a computação híbrida para parametrizações do modelo BRAMS, mas abrem oportunidades para alguns trabalhos futuros:

- Estudar outras estratégias de alocação de memória e uso dos *threads* para a implementação da parametrização de Smagorinsky em GPU.

- Implementação em CUDA de outras rotinas de parametrização presentes no modelo BRAMS.

- Integração da rotina desenvolvida no modelo BRAMS.

- Codificação em outras padrões para GP-GPU: OpenCL e OpenACC (um novo padrão para programação em computação híbrida), e análise comparativa de resultados.

REFERÊNCIAS BIBLIOGRÁFICAS

ASANOVIC, Krste, et al. The landscape of parallel computing research: A view from Berkeley. **Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley**, 2006.

BRAMS v4.2**. Brazilian Regional Atmospheric Modeling System**. Disponível em http://brams.cptec.inpe.br/. Acesso em 03/05/2013.

CCATT-BRAMS. **Monitoring the Transport of Biomass Burning and Anthropogenic Pollution in South America**. Disponível em: http//meioambiente.cptec.inpe.br/modelo_cattbrams.php. Acesso em: 30/09/2012.

EL-REWINI, H.; ABD-EL-BARR, M.; **Advanced Computer Architecture and Parallel Processing**. Hoboken, New Jersey: John Wiley & Sons, Inc, 2005.

FLYNN, M. J. Very high-speed computing systems**. Proceedings of the IEEE**, Vol. 54, p. 1901 – 1909, 1966

FREITAS, S. R. ; LONGO, K. M. ; SILVA DIAS, M. A. F; CHATFIELD, R.; DIAS, P. S.; , ARTAXO, P.; ANDREA, M. O. ; GRELL, G. ; RODRIGUES, L. F.; FAZENDA, A. L. and J. PANETTA. The Coupled Aerosol and Tracer Transport model to the Brazilian developments on the Regional Atmospheric Modeling System (CATT-BRAMS). **Atmos. Chem. Phys.** Vol. 7, p. 8525 – 8569, 2007.

HILL, G. E. Factors controlling the size and spacing of cumulus clouds as revealed by numerical experiments. **J. Atmos. Sci.,** v. 31, p. 646–673, 1974.

KALNAY, E. **Atmospheric modeling, data assimilation and predictability**. Cambridge, England: Cambridge University, 2003.

KIRK, David et al. **Programando para Processadores Paralelos: uma abordagem prática à programação de GPU.** Elsevier Brasil, 2010.

LILLY, D. K. On the numerical simulation of buoyant convection. **Tellus**, v. 14, p.168–172, 1962.

LUEBKE, David et al. GPGPU: general-purpose computation on graphics hardware. In: **Proceedings of the 2006 ACM/IEEE conference on Supercomputing.** ACM, 2006. p. 208.

NVIDIA CUDA: **NVidia CUDA C Programming Guide**. V4.0 2011. Disponível em: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf. Acesso em: 15/09/2012.

NVIDIA FERMI: **NVIDIA's Next Generation CUDA Architecture**: Fermi. V1.1 2009. Disponível em: http://www.nvidia.com/content/PDF/ fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Acesso em: 17/09/2012.

NVIDIA KEPLER: **NVIDIA's Next Generation CUDA Architecture**: Kepler GK 110. V1.0 2012. Disponível em: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf. Acesso em: 17/09/2012.

NVIDIA CUDA-MEMCHECK: **CUDA-MEMCHECK DU-05355-042_v01 2012**. Disponível em: http://people.maths.ox.ac.uk/gilesm/cuda/doc/cuda-memcheck.pdf. Acesso em: 20/10/2012.

PANETTA, J., BARROS, S. R. M., BONATTI, J. P., TOMITA, S.S. AND KUBOTA, P. Y. Computational cost of CPTEC AGCM. **Proceedings of the twelfth ECMWF workshopon use of high performance computing in meteorology.** Reading, UK. 30 Oct – 3 Nov2006. Edited by George Mozdzynski.

PIELKE, R. A., et al. A comprehensive meteorological modeling system – RAMS, **Meteorology and Atmospheric Physics.** Vol. 42, p. 69 – 91, 2006.

REYNOLDS, O. On the dynamical theory of incompressible viscous fluids and the determination of the criterion. **Phil. Trans. Roy. Soc. London**, A186, p.123–164, 1895.

SANDERS, J. and KANDROT, E. **Cuda by Example: an introduction to general-purpose gpu programming.** San Francisco, USA: Addison-Wesley, 2010.

SMAGORINSKY, J. General circulation experiments with the primitive equations: I. the basic experiment**. Mon. Weather Rev.,** v. 91, p. 99–164, 1963.