

Fabiano Cassol de Vargas

**Estudo de Técnicas de Otimização de
Desempenho para GPUs Utilizando CUDA
Aplicado a um Modelo Meteorológico**

Alegrete – RS

08/2014

Fabiano Cassol de Vargas

**Estudo de Técnicas de Otimização de Desempenho para
GPUs Utilizando CUDA Aplicado a um Modelo
Meteorológico**

Trabalho de Conclusão de Curso apresentado
ao Curso de Graduação em Ciência da Com-
putação da Universidade Federal do Pampa
como requisito parcial para a obtenção do tí-
tulo de Bacharel em Ciência da Computação.

Universidade Federal do Pampa – UNIPAMPA

Campus Alegrete

Curso de Graduação em Ciência da Computação

Orientador: Claudio Schepke

Alegrete – RS

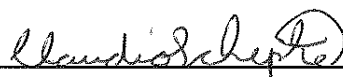
08/2014

Fabiano Cassol de Vargas

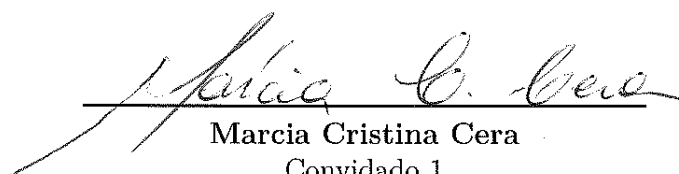
Estudo de Técnicas de Otimização de Desempenho para GPUs Utilizando CUDA Aplicado a um Modelo Meteorológico

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

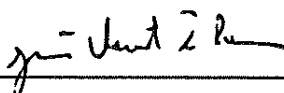
Trabalho de Conclusão de Curso defendido e aprovado em 26. de Agosto... de 2014



Claudio Schepke
Orientador



Marcia Cristina Cera
Convidado 1



João Vicente Ferreira Lima
Convidado 2

Alegrete – RS

08/2014

*Dedico este trabalho aos meus pais, os quais sempre
acreditaram na minha força e apoiaram as minhas escolhas,
me orientando e instruindo.*

*“Existe uma teoria que diz que, se um dia
alguém descobrir exatamente para que serve o universo
e porque ele está aqui, ele desaparecerá instantaneamente
e será substituído por algo ainda mais estranho e inexplicável.
Existe uma segunda teoria que diz que isso já aconteceu.”
(O Restaurante no fim do Universo, Douglas Adams)*

Resumo

Este trabalho teve como objetivo estudar técnicas de otimização de desempenho para um protótipo do *Ocean-Land-Atmosphere Model* (OLAM) implementado em CUDA utilizando GPUs. Foi feita uma ampla revisão bibliográfica acerca do modelo de programação CUDA e das respectivas arquiteturas de GPUs. Análises de desempenho foram realizadas para comparar as alterações aplicadas com uma versão sem paralelismo em CPU, bem como a versão inicial do protótipo. O modelo foi submetido a análises via *profiling*, onde foram feitas observações mais aprofundadas através de ferramenta de visualização gráfica. As alterações implementadas obtiveram pouco ganho de desempenho para o modelo, mas os estudos analisaram diversos fatores que ainda podem ser explorados para tirar proveito de um ambiente de execução massivamente paralelo como arquiteturas GPU.

Palavras-chave: Ocean-Land-Atmosphere Model (OLAM). Graphic Processing Unit (GPU). Compute Unified Device Architecture (CUDA). Técnicas de otimização.

Abstract

This work studied techniques for optimizing performance for a prototype of the Ocean-Land-Atmosphere Model (OLAM) implemented on CUDA using GPUs. One wide literature review about the CUDA programming model and its architecture of GPUs has been made. Performance analyzes were conducted to compare the changes implemented with a version without parallelism in CPU and the initial version of the prototype. The model was subjected to analysis via profiling, where more detailed observations were made through a graphical visualization tool. The changes implemented had little performance gain for the model, but the studies analyzed several factors that can still be exploited to take advantage of a massively parallel execution environment as GPU architectures.

Key-words: Ocean-Land-Atmosphere Model (OLAM). Graphic Processing Unit (GPU). Compute Unified Device Architecture (CUDA). Optimization techniques.

Lista de ilustrações

Figura 1 – Operações de ponto flutuante por segundo para CPUs e GPUs.	18
Figura 2 – Divisão do globo em uma malha de pontos triangulares.	23
Figura 3 – Diferentes resoluções horizontais de malha global.	23
Figura 4 – Representação de um ponto da malha como um prisma triangular.	24
Figura 5 – Globo terrestre com refinamento local de malha.	25
Figura 6 – Fases de execução do OLAM.	25
Figura 7 – Diferenças de projeto entre CPUs e GPUs.	28
Figura 8 – Comparativo da largura de banda de memória entre CPUs e GPUs.	29
Figura 9 – Esquema da arquitetura de uma GPU que utiliza CUDA.	30
Figura 10 – Arquitetura Fermi.	31
Figura 11 – Multiprocessador de <i>streaming</i> da arquitetura Fermi.	32
Figura 12 – Arquitetura Kepler.	33
Figura 13 – Multiprocessador de <i>streaming</i> da arquitetura Kepler.	34
Figura 14 – Organização de unidades de processamento em arquitetura de GPU AMD.	35
Figura 15 – Definindo uma função de <i>kernel</i>	37
Figura 16 – Invocando um <i>kernel</i>	37
Figura 17 – Esquema de organização de <i>threads</i> CUDA em um <i>grid</i> de <i>thread blocks</i>	39
Figura 18 – Execução de um programa CUDA comum.	39
Figura 19 – <i>NVIDIA Visual Profiler</i>	43
Figura 20 – <i>CUDA GPU Occupancy Calculator</i>	44
Figura 21 – Resultados dos testes na placa Tesla C2075.	46
Figura 22 – Resultados dos testes na placa Quadro 5000.	46
Figura 23 – Resultados dos casos de testes com 12 horas de simulação.	48
Figura 24 – <i>Timeline</i> do <i>NVIDIA Visual Profiler</i> para a aplicação com blocos de 128 <i>threads</i> para todos os <i>kernels</i>	48
Figura 25 – <i>Timeline</i> do <i>NVIDIA Visual Profiler</i> para a aplicação com blocos de 1.024 <i>threads</i> para todos os <i>kernels</i>	49
Figura 26 – Testes realizados com blocos de <i>threads</i> que atingem a maior ocupação possível da GPU.	50
Figura 27 – Execuções em GPU com um ou mais <i>streams</i>	51
Figura 28 – Execução assíncrona em GPU com múltiplos <i>streams</i>	51
Figura 29 – Exemplo de um laço iterativo FOR antes de ser alterado em um <i>kernel</i>	53
Figura 30 – Exemplo de um laço iterativo FOR depois de ser alterado em um <i>kernel</i>	53
Figura 31 – Comparação de tempos de execução da aplicação: <i>kernels</i> reestruturados VS <i>kernels</i> sem alteração.	54

Lista de tabelas

Tabela 1 – Aumento do número de transistores e núcleos CUDA nas GPUs NVIDIA.	35
Tabela 2 – Palavras-chave qualificadoras para declaração de funções, fornecidas por CUDA.	37
Tabela 3 – GPUs utilizadas para os testes.	42
Tabela 4 – Resultados dos testes do OLAM em CPU sem paralelismo.	47
Tabela 5 – Dependências de dados dos <i>kernels</i> implementados no OLAM.	52

Sumário

1	INTRODUÇÃO	17
1.1	Objetivos	19
1.2	Organização do Trabalho	19
2	MODELOS CLIMATOLÓGICOS	21
2.1	Weather Research and Forecasting Model	21
2.2	Regional Atmospheric Modeling System	21
2.3	Brazilian Developments on the Regional Atmospheric Modeling System	22
2.4	Ocean-Land-Atmosphere Model	22
3	GPUS - CONCEITO E PROGRAMAÇÃO	27
3.1	CPU vs GPU	27
3.2	As arquiteturas das GPUs NVIDIA com Suporte a CUDA	29
3.2.1	Arquitetura Fermi	30
3.2.2	Arquitetura Kepler	32
3.2.3	Evolução das GPUs NVIDIA	34
3.3	Comparativo entre a Organização de GPUs AMD e NVIDIA	35
3.4	Interfaces de Programação para GPU	36
3.5	Compute Unified Device Architecture (CUDA)	36
4	METODOLOGIA	41
4.1	Recursos de Hardware	42
4.2	Recursos de Software	42
5	ESTUDOS REALIZADOS E ANÁLISE DOS RESULTADOS	45
5.1	Fase 1 - Testes Iniciais	45
5.2	Fase 2 - Análise de Ocupação da GPU	47
5.3	Fase 3 - Análise de Dependência de Dados em cada Kernel	50
5.4	Fase 4 - Reestruturação Interna dos Kernels da Aplicação	52
5.5	Fase 5 - Cópia de Memória Assíncrona e Particionamento de Kernel	54
6	CONCLUSÃO	57
	Referências	59

1 Introdução

Há séculos a humanidade busca compreender o comportamento da natureza para prever a ocorrência de determinados fenômenos naturais. Entender, por exemplo, quando as chuvas ocorrem é um fator essencial em áreas como a agricultura, pois um mal planejamento pode comprometer o trabalho realizado. Nas últimas décadas, ocorreram muitas alterações no clima devido às ações do homem sobre a natureza, como o aumento da temperatura global. A poluição e o extrativismo de recursos naturais descontrolados contribuem para agravar a situação do clima mundial. Diversos fenômenos meteorológicos são de interesse das mais variadas áreas de estudo, bem como da população como um todo.

Visto isso, percebe-se que há uma grande importância nas pesquisas sobre previsões do tempo e em como melhorar o que já foi realizado na área de estudo. Prever como fenômenos atmosféricos irão se comportar é uma tarefa que depende de muitos fatores, os quais devem ser analisados com base em um histórico de um determinado período de tempo. Existem, atualmente, diversos modelos meteorológicos que estão em estudo e que são utilizados para a previsão do tempo. Esses modelos são programas executados em computadores que fazem uma aproximação da previsão dos fenômenos meteorológicos. Estes programas realizam muitos cálculos sobre diferentes dados repetidas vezes, o que atribui uma carga de trabalho imensa ao processador. Isso leva à necessidade de recorrer à computação de alto desempenho, como mencionado em [FINEP \(2013\)](#) e [IBM \(2014\)](#). Um simples processador não seria capaz de executar um programa que determinasse uma previsão em tempo hábil.

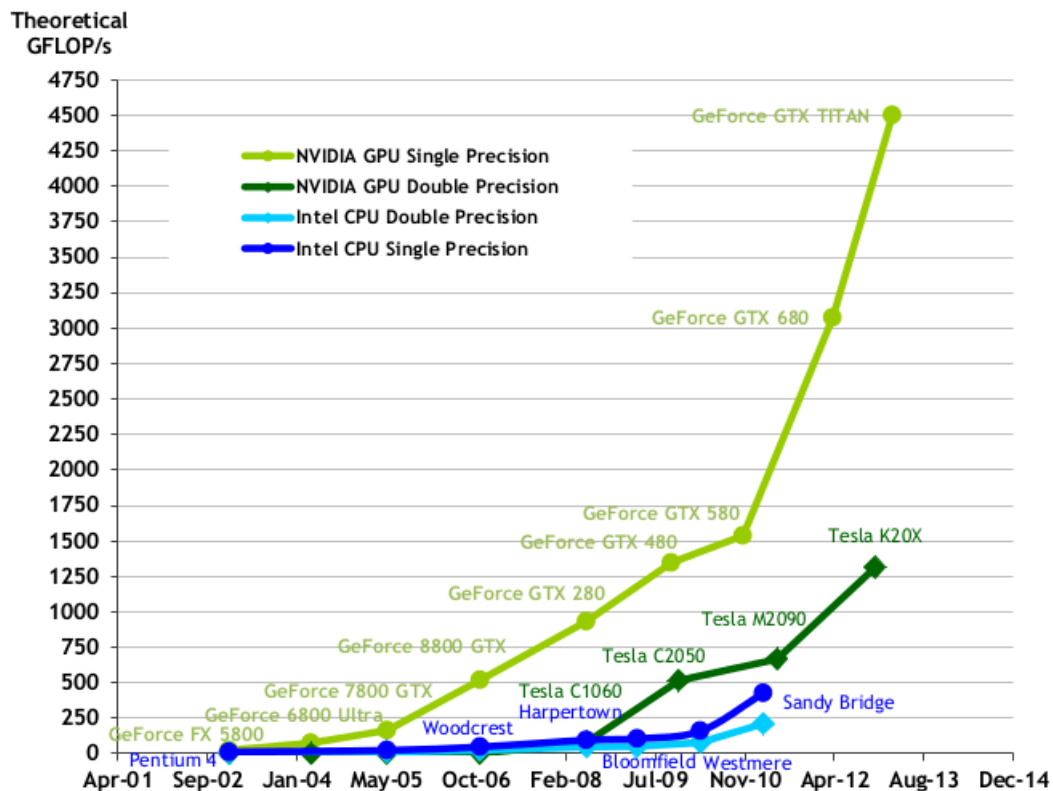
O Processamento de Alto Desempenho (PAD) faz uso da conexão física entre vários processadores/computadores, seja através de barramento em *chip* ou de uma rede física de cabos. Assim, muitos processadores/computadores podem ser usados a fim de realizar simultaneamente uma mesma computação, agilizando a execução de aplicações que demandam muito poder de processamento.

As arquiteturas *multicore* e processadores gráficos são comuns nos computadores atuais. Processadores com múltiplos núcleos de processamento - os mais comuns possuem entre 2 e 4 núcleos - são utilizados para realizar tarefas simultâneas, o que os torna intrínsecos aos processos computacionais de alto desempenho. Além disso, placas gráficas podem ser utilizadas para prover alto desempenho para aplicações variadas. Por conta de sua arquitetura específica, é possível tirar proveito de trechos de um programa que demandem muito processamento e que possam ser paralelizados. Diferentemente da arquitetura de uma *Central Processing Unity* (CPU), uma *Graphics Processing Unity* GPU

possui muitos núcleos (de dezenas à milhares), o que as torna um poderoso processador massivamente paralelo capaz de superar as CPUs em termos de paralelismo de dados.

Na [Figura 1](#) é mostrada a evolução da capacidade de CPUs e GPUs em realizar operações de ponto flutuante, onde percebe-se uma considerável vantagem das placas gráficas em relação às arquiteturas de CPUs. As CPUs dos computadores atuais são *multicore*, capazes de executar diversos processos simultaneamente, um em cada núcleo. O número de núcleos dos processadores varia, sendo 2, 4, 6, 8, ou até mais. Por outro lado, um processador gráfico (GPU) pode ter dezenas ou centenas de núcleos de processamento no *chip*, sendo capaz de operar sobre uma quantidade muito maior de tarefas simultâneas em relação à uma CPU. Hoje, CPU e GPU podem ser utilizados em conjunto para prover desempenho à diversas aplicações.

Figura 1 – Operações de ponto flutuante por segundo para CPUs e GPUs.



Fonte: NVIDIA (2013a)

Em um ambiente de execução massivamente paralelo, no caso das GPUs, determinadas aplicações podem ser adaptadas a fim de alcançar grandes ganhos de desempenho. Estudos de um determinado problema de uma classe podem beneficiar uma boa parte de sua categoria, dado que há determinadas características comuns à classe que podem ser aprimoradas ao se fazer um estudo específico de uma aplicação. O *Ocean-Land-Atmosphere Model* (OLAM) é um problema típico de decomposição de domínios

(SCHEPKE, 2012), representando uma classe de aplicações similares neste ponto. Neste sentido, optou-se por utilizar para os estudos uma aplicação de previsão meteorológica que demanda grande poder de processamento computacional para executar.

1.1 Objetivos

Considerando o uso de processadores gráficos para a computação de propósito geral a fim de prover desempenho para aplicações de alto custo computacional, este trabalho visa investigar que benefícios são obtidos com o uso de GPUs para uma aplicação de climatologia. Pretende-se, principalmente, avaliar o desempenho paralelo do OLAM implementado em CUDA e executado em GPUs NVIDIA, considerando as características das placas gráficas para encontrar meios possíveis de otimizar o código do programa. Espera-se que as otimizações propostas melhorem o desempenho da aplicação comparada com a versão sem o uso GPU e com o protótipo utilizando CUDA, no qual se baseia este trabalho.

1.2 Organização do Trabalho

O restante deste trabalho organiza-se da seguinte maneira. No [Capítulo 2](#), são apresentados alguns modelos climatológicos, com foco no OLAM, o qual é objeto alvo dos estudos deste trabalho. O [Capítulo 3](#) discorre sobre conceitos de unidades de processamento gráfico, comparações de projeto e performance entre CPUs e GPUs, arquiteturas de GPUs da NVIDIA para CUDA e finalizando com interfaces de programação para GPUs, examinando aspectos fundamentais do modelo de programação de CUDA. O [Capítulo 4](#) aborda a metodologia aplicada para o desenvolvimento deste trabalho e os recursos de *hardware* e de *software* utilizados. O [Capítulo 5](#) apresenta os estudos realizados e as análises feitas sobre os mesmos, ainda discutindo os resultados obtidos. O [Capítulo 6](#) finaliza com as conclusões as quais se chegou durante a realização deste trabalho, citando possibilidade de trabalhos futuros e menciona a publicação e submissão de artigos em eventos.

2 Modelos Climatológicos

Atualmente, existem diversos modelos climatológicos operacionais e em estudo. Estes modelos são programas de computador que foram criados a partir de estudos anteriores e que continuam sendo aprimorados. Dentre os mais difundidos mundialmente, alguns se destacam e estão fundamentalmente ligados. Grandes empresas e instituições aplicam estes modelos em prol de soluções para problemas reais, bem como pesquisam e implementam estes modelos para uso próprio e para distribuição aberta.

Nas seções deste capítulo são brevemente apresentados alguns modelos meteorológicos, como o *Weather Research and Forecasting Model*, o *Regional Atmospheric Modeling System* e o *Brazilian Developments on the Regional Atmospheric Modeling System*. No final é feita uma apresentação mais detalhada sobre o *Ocean-Land-Atmosphere Model*, o qual é o foco deste trabalho.

2.1 Weather Research and Forecasting Model

Weather Research and Forecasting Model (WRF) (SKAMAROCK et al., 2008) é um sistema de previsão numérica de tempo e simulação atmosférica. Seu suporte é feito por grupos de pesquisa e desenvolvimento que atuam em diversas funcionalidades do modelo, e é utilizado por empresas e instituições operacionais e de pesquisa. Através das simulações e previsões numéricas, o WRF atua em aplicações de escalas locais e globais. Essas aplicações incluem previsão numérica de tempo, em tempo real, simulações climáticas regionais, qualidade do ar, entre outras.

Os grupos de trabalho e desenvolvimento atuam, entre outros, nas seguintes áreas: arquitetura de *software*; desenvolvimento de assimilação de dados; pós-processamento, testes e verificação; modelos físicos e numéricos; química atmosférica; modelagem da superfície terrestre; e implementação do modelo.

2.2 Regional Atmospheric Modeling System

Regional Atmospheric Modeling System (RAMS) (PIELKE et al., 1992) foi criado na *Colorado State University* mesclando vários códigos de simulação numérica de clima, visando unificar vários tipos de análises em um só modelo. Em seu processo de criação, outras funcionalidades foram adicionadas para melhorar as simulações. Estudos feitos sobre o modelo por Cotton (2014) e outros parceiros proveem constantes melhorias à aplicação.

RAMS é um sistema climatológico que atua em regiões específicas de larga escala como um hemisfério inteiro do globo, podendo fazer aproximações em escalas muito menores. Entre as suas funcionalidades, estão a simulação de redemoinhos, simulação de tempestade de raios, campos de nuvens densas, tempestades e outras.

2.3 Brazilian Developments on the Regional Atmospheric Modeling System

Uma variação do RAMS versão 6 foi feita por um grupo brasileiro: BRAMS (INPE/CPTEC, 2014). O modelo é mantido principalmente por pesquisadores do Centro de Previsão de Tempo e Estudos Climáticos (CPTEC/INPE), e no seu desenvolvimento foram introduzidas novas funcionalidades e parâmetros específicos para regiões tropicais para que o modelo se adaptasse para ser utilizado operacionalmente no Brasil. Entre as modificações estão a melhoria em simulação de nuvens densas e finas, o uso do terreno e tipo do solo. A partir dos estudos realizados no Brasil com o BRAMS, os pesquisadores do CPTEC criaram o CATT-BRAMS (FREITAS et al., 2009), um modelo de análise da qualidade do ar que analisa os resíduos químicos provenientes de queimadas da vegetação. BRAMS é licenciado pela GNU General Public License (Free Software Foundation, 2014), e algumas partes do seu código podem ter outras licenças mais restritas.

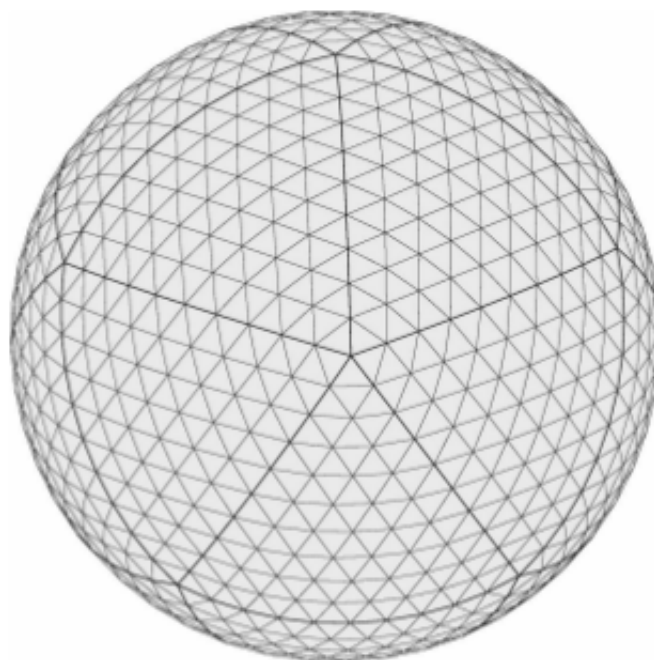
2.4 Ocean-Land-Atmosphere Model

O *Ocean-Land-Atmosphere Model* (OLAM) (WALKO; AVISSAR, 2008) foi criado por Robert L. Walko e Roni Avissar, da *Duke University*, como um novo modelo de simulação numérica baseado no *Regional Atmospheric Modeling System* (RAMS). O OLAM foi projetado para estender as capacidades do RAMS para um domínio de escala global, mantendo muitas características como inicialização de métodos, simulação de dados, lógica e estrutura de codificação e formatos de entrada e saída. Porém, seu novo funcionamento é baseado em uma grade global com uma malha de pontos triangulares para representação do espaço amostral do globo; baseia-se também na técnica de volumes finitos descrita por Marshall et al. (1997).

O OLAM representa o globo terrestre utilizando uma malha não estruturada, o que permite definir níveis de resolução horizontal variados, seja uma resolução maior ou menor, tanto no início de sua execução como durante o processo de simulação. Uma resolução maior significa uma malha global com mais pontos triangulares, e uma resolução menor implica o contrário, menos pontos, sendo os mesmos representados por triângulos maiores. Na Figura 2 é mostrada uma representação da malha global utilizada pelo modelo. Para que esta malha seja criada, inicialmente, a região do globo terrestre é dividida em 20

triângulos formando um icosaedro - são as linhas mais grossas da imagem. Cada um desses triângulos é então subdividido em triângulos menores conforme a resolução inicial estabelecida no código da aplicação - quanto maior a resolução da malha, mais preciso será o resultado das previsões, o que implica em ter mais pontos discretos e mais dados para se computar.

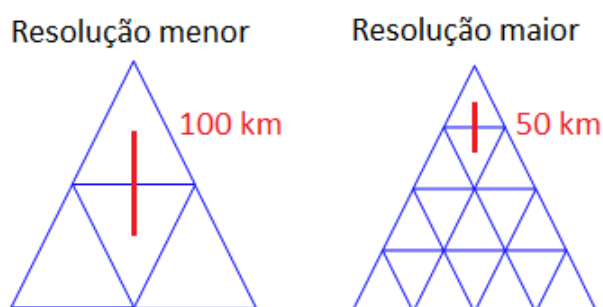
Figura 2 – Divisão do globo em uma malha de pontos triangulares.



Fonte: Walko e Avissar (2008)

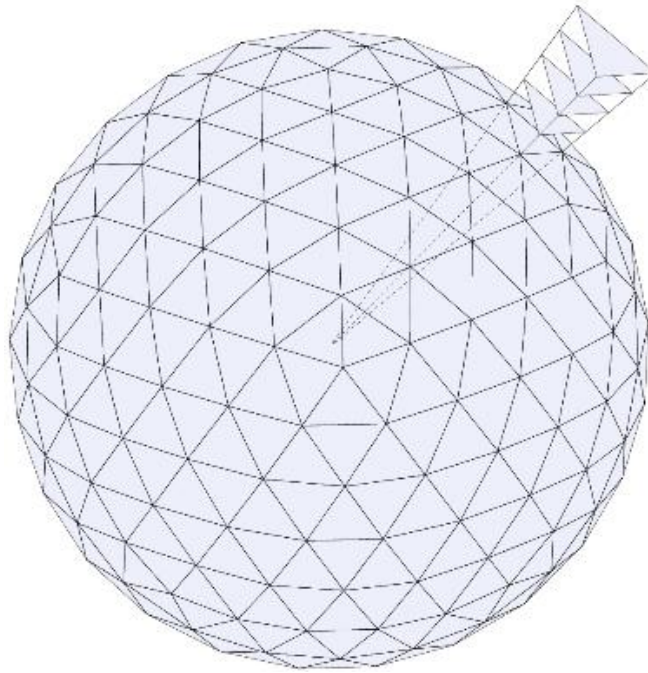
A resolução horizontal da malha global do OLAM é melhor explicada na [Figura 3](#). Uma resolução de 100 km significa que a distância do centro de cada ponto triangular para o centro dos pontos vizinhos é de 100 km. Logo, se tal distância for menor, maior será a resolução, pois haverá mais pontos de tamanho menor.

Figura 3 – Diferentes resoluções horizontais de malha global.



Para o OLAM, cada ponto da malha é representado, analogamente, por um prisma triangular com diversas camadas da divisão da atmosfera. Como os pontos são triangulares, sua projeção vertical cria essa forma de representação. Isso é representado na [Figura 4](#), o que define a resolução vertical da aplicação.

Figura 4 – Representação de um ponto da malha como um prisma triangular.



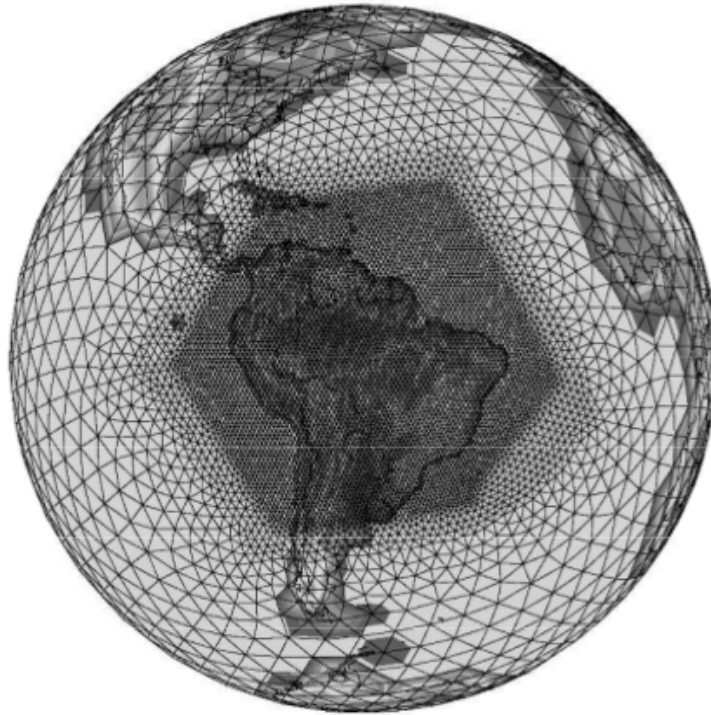
Fonte: [Walko e Avissar \(2008\)](#)

O modelo permite que se faça um refinamento em tempo de execução da resolução horizontal em uma determinada região do globo, conforme mostra a [Figura 5](#), onde se vê uma área refinada com triângulos menores. Isso é muito útil quando se deseja obter resultados mais precisos em regiões específicas, sem que haja a necessidade de executar toda a simulação com uma resolução grande. Para que isso ocorra, deve-se especificar qual a área a ser refinada. Assim, o modelo realiza um refinamento local com base na resolução horizontal definida inicialmente para o globo terrestre, duplicando essa resolução na região de refinamento. Por exemplo, se a resolução horizontal definida para a malha global no início da execução é 100km, no processo de refinamento dinâmico uma região passa ter uma resolução horizontal de 50km.

A [Figura 6](#) apresenta as fases da execução do OLAM. Na inicialização, são lidas as entradas e feitos os preparativos para o início da simulação. A etapa iterativa, com base nos dados de entrada, realiza a simulação do tempo futuro, com base nos dados passados, para chegar a uma previsão. Logo depois, caso não se deseje realizar o refinamento dinâmico da malha, o programa finaliza a execução. Caso se queira realizar o refinamento,

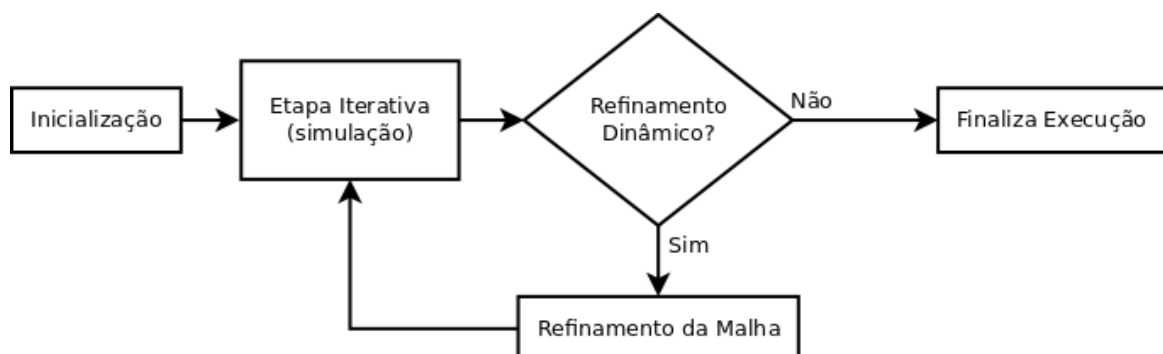
o programa manipula os pontos da malha para realizar a etapa de Refinamento da Malha em tempo de execução, prosseguindo com uma nova etapa iterativa de simulação. A etapa de refinamento pode ser realizada quantas vezes for necessária.

Figura 5 – Globo terrestre com refinamento local de malha.



Fonte: [Walko e Avissar \(2008\)](#)

Figura 6 – Fases de execução do OLAM.



3 GPUs - Conceito e Programação

As unidades gráficas de processamento - *Graphic Processing Unit* (GPU) - surgiram como *hardware* dedicados ao gerenciamento de imagens computacionais. Inicialmente, as GPUs possuíam pouco poder de processamento e eram capazes de operar com uma pequena variedade de cores em computadores em modo texto, o que foi mudando aos poucos, com o passar dos anos, devido à evolução dos computadores e seus componentes. As GPUs passaram a processar gráficos sofisticadas, como vídeos e jogos em computadores modernos, e até hoje estão cada vez mais sendo otimizadas para seu propósito de uso em processamento gráfico.

As placas gráficas vêm assumindo também outro papel: o de unidade gráfica de processamento para propósito geral - do inglês *General-Purpose Graphic Processing Unit* (GPGPU). O termo GPGPU é utilizado para referenciar o uso de placas gráficas a fim de executar aplicações que normalmente são executadas na CPU, ou ainda, para computar dados que não sejam referentes à imagens de computador, mas sim de outros programas. Isso se deve ao fato do grande poder de processamento paralelo presente nestes processadores, o que atraiu a atenção de cientistas e estudiosos das mais diversas áreas.

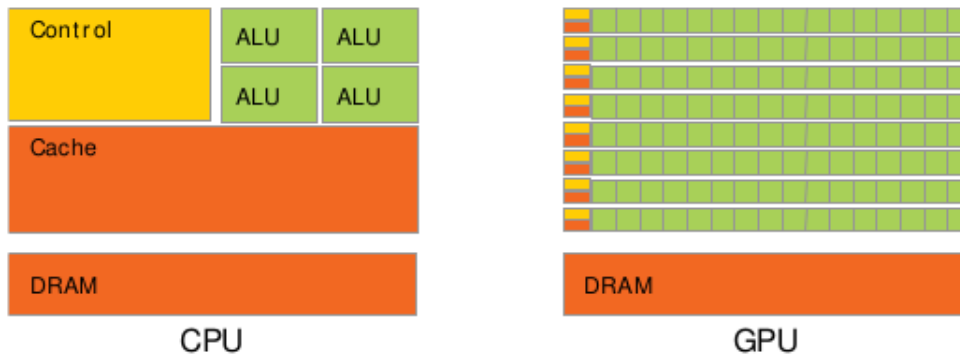
A seguir, este capítulo faz um comparativo de alguns aspectos fundamentais entre CPU e GPU, seguindo com exemplos de modelos de GPUs e apresentando sua arquitetura. Também são descritos alguns modelos de programação focando em *Compute Unified Device Architecture* (CUDA) e finalizando com alguns exemplos de aplicações que apresentam bom desempenho em GPUs.

3.1 CPU vs GPU

Uma questão pode ser levantada ao se comparar a performance de CPUs *multicore* com GPUs com muitos núcleos: o porque de tanta diferença de desempenho entre esses processadores (KIRK; HWU, 2011, 1.1). Esta divergência se deve ao fato de que CPUs e GPUs possuem diferenças fundamentais de projeto.

Na [Figura 7](#) tem-se um comparativo que mostra o diferencial básico entre CPU e GPU. As dimensões da memória *cache*, da unidade lógica e aritmética (ALU) e da unidade de controle da CPU são maiores do que as da GPU, pois nas GPUs os núcleos de execução (ALUs) são simplificados para efetuarem operações de ponto flutuante e de inteiro e alocados em grupos com *cache* e unidade de controle menores. A arquitetura de uma CPU é otimizada para se obter um bom desempenho sequencial, utilizando técnicas

Figura 7 – Diferenças de projeto entre CPUs e GPUs.



Fonte: [NVIDIA \(2013a\)](#)

para permitir que instruções de uma *thread* sejam executadas em paralelo ou mesmo fora de ordem. Memórias *cache* grandes são utilizadas para reduzir os tempos de acesso a instruções e dados dos programas. Segundo [Stallings \(2010\)](#), técnicas de projeto e organização da arquitetura do *chip* são desenvolvidas e aprimoradas a cada nova arquitetura, assim como uma crescente busca por aumentar a velocidade dos microprocessadores.

A filosofia de projetos das GPUs é focada acompanhando a ascendente indústria de videogames, pois os jogos cada vez mais demandam poder de cálculos de ponto flutuante. Isso motiva projetos de GPUs que otimizem a área do *chip* e o consumo de energia neste quesito. A principal solução é aumentar a vazão de execução maciça de *threads*, fazendo com que o *hardware* permita a execução de muitas *threads* enquanto algumas aguardam por acessos à memória DRAM, o que reduz a lógica de controle para cada *thread* ([KIRK; HWU, 2011, 1.1](#)). Como diversas *threads* compartilham dados para poderem executar um mesmo trecho de código de programa, pequenas memórias cache são utilizadas para armazenar esses dados compartilhados, reduzindo o tempo de acesso à memória sem necessitar recorrer à memória DRAM. Com isso, resta uma maior área útil do *chip* da GPU dedicada aos cálculos de ponto flutuante.

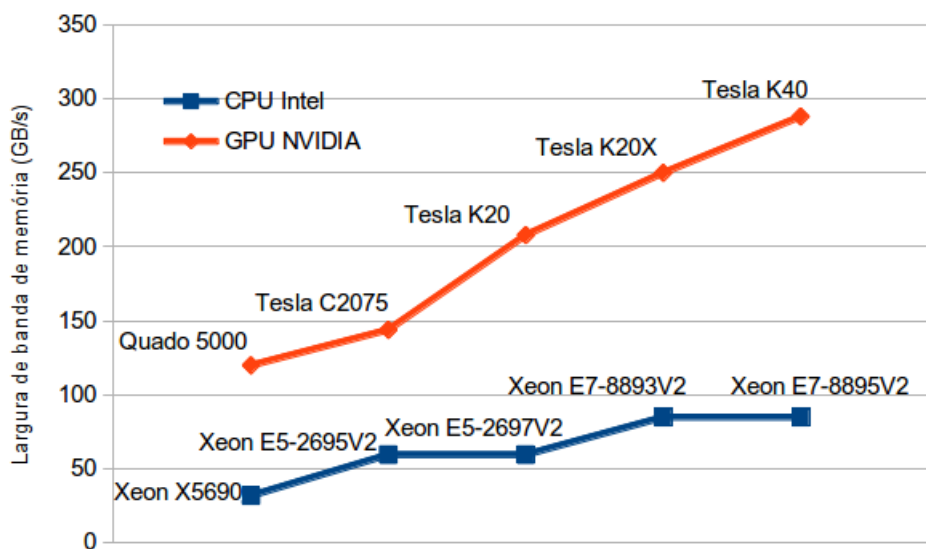
Um fator importante, conforme afirmam [Kirk e Hwu \(2011\)](#), é a largura de banda da memória. As GPUs possuem, em média, 10 vezes a largura de banda das CPUs ao longo do tempo. As CPUs precisam atender diversas tarefas, como do sistema operacional e dispositivos de entrada e saída, e isso impede a possibilidade de aumentar a largura de banda da memória, pois o processador não daria conta de utilizar um barramento maior. Por outro lado, as GPUs apresentam uma largura de banda de memória maior, pois existe um grupo menor de requisitos a serem atendidos, possibilitando um sistema de memória mais simples com maior largura de banda de memória.

A citação a seguir mostra que a diferença de largura de banda entre CPUs e GPUs

é fato de anos e ainda muito visível hoje em dia, conforme pode ser analisado na [Figura 8](#) com dados atuais.

Embora o ritmo de melhoria de desempenho dos micro-processadores de uso geral tenha desacelerado significativamente, o das GPUs continua a melhorar incessantemente. Em 2009, a razão entre GPUs com muitos núcleos e CPUs com múltiplos núcleos para a vazão máxima do cálculo de ponto flutuante era cerca de 10 para 1. (KIRK; HWU, 2011, p. 3)

Figura 8 – Comparativo da largura de banda de memória entre CPUs e GPUs.



Fonte: Intel (2014); NVIDIA (2013b); NVIDIA (2011); NVIDIA (2014b)

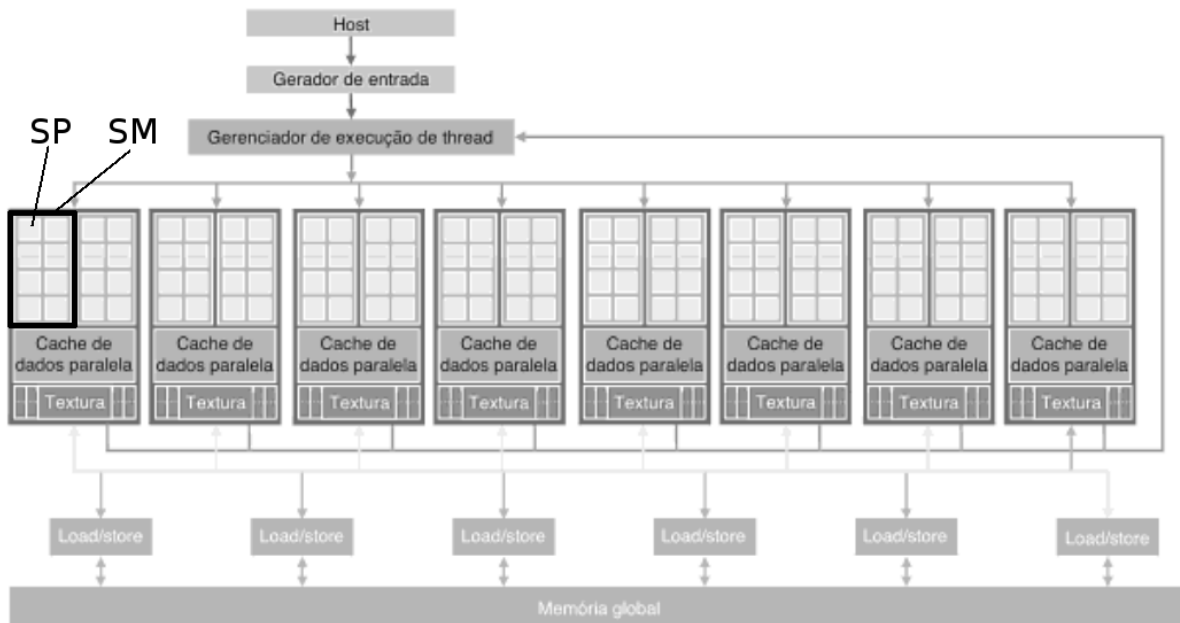
Vale ressaltar que algumas aplicações que são naturalmente criadas para funcionar em CPUs podem não funcionar bem em uma GPU. As placas gráficas são projetadas para aplicações de cálculo numérico, onde repetidas iterações de uma mesma parte da aplicação são realizadas. Por isso, é normal que um programa que utilize muito a GPU dependa ainda da CPU para realizar determinadas tarefas.

3.2 As arquiteturas das GPUs NVIDIA com Suporte a CUDA

As atuais GPUs da NVIDIA são baseadas em arquiteturas que suportam CUDA, desde a GeForce 8800 (NICKOLLS; DALLY, 2010) - também chamada G80 - lançada em 2006.

A [Figura 9](#) apresenta a arquitetura de uma placa gráfica preparada para CUDA. Pode-se ver que ela é organizada em 8 blocos; cada um desses blocos possui um par de multiprocessadores de *streaming* (SMs), o que, entretanto, varia de uma geração de

Figura 9 – Esquema da arquitetura de uma GPU que utiliza CUDA.



Fonte: [Kirk e Hwu \(2011\)](#)

GPUs para outra. Cada SM da GPU possui 8 processadores de *streaming* (SPs), também chamados de núcleos CUDA (*CUDA cores*); a quantidade de núcleos em cada SM também varia de uma GPU para outra. Cada GPU CUDA vem com uma determinada quantidade de memória DRAM GDDR (*Dynamic Random Access Memory Graphics Double Data Rare*), representada na [Figura 9](#) como “Memória global”, que possui uma grande largura de banda para transmitir dados para os núcleos - algumas informações sobre largura de banda foram apresentadas na [seção 3.1](#).

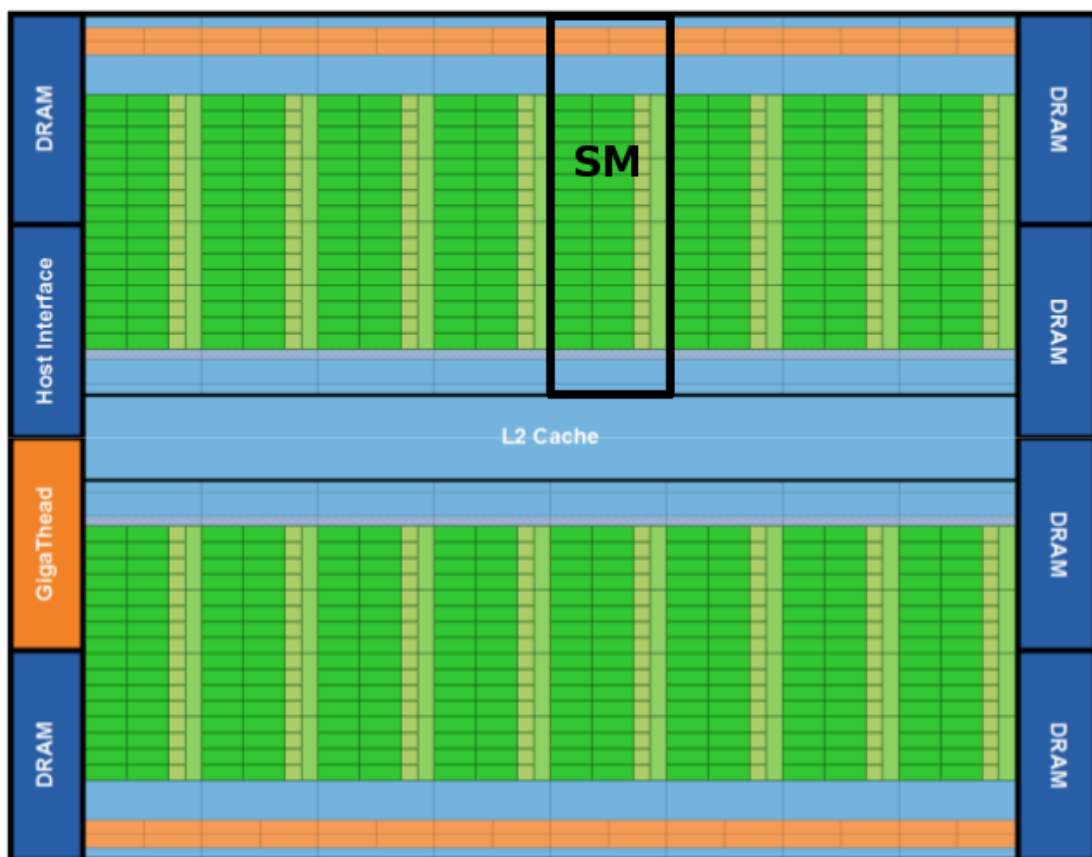
3.2.1 Arquitetura Fermi

Depois da G80, a NVIDIA vem aprimorando suas arquiteturas CUDA para prover cada vez mais poder de processamento às suas GPUs e aumento de desempenho à computação em GPU e jogos digitais. Com a evolução da G80, em 2008 a NVIDIA chegou à segunda geração de arquitetura CUDA: a arquitetura GT200, introduzida com as placas GTX 280, Quadro FX 5800 e Tesla T10 ([NVIDIA, 2009](#)), havendo um aumento de núcleos CUDA nos SMs e melhorias de acesso à memória. Com tudo que foi desenvolvido a partir das inovações de CUDA, a NVIDIA sempre procurou melhorar a eficiência e programabilidade da arquitetura.

Em 2009, uma nova arquitetura foi lançada, chamada Fermi. Com várias modificações e melhorias, a terceira geração de arquitetura CUDA trouxe a terceira geração de multiprocessadores de *streaming*, com mais núcleos CUDA e unidades lógicas aritméticas

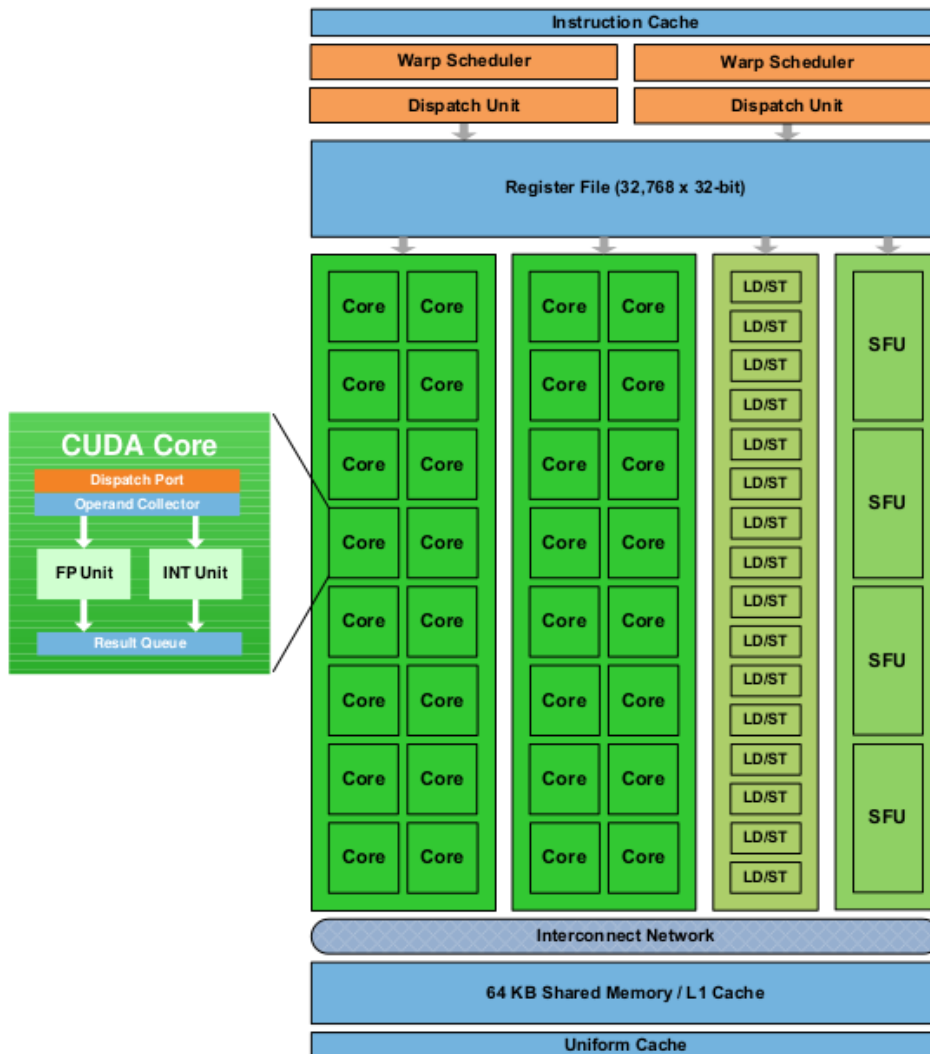
(ULAs) com suporte a instruções de 32 bits. Fermi implementa o novo padrão IEEE 754-2008 *floating-point* (IEEE, 2014), que especifica formato de números, operações básicas, converções, etc, aumentando a acurácia para operações aritméticas de precisão dupla e simples. Na Figura 10, tem-se uma representação da arquitetura Fermi, com destaque demonstrando um multiprocessador (SM), o qual é apresentado em detalhes na Figura 11. A primeira placa Fermi foi implementada com 3 bilhões de transistores, 512 núcleos CUDA - organizados em 16 SMs, cada um com 32 núcleos - e 6 blocos de memória de 64 bits com interface de 384 bits de comunicação suportando 6GB DRAM GDDR5. Uma interface *host* liga CPU e GPU via PCI-Express, e o distribuidor *GigaThread* atribui os blocos de *threads* para os multiprocessadores de *streaming*.

Figura 10 – Arquitetura Fermi.



Fonte: NVIDIA (2009)

Pode-se ver na Figura 11 cada componente de um SM da arquitetura Fermi. São 32 núcleos CUDA em cada SM, com uma unidade de inteiro e uma de ponto flutuante; 16 unidades de *load/store*, calculando endereços para 16 *threads* por ciclo; 4 unidades de função especial (SFUs), que realizam operações de seno, cosseno, raiz quadrada, etc, sendo que um SFU realiza uma operação por ciclo para uma *thread*; 2 *warp scheduler*, responsáveis pela execução simultânea de 2 *warps* - um *warp* de 32 *threads* executa em

Figura 11 – Multiprocessador de *streaming* da arquitetura Fermi.

Fonte: [NVIDIA \(2009\)](#)

16 núcleos CUDA, 16 unidades de *load/store* ou 4 SFUs - exceto se um *warp* realizar instruções de precisão dupla, pois não há suporte para isso; uma memória *cache* L1 de 64KB compartilhada, que pode ser usada por *threads* de um mesmo bloco para reduzir o acesso à memória.

3.2.2 Arquitetura Kepler

Mais recentemente, em 2012, a NVIDIA lançou a Kepler ([NVIDIA, 2012](#)), mais uma inovação da arquitetura CUDA. Com mais de 7 bilhões de transistores, a Kepler GK110 alcança maiores performances com um menor custo energético, o que foi o foco de seu projeto. Na [Figura 12](#) é mostrada uma implementação completa da arquitetura Kepler, com 15 multiprocessadores de *streaming* (SMX) e 6 controladores de memória de

64 bits; implementações diferentes podem ter um número menor de SMXs. Também há uma interface PCI-Express 3.0 para comunicar com a CPU e o *GigaThread Engine* que envia os blocos de *threads* para os SMXs.

Figura 12 – Arquitetura Kepler.



Fonte: [NVIDIA \(2012\)](#)

A [Figura 13](#) mostra o multiprocessador de *streaming*, o qual possui mais recursos de processamento em relação ao visto no multiprocessador da arquitetura Fermi na [Figura 11](#). São 192 núcleos CUDA de precisão simples, mantendo o padrão IEEE 754-2008; 64 unidades de precisão dupla; 32 unidades de função especial; e 32 unidades de carga/armazenamento. No SMX da Kepler também há mais *warp schedulers*; são 4, cada um com duas unidades de despacho de instrução. Cada *warp scheduler* seleciona um grupo de *threads* e duas instruções independentes para serem despachadas para execução. No entanto, é permitido que instruções de precisão dupla sejam executados com outras instruções, diferentemente da implementação da arquitetura Fermi.

Figura 13 – Multiprocessador de *streaming* da arquitetura Kepler.

Fonte: [NVIDIA \(2012\)](#)

3.2.3 Evolução das GPUs NVIDIA

A NVIDIA vem constantemente inovando com suas placas gráficas. Na [Tabela 1](#) é possível ver um comparativo das arquiteturas apresentadas quanto ao número de transistores na placa e núcleos CUDA, o que representa o crescimento do poder computacional dos dispositivos. Como visto nessa seção, a cada novo projeto, otimizações são feitas para se chegar a arquiteturas aprimoradas que possam melhorar o desempenho das aplicações que utilizam CUDA.

Tabela 1 – Aumento do número de transistores e núcleos CUDA nas GPUs NVIDIA.

Ano	Produto	Transistores	Núcleos CUDA
2006	GeForce 8800	681 milhões	128
2008	GeForce GTX 280	1,4 bilhões	240
2009	Fermi	3 bilhões	512
2012	Kepler GK110	7,1 bilhões	2880

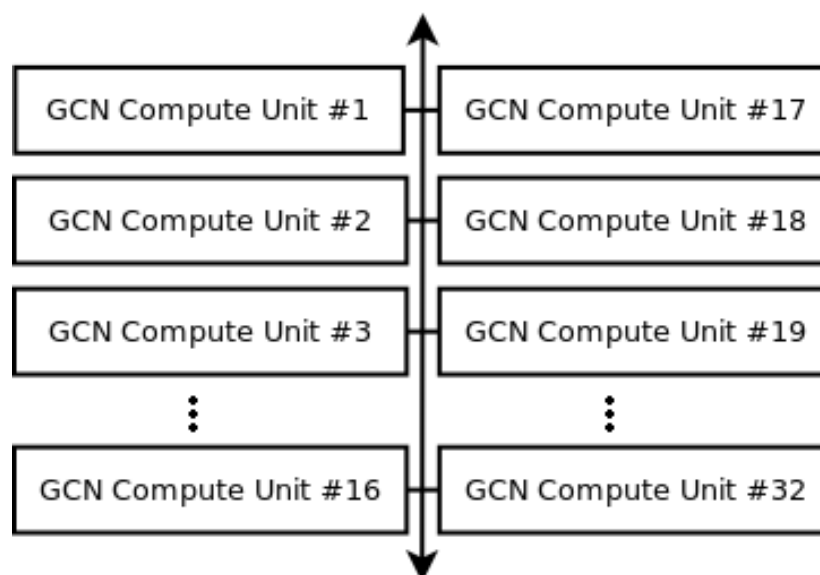
Fonte: [Nickolls e Dally \(2010\)](#); [NVIDIA \(2009\)](#); [NVIDIA \(2012\)](#)

3.3 Comparativo entre a Organização de GPUs AMD e NVIDIA

De maneira similar à NVIDIA, a AMD organiza a arquitetura de suas GPUs em muitos núcleos de processamento divididos em blocos. A arquitetura GCN (*Graphics Cores Next*) da AMD foi a sua primeira criada para computação de propósito geral ([AMD, 2014b](#)).

Na [Figura 14](#), que apresenta parte da organização da arquitetura da placa AMD Radeon HD 7970, se vê que existem 32 GCN *Compute Units* - ou simplesmente CU - interligados. A GPU em questão contém 2048 processadores de *streams*, sendo que cada CU possui 64 ([AMD, 2014a](#)). As setas na figura representam a ligação das unidades de processamento com os demais componentes da arquitetura, como memória *cache*. Esta organização adotada pela AMD, utilizando GCN *Compute Units* com muitos processadores de *streams*, é semelhante aos multiprocessadores (SMs) com muitos CUDA *cores* implementados nas placas NVIDIA

Figura 14 – Organização de unidades de processamento em arquitetura de GPU AMD.



Fonte: adaptado de [AMD \(2012\)](#)

3.4 Interfaces de Programação para GPU

Existem variadas tecnologias para se programar para GPU, dentre as quais três delas são bem difundidas e utilizadas atualmente: *Open Compute Language* (OpenCL), OpenACC e *Compute Unified Device Architecture* (CUDA).

OpenCL, que é gerenciada pelo Khronos Group ([KHRONOS, 2014](#)) e implementado por outras empresas de tecnologia, é o primeiro padrão aberto multi-plataforma para programação paralela entre CPUs, GPUs e outros processadores. Tal característica oferece uma boa portabilidade aos códigos escritos utilizando OpenCL, permitindo que uma mesma versão possa ser executada em diferentes plataformas, como GPUs AMD e NVIDIA ([HERDMAN et al., 2012](#)). Utilizado para placas gráficas, OpenCL utiliza códigos sequenciais que executam em um *host* (CPU) e códigos paralelos que executam no *device* (GPU) através de *threads* que compartilham informações e executam concorrentemente.

OpenACC ([OPENACC.ORG, 2014](#)) é uma organização sem fins lucrativos fundada pelo grupo de companhias que desenvolvem as especificações OpenACC para programação. A interface possui um modelo de programação de alto nível, utilizando um conjunto de diretivas de programação para especificar regiões de laços paralelos ao compilador - o qual realiza o trabalho de mapear a computação para dentro da GPU - sem necessidade de que programadores escrevam códigos específicos para o *device* ([REYES et al., 2013](#)) ([OPENACC.ORG, 2014](#)).

CUDA ([NVIDIA, 2014c](#)), a interface de programação que implementa o código da aplicação deste trabalho, é apresentada na seção a seguir.

3.5 Compute Unified Device Architecture (CUDA)

CUDA é uma plataforma de computação paralela e um modelo de programação para GPUs criada pela NVIDIA para as suas mais novas arquiteturas de placas gráficas ([NVIDIA, 2014c](#)). CUDA fornece extensões para linguagens de programação de alto nível, como C, C++ e Fortran, permitindo que estas sejam utilizadas para programar para GPUs. A fim de aumentar o poder de processamentos de seus chips gráficos, a NVIDIA vem constantemente aperfeiçoando as arquiteturas de suas GPUs, a API de programação CUDA e o seu compilador para CUDA (NVCC).

Ao programar usando CUDA, o sistema de computação consiste em um *host* (CPU) e um *device* (GPU). Para definir funções distintas de *host* e de *device*, CUDA oferece três palavras-chaves para identificar as funções declaradas, conforme é mostrado na [Tabela 2](#). A palavra-chave `__host__` é usada para declarar funções que serão chamadas e executada apenas no *host*. Por padrão, toda função que não possui uma dessas três palavras-chave

é considerada como uma função de *host*. A palavra-chave `__device__` define que uma função será chamada e executada apenas no *device*. No *host*, o programador lança a execução de um grande número de *threads* que executam paralelamente no *device*. É preciso definir uma função de *kernel* utilizando a palavra-chave `__global__`, conforme é visto na Figura 15. Então, para lançar a execução das *threads* paralelas, é feita uma chamada a esta função, como mostra a Figura 16.

Tabela 2 – Palavras-chave qualificadoras para declaração de funções, fornecidas por CUDA.

Palavra-chave		Função é executada no:	Função é chamada no:
<code>__device__</code>	<code>void deviceFunc()</code>	device	device
<code>__global__</code>	<code>void kernelFunc()</code>	device	host
<code>__host__</code>	<code>void hostFunt()</code>	host	host

Fonte: Kirk e Hwu (2011)

Figura 15 – Definindo uma função de *kernel*.

```
// Definição do kernel
__global__ void funcaoDeKernel(float A, float B, float C){
    ...
    ...
    ...
}
```

Fonte: adaptado de NVIDIA (2013a)

Figura 16 – Invocando um *kernel*.

```
{
    ...

    dim3 threadsPorBloco(4, 4);
    int numBlocos = 1;
    // Invocando o kernel
    funcaoDeKernel<<<numBlocos, threadsPorBloco>>>(A, B, C);

    ...
}
```

Fonte: adaptado de NVIDIA (2013a)

Pode-se ver ainda na Figura 16 que o *kernel* é chamado utilizando dois parâmetros de configuração, que apenas são utilizados em tempo de execução para organizar as suas *threads*; são eles: “numBlocos”, que define a dimensão do *grid*, e “threadsPorBloco”, que define a dimensão do bloco. Cada um pode ser declarado como um tipo básico “int”, ou

como um tipo “dim3” que é uma *struct* definida por CUDA. No exemplo, a dimensão dos blocos da grade (“threadsPorBloco”) é definido como um arranjo de 4 vezes 4 *threads*. A dimensão da grade (“numBlocos”) poderia ser definida de forma análoga, bem como utilizar uma terceira dimensão para qualquer um dos parâmetros.

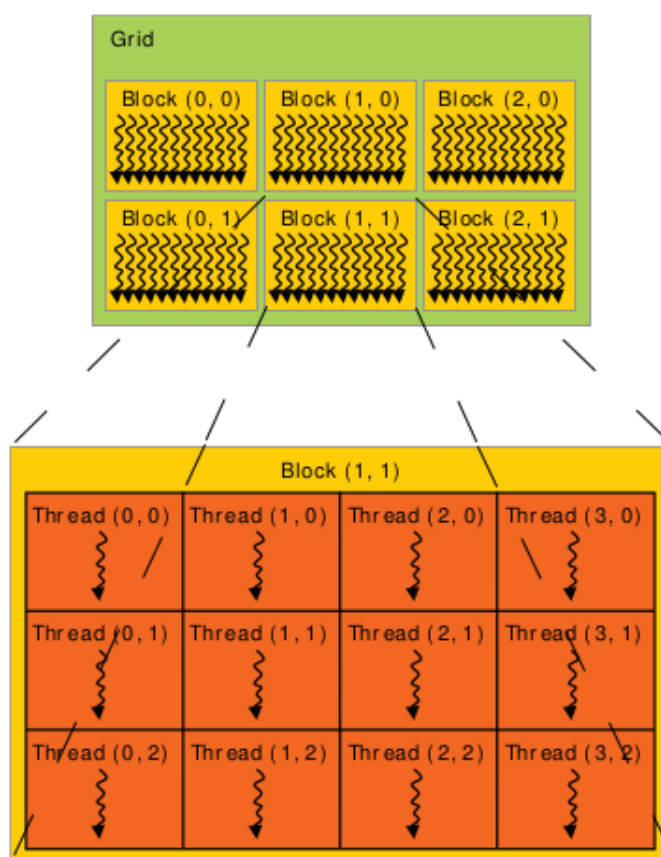
As *threads* de um *kernel* são executadas nos processadores da GPU em um modelo conhecido como único programa e múltiplos dados (SPMD - *Single Program, Multiple Data*). As unidades de processamento paralelo executam o mesmo código sobre diferentes conjuntos de dados, e não necessariamente precisam executar a mesma instrução simultaneamente (KIRK; HWU, 2011).

A Figura 17 mostra como as *threads* são organizadas dentro de um *grid*. Cada *thread* possui um identificador único dentro do bloco, estabelecido pelas palavras-chave *threadIdx.x*, *threadIdx.y* e *threadIdx.z*. Nota-se que na figura apenas duas coordenadas identificam as *threads*, o que significa que a terceira coordenada não foi utilizada. Da mesma forma é feito o mapeamento dos blocos dentro da grade, e cada bloco pode ser identificado através de palavras-chaves próprias, as quais são: *blockIdx.x*, *blockIdx.y* e *blockIdx.z*.

Porém, para diferenciar cada *thread* dentro de um *kernel*, uma combinação dos identificadores *threadIdx* e *blockIdx* deve ser utilizada. A citação a seguir faz uma breve consideração sobre isso.

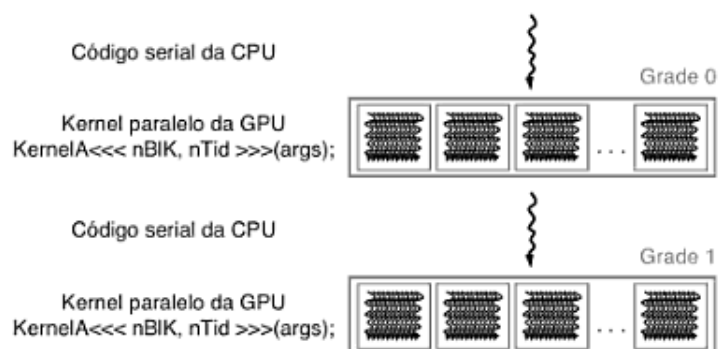
A configuração de execução do *kernel* define as dimensões de uma grade e seus blocos. Coordenadas exclusivas nas variáveis *blockIdx* e *threadIdx* permitem que as *threads* de uma grade identifiquem a si mesmas e seus domínios. É responsabilidade do programador usar essas variáveis nas funções do *kernel* de modo que as *threads* possam identificar corretamente a parte de dados que irão processar. Esse modelo de programação obriga o programador a organizar as *threads* e seus dados em organizações hierárquicas e multidimensionais. (KIRK; HWU, 2011, p. 61)

Por fim, para concluir esta apresentação sobre o modelo de programação CUDA, a Figura 18 representa a execução de um programa CUDA que inicia executando um código serial no *host* (CPU) e em um certo momento é disparada a execução de um *kernel*. Neste momento, o código do *kernel* passa a ser executado do *device* por centenas e milhares de *threads* que compoem um *grid* até que o trabalho seja concluído e o programa continue a executar no *host*. Esse processo pode acontecer várias vezes durante a execução de um programa.

Figura 17 – Esquema de organização de *threads* CUDA em um *grid* de *thread blocks*.

Fonte: NVIDIA (2013a)

Figura 18 – Execução de um programa CUDA comum.



Fonte: Kirk e Hwu (2011)

4 Metodologia

Para este trabalho, utilizou-se um protótipo do OLAM que foi implementado por Schepke (2012) em linguagem C utilizando as extensões de CUDA. Tal versão simplificada, refeita a partir da versão original escrita na linguagem Fortran, mantém as principais funções da aplicação, como decomposição de domínios, refinamento de malhas, distribuição paralela de dados e todas as estruturas de dados e funções necessárias.

Schepke (2012) implementou em seu trabalho todas as funcionalidades necessárias para executar o OLAM em GPUs CUDA. Alocação de memória em GPU para reservar endereços de memória para dados copiados da CPU para a GPU. Cópia de memória entre CPU e GPU nos dois sentidos, pois os dados precisam ser copiados para a GPU para que esta realize cálculos sobre tais dados e retorne os resultados para a CPU. Funções de *kernel* que são executadas em GPU, sendo três implementadas e que demandam um grande parte do tempo total de processamento da aplicação. Esses *kernels* possuem fortes dependências de dados entre si, bem como dos dados copiados da CPU para a GPU, devido ao uso de malhas não estruturadas para representar o globo terrestre no OLAM.

Visando realizar um estudo adequado sobre o modelo OLAM, foram definidos métodos analíticos a serem seguidos e estratégias de modificação do código da aplicação. As análises contribuem com a identificação de possíveis aspectos de implementação, configuração de execução em GPU e compilação do modelo que, se modificados, poderiam aumentar o desempenho do mesmo. Os critérios de análise, métodos de estudo aplicados e adequações de código são descritos a seguir:

- Testes de desempenho da aplicação com medida de tempo total de execução, bem como avaliar o tempo total dos *kernels*.
- Realizar análise da execução via *profiling*, utilizando ferramenta de visualização da execução em GPU.
- Analisar a ocupação da GPU durante execução do modelo, bem como utilizar ferramenta para estimar as configurações adequadas para prover a maior ocupação possível da GPU. Com isso, diferentes tamanhos de blocos de *threads* são aplicados a cada *kernel* implementado.
- Analisar as dependências de dados das funções executadas em GPU, de modo a investigar a viabilidade de sobrepor cópia de memória e execução de *kernels*.
- Verificar a estruturação interna de cada *kernel* para buscar melhor implementação e acesso a dados.

- Analisar a possibilidade de dividir as funções executadas em GPU em mais funções, visando a execução concorrente de *kernels* para atingir um possível ganho de desempenho.

4.1 Recursos de Hardware

Todos os testes foram realizados utilizando uma *workstation* Dell Precision T7600 da Universidade Federal do Pampa. A máquina possui 128 GB de memória RAM, dois processadores Intel Xeon E5-2650, cada um com 8 *cores* e suporte à tecnologia *Hyper-Threading*, totalizando 32 núcleos - 16 físicos e 16 lógicos. Os principais recursos disponíveis são duas GPUs da NVIDIA, uma Tesla C2075 e uma Quadro 5000, dispostas na [Tabela 3](#) com algumas especificações de *hardware* sendo mostradas.

Tabela 3 – GPUs utilizadas para os testes.

	Tesla C2075	Quadro 5000
Capacidade de computação	2.0	2.0
Número de CUDA <i>cores</i>	448	352
Taxa de <i>clock</i> da GPU	1147 MHz	1026 MHz
Número de multiprocessadores (SM)	14	11
Número de CUDA <i>cores</i> por SM	32	32
Memória global	6 GB GDDR5	2,5 GB GDDR5
Taxa de <i>clock</i> da memória	1566 MHz	1494 MHz
Largura do barramento de memória	384 bits	320 bits
Memória constante	65536 bytes	65536 bytes
Memória compartilhada por bloco	49152 bytes	49152 bytes
Número de registradores por bloco	32768	32768
Tamanho do <i>warp</i>	32	32
Máximo de <i>warp</i> suportados por um SM	48	48
Número máximo de <i>threads</i> por SM	1536	1536
Número máximo de <i>threads</i> por bloco	1024	1024
Dimensão máxima de um bloco de <i>threads</i> (x, y, z)	(1024, 1024, 64)	(1024, 1024, 64)
Dimensão máxima do tamanho de um <i>grid</i> (x, y, z)	(65535, 65535, 65535)	(65535, 65535, 65535)

Fonte: [NVIDIA \(2011\)](#); [NVIDIA \(2014b\)](#)

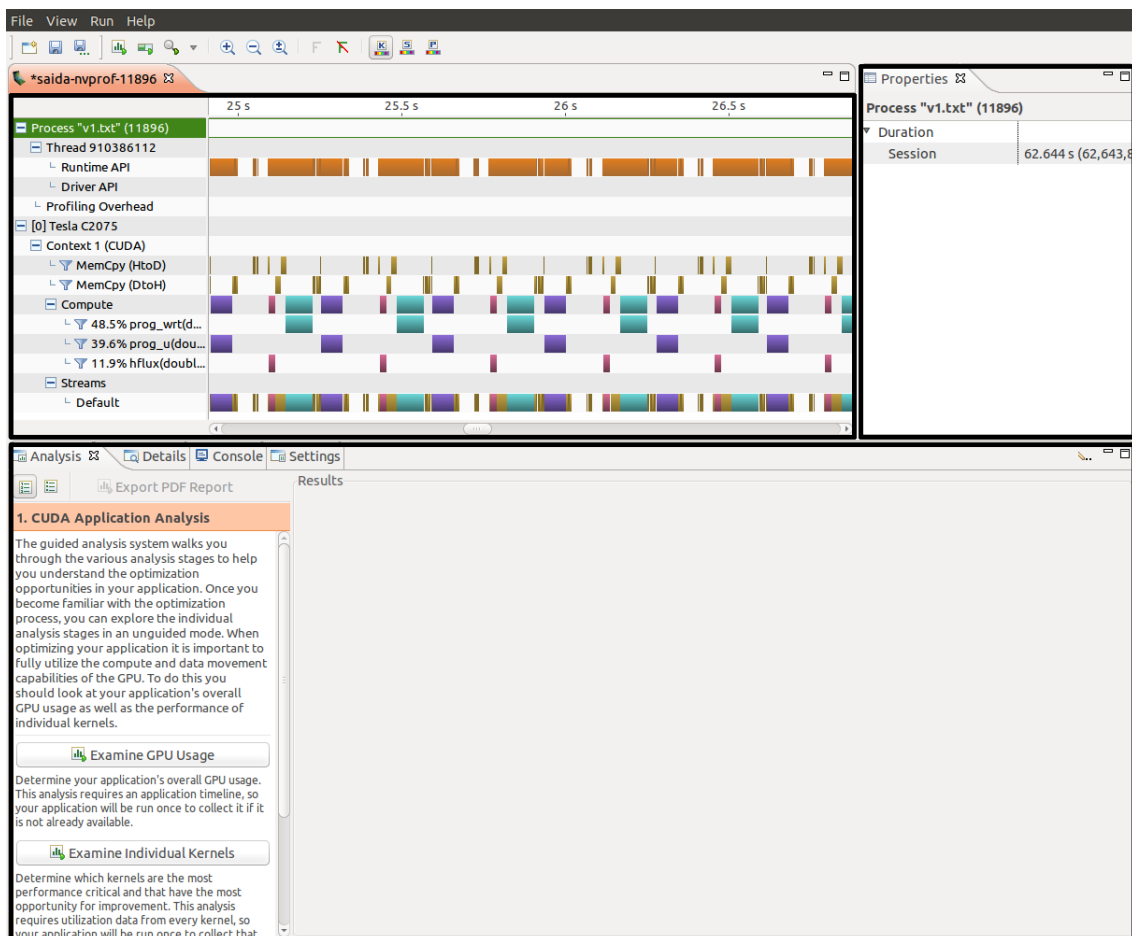
4.2 Recursos de Software

A máquina funciona com sistema operacional GNU/Linux Ubuntu 14.04 LTS de 64 bits. Possui instalado compilador GCC versão 4.8.2 para linguagem C e o NVCC V6.0, compilador NVIDIA para CUDA.

Como recursos para realizar *profiling* da aplicação, foram utilizados o *nvprof* e o *NVIDIA Visual Profiler*. O *nvprof* contribui para executar o programa e coletar dados que auxiliam na análise de desempenho. Os dados de saída foram direcionados para um

arquivo que pudesse ser importado no *NVIDIA Visual Profiler*, o qual provê uma interface gráfica intuitiva de grande ajuda na análise dos dados coletados. A [Figura 19](#) mostra a tela do *NVIDIA Visual Profiler*, onde se vê no retângulo superior esquerdo a linha do tempo da execução de um programa em GPU, identificando os intervalos de tempo onde os *kernels* executaram, as cópias de memória, etc. No retângulo superior direito, são apresentadas as propriedades de cada item da linha do tempo ao lado - se a linha de um *kernel* é selecionada, dados como número de chamadas e tempo total de execução são mostrados. No retângulo inferior, a aba *Analysis* é usada para controlar a análise da aplicação, e pode ser feita como o usuário preferir, ou este pode seguir um guia que o orienta. Já a aba *Details* mostra uma tabela com as informações de tempo, ocupação, etc, de todas ocorrências de um item da linha do tempo, como as chamadas de *kernels* e cópias de memória. As abas *Console* e *Settings* são utilizadas quando se executa uma aplicação a partir do próprio *NVIDIA Visual Profiler*, logo, não se aplicam neste trabalho, o qual executou *profiling* do programa utilizando o *nvprof*.

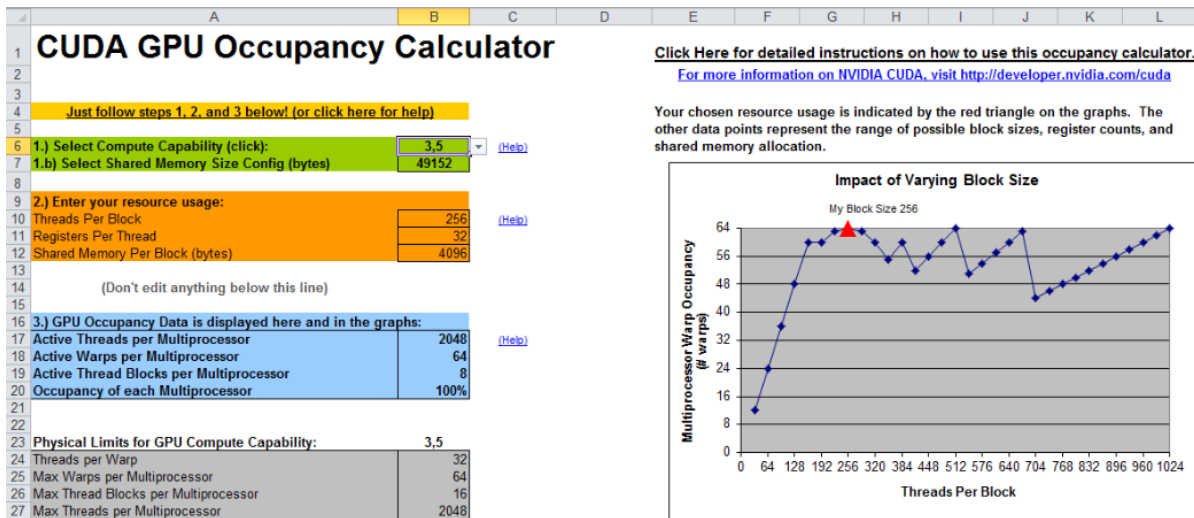
Figura 19 – *NVIDIA Visual Profiler*.



Outra ferramenta utilizada é a *CUDA GPU Occupancy Calculator* da NVIDIA, feita no *Microsoft Excel*, para cálculo de ocupação de GPU, disponível juntamente com

o CUDA Toolkit (NVIDIA, 2014a). Informando na planilha a capacidade de computação da GPU, o número de *threads* por bloco e o número de registradores utilizados por *thread* (obtido com a opção `-ptxas-options=-v` para o compilador), a planilha calcula a ocupação de cada multiprocessador da GPU para um determinado *kernel*. A Figura 20 apresenta uma parte da planilha, onde as seguintes informações devem ser especificadas adequadamente: na célula B6, deve-se selecionar a capacidade de computação da GPU; na célula B7, o tamanho da memória compartilhada em tempo de execução; na célula B10, o número de *threads* por bloco (triângulo vermelho no gráfico da figura); na célula B11, o número de registradores necessários para cada *thread* do *kernel*; na célula B12, a quantidade de memória compartilhada por bloco de *thread*. Na célula B20, é mostrada a estimativa de ocupação para um multiprocessador (SM) da GPU, calculada dividindo o número de *warps* ativos pelo número máximo de *warps* (célula B18 dividida pela célula B25). No gráfico da imagem, é mostrado um intervalo dos tamanhos de blocos possíveis, onde os pontos de pico representam a maior ocupação possível para um dado *kernel*.

Figura 20 – CUDA GPU Occupancy Calculator.



5 Estudos Realizados e Análise dos Resultados

Este capítulo apresenta os resultados obtidos a partir dos estudos realizados sobre técnicas de otimização de desempenho para o modelo OLAM. Também são discutidos tais resultados, bem como analisados os motivos que levaram até os mesmos, e demonstradas as alterações realizadas na aplicação visando melhorar sua performance.

5.1 Fase 1 - Testes Iniciais

Primeiramente, alguns testes iniciais foram realizados utilizando as duas GPUs descritas na [Capítulo 4](#). O objetivo destes testes foi analisar o desempenho do OLAM nesse tipo de arquitetura. Os casos de teste consistem em variar o número de *threads* por bloco com diferentes resoluções horizontais da malha global. O número de *threads* por bloco foi definido em 128, 256, 512 e 1.024, e as resoluções horizontais foram de 100 km, 50 km, 40 km e 30 km. Com isso, tem-se um total de 16 casos de teste. Nestes testes iniciais, optou-se por não realizar o refinamento da malha em tempo de execução. O número de níveis verticais adotados em todos os casos testados foi de 28. Além disso, para todos os casos, 72 horas de simulação da atmosfera foram considerados.

A [Figura 21](#) apresenta os tempos totais de execução obtidos para a execução do modelo utilizando-se a placa Tesla. Neste caso, os tempos de execução foram bastante próximos, exceto ao utilizar 1.024 *threads* por bloco, sendo analisado no uso de outra GPU se isso se repetiria. Os casos de testes de 1.024 *threads* apresentam desempenho quase 50% mais eficiente em relação aos outros casos com a mesma resolução horizontal.

A [Figura 22](#) mostra os tempos totais de execução obtidos com o uso da placa Quadro nas simulações. Neste segundo caso, tem-se apenas os tempos de execução para as resoluções de 100 km e de 50 km, pois houve falta de memória para armazenar os dados na GPU quando resoluções maiores foram utilizadas. Isso se deve ao fato de que quanto maior a resolução da malha, maior é a quantidade de dados a serem computados, implicando em mais dados a serem armazenados em memória. Da mesma forma que o caso da [Figura 21](#), o número de *threads* por bloco não interfere significativamente no tempo total de execução do modelo, e os casos de testes que utilizam 1.024 *threads* também apresentaram tempo de execução reduzido, agora com pouco mais de 55% em relação aos outros casos a mesma resolução horizontal.

Os fatores que reduzem o tempo de execução para este tamanho de bloco, bem como suas implicações, são discutidos na próxima seção. Pequenas diferenças nos tempos

Figura 21 – Resultados dos testes na placa Tesla C2075.

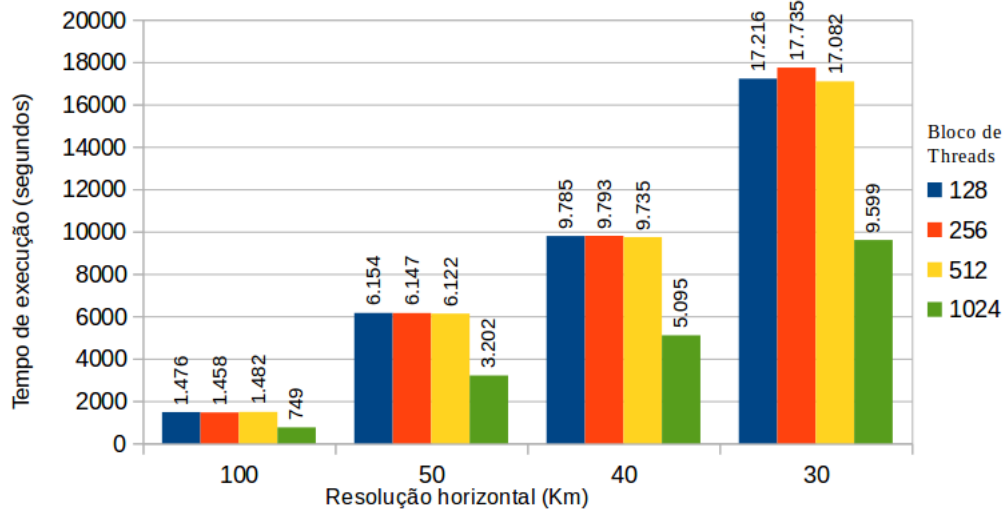
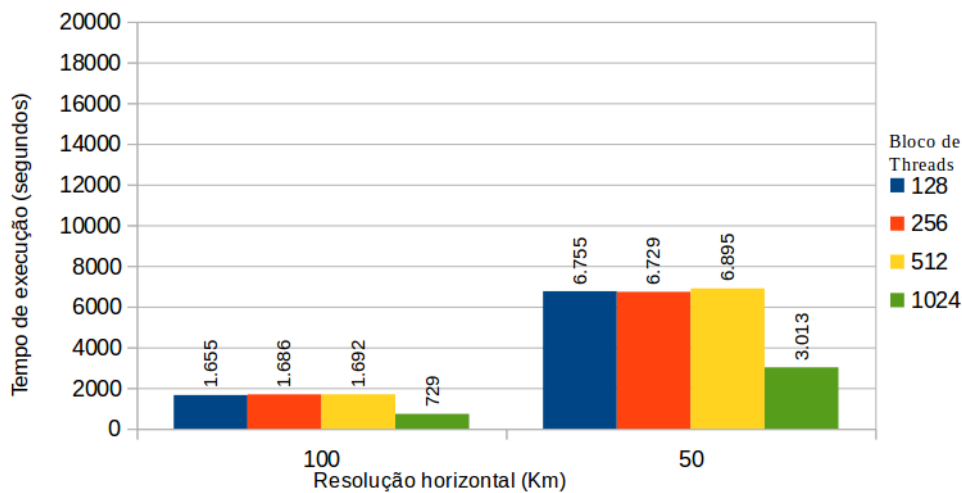


Figura 22 – Resultados dos testes na placa Quadro 5000.



obtidos ocorrem devido a uma melhor adequação da carga de processamento entre as *threads*, o que será melhor investigado na [seção 5.2](#).

A [Tabela 4](#) mostra os tempos da execução sequencial do OLAM, para cada resolução previamente definida, usando somente um *core* da CPU da arquitetura considerada. Comparando-se os tempos de execução dos testes realizados com GPUs em relação ao tempo sequencial, nota-se que o ganho de desempenho utilizando-se GPUs não é significativo. Os testes nas duas GPUs mostram que o ganho de desempenho é de quase 3X (cerca de 60%) em relação ao tempo sem paralelismo.

A comparação dos resultados em GPU com os resultados em CPU sem paralelismo indica que a utilização adequada das arquiteturas GPUs não foi realizada, pois o OLAM

Tabela 4 – Resultados dos testes do OLAM em CPU sem paralelismo.

Resolução (em km)	Tempo (em segundos)
100	3.959
50	16.003
40	25.378
30	45.075

não atinge um ganho expressivo de desempenho paralelo. Outras aplicações de GPU encontradas na literatura, a exemplo de *benchmarks* que utilizam o método de *Lattice-Boltzmann* em SUN, X. et al (2012), NITA, C. et al. (2013), e Kraus (2013), atingem ganhos de desempenho de 50X, 100X, 500X e até mais.

Como a GPU Tesla C2075 apresentou melhores resultados, isso devido as suas melhores configurações de *hardware*, a mesma passou a ser adotada nos demais experimentos apresentados a seguir.

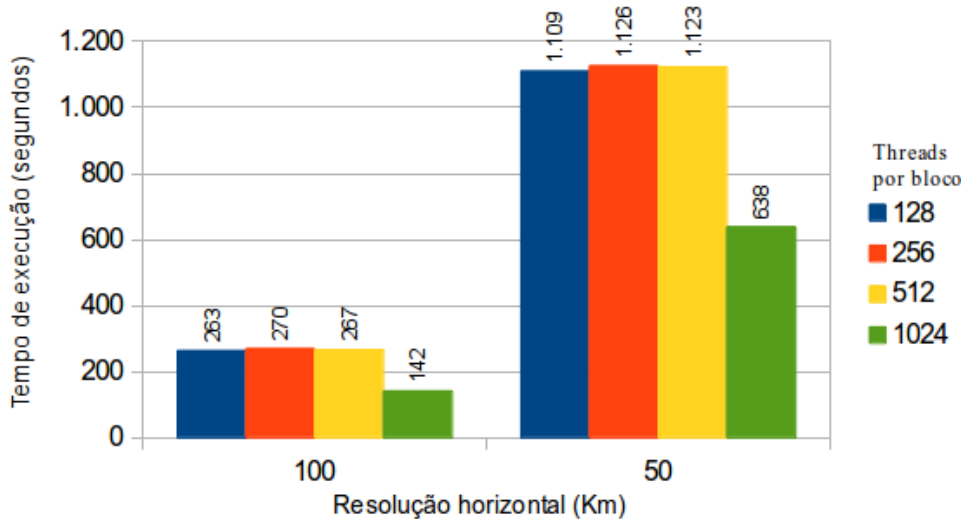
5.2 Fase 2 - Análise de Ocupação da GPU

Posteriormente, foi realizada uma análise utilizando a *CUDA GPU Occupancy Calculator* da NVIDIA. Três *kernels* CUDA são implementados na aplicação: *prog_wrt*, *prog_u* e *hflux*. Compilando o código com a opção `--ptxas-options=-v`, foram retornados os seguintes números de registradores para cada um deles: *prog_wrt*, 63 registradores; *prog_u*, 63 registradores; e *hflux*, 35 registradores. Analisando a planilha, notou-se que a ocupação dos multiprocessadores da GPU, para blocos de 1.024 *threads*, era 0% para todos os *kernels*, o que poderia explicar os tempos de execução cerca de 50% menores para os casos de 1.024 *threads* dos testes mostrados nas Figuras 21 e 22. Considerando que o número de registradores por multiprocessador da GPU é de 32.768, o uso de 1.024 *threads* por bloco implica na não execução dos *kernels* da aplicação. Por exemplo, 1.024 *threads* multiplicados por 35 registradores do *kernel hflux* é igual a 35.840 registradores, logo, um bloco desse tamanho não pode ser atribuído a um multiprocessador por necessitar de mais registradores do que o disponível. O mesmo cálculo vale para os outros dois *kernels*, que necessitam de 63 registradores.

Para provar o que se concluiu na análise da planilha, foram executados os seguintes casos de teste: o modelo foi configurado para executar com duas resoluções horizontais distintas para a malha global, sendo 100km e 50km; para cada resolução, utilizou-se blocos de *threads* de tamanhos diferentes, sendo 128, 256, 512 e 1.024 *threads*. No total são oito casos de teste e para todos o tempo de simulação do globo terrestre é 12 horas. Os resultados são vistos na Figura 23, onde novamente, percebe-se que o tempo de execução para as simulações cujos *kernels* são lançados com blocos de 1.024 *threads* é consideravelmente menor do que os demais. De acordo com a planilha da NVIDIA, blocos de 1.024 *threads*

não executam por demandarem mais registradores do que o disponível em *hardware*. De modo a concretizar tal expectativa, outra análise foi necessária.

Figura 23 – Resultados dos casos de testes com 12 horas de simulação.



Essa outra análise consiste em realizar *profiling* da aplicação utilizando o *nvprof*. As informações foram direcionadas para um arquivo de saída que pode ser importado no *NVIDIA Visual Profiler*. Esta ferramenta permite analisar uma linha do tempo da execução da aplicação, bem como dos *kernels*, onde percebeu-se que, de fato, não foram executados os *kernels* com blocos de 1.024 *threads*, apenas os menores (128, 256 e 512 *threads*). Na Figura 24, observa-se a *timeline* da execução do programa, onde um retângulo destaca os três *kernels*. Já na Figura 25, a *timeline* não mostra nenhum *kernel* executado, de acordo com o que se esperava ao analisar a planilha da NVIDIA.

Figura 24 – *Timeline* do *NVIDIA Visual Profiler* para a aplicação com blocos de 128 *threads* para todos os *kernels*.

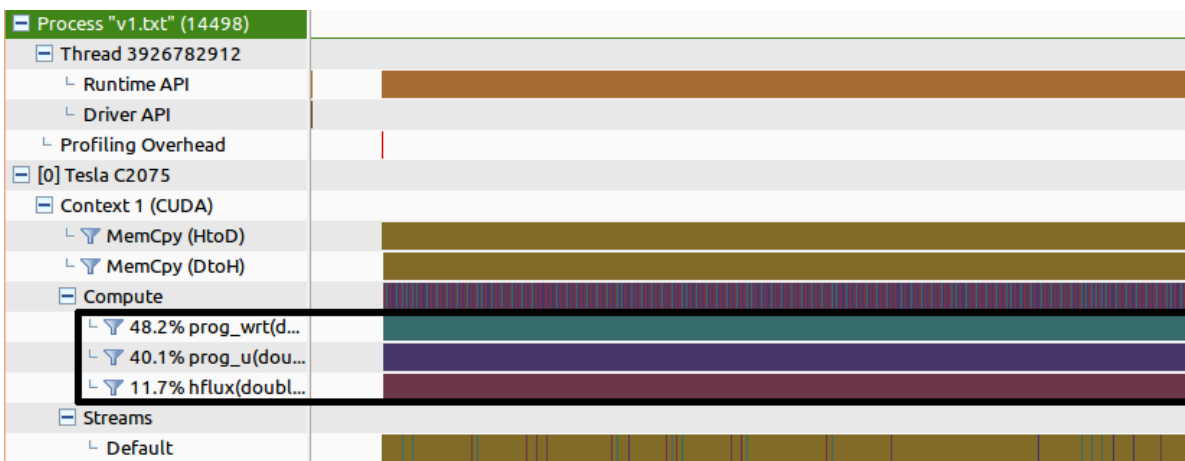
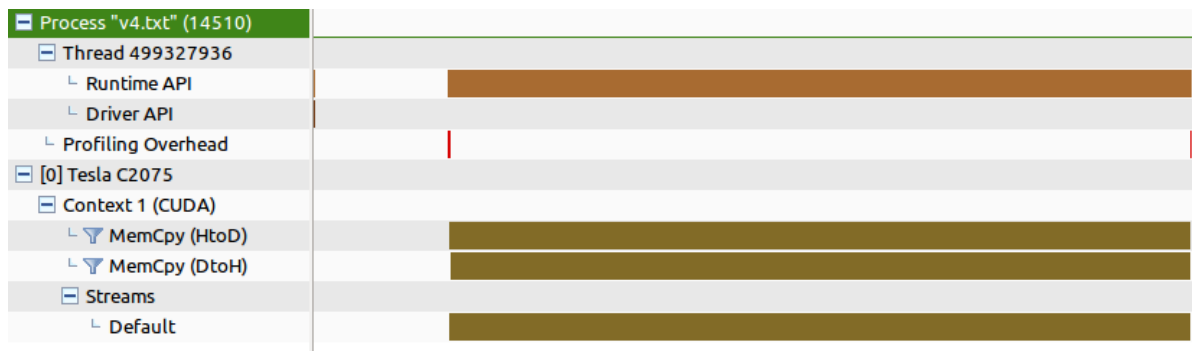


Figura 25 – *Timeline* do *NVIDIA Visual Profiler* para a aplicação com blocos de 1.024 *threads* para todos os *kernels*.

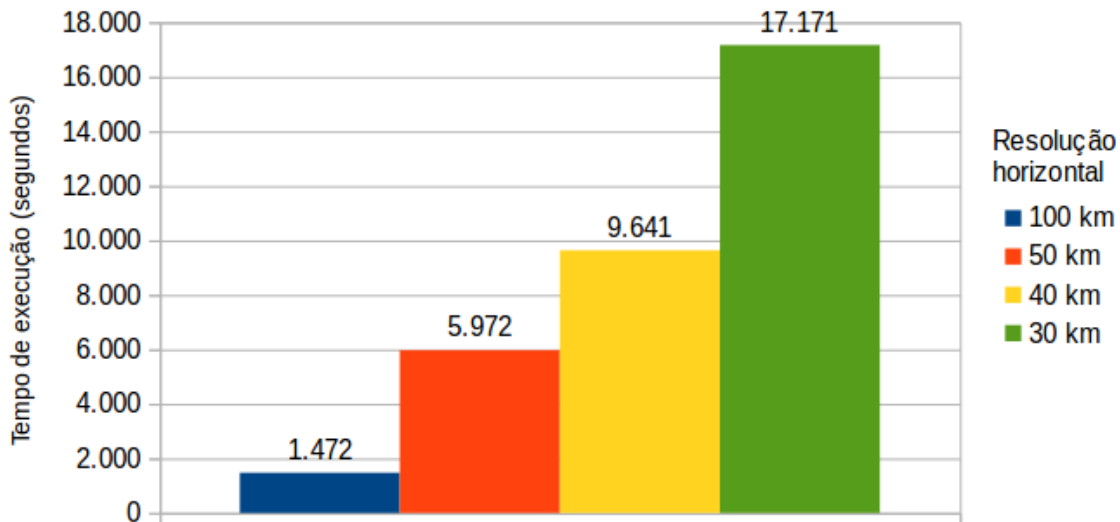


A partir disso, o modelo foi configurado para execução de modo que seus *kernels* consumissem a maior ocupação possível da GPU. Analisando a planilha de ocupação, constatou-se qual seria o tamanho do bloco de *threads* de cada *kernel* citado anteriormente para que isso ocorresse. Para os *kernels* *prog_wrt* e *prog_u* que utilizam 63 registradores por *thread*, blocos com 512 *threads* atingem o máximo de ocupação para estes *kernels*, sendo 33%. Para o *kernel* *hflux* que utiliza 35 registradores por *thread*, blocos com 896 *threads* atingem o máximo de ocupação para este *kernel*, sendo 58%. A Figura 26 mostra os resultados dos testes desta configuração do OLAM, onde utilizou-se os mesmos casos de teste mencionados na seção 5.1. Nota-se que os tempos de execução agora estão, em média, 1% menores que os testes iniciais vistos na Figura 21, demonstrando um pequeno ganho de desempenho.

Em relação à ocupação da capacidade de processamento dos multiprocessadores da GPU, é importante ressaltar que mesmo a ocupação dos SMs sendo alta, isso não necessariamente implica em melhorar a performance de uma aplicação. Um programa pode sofrer limitações quanto às cópias de memória entre CPU e GPU, e também por limitações do número de registradores necessários para cada *thread*.

Após esta fase, decidiu-se que o próximo passo nestes estudos seria analisar as funções de *kernel* da aplicação, tanto as chamadas de *kernel* e suas dependências de dados, quanto a estrutura interna de cada *kernel*, de modo a buscar meios de otimizar o código executado em GPU, bem como as cópias de memória entre *host* e *device* (e *vice-versa*).

Figura 26 – Testes realizados com blocos de *threads* que atingem a maior ocupação possível da GPU.



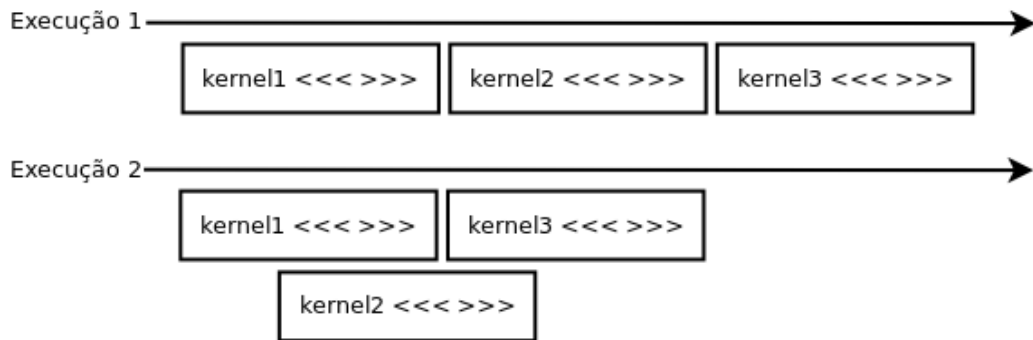
5.3 Fase 3 - Análise de Dependência de Dados em cada Kernel

Funcionalidades presentes na placa Tesla C2075 - mais especificadamente nas arquiteturas CUDA 2.0 (Fermi) e superiores - e que diminuem o tempo total de execução da uma aplicação quando podem ser utilizadas, são a execução sobreposta de cópia de memória e de *kernel*. Estes recursos permitem que a criação de diversos fluxos (*streams*) de execução realizem trabalho simultâneo na GPU. Um *stream* consiste em um conjunto de instruções, onde cada uma executa após o término da anterior.

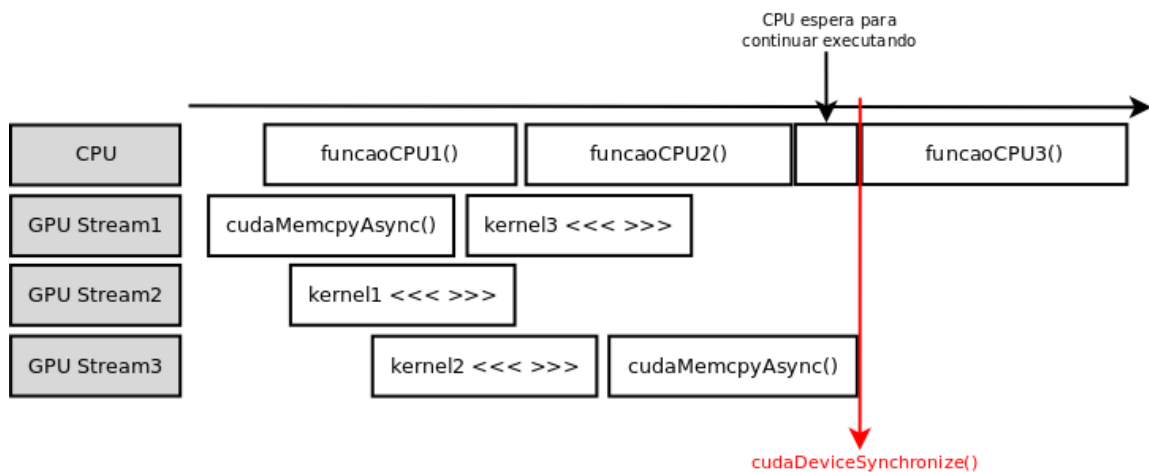
A Figura 27 representa duas linhas de execução. Em “Execução 1”, três *kernels* executam em ordem no mesmo fluxo. Já em “Execução 2”, dois *kernels* executam em um fluxo, enquanto o *kernel2* executa concorrentemente com os demais em um segundo fluxo, o que significa que o *kernel2* não possui dependência com os demais.

A Figura 28 representa como ocorre cópia de memória assíncrona entre CPU e GPU, de modo a sobrepôr a execução de *kernels* enquanto dados são copiados. Duas cópias de memória assíncrona também podem ocorrer ao mesmo tempo se uma for do *host* para o *device* (*HtoD*) e a outra do *device* para o *host* (*DtoH*), pois a arquitetura Fermi da placa Tesla C2075 possui duas *engines* de cópia de memória, uma *HtoD* e outra *DtoH*. Também observa-se, na Figura 28, a execução em CPU, pois a mesma é independente do que é executado em GPU. A não ser que uma barreira de sincronização (por exemplo, *cudaDeviceSynchronize*) sinalize que a CPU deve aguardar o término das tarefas executadas em GPU, a CPU continua a executar seu programa.

Tendo em vista tal características da GPU, foi executada uma análise do fluxo

Figura 27 – Execuções em GPU com um ou mais *streams*.

Fonte: adaptado de Taunay (2013)

Figura 28 – Execução assíncrona em GPU com múltiplos *streams*.

Fonte: adaptado de Taunay (2013)

de execução da função responsável por invocar os *kernels* do código, visando identificar dependências de dados dos mesmos. Este tipo de análise é fundamental quando pretende-se sobrepor cópia de memória e execução em GPU, o que ainda não está implementado na aplicação e poderia ser decisivo para melhorar sua performance.

Na Tabela 5, são mostradas as dependências identificadas em cada *kernel* da aplicação. Os três *kernels* são invocados na mesma ordem em que aparecem na tabela, e possuem dependências em todos seus dados. Os tipos de dependência de um *kernel* se devem a cópias de memória entre *host* e *device* (nos dois sentidos, *HtoD* e *DtoH*) e à utilização de dados em outro(s) *kernel(s)*. Exemplos: o *kernel hflux* executa apenas após cópias de memória *HtoD* de *var->rho_s_temp*, *var->umaru_temp* e *hflux_struct*; o *kernel prog_wrt* executa apenas depois do *kernel hflux* alterar os valores de *var->umaru_temp*, *var->hnum_w_temp* e *var->hflux_struct*.

Tabela 5 – Dependências de dados dos *kernels* implementados no OLAM.

Kernel	Dependência anterior	Dependência posterior
hflux	var->rho_s_temp var->umaru_temp hflux_struct	var->rho_s_temp var->umaru_temp var->hcnun_u_temp var->hcnun_w_temp var->hflux_t_temp
prog_wrt	var->umaru_temp var->wmarwsc_temp var->alpha_press_temp var->rhot_temp var->hflux_t_temp var->hcnun_w_temp var->rho_s_temp var->vadvwt1_temp var->vadvwt2_temp prog_wrt_struct	var->umaru_temp var->wmarw_temp var->wmarwsc_temp var->alpha_press_temp var->rhot_temp var->hflux_t_temp var->hcnun_w_temp var->vadvwt1_temp var->vadvwt2_temp
prog_u	var->umaru_temp var->wmarw_temp var->hcnun_u_temp var->rho_s_temp prog_u_struct	var->umaru_temp var->wmarw_temp var->hcnun_u_temp var->rho_s_temp

5.4 Fase 4 - Reestruturação Interna dos Kernels da Aplicação

Pensando em possíveis otimizações na implementação dos *kernels* do programa, foi realizada uma inspeção no código interno de cada um. Nesta análise, procurou-se analisar o impacto das modificações realizadas sobre o desempenho total da aplicação através dos tempos de execução.

As alterações internas aos *kernels* consistem tentar otimizar o acesso aos dados da memória da GPU. Isso foi pensado ao analisar os laços de repetição presentes no código, onde verificou-se diversos casos similares ao exemplo da [Figura 29](#). Assim como no exemplo, os laços do código realizam iterações alterando dados de variáveis diferentes, como *valor1* que é calculado e logo utilizado como parte do cálculo de *valor2*. Com isso, se o vetor *valor1* fosse inicializado antes de ser utilizado para calcular *valor2*, o acesso aos endereços de memória poderia ser otimizado, pois vetores são armazenados em espaços contíguos de endereçamento. A partir dessa ideia, os laços que apresentavam manipulações de dados como no exemplo citado foram fragmentados de acordo com o que se esperava quanto ao acesso à memória da GPU. Na [Figura 30](#) é representado, de forma geral, como essas alterações foram realizadas nos laços dos *kernels*, realizando a múltiplos laços iguais para calcular dados específicos.

Após feitas as alterações pretendidas nas três funções de *kernel*, a aplicação foi executada com casos de teste similares aos da [Figura 26](#) para uma comparação de desempenho, utilizando tamanhos de blocos de *threads* que obtenham a maior ocupação

Figura 29 – Exemplo de um laço iterativo FOR antes de ser alterado em um *kernel*.

```
//exemplo de laço antes de ser alterado
...

//'x' e 'y' representam cálculos quaisquer utilizando vários dados
for(i=0;i<N;i++){
    valor1[i] = x;
    valor2[i] = y * valor1[i];
}

...
```

Figura 30 – Exemplo de um laço iterativo FOR depois de ser alterado em um *kernel*.

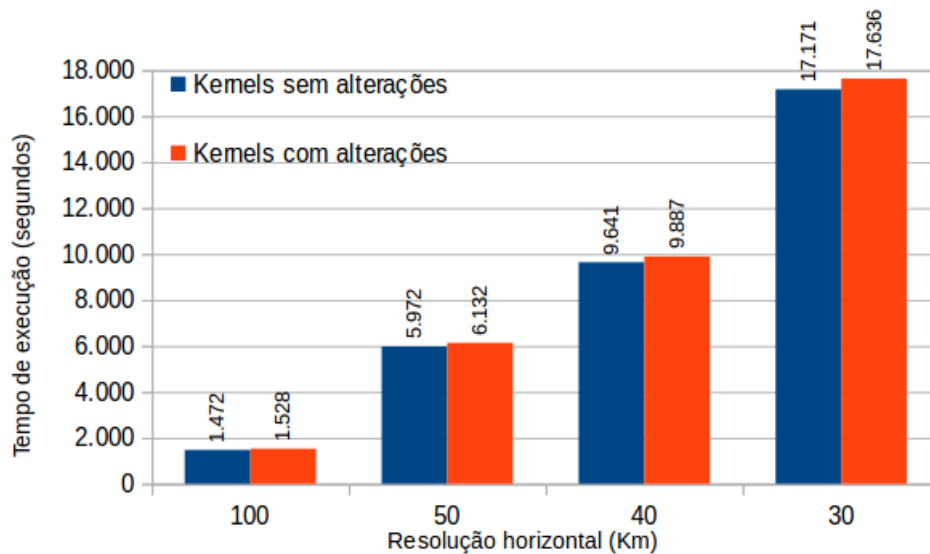
```
//exemplo de laço depois de ser alterado
...

//'x' e 'y' representam cálculos quaisquer utilizando vários dados
for(i=0;i<N;i++){
    valor1[i] = x;
}
for(i=0;i<N;i++){
    valor2[i] = y * valor1[i];
}

...
```

possível dos multiprocessadores da GPU. Como a aplicação foi modificada e compilada para uma nova execução, foi novamente analisado o número de registradores utilizados por *thread* em cada *kernel*. Para os *kernels prog_wrt* e *prog_u*, manteve-se o número de 63 registradores por *thread*, assim como a ocupação dos multiprocessadores estimada em 33% pela planilha de ocupação. Já para o *kernel hflux*, o número de registradores por *thread* diminuiu de 35 para 33, o que melhora a ocupação dos multiprocessadores de 58% para 63% utilizando blocos de 960 *threads*. A comparação dos resultados é mostrada na [Figura 31](#), onde se vê que o tempo de execução total para todos os casos onde os *kernels* foram alterados teve um aumento de em média 1%. Uma justificativa para isso é que mais tempo é necessário para realizar um número maior de laços iterativos, o que não compensa o possível acesso aprimorado à memória da GPU. Assim, a utilização dessas adaptações realizadas em código foi descartada por não prover melhor desempenho para a aplicação.

Figura 31 – Comparação de tempos de execução da aplicação: *kernels* reestruturados VS *kernels* sem alteração.



5.5 Fase 5 - Cópia de Memória Assíncrona e Particionamento de Kernel

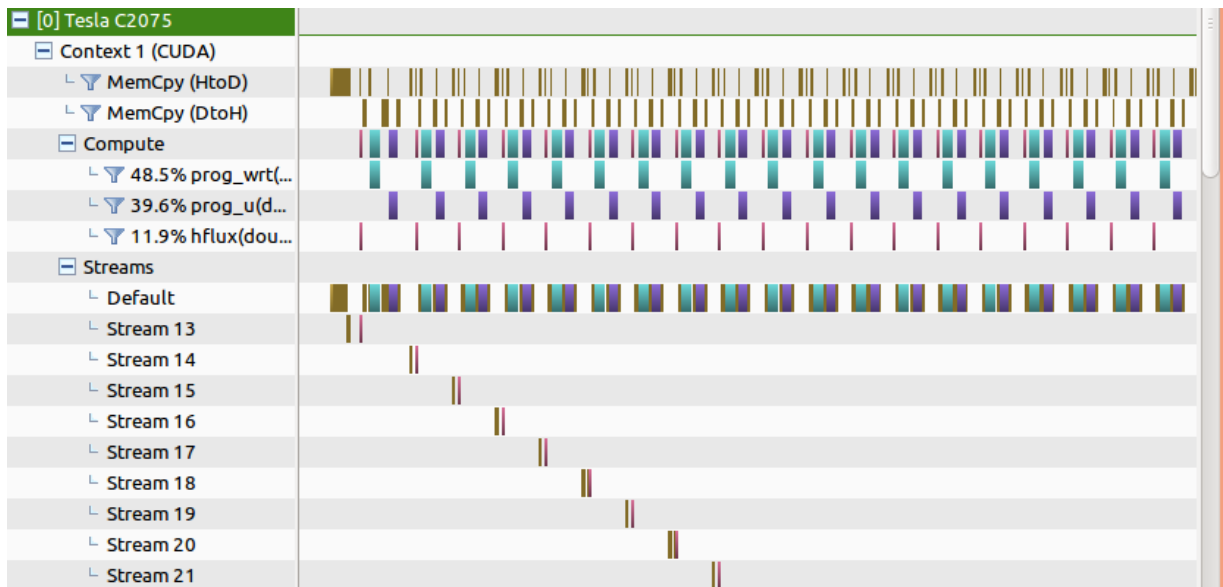
Como a etapa de estudos anterior não contribuiu com bons resultados para o desempenho do programa, a fase atual foi iniciada, onde pretende-se explorar as funcionalidades de execução em GPU sobreposta a cópias de memória assíncrona presentes na arquitetura da placa disponível, conforme foi visto na [seção 5.3](#). Para tal, a barreira de desempenho imposta pelas dependências de dados dos *kernels* deve ser transposta, mesmo que em parte.

Como foi dito na [seção 5.3](#), o uso de *streams* permite que diferentes tarefas sejam realizadas sobrepostas. Ao mesmo tempo, é possível transferir dados do *host* para o *device*, transferir dados do *device* para o *host*, e realizar a execução de *kernels*. Na implementação do OLAM, as funções de *kernel* são chamadas a partir de uma função dentro um laço iterativo, o que gera muitas chamadas a um *kernel* durante uma execução do modelo. Para aplicar uso de *streams* é necessário que os fluxos sejam criados e destruídos dentro desta função, controlando cópias de memória e chamadas de *kernel*.

Devido às dificuldades de contornar as dependências de dados para mascarar as transferências de memória, não foi possível implementar cópia de memória assíncrona neste trabalho. As cópias de memória são um fator limitante no desempenho da aplicação, fato já mencionado por [Osthoff et al. \(2012\)](#) que vem estudando o desempenho paralelo do OLAM utilizando CUDA. Na [Figura 32](#), pode-se ver parte da *timeline* de uma execução onde foi implementada uma *stream* para o *kernel hflux* e as cópias de memória anteriores

a este *kernel*, todas do *host* para o *device* com dados a serem utilizados pelos três *kernels*. Trata-se dos vários *streams* abaixo do *Default*, gerados a cada iteração do laço de invocação dos *kernels*.

Figura 32 – Exemplo de utilização de *streams*.



Quanto ao particionamento dos *kernels* já existentes em mais funções para a GPU, não foi possível eliminar as dependências internas para dividir o trabalho para *kernels* concorrentes. Uma análise mais rigorosa deve ser realizada, tanto na tentativa de dividir um *kernel* em mais funções quanto implementar a cópia de memória assíncrona, pois tais funcionalidade podem ser decisivas no desempenho paralelo do OLAM. Vale ressaltar que algumas cópias de memória entre *host* e *device* poderiam ser removidas desta implementação do OLAM, pois apenas são necessárias para transferir dados para a CPU comunicar com processos MPI quando a aplicação é executada sobre uma arquitetura de memória distribuída que utiliza GPUs.

6 Conclusão

Este trabalho buscou otimizar o desempenho do *Ocean-Land-Atmosphere Model* (OLAM) em um ambiente massivamente paralelo através de estudos acerca da arquitetura GPU para CUDA. O protótipo utilizado apresentou, inicialmente, desempenho 60% melhor em relação à sua versão sem paralelismo em CPU.

A partir de várias análises e testes realizados, diversos tópicos candidatos a prover desempenho à aplicação foram trabalhados, mas as fortes dependências de dados entre as funções executadas em GPU limitaram o desempenho do programa, apesar de que a definição adequada de tamanho do bloco de *threads* de cada *kernel* melhorou em cerca de 1% a performance da aplicação. Tais limitações se devem à implementação do código, o que necessitaria de grande profundidade nos estudos e reescrita de código, de modo a obter uma aceleração de desempenho considerável. Outros estudos são feitos sobre o OLAM utilizando GPUS, os quais também enfrentam barreiras que limitam a performance do mesmo.

Como trabalhos futuros, pode-se aprimorar as transferências entre memória do *host* e do *device* e reimplementar os *kernels* da aplicação. Com isso, a sobreposição de trabalhos distintos realizados pela GPU pode mascarar parte das cópias de memória, o que é parte fundamental em implementações de aplicações CUDA para GPUs.

Durante o Trabalho de Conclusão de Curso desenvolvido, foram enviados trabalhos para eventos relacionados à área de estudo com resultados parciais ou totais dos experimentos realizados. Estes trabalhos são:

- Avaliação de Desempenho de uma Implementação com CUDA do Ocean-Land-Atmosphere Model, apresentado na XIV Escola Regional de Alto Desempenho - RS (ERAD-RS), onde foi apresentado um trabalho sobre os testes de desempenho iniciais do modelo OLAM em arquiteturas GPU ([VARGAS; SCHEPKE, 2014](#));
- Proposta de Melhoria de uma Implementação Paralela para GPUs Usando CUDA - Estudo de Caso em Modelo Atmosférico, submetido para o XV Simpósio em Sistemas Computacionais de Alto Desempenho - WSCAD 2014;
- Propostas de Otimização de Códigos Implementados em CUDA para o Modelo Meteorológico OLAM, submetido para o XXVI Congresso Regional de Iniciação Científica e Tecnológica em Engenharia;
- Será produzido ainda um resumo para o VI Salão Internacional de Ensino, Pesquisa e Extensão da Universidade Federal do Pampa.

Referências

- AMD. *AMD Graphics Cores Next (GCN) Architecture*. 2012. Disponível em: <http://www.amd.com/br/Documents/GCN_Architecture_whitepaper.pdf>. Citado na página 35.
- AMD. *AMD Radeon HD 7970 Graphics*. 2014. Disponível em: <<http://www.amd.com/br/products/desktop/graphics/7000/7970/Pages/radeon-7970.aspx#3>>. Citado na página 35.
- AMD. *GCN Architecture*. 2014. Disponível em: <<http://www.amd.com/us/products/technologies/gcn/Pages/gcn-architecture.aspx>>. Citado na página 35.
- COTTON, W. *The RAMS Homepage - A Description of RAMS*. 2014. Disponível em: <<http://rams.atmos.colostate.edu/rams-description.html>>. Citado na página 21.
- FINEP. *Finep - Brasil avança na previsão do tempo com supercomputador*. 2013. Disponível em: <http://www.finep.gov.br/imprensa/noticia.asp?cod_noticia=3270>. Citado na página 17.
- Free Software Foundation. *GNU General Public License v2.0 - GNU Project*. 2014. Disponível em: <<http://www.gnu.org/licenses/gpl-2.0.html>>. Citado na página 22.
- FREITAS, S. R. et al. The Coupled Aerosol and Tracer Transport Model to the Brazilian Developments on the Regional Atmospheric Modeling System (CATT-BRAMS) – Part 1: Model Description and Evaluation. *Atmospheric Chemistry and Physics*, v. 9, n. 8, p. 2843–2861, 2009. Disponível em: <<http://www.atmos-chem-phys.net/9/2843/2009/>>. Citado na página 22.
- HERDMAN, J. et al. Accelerating Hydrocodes with OpenACC, OpenCL and CUDA. In: *2012 SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC)*. [s.n.], 2012. p. 465–471. Disponível em: <<http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6495848&tag=1>>. Citado na página 36.
- IBM. *IBM - Computação de Alto Desempenho do IBM Power Systems - Brasil*. 2014. Disponível em: <<http://www-03.ibm.com/systems/br/power/hardware/hpc.html>>. Citado na página 17.
- IEEE. *IEEE 754: Standard for Binary Floating-Point Arithmetic*. 2014. Disponível em: <<http://grouper.ieee.org/groups/754/>>. Citado na página 31.
- INPE/CPTEC. *BRAMS*. 2014. Disponível em: <<http://brams.cptec.inpe.br/>>. Citado na página 22.
- INTEL. *Comparação de Processadores Intel*. 2014. Disponível em: <<http://www.intel.com.br/content/www/br/pt/processor-comparison/compare-intel-processors.html?select=server>>. Citado na página 29.

KHRONOS. *OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems*. 2014. Disponível em: <<https://www.khronos.org/opencl/>>. Citado na página 36.

KIRK, D. B.; HWU, W.-M. W. *Programando para Processadores Paralelos: Uma Abordagem Prática à Programação de GPU*. [S.l.]: Elsevier Editora Ltda., 2011. Citado 7 vezes nas páginas 27, 28, 29, 30, 37, 38 e 39.

KRAUS, J. e. a. Benchmarking GPUs with a Parallel Lattice-Boltzmann Code. In: *XXV International Symposium on Computer Architecture and High Performance Computing - SBAC - PAD 2013*. [s.n.], 2013. Disponível em: <<http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6702593&queryText%3Dlattice-boltzmann+gpu>>. Citado na página 47.

MARSHALL, J. et al. A Finite-Volume Incompressible Navier-Stokes Model for Studies of Ocean on Parallel Computers. *Journal of Geophysical Research*, v. 102, p. 5753–5766, 1997. Citado na página 22.

NICKOLLS, J.; DALLY, W. J. The GPU Computing Era. *IEEE Micro*, v. 30, p. 56–69, 2010. Citado 2 vezes nas páginas 29 e 35.

NITA, C. et al. GPU Accelerated Blood Flow Computation using the Lattice Boltzmann Method. In: *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. [s.n.], 2013. Disponível em: <<http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6670324&queryText%3Dlattice-boltzmann+gpu>>. Citado na página 47.

NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. 2009. Disponível em: <http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>. Citado 4 vezes nas páginas 30, 31, 32 e 35.

NVIDIA. *NVIDIA Tesla C2075*. 2011. Disponível em: <<http://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-C2075.pdf>>. Citado 2 vezes nas páginas 29 e 42.

NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. 2012. Disponível em: <<http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>>. Citado 4 vezes nas páginas 32, 33, 34 e 35.

NVIDIA. *CUDA C Programming Guide*. 2013. Disponível em: <<https://developer.nvidia.com/cuda-downloads>>. Citado 4 vezes nas páginas 18, 28, 37 e 39.

NVIDIA. *NVIDIA Tesla - GPU Accelerators*. 2013. Disponível em: <<http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf>>. Citado na página 29.

NVIDIA. *CUDA Toolkit*. 2014. Disponível em: <<https://developer.nvidia.com/cuda-toolkit>>. Citado na página 44.

NVIDIA. *NVIDIA Quadro 5000*. 2014. Disponível em: <<http://www.nvidia.com/object/product-quadro-5000-us.html>>. Citado 2 vezes nas páginas 29 e 42.

NVIDIA. *Plataforma de Computação Paralela CUDA*. 2014. Disponível em: <http://www.nvidia.com.br/object/cuda_home_new_br.html>. Citado na página 36.

- OPENACC.ORG. *OpenACC Home Page*. 2014. Disponível em: <<http://www.openacc-standard.org/>>. Citado na página 36.
- OSTHOFF, C. et al. Atmospheric Model Cluster Performance Evaluation on Hybrid MPI/OpenMP/Cuda Programming Model Platform. In: *Proceedings of XXI International Conference of the Chilean Computer Science Society*. [S.l.: s.n.], 2012. Citado na página 54.
- PIELKE, R. et al. A Comprehensive Meteorological Modeling System - RAMS. *Meteorology and Atmospheric Physics*, v. 49, p. 69–91, 1992. Disponível em: <<http://link.springer.com/article/10.1007/BF01025401#>>. Citado na página 21.
- REYES, R. et al. A Preliminary Evaluation of OpenACC Implementations. *The Journal of Supercomputing*, v. 65, p. 1063–1075, 2013. Disponível em: <<http://link.springer.com/article/10.1007/s11227-012-0853-z>>. Citado na página 36.
- SCHEPKE, C. *Exploiting Multiple Levels of Parallelism and Online Refinement of Unstructured Meshes in Atmospheric Model Application*. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul, 2012. Citado 2 vezes nas páginas 19 e 41.
- SKAMAROCK, W. C. et al. *A Description of the Advanced Research WRF Version 3*. [S.l.], 2008. Disponível em: <http://www.mmm.ucar.edu/wrf/users/docs/arw_v3.pdf>. Citado na página 21.
- STALLINGS, W. *Arquitetura e Organização de Computadores*. [S.l.]: Pearson Education do Brasil, 2010. Citado na página 28.
- SUN, X. et al. Parallel active contour with Lattice Boltzmann scheme on modern GPU. In: *International Conference on Image Processing – ICIP 2012*. [S.l.: s.n.], 2012. Citado na página 47.
- TAUNAY, P.-Y. *Advanced CUDA Topics - Part 5: Concurrent Execution*. 2013. Disponível em: <https://rcc.its.psu.edu/education/seminars/pages/advanced_cuda/AdvancedCUDA5.pdf>. Citado na página 51.
- VARGAS, F. C. de; SCHEPKE, C. Avaliação de Desempenho de uma Implementação com CUDA do Ocean-Land-Atmosphere Model. In: *Anais da XIV Escola Regional de Alto Desempenho - ERAD/RS*. [S.l.: s.n.], 2014. Citado na página 57.
- WALKO, R. L.; AVISSAR, R. The Ocean–Land–Atmosphere Model (OLAM). Part I: Shallow-Water Tests. *Monthly Weather Review*, v. 136, p. 4033–4044, 2008. Disponível em: <<http://web.a.ebscohost.com/ehost/detail?sid=4e8b715e-ee11-4f90-af04-ff3825040aa3%40sessionmgr4003&vid=1&hid=4107&bdata=JnNpdGU9ZWwhvc3QtbGl2ZQ%3d%3d#db=aph&AN=35204804&anchor=GoToAllQVI>>. Citado 4 vezes nas páginas 22, 23, 24 e 25.