UNIVERSIDADE ESTADUAL DE PONTA GROSSA
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA

CIRO BARON NETO

SIMULAÇÃO CLIMÁTICA DE DADOS DE VENTO EM REDES P2P UTILIZANDO GPU

Ponta Grossa

2014

UNIVERSIDADE ESTADUAL DE PONTA GROSSA
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA

CIRO BARON NETO

SIMULAÇÃO CLIMÁTICA DE DADOS DE VENTO EM REDES P2P UTILIZANDO GPU

Dissertação apresentada para obtenção do título de Mestre em Computação Aplicada na Universidade Estadual de Ponta Grossa, Área de concentração: Computação para Tecnologias Agrícolas.

Orientador: Prof. Dr. Luciano José Senger

Ponta Grossa

2014

Ficha Catalográfica
Elaborada pelo Setor de Tratamento da Informação BICEN/UEPG

Baron Neto, Ciro
B265 Simulação climática de dados de vento em redes P2P utilizando GPU/ Ciro Baron Neto. Ponta Grossa, 2014.
73f.

Dissertação (Mestrado em Computação Aplicada - Área de Concentração: Computação para Tecnologias em Agricultura), Universidade Estadual de Ponta Grossa.
Orientador: Prof. Dr. Luciano José Senger.

1.Computação paralela. 2.GPGPU. 3.Peer-to-Peer. 4.Simulação climática. I.Senger, Luciano José. II. Universidade Estadual de Ponta Grossa. Mestrado em Computação Aplicada. III. T.

CDD: 004.22

TERMO DE APROVAÇÃO

Ciro Baron Neto

"SIMULAÇÃO CLIMÁTICA DE DADOS DE VENTO EM REDES P2P UTILIZANDO GPU"

Dissertação aprovada como requisito parcial para obtenção do grau de Mestre no Programa de Pós-Graduação em Computação Aplicada da Universidade Estadual de Ponta Grossa, pela seguinte banca examinadora:

Orientador: *Dr. Luciano José Senger*

UEPG

Dr. Jarim Sousa Dos Virgens Filho

UEPG

Dr. Renato Porfirio Ishii

UFMS

Ponta Grossa, 28 de fevereiro de 2014

Dedico aos meus pais Ulisses e Marilene, meus irmãos Pedro e Maria e a minha querida e amada esposa Fabi.

AGRADECIMENTOS

Agradeço primeiramente à Deus pela força de completar mais uma etapa em minha vida.

Ao meu orientador prof. Dr. Luciano José Senger pela dedicação, confiança e ajuda no desenvolvimento do trabalho.

Ao prof. Dr. Jorim Sousa das Virgens Filho por disponibilizar os materiais necessários para a realização do trabalho.

Ao prof. Dr. Márcio Augusto de Souza pelo apoio e contribuições que só ajudaram a melhorar o trabalho.

Aos meus pais Ulisses e Marilene, meus irmãos Pedro e Maria e minha amada esposa Fabi pelo apoio incondicional.

Aos meus grandes amigos Nelson, Eduardo, Leandro e Fábio pelo total apoio e entendimento da minha jornada.

RESUMO

Este trabalho apresenta uma avaliação das tecnologias de GPGPU (*General-Purpose Computing on Graphics Processing Unit*) e de redes P2P (*peer-to-peer*) para melhorar o tempo de resposta de simulações de dados climáticos. Para isso, uma aplicação utilizando a arquitetura CUDA (*Compute Unified Device Architecture*) e o modelo de simulação de dados de vento do software Venthor foram inicialmente adotados e após integrados ao *framework* P2PComp. Os resultados indicam um fator de aceleração igual a 70 para computadores isolados. Além disso, com a possibilidade do uso de uma rede P2P para compartilhamento de processamento, fatores de aceleração maiores podem ser obtidos. Modelos de simulação computacional geralmente demandam alto poder de processamento e este trabalho mostrou que a utilização do paralelismo em redes P2P e GPUs constitui uma alternativa que permite melhor desempenho quando comparado à computação sequencial.

Palavras-chave: Computação Paralela, GPGPU, *Peer-to-Peer*, Simulação Climática

ABSTRACT

This paper presents an approach of technologies GPGPU (General-Purpose Computing on Graphics Processing Unit) and P2P (peer-to-peer) networks in order to improve the response time of climate data simulations. Thus, an application using CUDA (Compute Unified Device Architecture) architecture and the simulation model of Venthor simulator were initially adopted and integrated into the P2PComp framework. The results indicate an acceleration factor equal to 70 for single computers. Furthermore, the possibility of using a P2P sharing network for processing, higher acceleration factors can be obtained. Computer simulation models usually demand high processing power and this work showed that the use of parallelism in GPUs and P2P networks is an alternative that allows better performance when compared to sequential computing.

Keywords: Parallel Computing, GPGPU, Peer-to-Peer, Climate Simulation

LISTA DE SIGLAS

ALU	Unidade Lógica Aritmética
API	*Application Programming Interface*
CPU	Unidade de Processamento Central
CUDA	*Compute Unified Device Architecture*
GPU	Unidade de Processamento Gráfico
GPGPU	*General-purpose Computing on Graphics Processing Units*
FLOP	*Floating-point Operations Per Second*
JXTA	Java *Juxtapose*
KS	Kolmogorov-Smirnov
LD/ST	*Load/Store*
NFS	*Network File System*
SFU	*Special Function Unit*
SIMD	*Single Instruction Multiple Data*
SLI	*Scalable Link Interface*
SM	*Stream Multiprocessor*
SP	*Stream Processor*
SMX	*Stream Processor* (versão da arquitetura Kepler)

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Variáveis climáticas como vento, chuva e temperatura do ar influenciam consideravelmente a produção agrícola de uma região. O uso de técnicas de simulação para a previsão dessas variáveis pode ajudar o produtor em tomadas de decisão para maximizar lucros, minimizar perdas e ainda reduzir a utilização de insumos agrícolas (BEDOS *et al.*, 2002).

Com relação à geração de energia alternativa, o vento é uma importante fonte renovável que pode ser utilizada para a geração de energia elétrica e mecânica. Evidentemente, a quantidade de energia gerada é definida pela velocidade e direção do vento (JACOBSONA; ARCHERB, 2012), portanto a instalação de turbinas deve ser feita em locais onde a incidência do vento é relevante. A energia gerada pode ainda ser utilizada em moinhos de ventos que podem auxiliar na captação e bombeamento de água para irrigação.

O vento é uma variável importante também na a área agrícola, pois influencia diretamente a utilização de insumos e ainda relaciona-se diretamente com a velocidade de difusão de doenças nas plantas. Essas doenças correspondem a perda de 31% a 42% da produção mundial causando um prejuízo de cerca de 220 bilhões de dólares (CONTRERAS *et al.*, 2009) e impactando setores econômicos e sociais. Para o controle das doenças, insumos químicos são utilizados nas plantações, acarretando muitas vezes em problemas ambientais. Para a redução da utilização desses insumos, é fundamental que entendamos a influência e a ação de variáveis climáticas na dinâmica da difusão das doenças. Modelos matemáticos e estatísticos têm sido usados para entender esses fatores e modelar dinâmicas epidemiológicas (CONTRERAS *et al.*, 2009), e a simulação computacional vem como uma aliada nesse processo, permitindo a utilização de diferentes modelos matemáticos e estatísticos para a análise do progresso de doenças em plantas. Esses modelos são resolvidos com cálculos matemáticos e a utilização de programas computacionais que permitam o processamento paralelo é encorajada.

Atualmente, o processamento paralelo em alta escala é feito por aglomerados de computadores (denominados *clusters*) interligados por um barramento de rede. Existem diversos problemas com esse tipo de arquitetura, como alto custo de aquisição, manutenção, e a necessidade de um espaço físico que comporte a estrutura (VALENTIN *et al.*, 2011). Uma alternativa a utilização de clusters é a utilização do processamento em GPU (*Graphics Processing Unit*). Esse novo paradigma tem sido usado, pois utiliza os núcleos de processamento disponibilizados na GPU para a realização de cálculos matemáticos. Apli-

cações que utilizam a GPU para essa finalidade são conhecidas como GPGPU (*General Purpose Computation on GPU*) e a arquitetura frequentemente utilizada para o desenvolvimento de GPGPUs é a CUDA (*Compute Unified Device Architecture*) da empresa NVIDIA[1]. Através das técnicas de computação paralela em simulação de dados climáticos, torna-se possível a redução do tempo de processamento dos cálculos estatísticos e matemáticos (SERCKUMECKA, 2012).

A diferença principal existente entre o processamento em CPUs e GPUs é que a primeira implementa uma arquitetura desenvolvida para o processamento de instruções de forma sequencial, com poucas *threads* e um *clock* alto, enquanto a segunda implementa uma arquitetura desenvolvida para o processamento massivamente paralelo, com muitas *threads* e com um *clock* inferior aos *clocks* existentes em CPUs. O número de *threads* que podem ser executadas ao mesmo tempo em uma GPU pode chegar à casa das dezenas de milhares, pois a arquitetura dos chips gráficos foi criada para o processamento de imagens e o processamento 3D em tempo real que demandam cálculos que são repetidos várias vezes.

A utilização de GPGPUs foi impulsionada pela empresa NVIDIA com a criação da arquitetura CUDA que é suportada em todas as GPUs a partir da arquitetura G80. A programação CUDA é feita na linguagem C, porém foram desenvolvidos *wrappers* para C#[2] e Java[3] . Em contrapartida à NVIDIA, a empresa ATI desenvolveu a arquitetura Brook+, que é suportado pelas GPUs a partir da arquitetura R5200 e a programação é feita na linguagem Brook, que é uma variação da linguagem C. A eficiência da utilização de GPGPUs, independente da arquitetura utilizada é comprovada, pois na data que esse trabalho está sendo escrito, supercomputadores equipados com GPUs estão entre os mais rápidos do mundo[4]. Empresas brasileiras, inclusive, investem em supercomputadores como é o caso da Petrobrás, que em 2012 criou o Grifo04, que possui 554 servidores sendo que cada um contempla duas GPUs NVIDIA Tesla M2050 (JOBSTRAIBIZER, 2012).

O desempenho nas arquiteturas paralelas é comumente expresso pela quantidade de operações de ponto flutuante por segundo (*FLoating-point Operations Per Second - FLOPS*). Fundamentalmente as CPUs e GPUs são construídas com bases em filosofias diferentes, por isso a diferença de desempenho entre elas é notável. A tabela 1 mostra um comparativo entre GPU e CPU onde é possível comparar o desempenho (em FLOPS) e a quantidade de transistores existentes:

[1] http://www.nvidia.com/object/cuda_home_new.html
[2] http://managedcuda.codeplex.com
[3] http://www.jcuda.org
[4] http://top500.org/lists/2013/11

Tabela 1: Diferença entre GPU e CPU

Intel Core i7 2600K	AMD Radeon HD 5870
14 GFLOPF	2.7 TFLOPS
730 Milhões de transistores	2.2 Bilhões de transístores

Fonte: O autor

O desempenho da GPU é quase 193x superior que a CPU e emprega 3x mais transistores. O trabalho de (LEE *et al.*, 2010) mostra com detalhes as diferenças e particularidades das duas arquiteturas.

Como o desempenho das GPUs é notavelmente superior às GPUs, este trabalho possui como objetivo a implantação de GPUs em redes P2P com o propósito de reduzir o tempo de processamento do simulador climático Venthor.

Grades computacionais foram desenvolvidas como uma forma de organizar e compartilhar recursos de domínios administrativos diferentes, frequentemente localizados em diferentes organizações, de forma que os recursos são interligados por uma infraestrutura de software comum. Essa infraestrutura disponibiliza mecanismos para localização, controle de acesso, monitoração e escalonamento de recursos e aplicações computacionais (FOSTER; KESSELMAN; TUECKE, 2001). Grades permitem a criação de sistemas computacionais que oferecem a possibilidade de crescimento em escala e potência elevada. Por isso, grades são atrativas como plataformas de execução de aplicações computacionais de diferentes áreas, como por exemplo: biologia molecular, física, modelagem do mercado financeiro, sistemas de predição de terremotos, sistemas de previsão de clima e bio-informática (RANJAN; HARWOOD; BUYYA, 2006). Dessa forma, o desenvolvimento da tecnologia de grades computacionais tem impacto direto em diferentes áreas de conhecimento.

Dentre as tecnologias que têm surgido para grades computacionais, o modelo de computação par-a-par (P2P) destaca-se por oferecer uma alternativa para a construção de sistemas que apresentam um eficiente uso dos recursos, possibilidade de crescimento em escala e capacidade de auto-organização (MILOJICIC *et al.*, 2002; HAUSWIRTH; SCHMIDT, 2005; AMORETTI, 2006; RANJAN; HARWOOD; BUYYA, 2006). O modelo de computação P2P permite a criação de sistemas distribuídos empregando um conjunto de computadores que cooperam entre si, chamados de pares, e que compartilham alguns de seus recursos sem a necessidade de um servidor central para a gerência. Arquivos, espaço de armazenamento e ciclos de processamento são exemplos comuns de recursos que podem ser compartilhados em sistemas distribuídos do tipo P2P. Sistemas P2P são construídos como uma rede

sobreposta (*overlay network*) sobre a rede real de comunicação, que frequentemente é a Internet. Tal rede sobreposta emprega um sistema de identificação plano, que permite que os pares tenham uma identificação única e independente da rede física na qual eles estão localizados. Além disso, a rede virtual permite que os pares possam encontrar outros pares, descobrir recursos e trocar informações, mesmo em redes que têm fronteiras protegidas por *firewalls* (MORRIS *et al.*, 2001).

Sistemas P2P apresentam assim três características importantes para a implementação de uma infraestrutura para grades computacionais: capacidade de auto-organização, comunicação simétrica entre os pares e controle distribuído (RISSON; MOORS, 2004). A capacidade de auto-organização permite que pares sejam incorporados ou retirados dos sistema de maneira transparente, tornando o sistema robusto mesmo que ocorram falhas de pares. A comunicação é simétrica pois cada par pode atuar como servidor ou consumidor de recursos. O controle é distribuído pois não é necessária a existência de um servidor central. Os projetos SETI (ANDERSON *et al.*, 2002) e BOINC (ANDERSON, 2004) são exemplos de sistemas P2P de sucesso. Esses projetos permitem que uma aplicação computacional seja particionada em diversas tarefas que são executadas em paralelo por computadores domésticos distribuídos, criando um computador paralelo virtual composto de computadores distribuídos pela Internet. Porém, as tarefas distribuídas que compõem as aplicações não trocam mensagens entre si para sincronização e troca de informações. Em outras palavras, não existe um suporte para troca e controle de execução das aplicações de propósito geral. Dentre os frameworks para desenvolvimento de aplicações, destaca-se o P2PComp (SENGER; SOUZA; FOLTRAN, 2010).

Este trabalho foca-se na paralelização do código do *software* de simulação de ventos Venthor (VIRGENS FILHO; LEITE, 2009) no contexto do desenvolvimento de uma aplicação GPGPU com a finalidade de ser executada em uma rede P2P. A contribuição que o *software* traz para o lado agronômico é o auxílio para o produtor em suas tomadas de decisões tornando possível a predição das condições de ventos locais. O produtor pode ainda levar em consideração questões como: quais dias e horários mais propícios para utilizar insumos agrícolas e ainda quais culturas melhor se adaptam para a condição climática da sua região (CONCEICAO, 1996). Do ponto de vista computacional, este trabalho contribui com a utilização do paralelismo de uma forma que ainda não é comum e que consiste na utilização de redes P2P com *peers* equipados com GPUs.

1.1 Justificativa e motivação

É fundamental que o código de um *software* de simulação envolva alguma forma de paralelismo para que o tempo de processamento seja reduzido. O *software* de simulação Venthor é utilizado neste trabalho, pois em sua versão inicial não existe nenhuma forma de paralelismo, portanto, a implementação do paralelismo poderá ser feita para a criação de uma GPGPU. As motivações para o desenvolvimento desse trabalho são os simuladores que já foram paralelizados para a execução em GPU onde os resultados obtidos na maioria deles foram satisfatórios. Como exemplo, pode-se citar o simulador molecular HAL'sMD[5] que obteve um *speedup* de 80x, o simulador QUICK[6] com o valor de *speedup* de 100x e ainda o GEOS-5[7], utilizado pela NASA teve um valor de *speedup* de 10x. Uma lista pode ser encontrada no site da NVIDIA[8] com as aplicações e os valores de *speedup* obtidos com a utilização da GPU.

1.2 Objetivos

Dentro deste contexto, o trabalho tem como objetivo geral a aplicação de tecnologias GPGPU em redes P2P. Os objetivos específicos são:

- Reduzir o tempo de processameto na simulação de ventos do *software* Venthor;
- Verificar se os resultados das rotinas criadas para a GPU são consistentes com os resultados das rotinas existentes no Venthor;
- Comparar o tempo de processamento da aplicação GPGPU com a versão original e a versão já existente paralelizada do Venthor;
- Executar o Venthor em uma rede P2P com duas GPUs e comparar os tempos de processamento com a versão original e a versão já paralelizada já existente do Venthor;

1.3 Trabalhos correlatos

Apesar do desenvolvimento de aplicações GPGPU ser um novo paradigma em computação paralela, existem trabalhos que abordam a criação de aplicações GPGPU no

[5]http://halmd.org
[6]http://www.merzgroup.org/quick.html
[7]http://gmao.gsfc.nasa.gov/systems/geos5
[8]http://www.nvidia.com/object/gpu-applications.html

contexto da simulação computacional (RADEKE; GLASSER; KHINAST, 2010; KALYANAPU *et al.*, 2011; ZWART; BELLEMAN; GELDOF, 2007; HEMERT; DICKERSON, 2011).

Kalyanapu *et al.* (2011) realizou um estudo onde o objetivo foi simular o comportamento de uma enchente caso uma represa fosse danificada. No estudo, o *speedup* com a utilização da GPU comparando implementações idênticas da CPU foi de 80x à 88x. Foi utilizada a arquitetura CUDA para a realização dos testes. O trabalho de Kalyanapu *et al.* (2011) possui relevância para o trabalho proposto, pois dá noções básicas de como mapear um problema real em um sistema GPGPU. Em outro trabalho, Radeke, Glasser e Khinast (2010) estudou o fluxo de substâncias granulares e qual o impacto no meio de substâncias com tamanhos, formas e flutuações diferentes. Foi possível a simulação de aproximadamente dois milhões de partículas por *Giga Byte* de memória de vídeo utilizando uma NVIDIA GT200. Para o cálculo massivo da simulação, foi utilizada também a arquitetura CUDA.

No estudo de Hemert e Dickerson (2011), existiu um *speedup* de 12x a favor das GPUs na realização de testes com o método estatístico de Monte Carlo. Hemert e Dickerson (2011) também demonstraram em seus trabalhos que um fator limitante em sua pesquisa foi a baixa quantidade de memória das GPUs.

Existem também estudos do desenvolvimento de versões paralelas do simulador Venthor. Serckumecka (2012) em seu trabalho desenvolveu a versão JAVA do simulador. Através do *framework* JXTA e do *framework* P2PComp, Serckumecka (2012) criou a versão paralela onde conseguiu um *speedup* de até 165x comparando o processamento de 26 *peers* com o sequencial. Apesar do trabalho do Serckumecka (2012) não abordar a utilização de GPGPUs a utilização do mesmo será referência para o desenvolvimento deste trabalho, pois trata de como foi feito o paralelismo do Venthor na utilização de redes P2P, e será utilizado como base para a implementação da versão do Venthor utilizando GPU com o *framework* P2PComp.

1.4 Organização do trabalho

Este trabalho foi organizado da seguinte forma:

No Capítulo 1 é mostrado uma visão geral das variáveis climáticas e das tecnologias GPGPU, grades computacionais e redes P2P. Nesse capítulo também é mostrado a justificativa e motivações, objetivos do trabalho e, finalmente, os trabalhos correlatos.

No Capítulo 2 é mostrado a revisão da literatura, onde está descrito o funcio-

namento das redes P2P com a utilização do *framework* P2PComp, o funcionamento das aplicações GPGPU com a arquitetura CUDA, e também o funcionamento do *software* Venthor.

O Capítulo 3 mostra quais foram os métodos utilizados para modificar o código do Venhor desenvolvendo uma GPGPU e integrando à uma rede P2P com o *framework* P2PComp.

O Capítulo 4 mostra o resultados das simulações feitas com a GPU utilizando rede P2P. É mostrado também o tempo de processamento comparado ao tempo de processamento em paralelo mostrado no trabalho de Serckumecka (2012).

O Capítulo 5 mostra as conclusões e também os trabalhos futuros que podem ser feitos com o Venthor e com o *framework* P2PComp.

Este trabalho conta ainda com o Apêndice A, que contém os códigos das chamadas do executável em C, o Apêndice B com os códigos das distribuições adaptadas à GPU, e o Anéxo A com as equações dos modelos de distribuições estatísticas.

2 REVISÃO DA LITERATURA

2.1 Simulação climática e a variável climática vento

O avanço da informática tem ajudado no desenvolvimento de modelos de simulações climáticas, permitindo assim simular sistemas reais e de alta complexidade. O vento é uma variável chave para diversos estudos climáticos como a mudança de temperatura em regiões, precipitação de chuva, movimento de massas quente entre outras (JIN-YOUNG; KIM; OH, 2013). No contexto da agricultura, é fundamental que exista um entendimento do clima da região em que uma determinada cultura será cultivada, pois permite o manejo adequado do solo e dos insumos agrícolas. Segundo Sediyama *et al.* (1978), o clima é um fator determinante para o controle do crescimento das plantas. Nesse contexto, a produção agrícola é um elemento probabilístico, pois depende de elementos climáticos durante a época de crescimento de uma cultura. Existem modelos de simulação matemática que podem descrever comportamentos climáticos e que têm sido aplicados com o objetivo principal de prever o comportamento das distribuições de probabilidade. Nesse trabalho, o modelo utilizado será o modelo implementado pelo *software* Venthor, que utiliza o teste de Kolmogorov-Smirnov para apontar qual é o melhor modelo entre as distribuições estatísticas Weibull, Gama, Beta e Rayleigh.

2.2 Processamento *multi-core* nos dias atuais

Nos últimos anos muito se tem feito na indústria de computação para mudar o paradigma de processamento puramente sequencial para a utilização de algum tipo de paralelismo. Em meados de 2007, processadores com dois núcleos eram comuns no mercado, porém eram utilizados especificamente em *desktops*. Atualmente existem *desktops* com 8 ou 16 núcleos[1] e a evolução do paradigma levou até mesmo dispositivos móveis como celulares[2] a terem mais de um núcleo. Nesse contexto, a necessidade do desenvolvimento de aplicações que possam ser compiladas para essa arquitetura é cada vez mais visível. Assim como a evolução dos processadores para *desktops*, nos últimos anos existiu uma evolução considerável nas placas de vídeo, conhecidas como GPU. As GPUs trabalham em uma arquitetura *multi-core* e geralmente implementam o número de núcleos superior aos das CPUs. Torna-se compreensível que a utilização das GPUs vá além do processamento

[1] http://www.amd.com/us/products/server/processors/6000-series-platform/6200/Pages/6200-series-processors.aspx

[2] http://www.samsung.com/br/consumer/cellular-phone/cellular-phone-tablets/smartphones/GT-I9295MOLZTO

puramente gráfico e que exista a flexibilidade da sua utilização para desenvolvimento de aplicações de propósito gerais. Existem atualmente GPUs com mais de 3000 núcleos[3] que implementam tecnologias necessárias para o desenvolvimento de aplicações GPGPU.

2.3 O início da utilização das GPGPUs

A criação de GPUs com APIs programáveis foi responsável por um salto de possibilidades no sentido de utilizar o hardware não apenas como unidades de renderização de imagens, mas sim como unidades totalmente programáveis. Antigamente a única forma de interação com a GPU era através das APIs OpenGL[4] e DirectX[5] , sendo que o programador ficava limitado apenas às funções existentes nessas bibliotecas. Os programadores eram obrigados a tentar resolver os problemas adaptando os programas de alguma forma que se aproximasse do processamento gráfico utilizando as APIs (SANDERS; KANDROT, 2004).

Em meados do ano 2000, as GPUs começaram a ser produzidas com unidades programáveis aritméticas chamadas *pixel shaders* que basicamente permitiam que o programador pudesse definir um conjunto de *inputs* que resultaria na cor de um *pixel* que seria exibido na tela. A entrada das funções *pixel shader*[6] eram geralmente a posição da tela onde o *pixel* seria exibido e outras opções que iriam exercer influência na cor final do *pixel*. Como os dados de entrada eram controlados pelos programadores, os pesquisadores começaram a utilizar essas unidades programáveis com *inputs* que, ao invés de representar as cores, representariam dados reais. Na sua essência a GPU começou a ser utilizada para processar dados que não eram mais puramente gráficos, aproveitando assim o desempenho das unidades aritméticas que era superior às encontradas nas CPUs da época. A partir desse momento percebeu-se que as GPUs teriam um futuro promissor (SANDERS; KANDROT, 2004).

Embora o avanço para a época fosse bastante visível, existiam ainda algumas limitações que impediam pesquisadores utilizar as GPUs em sua totalidade. Era impossível, por exemplo, gerenciar memória de uma forma eficiente, ou ainda predizer qual a exatidão que a GPU iria lidar com dados de ponto flutuante. Os pesquisadores e programadores ficavam novamente presos às APIs para tentar minimizar esses problemas.

Aproximadamente 6 anos depois ocorreu um fato que marcou o mundo da com-

[3]http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-690/specifications
[4]http://www.opengl.org
[5]http://windows.microsoft.com/pt-BR/Windows7/products/features/directx-11
[6]http://www.nvidia.com/object/feature_pixelshader.html

putação *multi-core*. A empresa NVIDIA lançou a GPU GeForce 8800 GTX que foi a primeira GPU criada na arquitetura CUDA. Tamanha importância se deve a essa arquitetura, pois essa inclui componentes desenvolvidos especialmente para terminar com as limitações existentes anteriormente, tornando assim uma arquitetura onde é possível o desenvolvimento de aplicações de propósito geral utilizando GPU. As ALUs se tornaram programáveis e foram desenvolvidas segundo as especificações IEEE de precisão simples e dupla, terminando assim com os problemas da imprecisão de dados de ponto flutuante (HWU; KIRK, 2011). A arquitetura CUDA hoje é utilizada em aplicações de diversas áreas como Química Computacional, Análise Numérica, Simulação de Fluídos e Simuladores climáticos. No contexto desse trabalho, a arquitetura CUDA será utilizada para executar o simulador climático Venthor.

2.4 A arquitetura CUDA

A arquitetura CUDA apresenta algumas particularidades quanto à execução de *threads*. Na programação paralela em CPUs, basicamente leva-se em consideração apenas o número de núcleos e a quantidade de *threads* enviadas a cada núcleo. Na arquitetura CUDA existem outros conceitos e definições que devem ser levados em consideração, portanto, a leitura dos próximos parágrafos é essencial para que exista o entendimento da metodologia deste trabalho.

2.4.1 O *kernel*

O primeiro conceito a ser considerado é a diferença entre *host* e *device*. O *host* é o processador e a memória RAM do *desktop* e o *device* é a GPU que está sendo utilizada.

O método inicial executado no *device* é chamado de *kernel*, ou seja, cada vez que o *host* precisar chamar alguma função ou método da GPU, o *kernel* será chamado. O *kernel* pode ainda fazer chamadas a outras funções que serão executadas pela GPU. Para que seja possível o compilador CUDA diferenciar quais as funções que devem ser compiladas para serem executadas na GPU ou na CPU, alguns identificadores são usados. O identificador **__global__** permite que o compilador entenda que a função será chamada pelo *host* e executada no *device*, portanto identifica o *kernel* da aplicação. As funções que são chamadas e executadas pelo *device*, iniciam com o identificador **__device__**.

2.4.1.1 Chamada e criação do kernel

A chamada do *kernel* sempre será feita pelo *host* por meio de alguma linguagem de programação. Geralmente é utilizada a linguagem C, porém existem *wrappers* em Java, C#, entre outras. A diferença entre a chamada de um *kernel* e da chamada de uma função qualquer em C é que no *kernel* devem-se definir quantos blocos e *threads* serão executados no *device*.

Na linguagem C, a chamada do *kernel* é feita da seguinte maneira (SANDERS; KANDROT, 2004):

Código Fonte 1: Chamada genérica do *kernel*

```
1 nome_kernel<<<num_blocos,num_threads>>>( param1, param2, ... );
```

O "nome_kernel"é o nome da função que será executada no *device*. Logo em seguida, é definida a quantia de blocos ("num_blocos") e *threads*("num_threads") que serão enviados à GPU. Enfim, são enviados os valores dos parâmetros, que como em qualquer outra função em C são definidos na própria função. A definição do *kernel* é igual a uma função em C, porém deve iniciar com o identificador **__global__**. O Código Fonte 2 mostra a definição genérica de um *kernel*.

Geralmente os parâmetros recebidos no *kernel* representam dados e ponteiros de memória. Os ponteiros de memória identificam qual é a posição da memória global que o *kernel* deve escrever os dados. Após a escrita na memória global, uma cópia dos dados é feita na memória RAM possibilitando o *host* fazer a leitura desses dados. Maiores detalhes do fluxo de dados entre o *host* e *device* podem ser vistos da seção 2.5.

Código Fonte 2: Definição genérica do *kernel*

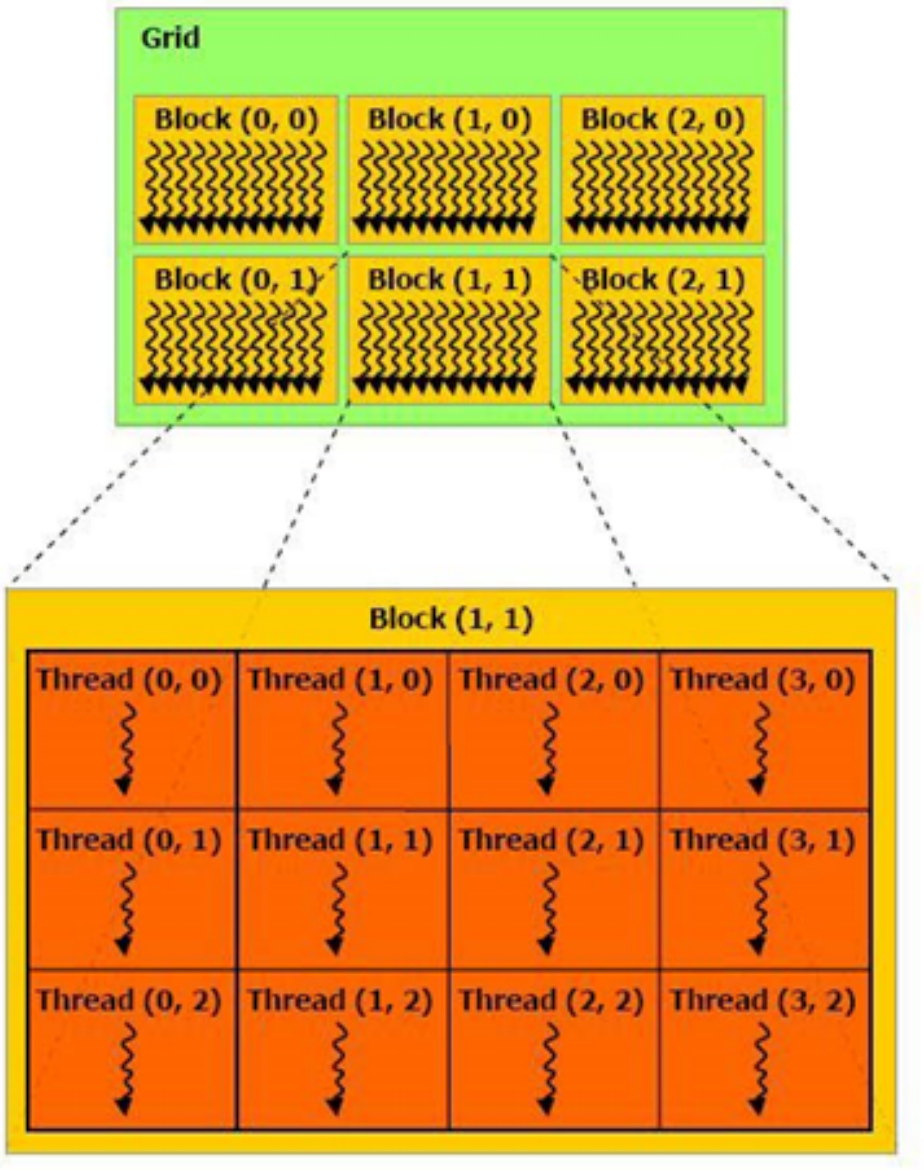
```
1 __global__ nome_kernel(param1, param 2, ...){
2 . . .
3 }
```

2.4.2 *Grids*, blocos e *threads*

Uma *grid* é formada por vários blocos podendo ter duas dimensões (x e y), onde, por exemplo, uma *grid* de dimensão 2x2 possui quatro blocos. Assim como a *grid*, um bloco é formado por um conjunto de *threads*, porém o bloco pode ter três dimensões (x, y e z). Um bloco com dimensão 3x3x4, por exemplo, possui 36 *threads*. A Figura 1 mostra uma *grid* de duas dimensões (x=3 e y=2) contendo seis blocos de duas dimensões (x=4 e

y=3) e cada bloco com 12 *threads*. A identificação de cada elemento (blocos e *threads*) é feito por um determinado *id* (identificador), por exemplo, na figura 1 a última *thread* da *grid* está no bloco com *id* (2, 1) e tem a *id* (3, 2).

Figura 1: Representação de *Grids*, blocos e *threads*



Fonte: *Cuda Programming Guide*

Dentro de um *kernel* é possível identificar as *threads* por variáveis que representam suas *ids*. A representação dos índices das *threads* é dada pelas seguintes variáveis:

threadIdx.x: pode ser entendida como *thread Index* x, e representa a posição x de uma determinada *thread* em um bloco.

threadIdx.y: pode ser entendida como *thread Index* y, e representa a posição y de uma

determinada *thread* em um bloco.

Assim como as *threads* são identificadas em um bloco, os blocos também podem ser identificados dentro de uma *grid*. As variáveis abaixo representam os índices dos blocos:

blockIdx.x: representa o índice x de um bloco na *grid*.

blockIdx.y: representa o índice y de um bloco na *grid*.

blockIdx.z: representa o índice z de um bloco na *grid*.

Para que seja possível percorrer os índices das *threads* em um determinado bloco, o programador deve ter o conhecimento do tamanho de cada dimensão do bloco. As variáveis abaixo representam o tamanho das dimensões em x , y e z.

blockDim.x: representa a dimensão em x do bloco na *grid*.

blockDim.y: representa a dimensão em y do bloco na *grid*.

blockDim.z: representa a dimensão em z do bloco na *grid*.

As variáveis acima permitem que o programador identifique uma *thread*. Para exemplificar como é feita a identificação das *threads*, o exemplo a seguir mostra um *kernel* chamado "principal". A função do *kernel* em questão é escrever em uma determinada posição da memória a *id* da *thread* executada multiplicado por 2. Como a chamada do *kernel* "principal"é feita com 5 blocos e 32 *threads*, a saída do *kernel* deve ser uma *array* de 160 posições (pois são 5 blocos com 32 *threads* cada) com os valores de 0 até 320. A chamada do *kernel* pelo *host* é feita da seguinte maneira:

Código Fonte 3: Chamada do *kernel*

```
1 principal<<<5, 32>>>(ponteiro_saida);
```

A definição do *kernel* citado acima é mostrado a seguir:

Código Fonte 4: Declaração do *kernel*

```
1 __global__ principal(*int ponteiro_saida){
2         int id;
3         id = blockDim.x * blockIdx.x + threadIdx.x;
4         ponteiro_saida[id] =  id * 2;
5 }
```

Portanto, o *kernel* "principal"será executado por 160 *threads*, e cada *thread* uma será identificada pela equação:

$$threadId = blockDim.x * blockIdx.x + threadIdx.x$$

Supondo, por exemplo, que a terceira *thread* do quarto bloco esteja sendo executada, portanto a *id* da *thread* seria:

$$threadId = 32 * 3 + 2$$

Sendo 32 o tamanho do bloco, 3 o número do bloco (os índices começam em zero, portanto o bloco 3 é o quarto bloco) e 2 é a terceira *thread* (as *threads* também começam com 0), a *array* com a posição 98, terá o valor de 196 (resultado da multiplicação 98*2). O código abaixo representa a versão equivalente sequencial do *kernel* "principal":

Código Fonte 5: Versão sequencial

```
1 for ( int i = 0;  i < 160; i++ ){
2         ponteiro_saida[i] = i * 2;
3 }
```

A principal diferença entre a versão sequencial e paralela, é que na versão paralela em apenas uma chamada do *kernel*, 160 *threads* serão lançadas, enquanto na versão sequencial existe um *loop* que será executado 160 vezes com uma única *thread*.

2.4.3 SM e SP

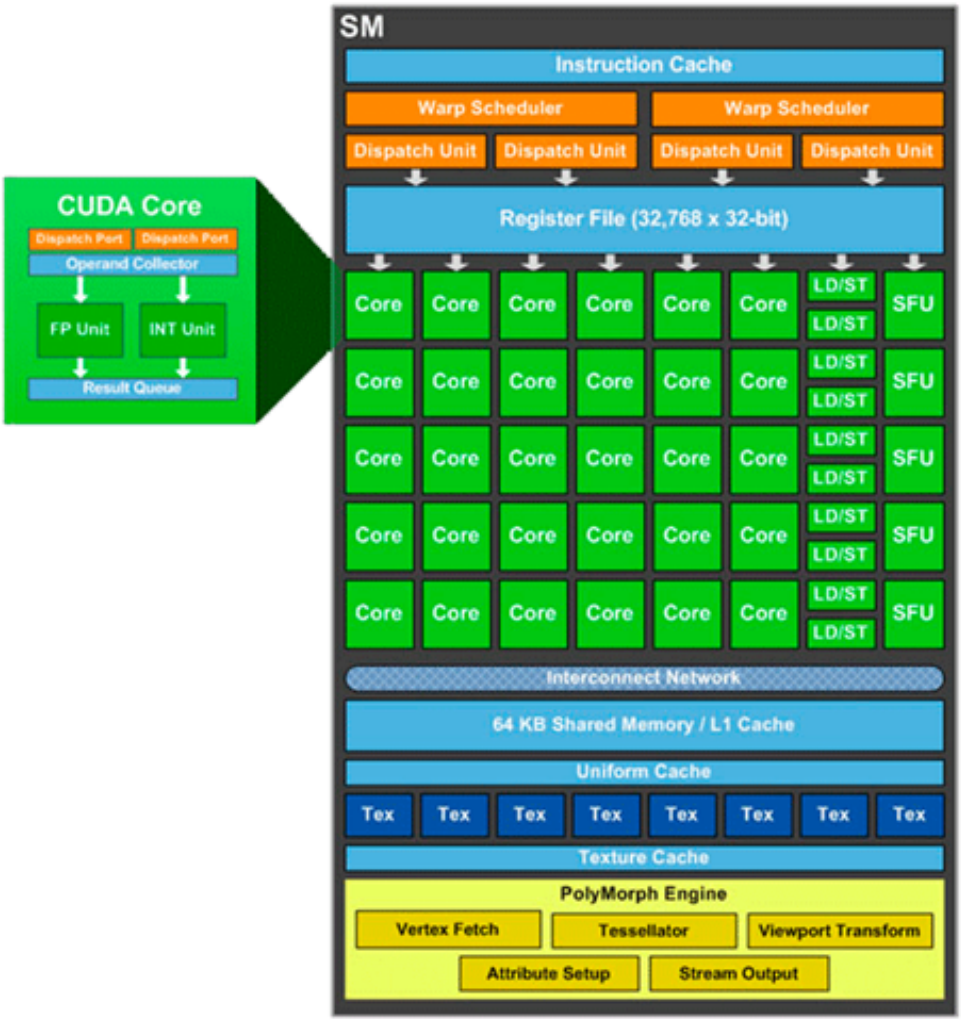
GPUs implementam uma ou mais unidades chamadas SM (*stream-multiprocessor*). Cada SM possui diversos SP (*stream-processors*) que são conhecidos pelo nome de núcleos CUDA, ou *cuda-cores*. Para que seja possível iniciar *threads* em CUDA cada SM deve receber um bloco de *threads*, onde, um bloco pode executar várias *threads* e o número máximo de *threads* varia de cada versão da arquitetura CUDA. O processamento das *threads* em si é feito pelos SP.

A figura 2 abaixo mostra maiores detalhes de uma SM.

Como pode ser visto na figura 2 os componentes principais de uma SM são:

Warp Scheduler: : responsável pelo escalonamento das *threads* nos núcleos. A GPU

Figura 2: Multiprocessador da arquitetura kepler



Fonte: *Cuda Programming Guide*

processa grupos de 32 *threads* chamados de *warp*;

Register File: : é a memória dos registradores e é a memória mais rápida da GPU. Quando uma variável é declarada dentro do *kernel* por exemplo, os dados ficam dentro dos registradores;

Core: São os núcleos, ou *cuda-cores* da GPU. Os núcleos são responsáveis pelo processamento das *threads*. Dentro de cada núcleo existem unidades responsáveis pelo processamento de inteiros (INT *unit*) e ponto flutuante (FP *unit*);

LD/ST: É a abreviação de *load/store*. Sempre que é necessário acesso à memória, essas unidades são utilizadas;

SFU: É a abreviação de *Special Function Unit*. É a unidade responsável pelo cálculo de funções matemáticas como cosseno (__cosf()) , exponencial (__expf()), etc;

Shared Memory: É a memória compartilhada dentro de um bloco. Todos os *cores* de uma SM tem acesso a essa memória;

A quantidade de SM disponível nas GPUs pode variar dependendo da versão do CUDA. Na versão Kepler, por exemplo, existem GPUs com até 15 SM. O próximo tópico irá tratar a arquitetura de memória e as classes de memórias utilizados em CUDA.

2.4.4 Arquitetura de memória CUDA

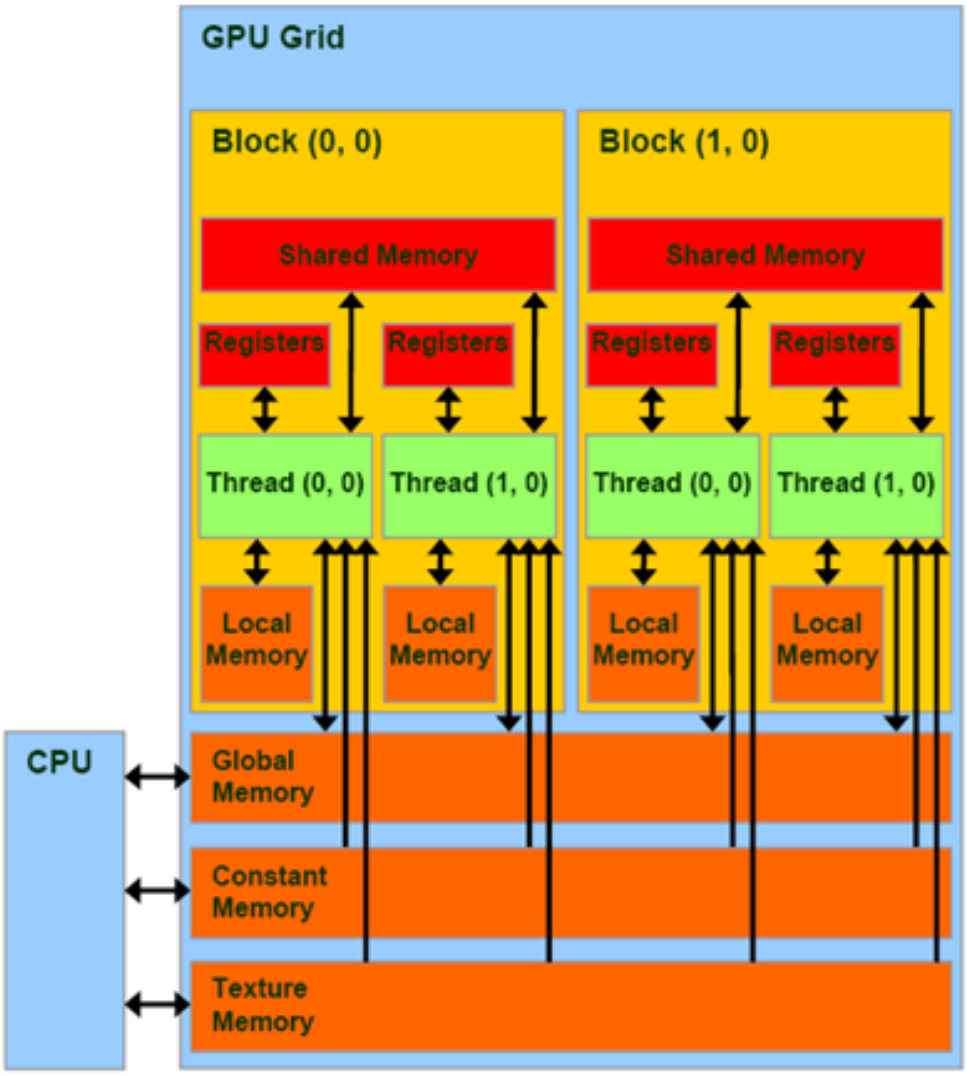
Em CUDA existem seis diferentes classes de memórias que vão desde memórias extremamente rápidas até as memórias mais abundantes que geralmente são mais lentas. É importante que no desenvolvimento de uma aplicação GPGPU, o programador tenha em mente quais os tipos de memória deverá utilizar. Uma *thread* pode acessar todos os tipos de memória. A diferença entre as memórias é a quantidade disponível e o tempo de acesso a cada uma. A Figura 3 mostra os tipos de memória com seus respectivos escopos.

Registers: O registrador é o tipo de memória mais rápida da GPU. O escopo de um registrador limita-se a *thread* em questão, ou seja, uma *thread* não pode ler ou escrever em um registrador de uma outra *thread*. O tempo de acesso para escrita é de aproximadamente 11 ciclos de *clock* (para a versão Kepler) e está localizada dentro do chip da GPU. O *host* não tem acesso a essa memória.

Shared Memory: É a memória compartilhada entre as *threads* de um mesmo bloco, e cada SM contém um banco. O tempo de acesso também é rápido, em torno de 40 ciclos de *clock* para acesso, pois como os registradores, é uma memória que está localizada dentro do chip da GPU. O *host* não tem acesso direto a essa memória.

Local memory: Assim como os registradores, o escopo da memória local limita-se a *thread* em questão. A diferença é que a memória local é localizada fora do chip da

Figura 3: Escopo das memórias CUDA



Fonte: *Cuda Programming Guide*

GPU, portanto o tempo de acesso é alto. Geralmente a memória local é utilizada quando é necessário armazenar grandes quantidades de dados, como *arrays* ou estruturas, que não cabem nos registradores. O *host* não tem acesso direto a essa memória.

Global Memory: É a memória global da GPU e *threads* de diferentes blocos possuem acesso a ela. Por se tratar de uma memória que também está localizada fora do chip da GPU, o tempo de acesso é bastante alto, em torno de 400 a 600 ciclos de *clock* para acesso. Tanto o *host* quanto o *device* tem acesso a essa memória.

Constant memory: Como o nome diz (memória constante), é um tipo de memória somente de leitura. Embora seja uma memória localizada fora do chip da GPU, o seu

processo de leitura é optimizado, portanto mais eficiente do que o processo de leitura da memória global. Tanto o *host* quanto o *device* tem acesso a essa memória.

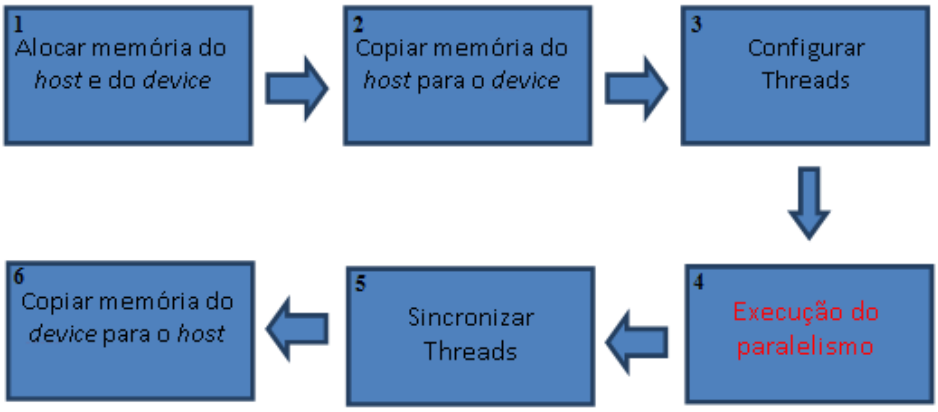
Texture memory: É a memória optimizada para guardar texturas 2D. A busca dessas texturas são eficientes, pois existe a utilização de memória cache que guardam as variáveis espaciais. Tanto o *host* quanto o *device* tem acesso a essa memória.

Explicações mais detalhadas sobre as memórias da GPU podem ser encontradas no *Cuda Toolkit Documentation*[7].

2.5 Fluxo de processamento entre *host* e *device*

O *device* não precisa ter conhecimento do código do *software* por completo, apenas o código paralelo deve ser copiado da memória do *host* para a memória do *device*. O fluxo de processamento de uma aplicação GPGPU é mostrado na figura 4.

Figura 4: Fluxo entre *host* e *device*



Fonte: O autor

Nota-se no fluxo que o primeiro passo é alocar memória no *host* e no *device*. O tamanho de espaço alocado no *device* deve ser igual o do *host*, pois os dados serão copiados do *host* para o *device* como é visto no passo 2. O terceiro passo é a definição de quantas *threads* e blocos serão executados na GPU. Após a execução das *threads*, as mesmas deves ser sincronizadas para que os resultados possam ser copiados do *device* para o *host* para que finalmente, possam ser avaliados pela aplicação.

[7]http://docs.nvidia.com/cuda/cuda-c-programming-guide/

2.6 P2PComp

O *framework* P2PComp (SENGER; SOUZA; FOLTRAN, 2010) foi criado e é mantido por pesquisadores da Universidade Estadual de Ponta Grossa (UEPG). O P2PComp implementa uma infraestrutura P2P utilizando a linguagem Java baseado na biblioteca JXSE[8] que implementa os padrões JXTA[9]. O padrão JXTA consiste em protocolos que permitem que exista uma comunicação de dispositivos interligados em rede, mesmo eles estando por trás de um *firewall*, em um sistema P2P. Essa biblioteca permite que sejam criados anúncios que podem ser obtidos por pares (elementos capazes de realizar processamento) que tem recursos para executar uma determinada aplicação. Cada par é reconhecido por um *peerId*, que consiste em uma *string*.

Quando o *framework* P2PComp é executado, cada *peer* cria um anúncio JXTA com determinadas características do par. As características são informações sobre a quantidade de memória disponível, a capacidade de processamento (em MFLOPS) da CPU, a carga de trabalho atual, e ainda se existe ou não GPU disponível. Além dessas informações, os pares mantém uma matriz de custos de comunicação, que contém o tempo de comunicação entre todos os pares. A frequência de troca de mensagens entre determinados pares pode ser definida em tempo de execução, permitindo que as trocas de mensagens sejam feitas entre pares que tem baixa latência de comunicação entre si. Assim como a frequência de troca de mensagens, a carga de trabalho também pode ser executada em tempo de execução através de algoritmos de escalonamento.

O modelo de programação do P2PComp define que as aplicações computacionais são organizadas em pares trabalhadores, escalonador e proprietário. O *peer* escalonador é responsável pelo escalonamento das *tasks* de uma aplicação entre os *peers* trabalhadores, que são responsáveis pela execução. O *peer* proprietário da aplicação recebe as mensagens com os resultados dos *peers* trabalhadores. Na execução de uma aplicação no P2PComp, é possível organizar os pares pelo uso do índice *rankId*. O *rankId* varia no intervalo de 0 à N - 1, sendo N o número de tarefas de uma aplicação. Por meio desse identificador as tarefas podem realizar comunicação e sincronização.

O P2PComp permite ainda a criação de *peersgroups*. Os *peersgroups* são um agrupamento de *peers* que possuem um conjunto em comum de serviços e interesses. Um exemplo que cabe ao contexto desse trabalho seria a criação de um *peergroup* onde todos os *peers* trabalhadores possuem GPUs. O ingresso de um *peer* em um grupo pode ser feito

[8] https://jxse.kenai.com/
[9] https://jxta.kenai.com/

de diversas maneiras, dependendo da política de membros que se impõe, onde pode ocorrer o livre acesso ou até mesmo autenticação utilizando infraestrutura de chaves públicas e privadas.

2.7 O *software* Venthor

O desenvolvimento de modelos para simulação de séries climáticas é de fundamental importância para o auxílio de tomadas de decisões de produtores, e a simulação de sistemas é importante para a validação destes modelos. O *software* Venthor é um sistema de simulação de vento que e em sua versão original executa os dados da simulação sequencialmente em apenas uma *thread*. O Venthor foi desenvolvido inicialmente na linguagem C# e posteriormente no trabalho de Serckumecka (2012) migrado para Java. O trabalho de Serckumecka (2012) foi de fundamental importância para o desenvolvimento desse trabalho, pois foi a partir da versão de Serckumecka (2012) que a migração de dados de Java para C foi realizada.

O processo completo da execução de uma simulação do *software* Venthor, pode ser dividido em quatro etapas: leitura, cálculo KS, simulação e calculo de direção, e, finalmente, escrita do arquivo de saída.

Na primeira etapa os dados de entrada são obtidos através de um arquivo CSV que contém os dados empíricos. Uma tabela de frequência é gerada contendo informações como média, desvio padrão, amplitude de classe, limites superiores e inferiores. Este trabalho não fez mudanças no código do Venthor que trata dessa primeira parte, portanto, o código que faz a leitura dos arquivos e gera as tabelas de frequência são os mesmos.

Baseado nos dados gerados anteriormente, a segunda etapa consiste em utilizar o teste de Kolmogorov-Smirnov para escolher qual a distribuição estatística dentre Weibull, Gama, Beta e Rayleigh será utilizada para o cálculos das distribuições. O valor de tolerância para a aceitação da distribuição é de 5%, sendo que, caso mais de uma distribuição passe pelo teste, a escolhida será a com menor valor de rejeição.

A terceira etapa é responsável pela maior parte do tempo de execução do Venthor. A simulação é realizada por meio de cálculos de funções inversas das distribuições indicados pelo teste KS. O tempo de processamento aumenta de acordo com o tempo de simulação desejada e o número total de chamadas de função inversa para n anos pode ser

calculada pela equação mostrada abaixo:

$$f(n) = nMeses * nHoras * nAnos * nDias \tag{1}$$

O número de meses (nMeses) e horas (nHoras) tem um valor constante de 12 e 24, respectivamente. Isso quer dizer que para qualquer simulação, o número de chamadas às funções inversas será:

$$f(n) = 288 * nAnos * nDias \tag{2}$$

onde o número da chamadas irá variar com a quantidade de anos a ser simulado. No pior dos casos, em um mês há 31 dias (nDias), portanto, a equação torna-se:

$$f(n) = 288 * nAnos * 31 \tag{3}$$

então

$$f(n) = 8928 * nAnos \tag{4}$$

A partir da equação acima, pode deduzir-se que, o número de chamadas de funções inversas está relacionada com a quantidade de anos (nAnos) a ser simulado. Por um período de 10 anos, por exemplo, Venthor executará pelo menos 89.280 chamadas de funções. Executá-las em paralelo em GPUs torna possível conseguir uma diminuição significativa do tempo de execução.

Na última etapa uma tabela de frequência é criada para direções do vento, e seus valores são definidos entre os oito pontos cardeais (norte, nordeste, leste, sudeste, sul, sudoeste, oeste e noroeste). Um novo banco de dados é criado contendo todas as informações simulação gerada (velocidade do vento simulado e direção), que podem ser exportados para uso no próprio Venthor ou outras aplicações.

2.7.1 Exemplo do Método Utilizado pelo Venthor

Ao iniciar a execução de uma simulação, o Venthor faz a leitura da base de dados empírica e agrupa os dados por hora, mês e ano. A tabela 2 mostra um exemplo dos dados agrupados de uma base de dados de 10 anos:

Tabela 2: Dados agrupados do Venthor

Mês / Hora	00:00	01:00	02:00		23:00
Jan	310 valores	310 valores	310 valores		310 valores
Fev	282 valores	282 valores	282 valores		282 valores
Mar	310 valores	310 valores	310 valores	...	310 valores
Abr	300 valores	300 valores	300 valores		300 valores
		...			
Dez	310 valores	310 valores	310 valores		310 valores

Fonte: O Autor

Os dados são agrupados da hora 00:00 até a hora 23:00 de cada mês do ano. Nota-se que existem 310 valores para a hora 00:00 de janeiro, pois janeiro tem 31 dias e a base é de 10 anos, para fevereiro existem 282 valores pois considerou-se dois anos bissextos, março 310 valores e assim por diante até o mês de dezembro.

Um conjunto de dados como mostrado na tabela 2 é enviado como parâmetro para o método que irá executar o cálculo da tabela de frequência e calcular valores como: média, desvio padrão, valor máximo, valor mínimo, amplitude de classe, e limites inferior e superior. Na sequência é executado para cada tabela as quatro distribuições de probabilidade e é aplicado o teste de Kolmogorov-Smirnov para verificar qual das distribuições melhor representa os dados empíricos. Após definida qual a distribuição vencedora, é executada a função inversa da distribuição que resultará os dados simulados para o período de tempo solicitado.

A descrição do modelo de cada uma das distribuições utilizadas (Weibull, Gama, Beta e Rayleigh) pode ser encontrado no Anexo A.

3 MATERIAIS E MÉTODOS

O Venthor utiliza o modelo de programação *Single Instruction Multiple Data* (SIMD) que é uma característica que o torna um forte candidato a ser paralelizado com CUDA. A migração dos códigos do Venthor foi feita da linguagem Java para a linguagem C, para que posteriormente fosse compilado com CUDA. No geral, o trabalho consistiu em seis etapas principais:

1. Identificar os códigos a serem paralelizados no Venthor;
2. Migrar os métodos identificados da linguagem Java para a linguagem C;
3. Testar isoladamente os códigos em C e comparar os resultados obtidos com os resultados obtidos pelo Venthor;
4. Executar o código com paralelismo na GPU;
5. Otimizar o código realizando os ajustes necessários;
6. Integrar os códigos desenvolvidos ao framework P2PComp.

Os experimentos foram realizados em GPUs NVIDIA GeForce GTX 650, com 1 GB de memória. O computador utilizado é um Intel i7-2600 com sistema operacional Linux, com 3,4 GHz de clock e 8 GB de memória. Na execução em múltiplas GPUs com o framework P2PComp os dois *workers* possuem a mesma configuração acima. O arquivo de entrada utilizado contém 30 anos de dados históricos do clima da região de Lapa / PR. *Drivers* de vídeo diferentes interferem na performance do sistema como um todo, portanto o driver utilizado foi o mesmo nos dois *workers* (v. 331.38).

3.1 Identificação dos códigos a serem paralelizados

Para que fosse possível paralelizar métodos do Venthor com a utilização do CUDA, a primeira etapa foi identificar os principais métodos que são responsáveis pela simulação e que são executados de forma sequencial. Para uma maior eficiência, também foi criado um *profile* utilizando o *profiler* do Visual Studio 2010[1], e após verificar a análise foi concluído que o bloco de código abaixo é o responsável pela parte da simulação e que pode ser paralelizada.

Código Fonte 6: Código a ser paralelizado

[1] http://www.visualstudio.com/pt-br/visual-studio-homepage-vs.aspx

```
1  while (dInicio<dFinal)
2          {
3          b1.ProgressBarStep();
4          for (int j = 0; j <= 23; j++)
5          {
6                  char ajuste = matriz[dInicio.Month, j].distribuicao;
7                  switch (ajuste)
8                  {
9                          case 'G':
10                         velocidade = f1.InvDistGamma(matriz[dInicio.Month,
                               j].p1, `       matriz[dInicio.Month, j].p2);
11                         break;
12                         case 'R':
13                         velocidade = f1.InvDistRaylegh(matriz[dInicio.Month
                               , j].p1);
14                         break;
15                         case 'B':
16                         {
17                         velocidade = f1.InvDistBeta(matriz[dInicio.Month, j
                               ].p1, matriz[dInicio.Month, j].p2);
18                         velocidade = velocidade * matriz[dInicio.Month, j].
                              H + matriz[dInicio.Month, j].menor;
19                         }
20                         break;
21                         case 'W':
22                         velocidade = f1.InvDistWeibull(matriz[dInicio.Month
                               , j].p1, matriz[dInicio.Month, j].p2);
23                         break;
24                 }
25         . . .
26         }
27 }
```

O código consiste em um *loop* que será repetido conforme a quantidade de número de anos a ser simulado (linha 1), por exemplo, se a simulação for do ano 1900 até o ano 2000, o bloco será repetido 100x.

O trabalho do Serckumecka (2012) consistiu em transcrever os códigos do Venthor de C# para Java. Foi feita uma análise da migração dos códigos e decidido que a migração dos códigos desse trabalho seria feito da versão Java e não mais da versão C#. Isso se deve ao fato da versão do Serckumecka (2012) possuir dois pontos relevantes: a) estar mais

bem estruturada quanto a orientação a objetos, tornando a tarefa de reaproveitamento de códigos mais simples e, b) após a migração dos códigos, a performance da versão em Java foi bastante superior à versão em C#.

3.2 Migrar os códigos da linguagem Java para a linguagem C

Após encontrar o método responsável pela simulação, o próximo passo foi a migração dos métodos escritos em Java para a linguagem C. Isso é necessário, pois a API do CUDA é desenvolvida para a linguagem C. Os métodos chamados pelo código (ex: InvDistGamma, InvDistRaybull, etc) foram implementados como funções em C. Todos os atributos foram declarados como variáveis, respeitando seus respectivos escopos.

3.3 Testar isoladamente os códigos em C e comprar os resultados obtidos com os resultados obtidos pelo Venthor

Feita a migração, os códigos em C foram testados sequencialmente com o intuito de comparar os resultados do novo código em C com o código original do Venthor. É fundamental que todos os métodos estatísticos (Weibull, Rayleigh, Beta e Gama) e seus inversos sejam comparados e validados.

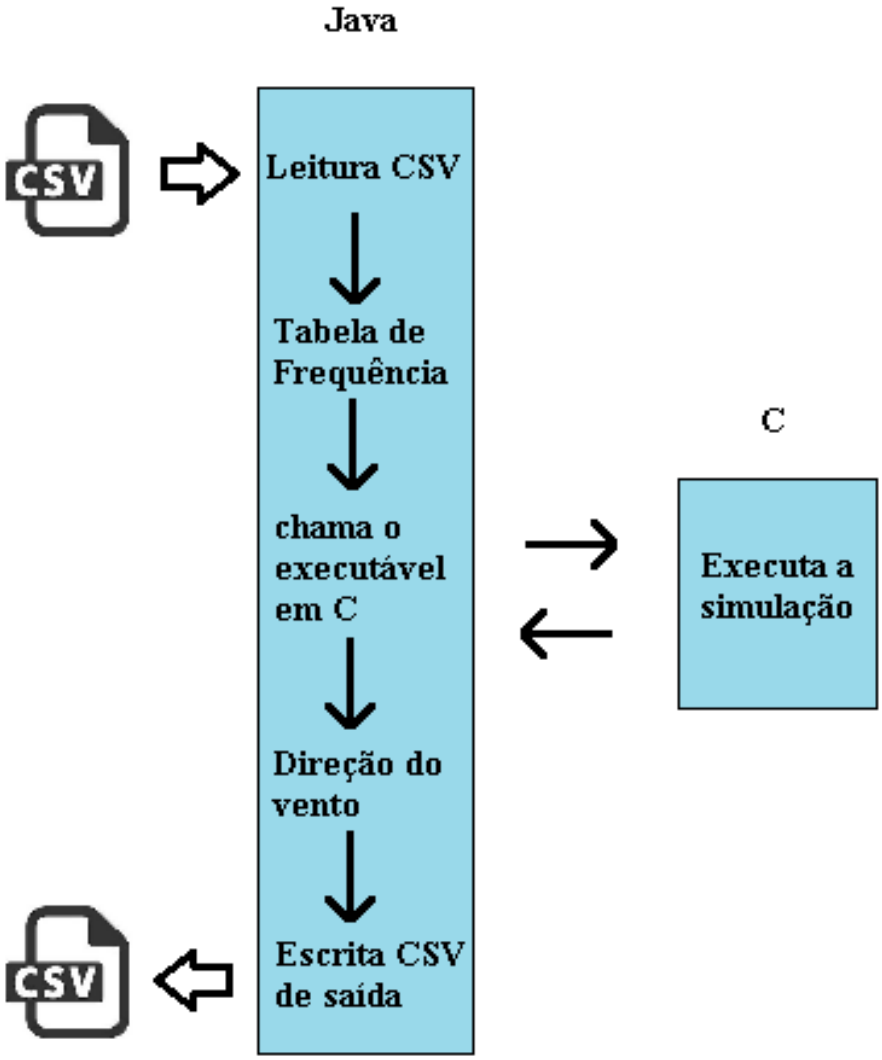
A estrutura do projeto no determinado momento consiste em duas aplicações: a aplicação Java com os códigos responsáveis pela leitura do *input* do Venthor e escrita dos arquivos; e o executável em C responsável pela execução dos métodos estatísticos.

Os testes foram realizados da seguinte maneira: A aplicação Java faz a leitura do arquivo de *input* do Venthor e prepara os dados para a chamada do executável em C. Através do método *ProcessBuilder* do Java o executável em C é chamado enviando como argumento os dados necessários para a simulação. Após a simulação, utiliza-se um *BufferedReader* para ler os resultados da simulação e assim continuar com a execução da aplicação em Java, e finalmente, após o cálculo da direção do vento a aplicação Java faz a escrita dos arquivos de saída do Venthor. A figura 5 mostra o esquema onde a aplicação em C é chamada apenas para executar a simulação.

O Apêndice A mostra o código responsável pela chamada do executável.

Foram comparados os arquivos de saída da versão do Serckumecka (2012) com a versão feita em C utilizando a ferramenta *diff* do Linux, e a ferramenta não apontou diferenças.

Figura 5: Aplicação Java e C



Fonte: O Autor

3.4 Executar o código com paralelismo na GPU

Modificar o código escrito em C para ser executado na GPU foi uma das fases mais importantes do trabalho. Todas as funções escritas em C foram adaptadas para serem executadas na GPU, para isso, foi adicionado o identificador **__device__** nas funções que são chamadas pelo *kernel*. Os valores declarados como constantes que foram utilizados na GPU também receberam o identificador **__device__**. As funções que anteriormente eram utilizadas da biblioteca math.h foram adequadas para serem executadas pelo próprio compilador CUDA.

A chamada do *kernel* feita pelo *host* foi definida:

Código Fonte 7: Chamada do *kernel*

```
1  Principal<<<nAnos, dia>>>(a, nAnos, dia, ajuste, p1, p2, H, menor, PiDir_d,
       randomicos_d, dados_d);
```

A explicação de todos os parâmetros passados para o *kernel* não é de fundamental importância para este trabalho, no entanto, o que realmente deve ser notado é que a cada chamada do *kernel* é enviada a quantidade de blocos igual à quantidade de anos que está sendo simulado e a quantidade de *threads* por bloco como o total de dias do mês que está sendo simulado. A criação do *kernel* foi baseada no *loop* sequencial existente na versão original. Após realizar as adaptações necessárias (como declaração de ponteiros e estruturas de dados) a versão final do *kernel* é mostrada abaixo:

Código Fonte 8: Versão final do *kernel*

```
1  __global__ void Principal(int a, int nAnos, int dia, char ajuste, double
       p1, double p2, double H, double menor, double *PiDir, double *
       randomicos, Dados *dados) {
2
3          double velocidade;
4
5          int i = blockIdx.x;
6          int j = threadIdx.x;
7
8          switch (ajuste) {
9                  case 'G': {
10                 velocidade = ArredondarParaCima(InvDistGamma(p1, p2), 1);
11                         break;
12                 }
13                 case 'R': {
14                         velocidade = ArredondarParaCima(InvDistRaylegh(p1),
                               1);
15                         break;
16                 }
17                 case 'B': {
18                         velocidade = InvDistBeta(p1, p2);
19                         velocidade = ArredondarParaCima((velocidade * H +
                               menor), 1);
20                         break;
21                 }
22                 case 'W': {
23                         velocidade = ArredondarParaCima(InvDistWeibull(p1,
                               p2), 1);
24                         break;
```

```
25                  }
26          }
27          if (velocidade < 0.1) {
28                  velocidade = 0.1;
29          }
30          int idx = blockIdx.x * blockDim.x + threadIdx.x;
31          dados[idx].j = j + 1;
32          dados[idx].i = i + 1;
33          dados[idx].direcao2 = SimulaDirecao(PiDir);
34          dados[idx].direcao = SimulaDirecao2(dados[idx].direcao2);
35          dados[idx].velocidade = velocidade;
36  }
```

Na linha 1 o identificador **__global__** permite que o compilador identifique que a função em questão deve ser compilada para ser executada na GPU. Na linha 3 é criada a variável velocidade, que armazena a velocidade do vento que está sendo simulado na *thread* em questão. A variável é do tipo registrador, pois é utilizada diversas vezes e o seu escopo é utilizada apenas pela própria *thread*. As linhas 5 e 6 representam o índice i, que é o bloco que está sendo executado e o índice j que representa o dia do mês em questão. Da linha 8 até a 26 é feita a execução da simulação propriamente dita. As funções inversas são chamadas de acordo com a variável "ajuste". A linha 30 cria o índice da *thread*, que é utilizado para guardar os valores em uma posição de memória única. Maiores detalhes da equação da linha 30 pode ser visto na seção 2.4.2. Por fim, da linha 31 até a 35, os dados referentes ao ano, dia, direção e velocidade do vento são armazenados em uma posição de uma estrutura de dados.

A estrutura de dados do retorno do *kernel* tem o seguinte formato:

Código Fonte 9: Estrutura que o *device* retorna

```
1
2   struct Dados{
3           // j possui o valor do dia
4           int j;
5
6           // mes que está sendo executado
7           int mes;
8
9           // i possui o valor do ano
10          int i;
11
12          // velocidade encontrada
```

```
13          double velocidade;
14
15          // direção 1 e 2
16          int direcao;
17
18          int direcao2;
19  };
```

Para alocar o espaço de memória do *host*, foi utilizado o malloc():

Código Fonte 10: Alocação de memória no *host*

```
1  dados_h = (Dados *) malloc(nAnos * dia * sizeof(Dados));
```

Para alocar o espaço de memória no *device*, foi utilizado o cudaMalloc():

Código Fonte 11: Alocação de memória no *device*

```
1  cudaMalloc((void **) &dados_d, nAnos * dia * sizeof(Dados));
```

Após o fim da simulação e a sincronia das *threads*, é necessário transferir os dados da memória global do *device* para a memória do *host*. Para a transferência de dados foi utilizada a função cudaMemcpy():

Código Fonte 12: Cópia de dados entre *device* e *host*

```
1  cudaMemcpy(dados_h, dados_d, sizeof(Dados) * dia * nAnos,
       cudaMemcpyDeviceToHost);
```

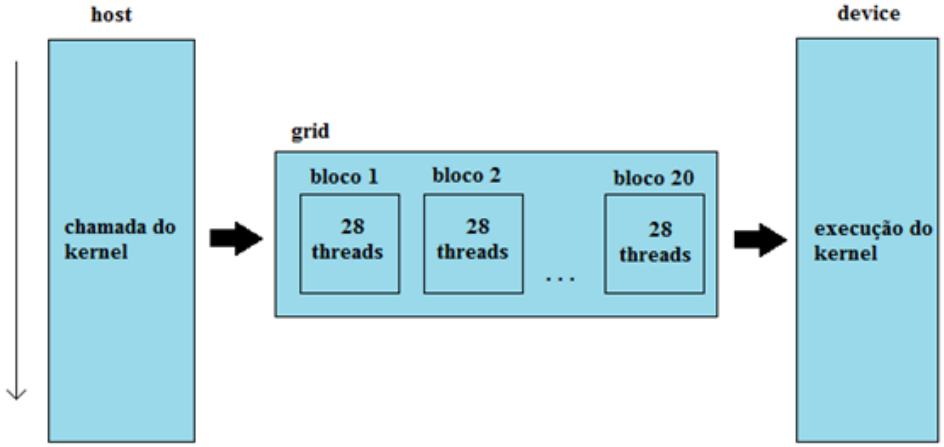
3.4.1 Quantidade de *threads* e blocos

Para que a aplicação seja executada com o máximo de eficiência na GPU, é fundamental que seja definido quantos blocos e *threads* serão criados, pois a variação desses pode influenciar na eficiência do processamento. Existem diferentes tipos de abordagens que poderiam ter sido utilizadas, e a abordagem escolhida, como mostrado na seção 3.4, foi o número de *threads* enviados à GPU é igual ao número de dias do mês que está sendo simulado, e a quantidade de blocos é igual ao número de anos da simulação. CAMARGO e M.A. (2013) fez uma avaliação da estrutura do Venthor na execução em GPGPU e concluiu que a abordagem utilizada é satisfatória.

A figura 6 ilustra uma simulação de 20 anos onde o mês em questão é o mês de fevereiro. O *host* cria uma *grid* que é composta de 20 blocos e cada bloco com 28 *threads*.

Considerando que o *warp* possui o tamanho de 32, essa abordagem poderia ser melhorada tentando criar blocos com o número de *threads* múltiplos de 32.

Figura 6: *Grid* sendo enviada ao *device*



Fonte: O Autor

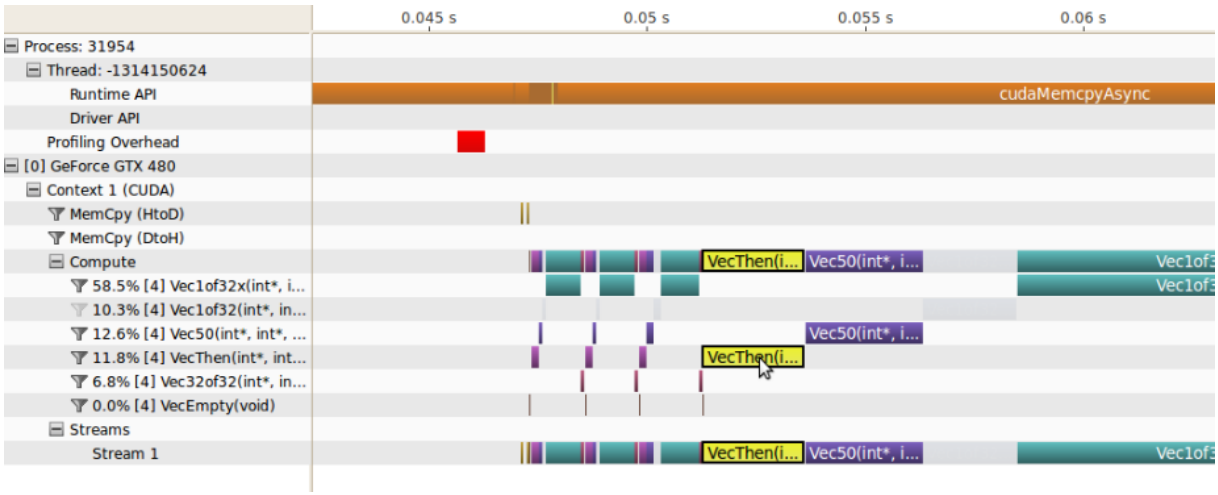
3.5 Optimização dos códigos realizando os ajustes necessários

Nessa etapa foi realizado um estudo a fundo dos manuais CUDA para que as melhores práticas fossem identificadas e aplicadas nos códigos. Desta maneira, foi possível detectar que melhorias poderiam ser feitas, como por exemplo aumentar a eficiência entre a comunicação da memória do *host* e *device*. Todavia, a melhor forma de encontrar gargalos de uma aplicação GPGPU utilizando CUDA é criando um *profile* do código através do software Nsight[2]. Com o Nsight é possível obter informações de taxa de transferência de dados entre memórias e taxa de utilização dos *cores*. É possível ainda criar uma *timeline* mostrando todas essas informações. Outra característica notável do Nsight é a possibilidade de debugar o código ao nível de GPU, criando a possibilidade de visualizar erros a um nível bastante baixo. A figura 7 mostra o profiler do Nsight sendo executado:

Ao lado esquerdo é possível observar os nomes das funções que são executadas pelo *host* e pelo *device*. É possível observar, por exemplo, que a função MemCpy(HtoD) que é responsável por fazer a cópia dos dados da memória do *host* para o *device*, tem um tempo de execução baixo quando comparada, por exemplo, com a função Vec50, que ocupou 12,6% do tempo total de processamento da GPU. Esse tipo de análise torna

[2]https://developer.nvidia.com/nvidia-nsight-visual-studio-edition

Figura 7: NSight *Profiler*



Fonte: NVIDIA [3]

possível detectar quais são os possíveis gargalos da aplicação.

A aplicação nesta parte do projeto consiste em três partes principais. A primeira é a aplicação Java, a qual é responsável pela leitura/escrita dos arquivos e pela montagem de tabelas de frequência e análise de dados. A segunda é o executável em C que é responsável por receber e enviar os dados da simulação e ainda se comunicar com o *kernel*, e, a terceira parte é composta pelo *kernel* e as funções executadas pela GPU. A figura 8 mostra o esquema entre a aplicação JAVA e C.

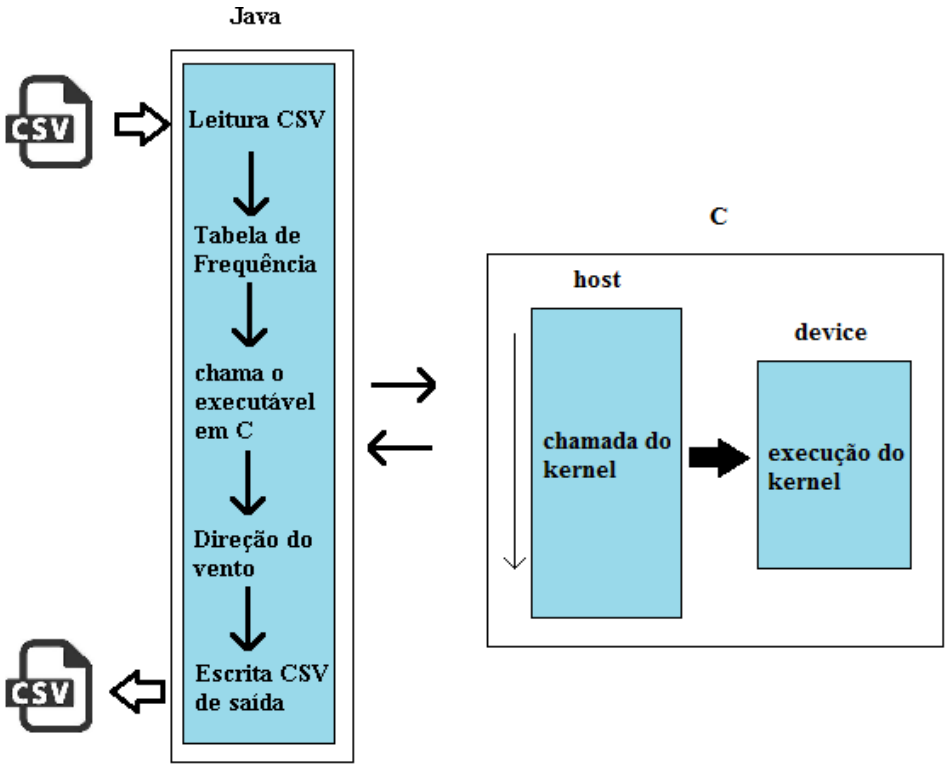
3.6 Execução no P2PComp

A última fase deste trabalho foi a implementação da GPGPU para ser executada em múltiplas GPUs em uma infraestrutura de rede. O *framework* P2PComp implementa uma API similar ao do MPI (BURNS *et al.*, 1994), que é um padrão para o desenvolvimento de aplicações paralelas, portanto a adaptação da aplicação para o *framework* não apresentou muitos problemas. O código abaixo mostra um trecho da classe da implementação do Venthor no *framework*:

Código Fonte 13: P2PComp instanciando a classe do Venthor

```
1 if (this.getSize() == 3) {
2         numAnos = (Integer) vv.get(144);
3         dataInic = (String) vv.get(145);
4         try {
5                 List<String> args2 = newArrayList<String>();
6                 args2.add("/home/usuario/Venthor" + this.getRank());
7                 Process process = newProcessBuilder(args2).start();
```

Figura 8: Aplicação Java e C - *host* e *device*



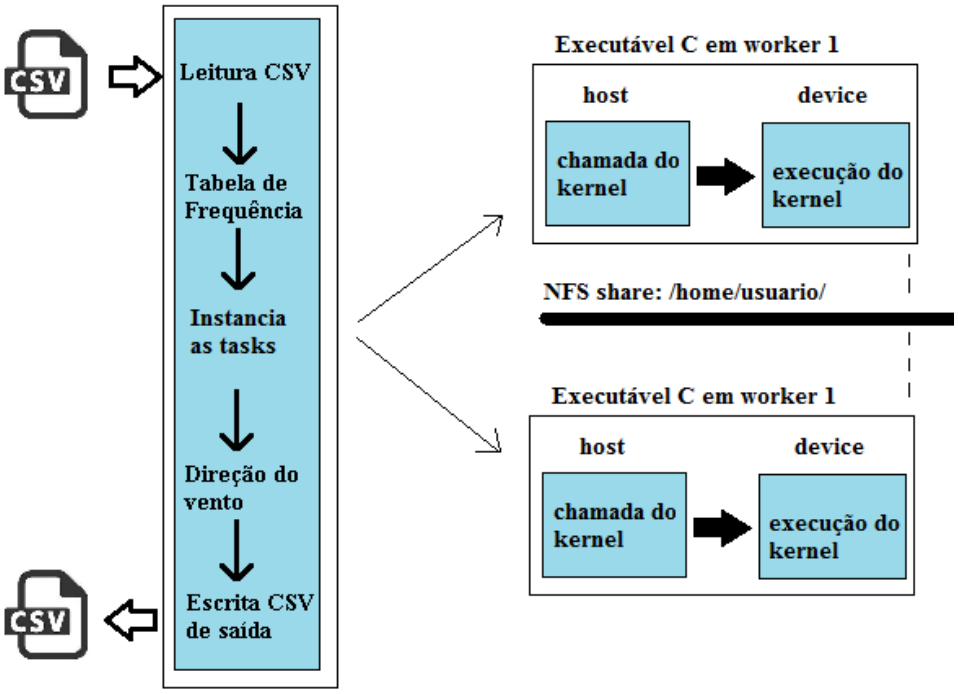
Fonte: O autor

```
8                   InputStream in = process.getInputStream();
9                   InputStreamReader isr = newInputStreamReader(in);
10                  BufferedReader br = newBufferedReader(isr);
11                  OutputStreamWriter ow = newOutputStreamWriter(
12                                          process.getOutputStream());
13                  BufferedWriter out = newBufferedWriter(
14                                          newOutputStreamWriter(process.
                                                getOutputStream()));
15
16                  synchronized (ret) {
17                          for (int i = 0; i< 144; i++) {
18                                  base = (String) vv.get(i);
19                                  ret.add(simul.start(base, numAnos, dataInic
                                        , process, out, in,br));
20                                  }
21                          }
22                  } catch (Exception e) {
23                  System.out.println(e.getMessage());
24          }
25  }
```

A linha 1 verifica se o tamanho do *rank* é igual a três. O tamanho três quer dizer

que existem dois processos que serão trabalhadores e um que será mestre. O processo mestre é responsável por enviar e receber os dados dos processos trabalhadores. As linhas 2 e 3 guardam o número de anos e a data inicial que será iniciada no trabalhador em questão. A linha 6 define qual é o arquivo executável que cada trabalhador irá utilizar, caso seja o trabalhador 1 irá utilizar o executável "Venthor1"e caso seja o *worker 2* irá utilizar o "Venthor2". A abordagem de utilizar executáveis diferentes para cada *worker* é necessária, pois duas máquinas virtuais java não conseguem abrir o mesmo arquivo ao mesmo tempo. Das linhas 8 à 14 são instanciados o *ProcessBuilder*, *InputStreamReader*, *OutputStreamReader* e seus respectivos *buffers*. A linha 18 é um *loop* de 144 repetições, que consiste em exatamente metade do trabalho de uma simulação completa, portanto, cada *worker* irá processar 6 meses com 24 horas diárias, completando assim 144 repetições. Por fim, linha 19 chama a classe que trata os dados e faz a simulação. Os arquivos acessados pelos *workes* estão em um diretório compartilhado pelo protocolo NFS. Sendo assim, todos os *workers* tem acesso aos executáveis, arquivos de entrada e saída. A figura 9 ilustra o esquema de funcionamento do Venthor no P2PComp.

Figura 9: Aplicação Java e C no P2PComp



Fonte: O autor

4 RESULTADOS E DISCUSSÕES

Os resultados apresentados a seguir mostram os tempos de execução do Venthor. Foram simulados 100, 200, 300, 400 e 500 anos na versão sequencial e na versão com *peers* equipados com uma e duas GPUs. Para a comparação com os tempos de execução do trabalho do Serckumecka (2012), foram executados também 10, 20, 50 e 100 anos, e todos os testes foram executados 10 vezes. Para comparar as amostras, foi utilizados o teste não paramétrico de Kruskal-Wallis.

4.1 Venthor executado com GPU e *framework* P2PComp

Os primeiros resultados mostram os tempos de execução sequencial e utilizando o *framework* P2PComp.

4.1.1 Execução do Venthor Sequencial

A tabela 3 mostra os tempos de execução do Venthor para a versão sequencial (em C).

Tabela 3: Tempos de execução: Sequencial em C

Teste/Anos	**500**	**400**	**300**	**200**	**100**
Teste 1	01:56:41	01:14:22	00:42:21	00:19:46	00:04:09
Teste 2	01:54:51	01:14:50	00:43:21	00:19:49	00:04:11
Teste 3	01:56:06	01:15:21	00:42:02	00:19:25	00:04:10
Teste 4	01:57:02	01:15:06	00:42:08	00:19:35	00:04:00
Teste 5	01:57:29	01:15:08	00:41:20	00:19:40	00:04:29
Teste 6	01:56:35	01:14:53	00:42:39	00:19:33	00:04:05
Teste 7	01:56:19	01:13:52	00:41:57	00:19:09	00:04:01
Teste 8	01:56:43	01:14:15	00:41:50	00:19:32	00:04:03
Teste 9	01:55:55	01:14:41	00:41:57	00:19:54	00:04:06
Teste 10	01:58:17	01:14:10	00:42:07	00:19:00	00:04:00
Média	01:56:36	01:14:40	00:42:10	00:19:32	00:04:07
Desvio Padrão	00:00:55	00:00:29	00:00:32	00:00:17	00:00:09
p-value (uma GPU)	<0,05	<0,05	<0,05	<0,05	<0,05
p-value (duas GPU)	<0,05	<0,05	<0,05	<0,05	<0,05

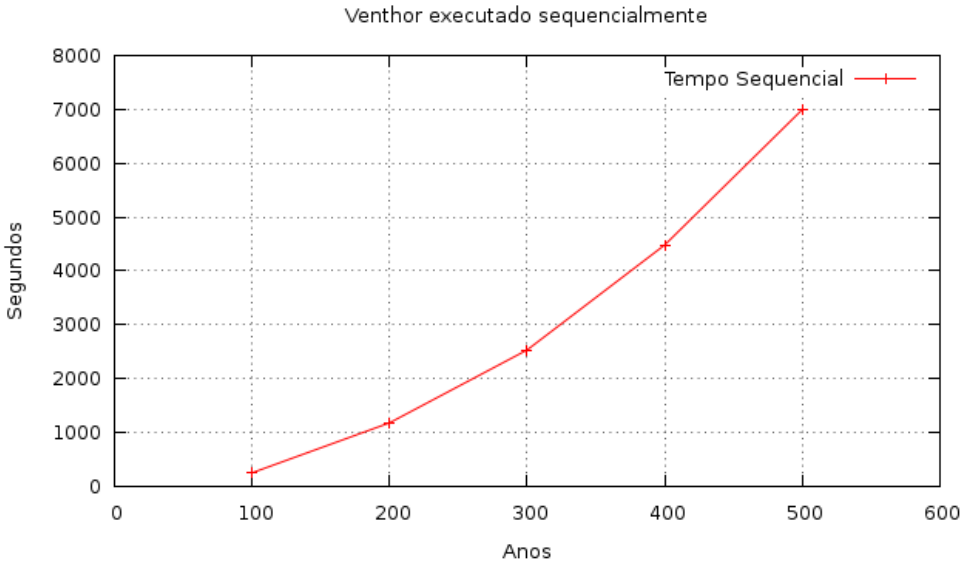
Fonte: O autor

Os valores de *p-value* foram obtidos comparando a amostra sequencial com as amostras obtidas com uma e duas GPUs. Nota-se que o *p-value* em todos os casos foi menor que 0,05, portanto, estatisticamente as amostras diferem-se entre si. O maior desvio

padrão encontrado foi de 55 segundos para a simulação de 500 anos, e o menor desvio padrão foi de 9 segundos para a simulação de 100 anos.

A figura a seguir mostra graficamente os tempos de execução médios da tabela acima.

Figura 10: Tempo médio sequencial



Fonte: *O autor*

4.1.2 Execução do Venthor com uma GPU

Na tabela 4 são mostrados os tempos de execução em uma GPU para as 10 repetições comparando com a execução sequencial e com duas GPUs. A tabela mostra ainda o *speedup* comparando com a versão sequencial.

Os valores de *p-value* foram obtidos comparando a amostra sequencial e as amostras obtidas com duas GPUs. Nota-se que o *p-value* em todos os casos foi menor que 0,05, portanto, estatisticamente as amostras diferem-se entre si. Nota-se também, que, como o tempo de execução é significativamente baixo, o desvio padrão máximo foi de 1,03 segundos para a simulação de 500 anos. O maior *speedup* alcançado foi de 76x para a simulação de 500 anos. Além disso, nota-se que o aumento de velocidade melhora à medida que o número de anos aumenta. Esse fato pode ser explicado porque o tempo de processamento na aplicação Java é constante, e, aumentando a quantidade de anos a ser simulado, maior será o tempo gasto na GPU.

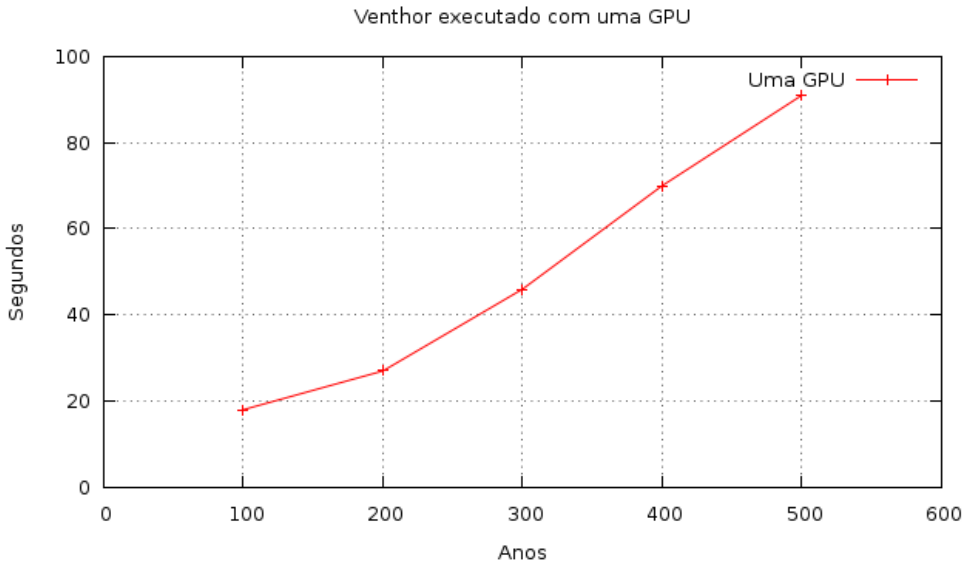
A figura 11 mostra graficamente os tempos de simulação para a execução com uma GPU.

Tabela 4: Tempos de execução: Um *peer* com GPU

Teste/Anos	**500**	**400**	**300**	**200**	**100**
Teste 1	00:01:31	00:01:09	00:00:46	00:00:28	00:00:19
Teste 2	00:01:32	00:01:11	00:00:47	00:00:28	00:00:19
Teste 3	00:01:31	00:01:11	00:00:45	00:00:29	00:00:18
Teste 4	00:01:33	00:01:12	00:00:46	00:00:27	00:00:21
Teste 5	00:01:33	00:01:09	00:00:46	00:00:27	00:00:19
Teste 6	00:01:32	00:01:10	00:00:45	00:00:29	00:00:18
Teste 7	00:01:33	00:01:09	00:00:47	00:00:28	00:00:19
Teste 8	00:01:31	00:01:10	00:00:45	00:00:26	00:00:18
Teste 9	00:01:30	00:01:11	00:00:46	00:00:28	00:00:18
Teste 10	00:01:32	00:01:09	00:00:47	00:00:28	00:00:19
Média	00:01:32	00:01:10	00:00:46	00:00:28	00:00:19
Desvio Padrão	00:00::01	00:00::01	00:00::01	00:00::01	00:00::01
speedup (sequencial)	**76,20**	63,90	55,00	42,16	13,15
p-value obtido (sequencial)	<0,05	<0,05	<0,05	<0,05	<0,05
p-value obtido (2 GPUs)	<0,05	<0,05	<0,05	<0,05	<0,05

Fonte: O autor

Figura 11: Tempo médio com uma GPU



Fonte: *O autor*

4.1.3 Execução do Venthor com duas GPUs

A tabela 5 mostra a execução do Venthor com a execução de dois *peers* com uma GPU cada utilizando o *framework* P2PComp. Também é mostrado o *speedup* comparando com a versão sequencial e com o *peer* equipado com uma GPU.

Os valores de *p-value* foram obtidos comparando a amostra sequencial e as amostras obtidas com uma GPU. Nota-se que o *p-value* em todos os casos foi menor que 0,05,

Tabela 5: Tempos de execução: Dois *peers* com GPU

Teste/Anos	**500**	**400**	**300**	**200**	**100**
Teste 1	00:00:45	00:00:37	00:00:25	00:00:17	00:00:16
Teste 2	00:00:44	00:00:36	00:00:26	00:00:16	00:00:16
Teste 3	00:00:46	00:00:36	00:00:23	00:00:16	00:00:16
Teste 4	00:00:42	00:00:33	00:00:21	00:00:13	00:00:17
Teste 5	00:00:43	00:00:34	00:00:22	00:00:16	00:00:16
Teste 6	00:00:47	00:00:36	00:00:21	00:00:16	00:00:16
Teste 7	00:00:45	00:00:35	00:00:21	00:00:15	00:00:16
Teste 8	00:00:44	00:00:34	00:00:24	00:00:16	00:00:16
Teste 9	00:00:46	00:00:36	00:00:25	00:00:16	00:00:17
Teste 10	00:00:42	00:00:34	00:00:24	00:00:16	00:00:17
Média	00:00:44	00:00:35	00:00:23	00:00:16	00:00:16
Desvio Padrão (s)	00:00:02	00:00:01	00:00:02	00:00:01	00:00:01
speedup (sequencial)	**157,56**	127,62	109,06	74,66	15,17
speedup (uma GPU)	**2,067**	1,99	1,98	1,77	1,15
p-value obtido (sequencial)	<0,05	<0,05	<0,05	<0,05	<0,05
p-value obtido (uma GPU)	<0,05	<0,05	<0,05	<0,05	<0,05

Fonte: O autor

portanto, estatisticamente as amostras diferem-se entre si. Pela tabela, é possível notar que executando o Venthor em 2 GPUs existe um *speedup* de 157x quando comparada a versão sequencial em uma simulação de 500 anos. Quando a comparação é feita com uma GPU, é possível notar que o *speedup* para 300, 400 e 500 anos é próximo de dois, pois a carga é dividida igualmente entre as GPUs.

A figura 12 mostra graficamente os tempos de simulação para a execução com dois *peers* equipados com uma GPU cada.

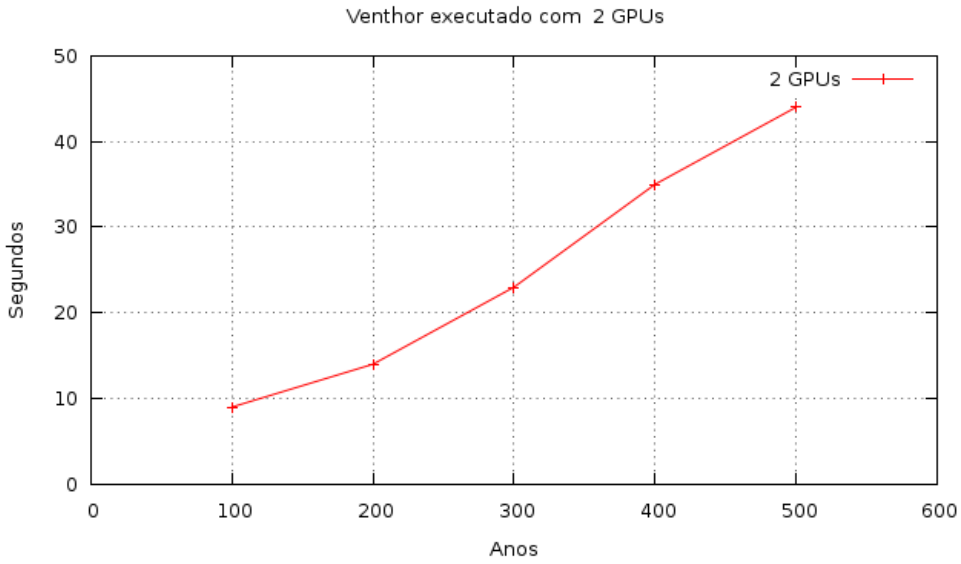
4.2 *Peers* GPU vs. *peers* CPU

O trabalho do Serckumecka (2012) simulou o Venthor com o *framework* P2PComp utilizando a mesma base de dados utilizada nesse trabalho, entretanto foram utilizadas apenas *peers* tradicionais, sem GPUs. A comparação entre os resultados foi feita com simulações de 10, 20, 50 e 100 anos.

4.2.1 Tempos de execução de 10 anos

A tabela 6 mostra a médias dos resultados de simulações de 10 anos onde foi comparado o tempo de execução sequencial, com 6, 14 e 26 *peers* (CPU) com o tempo da execução de uma e duas GPUs.

Figura 12: P2PComp com duas GPUs



Fonte: *O autor*

Tabela 6: Tempos de execução: GPU vs. CPU *peers* - 10 anos

10 anos	**Sequencial**	**6 *peers***	**14 *peers***	**26 *peers***	**GPU**	**2 GPU**
1ª	00:00:07	00:00:13	00:00:11	00:00:13	00:00:17	00:00:16
2ª	00:00:06	00:00:12	00:00:10	00:00:12	00:00:16	00:00:16
3ª	00:00:06	00:00:12	00:00:10	00:00:12	00:00:15	00:00:17
4ª	00:00:06	00:00:12	00:00:10	00:00:12	00:00:16	00:00:16
5ª	00:00:06	00:00:12	00:00:11	00:00:12	00:00:15	00:00:16
6ª	00:00:06	00:00:13	00:00:10	00:00:12	00:00:17	00:00:16
7ª	00:00:07	00:00:12	00:00:10	00:00:12	00:00:15	00:00:17
8ª	00:00:06	00:00:12	00:00:10	00:00:13	00:00:16	00:00:16
9ª	00:00:06	00:00:12	00:00:10	00:00:12	00:00:16	00:00:16
10ª	00:00:06	00:00:12	00:00:10	00:00:12	00:00:16	00:00:16
Média	00:00:06	00:00:12	00:00:10	00:00:12	00:00:16	00:00:16
Desvio Padrão	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
p-value obtido (1 GPU)	<0,05	<0,05	<0,05	<0,05		
p-value obtido (2 GPU)	<0,05	<0,05	<0,05	<0,05		

Fonte: O autor

Os valores de *p-value* foram obtidos comparando as amostras do trabalho de Serckumecka (2012) com as amostras da execução de uma e duas GPUs. Nota-se que o *p-value* em todos os casos foi menor que 0,05, portanto, estatisticamente as amostras diferem-se entre si. É possível notar que na simulação de 10 anos, os *peers* equipados com uma e duas GPUs tem um desempenho inferior às execuções dos *peers* sem GPU. É fundamental ressaltar que existe um tempo fixo necessário para iniciar a execução nas GPUs, portanto em simulações onde a quantidade de anos é baixa, não existe vantagem de tempo para a GPU.

4.2.2 Tempos de execução de 20 anos

A tabela 7 mostra os tempos de execução para simulações de 20 anos.

Tabela 7: Tempos de execução: GPU vs. CPU *peers* - 20 anos

20 anos	**Sequencial**	**6 *peers***	**14 *peers***	**26 *peers***	**GPU**	**2 GPU**
1ª	00:00:18	00:00:24	00:00:15	00:00:16	00:00:17	00:00:16
2ª	00:00:17	00:00:23	00:00:14	00:00:15	00:00:18	00:00:17
3ª	00:00:17	00:00:23	00:00:14	00:00:15	00:00:18	00:00:17
4ª	00:00:17	00:00:22	00:00:14	00:00:15	00:00:17	00:00:17
5ª	00:00:17	00:00:23	00:00:14	00:00:14	00:00:18	00:00:16
6ª	00:00:16	00:00:23	00:00:14	00:00:15	00:00:17	00:00:16
7ª	00:00:17	00:00:23	00:00:14	00:00:15	00:00:19	00:00:16
8ª	00:00:16	00:00:23	00:00:14	00:00:15	00:00:18	00:00:16
9ª	00:00:17	00:00:23	00:00:13	00:00:15	00:00:18	00:00:16
10ª	00:00:17	00:00:23	00:00:14	00:00:14	00:00:18	00:00:17
Média	00:00:17	00:00:23	00:00:14	00:00:15	00:00:18	00:00:16
Desvio Padrão	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
p-value obtido (1 GPU)	<0,05	<0,05	<0,05	<0,05		
p-value obtido (2 GPU)	0,695	<0,05	<0,05	<0,05		

Fonte: O autor

É possível notar que mesmo com uma simulação de 20 anos, a diferença de tempo entre os *peers* com e sem GPU é pequena. O *p-value* encontrado quando comparado a amostra sequencial com a amostra de uma GPU é de 0,695. Esse foi o único caso em que é possível afirmar que não existe diferença notável entre a execução com e sem GPU. Para todos os outros *p-values* o valor é menor que 0,05, portanto as amostras diferem-se estatisticamente.

4.2.3 Tempos de execução de 50 anos

A tabela 8 mostra os tempos de execução para simulações de 50 anos.

Nota-se que o *p-value* em todos os casos foi menor que 0,05, portanto, estatisticamente as amostras diferem-se entre si. É possível notar que para a simulação de 50 anos, o tempo de execução com *peers* equipados com GPU é menor que os *peers* sem GPU.

4.2.4 Tempos de execução de 100 anos

A tabela 9 mostra os tempos de execução para simulações de 100 anos.

Nota-se que o *p-value* em todos os casos foi menor que 0,05, portanto, estatisti-

Tabela 8: Tempos de execução: GPU vs. CPU *peers* - 50 anos

50 anos	**Sequencial**	**6 *peers***	**14 *peers***	**26 *peers***	**GPU**	**2 GPU**
1ª	00:01:36	00:01:38	00:00:44	00:00:34	00:00:20	00:00:16
2ª	00:01:34	00:01:36	00:00:42	00:00:33	00:00:19	00:00:17
3ª	00:01:34	00:01:36	00:00:43	00:00:33	00:00:20	00:00:17
4ª	00:01:33	00:01:36	00:00:43	00:00:33	00:00:21	00:00:17
5ª	00:01:34	00:01:37	00:00:43	00:00:33	00:00:20	00:00:17
6ª	00:01:34	00:01:36	00:00:43	00:00:33	00:00:20	00:00:17
7ª	00:01:34	00:01:36	00:00:43	00:00:33	00:00:19	00:00:16
8ª	00:01:34	00:01:36	00:00:43	00:00:32	00:00:19	00:00:17
9ª	00:01:34	00:01:35	00:00:42	00:00:33	00:00:20	00:00:16
10ª	00:01:34	00:01:36	00:00:43	00:00:33	00:00:20	00:00:17
Média	00:01:34	00:01:36	00:00:43	00:00:33	00:00:20	00:00:17
Desvio Padrão	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
p-value obtido (1 GPU)	<0,05	<0,05	<0,05	<0,05		
p-value obtido (2 GPU)	<0,05	<0,05	<0,05	<0,05		

Fonte: O autor

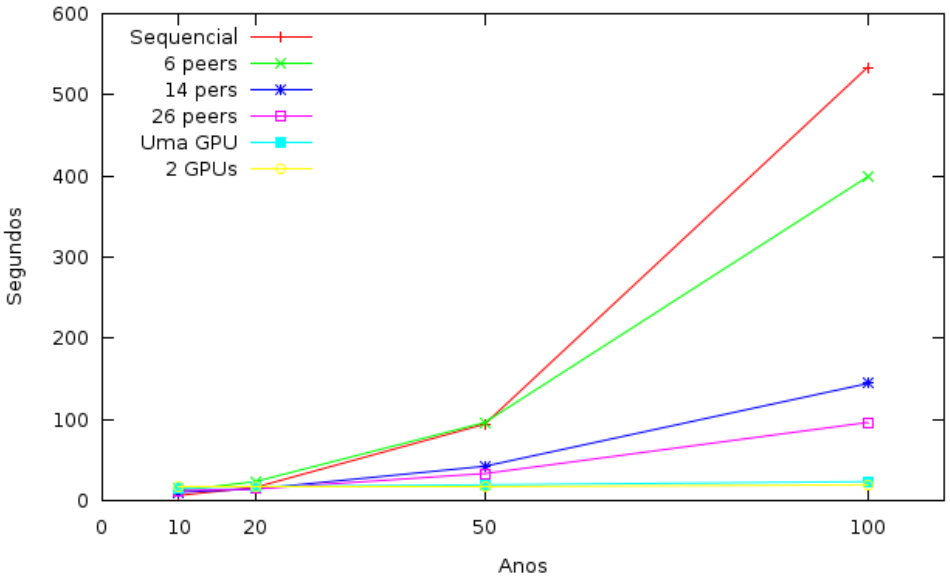
Tabela 9: Tempos de execução: GPU vs. CPU *peers* - 100 anos

100 anos	**Sequencial**	**6 *peers***	**14 *peers***	**26 *peers***	**GPU**	**2 GPU**
1ª	00:08:57	00:06:41	00:02:25	00:01:38	00:00:24	00:00:17
2ª	00:08:54	00:06:38	00:02:24	00:01:36	00:00:24	00:00:19
3ª	00:08:54	00:06:39	00:02:24	00:01:36	00:00:22	00:00:19
4ª	00:08:53	00:06:39	00:02:24	00:01:36	00:00:23	00:00:19
5ª	00:08:54	00:06:39	00:02:24	00:01:36	00:00:24	00:00:18
6ª	00:08:54	00:06:39	00:02:24	00:01:35	00:00:24	00:00:19
7ª	00:08:54	00:06:39	00:02:24	00:01:36	00:00:23	00:00:19
8ª	00:08:53	00:06:39	00:02:23	00:01:36	00:00:23	00:00:18
9ª	00:08:54	00:06:39	00:02:24	00:01:36	00:00:24	00:00:19
10ª	00:08:54	00:06:39	00:02:24	00:01:36	00:00:23	00:00:19
Média	00:08:54	00:06:39	00:02:24	00:01:36	00:00:23	00:00:19
Desvio Padrão	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
p-value obtido (1 GPU)	<0,05	<0,05	<0,05	<0,05		
p-value obtido (2 GPU)	<0,05	<0,05	<0,05	<0,05		

Fonte: O autor

camente as amostras diferem-se entre si. Evidentemente, para simulação de 100 anos, o tempo de processamento é menor para os *peers* equipados com GPUs. É possível notar na figura 13 que o tempo de execução aumenta mais rapidamente em *peers* que não possuem GPUs.

Figura 13: Tempo de execução para *peers* com e sem GPU



Fonte: *O autor*

5 CONCLUSÕES

É fundamental que simuladores tenham o menor tempo de resposta possível. Nesse sentido, as constantes melhorias do Venthor em relação ao tempo de resposta permitem que pesquisadores ou até mesmo produtores utilizem desse simulador como um sistema de auxílio a decisões. Os modelos matemáticos utilizados pelo Venthor para a execução de simulações climáticas exigem alto poder de processamento, que pode ser viabilizado pela utilização do processamento paralelo. A abordagem desse trabalho torna possível que as simulações possam ser executadas em um *desktop* equipado com uma GPU da NVIDIA, que é considerado financeiramente acessível para qualquer pesquisador, ou mesmo produtores de pequeno porte.

O aumento do volume de dados da entrada do Venthor torna as simulações mais lentas. A utilização da GPU como uma alternativa de método de processamento aumenta a eficiência nas simulações. As bases de dados utilizadas hoje como entrada para o Venthor são bases horárias, ou seja, os dados empíricos foram dados adquiridos no intervalo de uma em uma hora. Futuramente a coleta desses dados podem ser feitas de minuto a minuto, ou até menos, portanto, com bases de dados maiores, o tempo de processamento da simulação aumenta proporcionalmente. A utilização da abordagem deste trabalho, permite que o processamento de bases maiores seja feita com uma única GPU.

Em situações que o processamento de uma única GPU não seja suficiente, pode ser utilizado o *framework* P2PComp para a utilização de múltiplas GPUs. A utilização das GPUs em rede P2P demonstrou ser uma alternativa eficaz de processamento paralelo, permitindo escalabilidade simples para a utilização de múltiplas GPUs. Foi utilizado tempos de simulação grande, de 100 a 500 anos para que fosse possível ter uma dimensão real da eficiência da GPU.

Os resultados mostraram também que quanto maior a quantia de dados a ser simulado, maior é a eficiência da GPU. Foi possível constatar que em simulações pequenas (10 ou 20 anos) não existe vantagens na utilização da GPU.

5.1 Trabalhos Futuros

Existem alternativas que não são abordadas neste trabalho, como a utilização de GPUs interligados pelo barramento SLI (*Scalable Link Interface*) da NVIDIA, onde duas placas de vídeo são ligadas na mesma placa mãe em diferentes *slots* PCI Express. Segundo o próprio site da NVIDIA, a utilização da SLI em determinadas aplicações pode

aumentar o desempenho em até duas vezes. A melhoria da utilização dos *warps* também parece ser um trabalho futuro promissor. O pré-processamento que pudesse ordenar quais as distribuições a serem utilizadas na GPU, teoricamente teria um impacto ainda maior no *speedup*.

O *framework* P2PComp também pode ser melhorado com a criação da funcionalidade de compartilhamento de arquivos entre *peers*. Os experimentos deveriam ser executados sem a necessidade de utilizar um sistema de arquivos distribuído, como o NFS.

REFERÊNCIAS

AMORETTI, M. *Peer-to-peer based grid architectures*. Tese (Doutorado) — Universit? Degli Studi Di Parma, Janeiro 2006.

ANDERSON, D. P. Boinc: A system for public-resource computing and storage. In: *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2004. p. 4–10. ISBN 0-7695-2256-4.

ANDERSON, D. P. *et al.* Seti@home: an experiment in public-resource computing. *Commun. ACM*, ACM, New York, NY, USA, v. 45, n. 11, p. 56–61, 2002. ISSN 0001-0782.

ASSIS F. N.; ARRUDA, H. V. P. A. R. *Aplicações de estatística à climatologia: teoria e prática*. [S.l.]: Pelotas: Editora Universitária/UFPel, 1996. 161 p.

BEDOS, C. *et al.* Rate of pesticide volatilization from soil: an experimental approach with a wind tunnel system applied to trifluralin. *Atmospheric Environment*, v. 36, n. 39 - 40, p. 5917 – 5925, 2002. ISSN 1352-2310. Disponível em: <http://www-.sciencedirect.com/science/article/pii/S1352231002007756>.

BURNS *et al.* An open cluster environment for mpi. *In Proceedings of Supercomputing Symposium*, p. 379 – 386, 1994.

CAMARGO, L.; M.A., S. *Avaliação da Estrutura do Simulador Venthor para GPGPU*. Ponta Grossa, 2013.

CONCEICAO, M. A. F. *Critérios para a instalação de Quebra-Ventos*. [S.l.], 1996. 1 - 2 p. Disponível em: <http://ainfo.cnptia.embrapa.br/digital/bitstream/item-/25946/1/Comt18.pdf>.

CONTRERAS, L. M. *et al.* Mathematical modeling tendencies in plant pathology. *African Journal of Biotechnology*, v. 8, n. 25, p. 7399 – 7408, 2009.

FALLS, L. W. The beta distribution: a statistical model for world cloud cover. *NASA Tecnical Memorandum, TMX-64714, NASA*, p. 1 – 6, 1973.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, Sage Publications, Inc., Thousand Oaks, CA, USA, v. 15, n. 3, p. 200–222, August 2001. ISSN 1094-3420. Disponível em: <http://portal.acm.org/citation.cfm?id=1080667>.

GREENWOOD J. A.; DURAND, D. Aids for fitting the gamma distribution by maximum likelyhood. *Technometrics*, v. 2, n. 1, 1960.

HAUSWIRTH, M.; SCHMIDT, R. An overlay network for resource discovery in grids. In: *Proceedings of Sixteenth Workshop on Database and Expert Systems Applications, 2005*. [s.n.], 2005. p. 343–348. Disponível em: <http://ieeexplore.ieee.org/xpls-/abs_all.jsp?arnumber=1508296>.

HEMERT, J. L. V.; DICKERSON, J. A. Monte carlo randomization tests for large-scale abundance datasets on the gpu. *Computer Methods and Programs in Biomedicine*, v. 101, n. 1, p. 80 – 86, 2011. ISSN 0169-2607. Disponível em: <http://www-.sciencedirect.com/science/article/pii/S0169260710000957>.

HWU, W.-M. W.; KIRK, D. B. *Programando para Processadores Paralelos. Uma Abordagem Prática à Programação de GPU*. [S.l.]: Elsevier, 2011.

JACOBSONA, M. Z.; ARCHERB, C. L. Saturation wind power potential and its implications for wind energy. *Proceedings of the National Academy of Sciences of the United States*, v. 109, n. 39, p. 15679 – 15684, 2012.

JIN-YOUNG; KIM, D.-Y.; OH, J.-H. Projected changes in wind speed over the republic of korea under a1b climate change scenario. *International Journal of Climatology*, John Wiley & Sons, Ltd, p. n/a–n/a, 2013. ISSN 1097-0088. Disponível em: <http:-//dx.doi.org/10.1002/joc.3739>.

JOBSTRAIBIZER, F. *Grifo 4, o Supercomputador da Petrobrás*. 2012.

JUSTUS, C. G. *et al.* Methods for estimating wind speed frequency distributions. *Journal of Aplied Meteorology*, v. 17, n. 3, p. 350 – 353, 1978.

KALYANAPU, A. J. *et al.* Assessment of gpu computational enhancement to a 2d flood model. *Environmental Modelling & Software*, v. 26, n. 8, p. 1009 – 1016, 2011. ISSN 1364-8152. Disponível em: <http://www.sciencedirect.com/science/article-/pii/S1364815211000582>.

LEE, V. W. *et al.* Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 38, n. 3, p. 451–460, jun. 2010. ISSN 0163-5964. Disponível em: <http:/-/doi.acm.org/10.1145/1816038.1816021>.

MILOJICIC, D. *et al. Peer-to-Peer Computing*. [S.l.], March 2002.

MORRIS, R. *et al.* Chord: A scalable peer-to-peer lookup service for internet applications. In: *ACM SIGCOMM 2001*. San Diego, CA: [s.n.], 2001.

PEARSON, K. Tables of the incomplete beta function. *University College of London, Biometriks Office*, n. S.1, 1934.

RADEKE, C. A.; GLASSER, B. J.; KHINAST, J. G. Large-scale powder mixer simulations using massively parallel gpu architectures. *Chemical Engineering Science*, v. 65, p. 6435 – 6442, 2010.

RANJAN, R.; HARWOOD, A.; BUYYA, R. *A Study on Peer-to-Peer Based Discovery of Grid Resource Information*. [S.l.], Dezembro 2006.

RISSON, J.; MOORS, T. *Survey of research towards robust peer-to-peer networks:search methods*. Austrália, Setembro 2004.

SANDERS, J.; KANDROT, E. *Cuda by Example. An Introduction to General-Purpose GPU Programming*. [S.l.]: Addison Wesley, 2004.

SCHEFLER, W. C. *Statistics: Concepts and Applications.* [S.l.]: Benjamin-Cummings Publishing Co. Redwood City, CA, USA, 1988.

SEDIYAMA, G. C. *et al.* Simulação de parâmetros climáticos para a época de crescimento das plantas. *Revista Ceres*, v. 25, p. 455 – 466, 1978.

SENGER, L. J.; SOUZA, M. A. de; FOLTRAN, D. Towards a peer-to-peer framework for parallel and distributed computing. *Computer Architecture and High Performance Computing (SBAC-PAD)*, 2010.

SERCKUMECKA, A. *Avaliação do Uso de Computação Paralela Utilizando uma Rede P2P as Simulação de Dados Climáticos: Velocidade e Direção do Vento.* 56 p. Tese (Doutorado) — Universidade Estadual de Ponta Grossa, Ponta Grossa, 2012.

VALENTIN, G. L. *et al.* An overview of energy eficiency techniques in cluster computing systems. *Cluster Computing*, 2011.

VIRGENS FILHO, J. S. das; LEITE, M. de L. Ajuste de modelos de distribuição de probabilidade a séries horárias de velocidade do vento para o município de ponta grossa, estado do paraná. *Acta Scientiarum Technology*, v. 33, n. 4, p. 447 – 455, 2009.

ZWART, S. P.; BELLEMAN, R.; GELDOF, P. High performance direct gravitational n-body simulations on graphics processing unit i: An implementation in cg. *New Astronomy*, v. 12, p. 641 – 650, 2007.

APÊNDICE A – CHAMADA DO EXECUTÁVEL EM C

O código JAVA abaixo faz a chamada do executável C enviando como parâmetro os dados necessários para a simulação.

Código Fonte 14: Chamada do executável C

```
1
2  // flag para usar o executavel em C
3  if (C) {
4          try {
5
6                  \\ Monta uma ArrayList com os dados a serem enviados ao
                       executável
7
8                  List<String> args = new ArrayList<String>();
9                  args.add("C:\\Users\\ciro\\workspacec++\\ciro\\Debug\\
                       Venthor.exe");
10                 args.add(String.valueOf(ano));
11                 args.add(String.valueOf(nAnos));
12                 args.add(String.valueOf(dia));
13                 args.add(String.valueOf(param.distribuicao));
14                 args.add(String.valueOf(param.p1));
15                 args.add(String.valueOf(param.p2));
16                 args.add(String.valueOf(param.H));
17                 args.add(String.valueOf(param.menor));
18                 args.add(String.valueOf(hora));
19                 args.add(String.valueOf(mes));
20                 args.add(String.valueOf(param.PiDir.length));
21                 for (int w = 0; w <param.PiDir.length; w++) {
22                         args.add(String.valueOf(param.PiDir[w]));
23                 }
24                 // Cria o processo do executável
25
26                 Process process = new ProcessBuilder(args).start();
27                 InputStream is = process.getInputStream();
28                 InputStreamReader isr = newInputStreamReader(is);
29                 BufferedReader br = newBufferedReader(isr);
30
31                 // Lê os dados retornados pelo executável
32
33                 while ((line = br.readLine()) != null) {
34                         simulacao = line;
35                 }
36         } catch (Exception e) {
37
```

```
38              System.out.println(e.getMessage());
39      }
```

APÊNDICE B – DISTRIBUIÇÕES ADAPTADAS PARA A GPU

As próximas seções mostram os códigos em C das distribuições estatísticas e suas inversas adaptadas para GPU. Os tipos de dados retornados são de precisão dupla, embora algumas GPUs ainda não tenham arquitetura desenvolvida para a precisão dupla de dados. Atualmete, as placas de vídeos Tesla são recomendadas para simulações por suporterem esse tipo de caracterítica.

Todos os códigos escritos para CUDA foram compilados utilizando o compilador *nvcc*, desenvolvido pela própria NVIDIA e que pode ser adquirido sem cusos pelo próprio site da empresa.

B.1 Weibull

Os códigos em C da distribuição estatística Weibull e sua inversa são mostrados a seguir.

Código Fonte 15: Distribuição Weibull

```
1 __device__ double DistWeibull(doubleLs, doublealfa, double beta) {
2         return (1 - exp(-(powf((Ls / beta), alfa))));
3 }
```

B.2 Weibull Inversa

Código Fonte 16: Distribuição Weibull Inversa

```
1 __device__ double InvDistWeibull(double alfa, double beta) {
2         double aleatorio = Proximo();
3         return (beta * powf(-(log(1 - aleatorio)), (1 / alfa)));
4 }
```

Código Fonte 17: Distribuição Weibull Inversa 2

```
1 __device__ double InvDistWeibull2(double prob, double alfa, double beta) {
2         return (beta * powf(-(log(1 - prob)), (1 / alfa)));
3 }
```

B.3 Distribuição Gama

Os códigos em C da distribuição estatística Gama e sua inversa são mostrados a seguir.

Código Fonte 18: Distribuição Gama

```
1  __device__ doubleDistGamma(double Valor, double Palfa, double Pbeta) {
2          double z, Ft, t;
3          double prod = 1, soma = 0;
4          //int i=1, j=1;
5          t = Valor / Pbeta;
6  z = sqrt(2.0 * 3.14159265358979 / Palfa)          * exp(  Palfa * (log(Palfa)
          - (1 - (1 / (12 * Palfa * Palfa))+ (1 / (360 * powf(Palfa, 4.0)))
           - (1 / (1260 * powf(Palfa, 6))))));
7
8          int i = 1;
9          int j = 1;
10         for (i = 1; i<= 100; i++) {
11                 for (j = 1; j <= i; j++) {
12                         prod *= (Palfa + j);
13                 }
14
15                 soma += powf(t, i) / prod;
16                 prod = 1;
17         }
18         Ft = ((powf(t, Palfa) / (Palfa * z * exp(t))) * (1 + soma));
19
20         return Ft;
21 }
```

B.4 Distribuição Gama Inversa

Código Fonte 19: Distribuição Gama Inversa

```
1  __device__ double InvDistGamma2(double prob, double PAlfa, double PBeta) {
2
3          // declara os valores temporários
4
5          double a = 0, b = 0, x3 = 0, y = 0, gy = 0, dgama = 0;
6          x3 = 1;
7          if (PAlfa< 1.5) {
8                  b = 0.24797 + (1.3473574 * PAlfa) - (1.00004204 * powf(
                       PAlfa, 2))
9                                  + (0.53203176 * powf(PAlfa, 3)) -
                                     (0.13671536 * pow(PAlfa, 4))
10                                 + (0.01320864 * powf(PAlfa, 5));
11         } else if (PAlfa< 19) {
```

```
12                  b = 0.6435 + (0.45839602 * PAlfa) - (0.02952801 * powf(
                        PAlfa, 2))
13  + (0.00172718 * powf(PAlfa, 3)) - (0.0000581 * pow(PAlfa,4))
14                                  + (0.00000082 * powf(PAlfa, 5));
15          } else {
16                  b = 1.33408 + (0.22499991 * PAlfa) - (0.00230695 * powf(
                        PAlfa, 2))
17                                  + (0.00001623 * powf(PAlfa, 3)) -
                                        (0.00000006 * pow(PAlfa, 4));
18          }
19          y = 1 + (1 / b);
20
21          if (y - 1 > 0) {
22                  while (y - 1 > 0) {
23                          y = y - 1;
24                          x3 = x3 * y;
25                  }
26          }
27
28          if (y - 1 < 0) {
29                  gy =1+ y*(-0.5771017+ y* (0.985854+ y* (-0.8764218+ y*
                        (0.8328212+ y* (-0.5684729+ y* (0.2548205+ y*
                        (-0.0514993))))))));
30                  x3 = x3 * gy / y;
31          }
32
33          a = powf((x3 / (PAlfa * PBeta)), b);
34          b = 1 / b;
35          a = 1 / a;
36          dgama = powf((-a * log(1 - prob)), b);
37
38          double ret = ArredondarParaCima(dgama, 1);
39
40          return ret;
41  }
```

B.5 Distribuição Rayleigh

Os códigos em C da distribuição estatística Rayleigh e sua inversa são mostrados a seguir.

Código Fonte 20: Distribuição Rayleigh

```
1  __device__ double DistRaylegh(double Ls, double media) {
2          double aux = (exp(-(PI * (powf(Ls, 2))) / (4 * (pow(media, 2)))));
3          return (1 - aux);
4  }
```

B.6 Distribuição Rayleigh Inversa

Código Fonte 21: Distribuição Rayleigh Inversa

```
1
2  __device__ double InvDistRaylegh2(double prob, double media) {
3          double scala = sqrt(2 / PI) * media;
4          return sqrt(2 * powf(scala, 2) * log(1 / (1 - prob)));
5  }
```

B.7 Distribuição Beta

Os códigos em C da distribuição estatística Beta e sua inversa são mostrados a seguir.

Código Fonte 22: Distribuição Beta

```
1
2  __device__ double DistBeta(double x, double p, double q) {
3          return incompleteBeta(x, p, q);
4  }
```

B.8 Distribuição Beta Inversa

Código Fonte 23: Distribuição Beta Inversa

```
1
2  __device__ double InvDistBeta(double Pp, double Pq) {
3          double aleatorio = Proximo();
4          //aleatorio /= 100;
5          return inverse(aleatorio, Pp, Pq);
6  }
```

Código Fonte 24: Distribuição Beta Inversa 2

```
1 __device__ double InvDistBeta2(double prob, double Pp, double Pq) {
2         return inverse(prob, Pp, Pq);
3 }
```

ANEXO A – EQUAÇÕES DO MODELO

A.1 Distribuições de Probabilidade

A.1.1 Distribuição de Weibull

A distribuição de Weibull 2 parâmetros para a velocidade do vento é expressa pela função de densidade de probabilidade:

$$f(v) = \left(\frac{k}{c}\right)\left(\frac{v}{c}\right)^{k-1} \exp\left[-\left(\frac{v}{c}\right)^{k}\right] \tag{1}$$

em que a função cumulativa de probabilidade dada por:

$$F(v) = 1 - \exp\left[-\left(\frac{v}{c}\right)^{k}\right] \tag{2}$$

em que: c é o fator de escala em unidades de velocidade do vento, k, o fator de forma adimensional e v, a variável aleatória velocidade do vento. O fator de forma k está inversamente relacionado à variância σ^2 das velocidades eólicas em torno da média. Os parâmetros c e k, conforme discutidos em Justus *et al.* (1978), podem ser determinados a partir da transformação da equação 2 na forma linear.

$$ln(-ln(1 - F(v))) = -kln(c) + kln(v) \tag{3}$$

que pode ser representada pela reta: $Y = a + bX$

em que:

Y = ln [- ln (1 - F(v)];

X = ln (v);

a = - k ln (c) e

b = k

Assim, a determinação dos parâmetros c e k fica condicionada aos cálculos dos coeficientes a e b da reta. Esses podem ser obtidos pelo método dos mínimos quadrados aplicado ao conjunto de dados $X = ln(v)$ e $Y = ln[-ln(1 - F(v))]$ obtidos dos valores de v e $F(v)$ que, por sua vez, são determinados a partir das séries observadas da velocidade do vento distribuídas em n intervalos de classe com suas respectivas frequências.

A.1.2 Distribuição de Rayleigh

A expressão matemática da função de densidade de probabilidade de Rayleigh é dada por:

$$f(v) = -\frac{v}{a^2}\exp\left(-\frac{v^2}{2a^2}\right) \tag{4}$$

em que:

$$a = \sqrt{\frac{2}{\pi}}E(v) \tag{5}$$

e E (v) é a esperança matemática da variável aleatória v (velocidade do vento). A função cumulativa de probabilidade Rayleigh é:

$$F(v) = 1 - \exp\left(-\frac{\pi v^2}{4E(v)^2}\right) \tag{6}$$

A.1.3 Distribuição de Beta

Essa função de densidade de probabilidade pode ser expressa da seguinte forma (FALLS, 1973):

$$F(v) = \frac{1}{(b-a)}\frac{\Gamma(p+q)}{\Gamma(p)\Gamma(q)}\left(\frac{v-a}{b-a}\right)^{p-1}\left(1-\frac{v-a}{b-a}\right)^{q-1} \tag{7}$$

em que: a e b correspondem ao menor e maior valor da série de dados, respectivamente, Γ é o símbolo da função Gama das respectivas variáveis, p e q são parâmetros da distribuição Beta e v é um valor qualquer da variável em análise. A estimativa dos parâmetros p e q pode ser realizada a partir do método dos momentos (PEARSON, 1934).

$$p = \frac{\mu_1(\mu_1 - \mu_2)}{[\mu_2 - (\mu_1)^2]} \tag{8}$$

$$q = \frac{(1-\mu_1)(\mu_1 - \mu_2)}{[\mu_2 - (\mu_1)^2]} \tag{9}$$

em que: μ_1 corresponde ao momento de ordem 1 para a variável v e μ_2 ao momento de ordem 2 para a variável v, dentro de uma série de n dados. Estes termos podem ser

estimados a partir da seguinte análise:

$$\mu_1 = \frac{\sum_{i=1}^{N} v_i}{N} \tag{10}$$

$$\mu_2 = \frac{\sum_{i=1}^{N} v_i^2}{N} \tag{11}$$

Para a estimativa dos valores de ocorrência de probabilidade, por meio da distribuição Beta, a equação 7 deve ser adimensionalizada para um intervalo compreendido entre [0 e 1]. A variável adimensionalizada v toma então a seguinte forma:

$$v' = \frac{v - a}{b - a} \tag{12}$$

sugerindo que a função de densidade de probabilidade Beta assuma a seguinte forma:

$$f(v') = \frac{\Gamma(p+q)}{\Gamma(p)\Gamma(q)}(v')^{p-1}(1-v')^{q-1} \tag{13}$$

em que: $0 \leq v' \leq 1$, para $p > 1$ e $q > 1$. A integração numérica da equação 13 confere os valores da probabilidade de ocorrência para um valor de v qualquer dentro do intervalo considerado.

A.1.4 Distribuição de Gama

A função de densidade de probabilidade Gama é expressa por:

$$f(v) = \frac{v^{\alpha-1}e^{\frac{-v}{\beta}}}{\beta^{\alpha}\Gamma(\alpha)} \tag{14}$$

em que: α = parâmetro de forma, estimado pelo método de GREENWOOD J. A.; DURAND, dado por:

$$\alpha = \frac{8,898919 + 9,05995y + 0,9775373y^2}{y(17,79728 + 11,968477y + y^2)} \tag{15}$$

para $0 \leq y \leq 0,5772$. Ou

$$\alpha = \frac{0,5000876 + 0,1648852y - 0,0544274y^2}{y} \tag{16}$$

para $0,5772 < y \leq 17,0$ em que:

$$y = \ln \left(\frac{\frac{1}{N} \sum_{i=1}^{N} v_i}{\left(\prod_{i=1}^{N} v_i \right)^{\frac{1}{N}}} \right) \tag{17}$$

β = parâmetro de escala estimado a partir de:

$$\beta = \frac{\sum_{i=1}^{N} v_i}{N\alpha} \tag{18}$$

$\Gamma(\alpha)$ = Função Gama ordinária de α. A função cumulativa de probabilidade é:

$$F(v) = \frac{1}{\Gamma(\gamma)\beta^{\gamma}} \int_0^v v^{\gamma-1} e^{-\frac{v}{\beta}} dv \tag{19}$$

A.2 Teste de Kolmogorov-Smirnov

O teste de Kolmogorov-Smirnov é aplicado para verificar se os valores de uma certa amostra de dados podem ser considerados como provenientes de uma população com distribuição teórica pré-estabelecida, sob uma hipótese: a hipótese de nulidade (H0). O teste confronta duas distribuições de frequência acumuladas, uma F(X), teórica, e outra, F(X), derivada dos dados amostrais, tal que: - Seja F(X) uma função de distribuição de probabilidade teórica com seus parâmetros especificados; - Seja F(X) uma distribuição de probabilidade empírica, ou seja, para uma determinada classe de frequência F(X) = fa/N, onde fa é a frequência acumulada da classe;

Determina-se:

$$Dmax = MAX|F'(X) - F(X)| \tag{20}$$

Se, ao nível de significância estabelecido, o valor observado de Dmáx (calculado) for maior ou igual ao valor crítico de Dmáx (tabelado), a hipótese de nulidade, ou seja, a hipótese de que os dados amostrais provêm de uma população com distribuição teórica, $F(X)$ é rejeitada (ASSIS F. N.; ARRUDA, 1996).

A.3 Cálculo da Direção do Vento

Já para a análise da direção predominante do vento, os dados foram descritos probabilisticamente através da frequência relativa simples (fr_d) para cada mês do ano do período em estudo, cuja fórmula esta descrita abaixo:

$$fr_d = \frac{f_d}{N} \tag{21}$$

onde f_d é a frequência de cada direção e N é o numero de observações dentro do mês analisado. Para determinar as direções dos ventos, foram utilizados os pontos cardeais Norte, Sul, Leste e Oeste e também os pontos colaterais Nordeste, Sudoeste, Sudeste e Noroeste, através da rosa-dos-ventos.