Universidade do Estado da Bahia

Departamento de Ciências Exatas e da Terra

Colegiado de Sistemas de Informação

Análise da Efetividade de uma Arquitetura Paralela Híbrida utilizando Multicore e Multi-GPU

Anderson da Conceição Soares

Salvador, Bahia, Brasil

Julho de 2014

Anderson da Conceição Soares

Análise da Efetividade de uma Arquitetura Paralela Híbrida utilizando Multicore e Multi-GPU

Monografia submetida ao Colegiado de Sistemas de Informação da Universidade do Estado da Bahia como parte dos requisitos necessários para obtenção do grau de Bacharel em Sistema de Informação.

Área de Concentração: Sistema de Informação

Linhas de Pesquisa: Computação paralela e distribuída

Murilo do Carmo Boratto

(Orientador)

Salvador, Bahia, Brasil

Julho de 2014

Anderson da Conceição Soares

Análise da Efetividade de uma Arquitetura Paralela Híbrida utilizando Multicore e Multi-GPU

Monografia submetida ao Colegiado de Sistemas de Informação da Universidade do Estado da Bahia como parte dos requisitos necessários para obtenção do grau de Bacharel em Sistema de Informação.

Aprovada em: _____ / _____ / _____

Murilo do Carmo Boratto
Doutorando em Ciência da Computação
Universidade do Estado da Bahia

Robespierre Dantas
Mestrando em Ciência da Computação
Universidade Federal da Bahia

Ivan de Mattos Lessa
Mestrando em Sistemas e Computação
Universidade Federal da Bahia

Salvador, Bahia, Brasil

Julho de 2014

Resumo

Com a possibilidade de representar sistemas reais através de equações matemáticas o estudo dos modelos computacionais se tornou um campo atrativo nos últimos anos. Porém esses modelos demandam um grande poder de processamento e acabam por levar um grande tempo para serem completamente executados. Uma das formas para a otimização desses modelos é a utilização da computação de alto desempenho, que conta com uma série de novos dispositivos com características computacionais de aceleração de processamento. Aliado a isso surgiram novas *APIs* de programação paralela que permitem unir mais de um dispositivo com poder computacional em um ambiente híbrido, onde há uma tendência de obter um ganho de desempenho. Dessa maneira, o presente trabalho tem por objetivo estudar, experimentar e validar um modelo de programação paralela e distribuída, baseado em uma arquitetura heterogênea sobre multicore e GPUs.

Palavras chave: Programação paralela, multicore, multi-GPU, programação paralela híbrida, CUDA.

Abstract

The possibility to represent real systems through mathematical equations the study computational models has become an attractive field in recent years. That models demand a lot of processing power and eventually take a long time to be fully implemented. One ways to optimize these models is the use of high performance computing, which has a number of new computing devices with features such as the GPU. Along with that came new parallel programming *APIs* that allow you join more than one device with computational power in a hybrid environment, where there is a tendency to get performance gain. Thus the present work aims to study, experiment, contribute and validate a model of parallel and distributed programming, based a heterogeneous architecture on multicore and GPUs.

Keywords: Parallel programming, multicore, multi-GPU, hybrid parallel programming, CUDA .

Agradecimentos

- Em primeiro lugar a Deus por ter me amparado em cada momento.
- Agradeço a minha família, em especial aos meus pais (Maria Conceição e Rosalvo Soares), minha madrinha (Cleide Lima) e meu padrinho (João Francisco), por terem me dado acesso a uma boa educação, pelo apoio, pelo carinho e pela confiança que sempre tiveram em mim.
- Ao meu orientador, Murilo Boratto, meu agradecimento especial, por todo o apoio, ajuda e pela paciência durante todas as etapas deste trabalho. Muito obrigado pelos ensinamentos!
- Agradeço também a todos os professores e funcionários do colegiado de Sistemas de Informação da UNEB, pela contribuição na minha formação.
- Aos meu colegas, pelo apoio nos estudos e pela amizade formada durante o curso, em especial as turmas de 2007, 2008 e 2009 do curso de Sistemas de Informação.
- A todos que contribuíram de forma direta ou indireta para a realização deste trabalho, o meu muito obrigado!

"A curiosidade é mais importante do que o conhecimento."

Albert Einstein

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Abreviaturas e Siglas

API	*Application Programming Interface*
CAD	Computação de Alto Desempenho
CPU	*Central Processing Units*
CUBLAS	*Cusa Basic Linear Algebra Subroutines*
CUDA	*Compute Unified Device Architecture*
FLOPS	*Floating Point Operations Per Second*
GPGPU	*General Purpose Graphics Processing Unit*
GPU	*Graphics Processing Units*
LAPACK	*Linear Algebra Package*
OPENMP	*Open Multi-Processing*
SM	*Stream Multiprocessors*
SP	*Stream Processors*

Capítulo 1

Introdução

A necessidade de representar, através de modelos computacionais, sistemas e fenômenos observados sempre foi um desafio para o homem. Dessa forma, procura-se descrever matematicamente sistemas reais, para facilitar o seu entendimento e assim resolver problemas que são comuns nesses tipos de sistemas. Segundo Pereira, Bonganha e Guiguer (2007), um modelo computacional consiste na representação matemática do que acontece na natureza a partir de um modelo conceitual, idealizado com base no levantamento e na interpretação de dados a partir da observação de um sistema real.

Diversas áreas de estudo, que possuem uma grande relevância científica e econômica, como por exemplo, as ciências e as engenharias, utilizam modelos computacionais para simular sistemas. Os modelos que são desenvolvidos utilizando equações matemáticas geralmente necessitam de uma quantidade significativa de parâmetros, para que assim seja possível obter uma maior fidelidade dos resultados. Porém estes modelos demandam cada vez mais um maior poder de processamento computacional e consequentemente um maior tempo de execução (BORATTO et al., 2012a).

Uma das formas de reduzir o tempo de processamento é paralelizar a execução das instruções, onde uma tarefa que possui um alto custo computacional pode ser realizada de maneira paralela. De acordo com (SANTANA et al., 1997), o processamento paralelo implica na divisão de uma tarefa de maneira que ela possa ser executada por uma ou mais unidades de processamento, que por sua vez deverão cooperar entre si na resolução do problema.

Com o surgimento de novas tecnologias é possível utilizar o poder dos dispositivos gráficos, que são massivamente paralelos, para realizar uma grande quantidade de cálculos aritméticos. De acordo com Nvidia (2013), as GPUs evoluíram a um ponto em que muitos aplicativos do mundo real são facilmente implementados e operarem com velocidade significativamente maior do que os sistemas com multi-processador. Para tal, foram desenvolvidas arquiteturas que possibilitam a programação da GPU para propósitos gerais, e não mais exclusivamente para o processamento de imagens. Segundo Yano (2010), a principal arquitetura desenvolvida é o CUDA, onde é possível programar a GPU para executar aplicativos que não estejam exclusivamente ligados ao processamento de imagens, como por exemplo, aplicativos comerciais onde são realizadas ordenação de valores, cálculo de finanças, dentre outros (DUTRA; VARINI; CANAL, 2012).

O crescente uso dos dispositivos gráficos para a computação de propósito geral acabou por expandir dentro da Computação Paralela, o paralelismo híbrido, que utiliza tanto a CPU como a GPU para processar algum tipo de informação. Segundo Pinto (2011) a programação paralela híbrida tem por objetivo unir dispositivos com diferentes arquiteturas para que os mesmos trabalhem em conjunto e assim possam atingir um maior desempenho. Desta forma, é possível executar cada conjunto de instruções na arquitetura que melhor se adapte.

Nesse contexto, esta pesquisa busca experimentar, validar e propor otimizações a um modelo de programação paralela e distribuída, baseado em uma arquitetura heterogênea sobre multicores e GPUs. Dessa forma será realizada uma tentativa de otimização utilizando conceitos de paralelismo híbrido sobre o modelo proposto por Boratto et al. (2013), que realiza o cálculo de polinômios matriciais. Para isso é necessário analisar qual a técnica de paralelização em ambientes híbrido que será utilizada e definir os critérios de avaliação dos resultados além de desenvolver o protótipo híbrido do modelo.

Para a realização dessa pesquisa a etapa metodológica foi dividida em cinco fases. Na **primeira fase** será feito um levantamento sobre os trabalhos que já foram desenvolvidos e publicados na área de computação paralela híbrida. Na **segunda fase**, de acordo com o moledo computacional escolhido para a validação da tese, serão realizados testes utilizando somente uma GPU para processamento, visando assim analisar o comportamento do modelo. Na **terceira fase** será desenvolvido protótipos paralelos híbridos do modelo computacional.

Na **quarta fase** irá se discorrer sobre os resultados encontrados. De acordo com a métrica de desempenho escolhida, será feita uma análise do comportamento do algoritmo em ambos os modelos. E por fim, na **quinta fase** serão propostos possíveis trabalhos futuros relacionados a área de pesquisa além da conclusão do trabalho.

Este trabalho está estruturado em 6 capítulos. No **capítulo 2** descrevem-se as ferramentas que serão utilizadas no decorrer do trabalho assim como os modelos da computação paralela de alto desempenho. O **capítulo 3** descreve alguns trabalhos que possuem relação com o problema apresentado e apresentada as oportunidades de pesquisa encontradas. No **capítulo 4** é apresentado o projeto experimental para a validação trabalho, os polinômios matriciais, o qual é otimizado através de um modelo de computação paralela. No **capítulo 5** haverá uma discussão sobre os resultados obtidos. E por fim, serão apresentadas as conclusões do trabalho assim como propostas para trabalhos futuros.

Capítulo 2

Descrição teórica e ferramentas básicas para computação paralela

Neste capítulo será descrito de maneira teórica os modelos de computação de alto desempenho e também softwares e hardwares que serão utilizados no decorrer desse trabalho.

2.1 Computação paralela

Por um certo período, o poder de processamento da CPU era baseados no aumento constante de sua frequência. Porém ,no ano de 2004 a principal fabricante de processadores desistiu de lançar sua nova linha de processadores por problemas relacionados ao elevado consumo de energia e a alta dissipação térmica que os mesmos apresentavam (YANO, 2010).

Como até então os processadores trabalhavam executando instruções em sequência, bastava a elevação da frequência de processamento para que as instruções fossem executadas mais rapidamente. Com a limitação da elevação da quantidade máxima da frequência passou-se a investir em CPUs que operassem de maneira paralela, para que assim pudessem ser executas mais de uma instrução ao mesmo tempo. Essa tecnologia foi denominada multicore (YANO, 2010).

De acordo com (PEREIRA, 2012), os computadores que tem vários núcleos tornaram-se o padrão industrial usando atualmente. Apesar disso, a maioria dos algoritmos é concebida de forma a ser executada em apenas um núcleo, causando assim um pequeno aumento de desempenho, sendo que em algumas situações o desempenho permanece o mesmo. Assim sendo, uma forma de obter uma melhora significativa seria a criação ou adaptação dos algoritmos para que os mesmos possam ser divididos e executados por várias unidades de processamento ao mesmo tempo. Deste modo, a principal motivação para o desenvolvimento de aplicações com alto nível de paralelismo é para que estas possam usufruir de um contínuo aumento de velocidade nas gerações de hardware futuras.

Segundo (YANO, 2010), com o uso da programação paralela a aplicação é desenvolvida para aproveitar o paralelismo desde o algoritmo. A vantagem dessa abordagem é que ela fornece maior ganho de desempenho, mas tem como desvantagem a exigência um maior esforço no desenvolvimento na concepção do algoritmo.

A programação paralela apresenta os mesmos desafios tanto de correção e segurança quanto aos programas sequenciais, ainda assim podendo existir problemas como a concorrência, o bloqueio e as disputas. Por isso o desenvolvedor deve ficar atento a essas situações que podem causar uma divergência de dados entre o modelo sequencial e o paralelo (PEREIRA, 2012) . Para que programas paralelos possam ser executados de maneira correta é preciso que não haja dependência entre os dados a serem processados. Uma instrução y depende de x quando ela necessita do resultado computado de x para a sua execução (YANO, 2010).

2.1.1 Classificação dos sistemas paralelos

Os sistemas paralelos podem ser classificados de várias maneiras, dentre elas a mais conhecida é a taxonomia de Flynn (Figura 2.1), o qual classificou os computadores de acordo com a organização de instruções e de dados no seu determinado tipo de processamento. Para maiores detalhes desse taxonomia pode ser encontrado em (FERRÃO; PLOTZE, 2012). O modelo SIMD *(Single Instruction, Multiple Data)* , é o que mais se assemelha a tecnologia de processamento em GPUs usada pela nvidia que é denominada SIMT *(Single Instruction Multiple Threads)*(NVIDIA, 2009) .

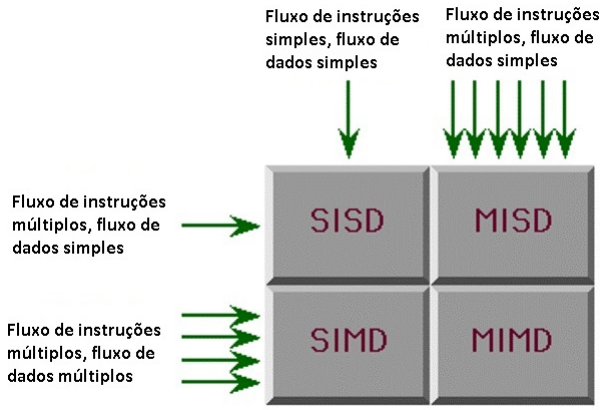

Figura 2.1: Taxonomia de Flynn
Fonte:(MORAIS, 2012)

Os tipos de processamento de acordo com a taxonomia de Flynn são:

- SISD *(Single Instruction Single Data)*: Nesse processamento um único processador executa uma instrução completa de cada vez, para operar em dados armazenados em uma única memória referente a essa operação. Isto corresponde a Arquitetura de Von Neumann.

- MISD *(Multiple Instruction Single Data)*: Já esse processamento corresponde a um tipo de arquitetura de computação paralela, em que unidades funcionais executam operações diferentes sobre os mesmos dados, como por exemplo um array.

- SIMD *(Single Instruction Multiple Data)*: Tais máquinas são caracterizadas por possuírem apenas uma unidade de controle que executa uma instrução de cada vez, mas cada instrução opera sobre vários dados.

- MIMD *(Multiple Instruction Multiple Data)*: Nesse tipo de processamento que é construído a partir de vários processadores operando de forma cooperativa ou concorrente, na execução de um ou vários aplicativos, ou seja, computadores que operam com múltiplos fluxos de instruções e múltiplos fluxos de dados.

A GPU possui um núcleo baseado no modelo computacional denominado SIMD *(Single Instruction Multiple Data)*, onde existem vários processadores mas somente uma unidade

de processamento. Utilizada principalmente para a resolução de problemas computacionais da área científica e de engenharias, onde pode-se encontrar estruturas de dados regulares ao exemplo de vetores e matrizes. Essa característica classifica o processador das GPUs como sendo processadores vetoriais (MORAIS, 2012).

2.2 GPGPU (*General Purpose Graphics Processing Unit*)

Os chips gráficos trabalhavam unicamente com o processamento de imagens. Porém com o passar dos anos se tornou cada vez mais programável e computacionalmente poderoso. Tanto que, em meados do ano 2000, alguns cientistas passaram a utilizar placas gráficas para acelerar algumas aplicações cientificas. Com isso surgiu o conceito de GPGPU, onde processamentos que antes só eram realizados nas CPUs agora passam a ser realizados também nas GPUs, eliminando assim a fronteira unicamente gráfica a que as placas de vídeos estavam submetidas (KIRK; HWU, 2011).

Desde o lançamento das primeiras GPUs, vem se pesquisando formas de aproveitar a excelente performance que as placas de vídeo tem para realizar cálculos de pontos flutuantes, que foi denominado de Movimento da GPU de Propósito Geral .

Após anos de pesquisa, em 2003 um grupo de pesquisadores apresentou um modelo de (GPGPU) baseado na linguagem C e com o uso de paralelismo de dados. Este modelo era mais simples de ser codificado que os códigos de GPUs ajustados manualmente e eram certa de sete vezes mais rápidos que os códigos similares existentes. Partindo disso a Nvidia desenvolveu uma ferramenta intuitiva que visava desenvolver uma solução para que fosse possível executar a linguagem C de forma satisfatória na GPU. E assim no ano de 2006 a Nvidia mostrou ao mercado o CUDA, uma solução para o GPGPU, que acabou por ajudar a popularizar e impulsionar a utilização desse novo modelo (NVIDIA, 2013).

O uso de GPUs para o processamento paralelo tem se mostrado eficiente quando há grandes quantidades de dados que devem apresentar um tempo de resposta rápido ou satisfatório na obtenção de resultados. É cada vez mais comum se encontrar pequenos *clusters* de GPUs tendo como fator determinante para esse fato o alto custo-benefício (tem um preço acessível e

um alto poder de processamento). Na Figura 2.2 é possível notar a evolução do desempenho das GPUs em relação as CPUs (GRAMMELSBACHER; MEDRADO, 2009).

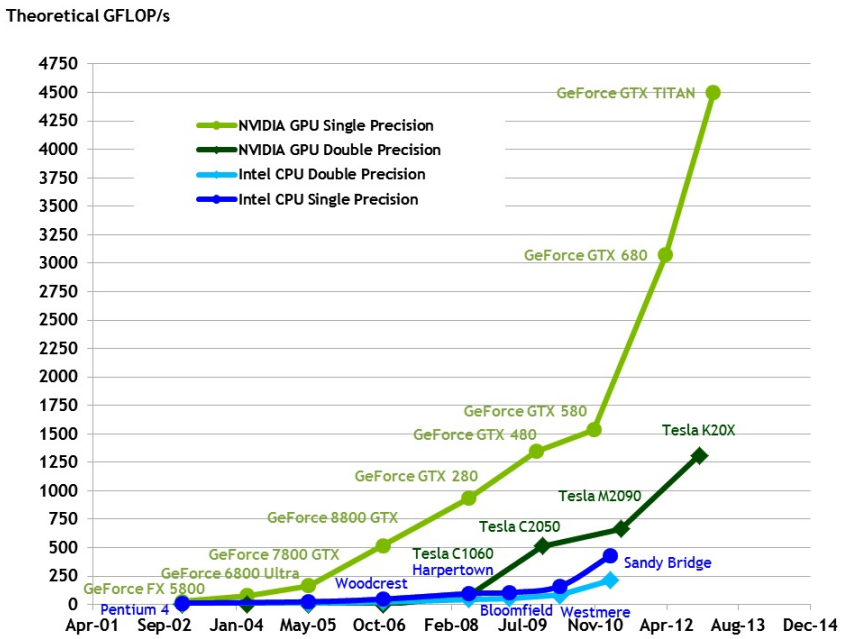

Figura 2.2: Desempenho GPU vs CPU
Fonte: (NVIDIA, 2009)

2.3 Paradigma de memória compartilhada

No modelo de memória compartilhada, as tarefas compartilham um mesmo espaço de endereçamento, sendo possível ler os dados armazenados por diferentes processadores. Quando se utiliza esse tipo de comunicação, deverão ser adotados métodos que possam prover o controle de acesso a memória (semáforos) para evitar que dois ou mais processadores atualizem a mesma região da memória simultaneamente (para garantir a integridade dos dados). Esse tipo de controle de memória acaba onerando desempenho geral das aplicações que utilizam esse paradigma (RIBEIRO, 2009). Em geral os processadores podem operar de maneira independente porém eles compartilham os mesmos recursos de memória.

2.4 Ferramentas de hardware

Este trabalho irá utilizar um sistema formado por nodos multiprocessados com um número de núcleos igual a seis e que faz o uso de uma mesma memória física.

2.4.1 Descrição dos sistemas paralelos utilizado

O ambiente computacional usados no desenvolvimento do projeto foi selecionado de acordo com as necessidades computacionais exigidas, sendo que em algumas situações somente partes das plataformas são utilizadas.

- O **Sistema 1** é formado por 1 nodo biprocessador Intel Xeon X5680 que possui um *clock* de 3.33 GHz e 96GB DDR3 de memória principal. Cada processador contém um núcleo de 12MB de memória cache. O sistema também contém 2 GPUs NVIDIA Tesla C2070, com 14 SM *(Stream multiprocessors)* e 32 SP *(stream processors)*, cada um com um total de 448 núcleos, tendo 16 unidades de *load/store*, com 32K de palavra no registrador e 64K de memória ram configurável. Cada SM dispõe de unidades simples e duplas de ponto flutuante. Sendo que para cada operação segue o padrão IEEE 754-2008 de ponto flutuante.

2.5 Ferramentas de software

As ferramentas de *softwares* que serão utilizadas no decorrer desse trabalho podem ser divididas em dois grupos, as linguagens de programação e as bibliotecas utilizadas.
Irá se utilizar a linguagem ANSI C/C++ para a programação sequencial, a biblioteca MPI para a programação em memória distribuída e as API OpenMP e CUDA para a programação de memória compartilhada.

OpenMP (Open Multi-Processing)

O OpenMP é um padrão atual para programação utilizando o modelo de memória compartilhada, que incluem os sistemas *multithreads* e computadores de alto desempenho com memória virtual compartilhada. Esta API *(Application Programming Interface)* através de chamadas de funções da biblioteca, possui uma grande variedade de características. Existem funções que determinam o número de processos e subprocessos para estabelecer o número de filhos que se pode usar, funções que mensuram o tempo, funções para paralelismo, dentre outras. As regiões paralelas são identificadas pelas diretivas, que consistem em linhas de código que possuem algum significado para o compilador. Como exemplo nas linguagens C e C++ as diretivas OpenMP são identificadas pelo *pragma omp* (CALDAS; SENA, 2008).

Como o OpenMP é baseado no paradigma de programação de memória compartilhada, o paralelismo consiste então de múltiplas *threads*. Assim, pode-se dizer que OpenMP é um modelo de programação paralelo explicito, já que oferece um total controle ao programador (RIBEIRO, 2009).

O modelo de programação do OpenMP se inicia com uma única *thread* que executa sozinha as instruções até encontrar uma região paralela (identificada pela diretiva), ao encontrar essa região ela cria um grupo de *thread*, que juntas executam o código dentro dessa região. Quando as *thread* completam a execução do código na região paralela, elas sincronizam-se e somente a *thread* inicial segue na execução do código até que uma nova região paralela seja encontrada ou que o programador decida encerrar essa *thread* (CALDAS; SENA, 2008). Esse modelo de programação é conhecido na como *fork-join* e pode ser visualizado na Figura 2.3.

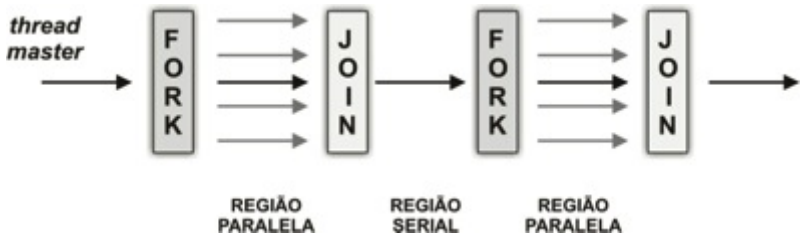

Figura 2.3: Modelo fork-join
Fonte: (CALDAS; SENA, 2008)

O uso do OpenMP se deve a algumas vantagens, como:

- Normalmente são feitas poucas alterações no código serial existente.
- Possui uma robusta estrutura para suporte a programação paralela.
- Fácil compreensão e uso das diretivas.
- Suporte a paralelismo aninhado.
- Possibilita o ajuste dinâmico do número de *threads*.

2.5.1 CUDA (Compute Unified Device Architecture)

A ferramenta CUDA é uma plataforma de computação paralela desenvolvida pela NVIDIA que permite um aumento do desempenho computacional, aproveitando o poder das unidades de processamento gráfico (GPU). A linguagem padrão de utilização do CUDA é a linguagem C, onde se separa a parte que será executa na CPU apelidado de *Host* e a parte que será executada na GPU, apelidada de *Device*. No contexto de CUDA as funcções executadas na GPU são chamadas de *Kernel*.

Com o CUDA a CPU é utilizada como processador central (responsável pelo controle de fluxo da aplicações) e a GPU serve como co-processador realizando o processamento paralelo de cálculos necessários para a execução do programa (ROCHA; FILHO, 2010). Essa interação na maioria das vezes resulta em um ganho de desempenho. Existem vários setores que necessitam da tecnologia de alto desempenho e viram no CUDA a oportunidade de acelerar a execução dos seus programas. Abaixo alguns setores que utilizam a tecnologia CUDA segundo Nvidia (2013):

- Identificação de placas ocultas em artérias: Ataques cardíacos são a maior causa de mortes no mundo todo. *A Harvard Engineering, a Harvard Medical School e o Brigham e Women's Hospital* se reuniram para usar GPUs com o objetivo de simular o fluxo sanguíneo e identificar placas arteriais ocultas sem fazer uso de técnicas de imageamento invasivas ou cirurgias exploratórias.

- Análise do fluxo de tráfego aéreo: O Sistema de Espaço Aéreo Nacional dos Eua gerencia a coordenação do fluxo de tráfego aéreo em âmbito nacional. Modelos computacionais ajudam a identificar novas maneiras de aliviar congestionamentos e manter o tráfego de aeronaves fluindo de forma eficiente. Utilizando o poder computacional das GPUs, uma equipe da NASA obteve grande ganho de performance, reduzindo o tempo de análise de dez minutos para três segundos.

- Visualização de moléculas: Uma simulação molecular denominada NAMD (Dinâmica Molecular em Nanoescala) alcança um grande aumento de performance com o uso de GPUs. Essa aceleração é resultado da arquitetura paralela das GPUs, que permite que desenvolvedores NAMD migrem partes de aplicativos com alta demanda computacional para a GPU utilizando o CUDA.

Na Figura 2.4 é possível visualizar a estrutura da API CUDA. Como ela se baseia no compilador LLVM, que é um compilador de código aberto amplamente utilizado, com um design modular que torna fácil adicionar suporte para linguagens de programação e que nesse caso foi adaptado para trabalhar com essa API. O compilador também tem a função de separar e repassar os trechos de códigos através de marcações da propria API que vão ser executados no *Host* (CPU) ou no *Device* (GPU).

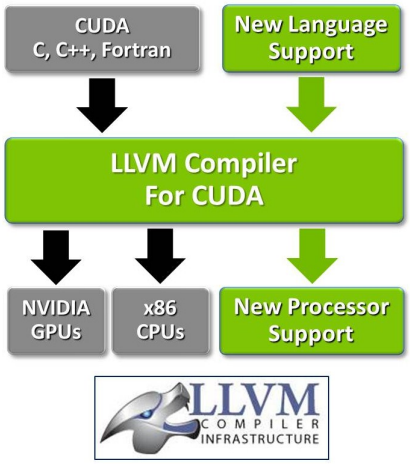

Figura 2.4: Estrutura CUDA
Fonte: (NVIDIA, 2013)

Há também a biblioteca *Runtime* que gerencia as comunicações ente a CPU e a GPU além de ter grande parte das funções especificas que podem ser executadas pelo *device*.

De acordo com a Figura 2.5 as *threads* que são a unidade básica de processamento, estão localizadas dentro de blocos (que pode ter até três dimensões) , que por sua vez estão localizado dentro de um *grid* seguindo algumas características:

- Cada *thread* de um bloco pode ter até três dimensões.
- Blocos são organizados em *grids*.
- Os blocos de um *grid* tem o mesmo número de *threads*.

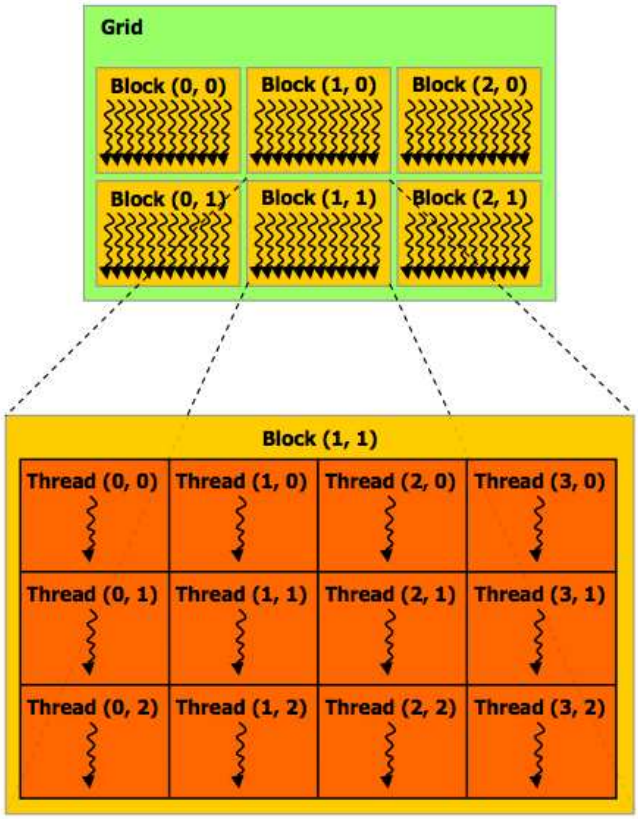

Figura 2.5: Arquitetura CUDA
Fonte: (NVIDIA, 2013)

Cada *thread* possui uma memória local que pode ser acessada unicamente por ela. Há também uma memória compartilhada por bloco, onde todas as *threads* pertencentes a um determinado bloco, podem acessar essa memória. Para *threads* de blocos diferentes existe a memória global que é acessível a todas as *threads*. A independência que existe entre os blocos permite que estes possam ser alocados a GPU em qualquer ordem. Dessa maneira um

código executado na GPU pode executar em uma quantidade qualquer de núcleos, evitando assim a ociosidade dos mesmos (NVIDIA, 2009).

Existe um limite de *threads* por blocos, pois o bloco de *threads* deve residir no mesmo núcleo do processador e deve compartilhar os recursos de memória que é limitada nos núcleos. Assim a quantidade total de *threads* designada a um bloco deve ser compatível com os recursos de memória disponíveis no núcleo de processamento da GPU, para que não ocorra a subutilização dos mesmos. A plataforma CUDA durante toda sua definição sempre visa a utilização por completo dos recursos disponíveis de *hardware* para ganho de desempenho. Em GPUs atuais, um bloco de *thread* pode conter até 512 *threads*, mas a definição do número total de blocos e *thread* por bloco é geralmente especificada de acordo com o tamanho dos dados que estão sendo processados ou o número de processadores no sistema (ROCHA; FILHO, 2010).

Todas as *threads* de um bloco executam uma mesma função do *kernel*. Sendo assim elas necessitam ter coordenadas exclusivas para que se possa distinguir uma das outras e também para identificar a parte de dados que cada uma deve processar. Essas *threads* se organizam em níveis de hierarquia utilizando dois níveis de coordenadas que são o *blockIdx* (índice do bloco) e o *threadIdx* (índice da thread), vide Figura 2.5. Essas coordenadas são atribuídas as *threads* pelo sistema de *runtime* do CUDA e podem ser acessadas nas funções do *kernel* já que no momento da execução do *kernel* as mesmas são pré-inicializadas (KIRK; HWU, 2011).

Uma vantagem do CUDA juntamente com a arquitetura da GPU é a maneira eficiente que possibilita o desenvolvimento de aplicações que podem explorar ao máximo o paralelismo dos dados. Um mesmo trecho de código é executado em paralelo para pequenos blocos de dados, com a existência de várias pequenas memórias *cache*. Os níveis de hierarquia de memória ajudam a esconder a latência de acesso a estes blocos. A latência refere-se ao tempo de acesso a memória, esse tempo é influenciado diretamente pela distância física que a unidade de memória está da unidade que deseja acessá-la. Assim quanto mais próximo do processador a unidade de memória estiver mais rápido será o acesso aos dados devido a menor latência (NVIDIA, 2009).

Ainda existem outros dois espaços de memória somente de leitura, textura e constante, que são otimizados para diversos usos da memória. A memória global, textura e constante são persistentes, não deixando de existir quando a execução do *kernel* termina. A memória de textura oferece diferentes modos de endereçamento, bem como dados de filtragem para alguns formatos de dados específicos (ROCHA; FILHO, 2010).

Em CUDA uma função do tipo *kernel* especifica o código que será executado por todas as *threads* durante a etapa paralela na GPU. Dessa forma todas as *threads* executam o mesmo código, o que faz a programação CUDA ser do tipo SIMT *(Single Instruction Multiple Threads)* o qual é um estilo de programação normalmente utilizado por processadores massivamente paralelos (NVIDIA, 2009).

Através do *kernel* o *hardware* da GPU é acessado e o trecho do programa passado é executado em paralelo numa quantidade N de *threads*. Ele pode receber argumentos como valores ou ponteiros de memórias globais e possui uma série de diretivas que permitem a uma determinada *thread* identificar quais elementos deverão ser processados por ela. A função que define o *kernel* é definida usando a declaração *_global__* e obrigatoriamente terá de ser chamada a partir do *host*. Nessa chamada devem ser definidos alguns parâmetros para a execução da função como, por exemplo, o tamanho e a dimensão do grid e dos blocos, representados pelo tipo de variável $dim3$. O *kernel* também pode invocar outras funções exclusivas do *device*. Para isso as funções deverão estar usando a declaração *__device__* e essas funções podem invocar outras funções que também tenha esse mesmo marcador (YANO, 2010).

A principio os dados não se encontram na memória do *device* e sim na memória do *host*. Sendo assim é necessário realizar a cópia dos dados do *host* para o *device*. Após a realização da cópia através da função *cudaMemCpy()* com o marcador *cudaMemcpyHostToDevice* é que se chama a função *kernel*. Após o processamento e conclusão do trecho enviado a GPU, é necessário mais uma vez fazer a cópia de dados, só que dessa vez no sentido inverso, do *device* para o *host*. Para isso é utilizada a função *cudaMemCpy()* com o marcador *cudaMemcpyDeviceToHost* (MORAIS, 2012).

2.5.2 Paralelismo híbrido

A idéia principal da programação paralela híbrida é ter dispositivos computacionais com arquiteturas diferentes trabalhando juntamente para resolver um determinado problema. Dessa forma, o código deve ser desenvolvido para que se adapte a arquitetura em que será executado. Com a ascensão dos modelos de GPGPU, surgiram muitas plataformas de computação híbridas compostas principalmente por CPU e GPUs.

A Figura 2.6 representa um modelo híbrido, onde a partir de uma *thread master* são inicializadas outras *threads* de acordo com a necessidade do programador. Essas *threads* escravas são responsáveis por receber e repassar os dados que serão processados pelos dispositivos. Neste modelo o *master* fica responsável pela divisão do problema e pela sincronização dos dados quando os mesmos retornam das *threads* escravas.

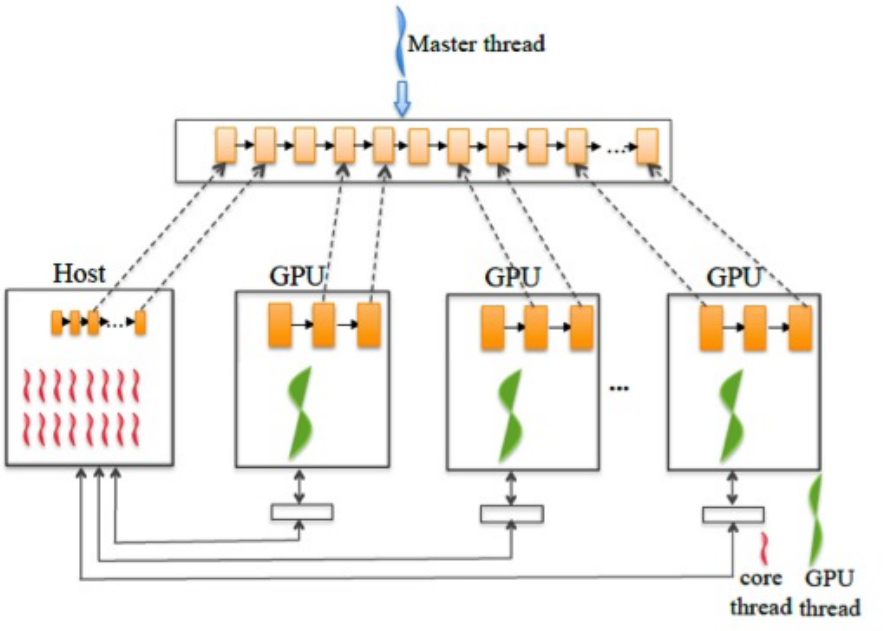

Figura 2.6: Modelo híbrido CPU e GPU
Fonte:(BORATTO et al., 2012b)

De acordo com o modelo proposto na Figura 2.6, pode-se chegar a um algoritmo genérico que utiliza o referido modelo. O Algoritmo 1 mostra o esquema utilizado para realizar o balanceamento de carga entre a CPU e a GPU, através da estrutura condicional `se`, que começa na linha 7. A ideia é gerenciar os núcleos dos multicore e das GPUs, levando-se

em conta um número total de *threads* (nthreads). Isso pode ser feito através das diretivas OpenMP (linhas 1 e 2). As primeiras *threads* são associadas as GPUs e as demais ao multicore. As vezes é possível conseguir melhores resultados, com uma quantidade de *threads* maior que o número de núcleos totais do sistema, devido ao conceito de *Intel Hyper-Threading* (INTEL, 2001). Foi empregado uma estratégia estática para enviar os dados e as tarefas para os núcleos dos multicores e das GPUs. A porcentagem de carga de trabalho que será repassada a cada sistema, é uma entrada do algoritmo e é controlada pelo usuário. Uma vez repassada a porcentagem desejada, se calcula o balanceamento de carga, tendo como base o tamanho do problema e logo após essa etapa se chama o Algoritmo 1. Dessa forma a variável `sizeGPU` armazena o número total de termos que serão calculados por cada GPU e a variável `sizeCPU` armazena a quantidade de dados que serão calculados pelo multicore.

Algoritmo 1 Algoritmo híbrido utilizando multicore e multi-GPU.

```
    ENTRADA
  numberCPUcores /* número de cores na CPU  */
  numberGPUs     /* número de GPUs utilizadas            */
  sizeGPU        /* Tamanho do problema repassado a cada GPU */
  problemSize    /* tamanho total do problema para ser
    dividido entre CPU e GPU                    */

1: nthreads = numberCPUcores + numberGPUs;
2: sizeCPU  = (problemSize-numberGPUs*sizeGPU)/numberCPUcores;
3: omp_set_num_threads(nthreads);
4: #pragma omp parallel
5: {
6:    thread_id = omp_get_thread_num();
7     if(thread_id < numberGPUs) { // cálculo realizado por GPU
8:      first = thread_id * sizeGPU;
9:      cudaSetDevice(thread_id);
10:     Calculo_GPU(thread_id, first, sizeGPU);
11:   } else {                    // cálculo realizado pela CPU
12:     first = numberGPU*sizeGPU + (thread_id-numberGPU)*sizeCPU;
13:     Calculo_CPU(thread_id, first, sizeCPU);
14:   }
15: }
```

2.5.3 Ferramentas matemáticas

LAPACK

O LAPACK *(Linear Algebra Package)*, é uma biblioteca que possui funções escritas em FORTRAN e resolve problemas relacionados a álgebra linear (problemas de valores próprios, sistemas de equações lineares dentre outros). As rotinas LAPACK são escritas de modo que o cálculo possa ser feito através de chamadas BLAS. Essas rotinas foram projetadas para explorar o nível 3 de BLAS *(Basic Linear Algebra Subprograms)* que contém um conjunto de especificações em FORTRAN que são capazes de realizar vários tipos de multiplicações envolvendo matrizes. Dessa forma, a sua utilização promove uma grande eficiência em computadores de alto desempenho (BORATTO et al., 2012b).

CUBLAS

Com a utilização do cuda, a nvidia disponibiliza uma biblioteca equivalente ao LAPACK, porém otimizadas para o uso em GPUs, denominada CUBLAS (*CUDA Basic Linear Algebra Subroutines*) (NVIDIA, 2013). A biblioteca CUBLAS é implementada usando o conjunto de ferramentas CUDA baseado na linguagem C, e portanto fornece uma *API* no estilo da *API* do C. Pode ser usada por aplicações escritas na linguagem FORTRAN (NVIDIA, 2009).

2.6 Métrica de desempenho

Com o uso de de sistemas paralelas para resolver um determinado problema, o principal objetivo é comparar o desempenho e saber se há ou não um ganho de desempenho do modelo paralelo em relação ao modelo sequencial. Uma das principais métricas utilizadas para obter essa informação é o fator *speedup* o qual representa o ganho de velocidade de processamento da aplicação quando a mesma é executada com uma quantidade x de processadores. Sendo assim quanto maior o *speedup*, mais rápido será o código paralelo (PINTO, 2011). O *speedup* **(S)** é obtido quando se divide o tempo de execução sequêncial **(Ts)** pelo tempo de execução paralelo **(Tp)** de uma aplicação, como é exibido na Equação 2.1.

$$S = \frac{Ts}{Tp} \tag{2.1}$$

O *speedup* é um fator que depende de dois itens: o número de processadores da arquitetura e do software paralelizado. Esta métrica só tem sentido a partir do momento em que a arquitetura possua ao menos dois processadores. Esta métrica terá pouca utilidade se o software não tiver um código paralelo compatível com o número de processadores. Isto porque alguns processadores podem ficar ociosos durante o processamento e não contribuirão na redução do tempo de processamento (PINTO, 2011).

2.7 Resumo do capítulo

Neste capítulo foram descrias as ferramentas básicas que serão utilizadas no decorrer do presente trabalho. Também foi descrito o hardware onde serão realizados os experimentos e as bibliotecas de *software* que serão utilizadas. Também foi descrito as ferramentas matemáticas. Na parte final do capítulo é mostrada a métrica de desempenho que será utilizada no decorrer desse trabalho.

Capítulo 3

Trabalhos relacionados

Para contextualizar o que será abordado nesta monografia é necessário uma revisão dos trabalhos disponíveis na literatura. O principal objetivo é avaliar os aspectos determinantes que motivaram e impulsionaram a criação e o desenvolvimento de aplicações relacionadas com os problemas estudados. Desta forma, pode-se classificar em três grupos os problemas tratados neste trabalho, os quais são divididos em tópicos.

3.1 Modelos paralelos de alto desempenho

O primeiro grupo de trabalho se encaixa na computação de alto desempenho onde partindo de um problema de alto custo computacional se obtém um estudo comparativo entre versões sequenciais e paralelas de um algoritmo, visando assim comparar o potencial de processamento das GPUs frente as CPUs. Neste grupo de trabalho utilizam algoritmos ou modelos computacionais que demandam uma grande quantidade de processamento. A partir da escolha do algoritmo são desenvolvidas versões paralelas e sequenciais e calcula-se o tempo de execução entre as diferentes técnicas de paralelização escolhidas pelos autores.

Em (YANO, 2010), se utiliza um algoritmo de multiplicação de matrizes onde o autor desenvolve uma versão sequencial e uma versão paralela para ambientes de GPU . O modelo voltado para GPU utiliza o esquema de divisão das *threads*, onde cada *thread*

fica responsável por uma parte do problema, que é obtida de acordo com o tamanho da matriz e a quantidade total de *threads* que serão utilizada. Em (BORATTO et al., 2012b) visando obter uma melhor performance em um modelo matemático de espacialização do relevo, utiliza-se ferramentas baseadas no paradigma de memória compartilhada, as quais se mostraram eficientes e obtiveram os resultados dentro do esperado principalmente com o uso da *API* CUDA, servindo assim como referência para serem aplicados a este trabalho.

3.2 Computação paralela heterogênea

Já o segundo grupo de trabalhos remete a área da computação paralela heterogênea onde um mesmo algoritmo é executado em ambientes híbridos. Em (CHATAIN, 2012) são realizados testes com o OpenACC, que é um novo padrão de programação paralela onde através de algumas diretrizes é possível selecionar onde determinado trecho de código será executado. Nestes trabalho procura-se comparar o desempenho das aplicações que são executadas diretamente na GPU (especialmente as que utilizam o CUDA), além das características e a facilidade desse novo padrão de programação. Por fim, discorrem também sobre as limitações do OpenACC, fazendo uma análise do seu impacto na performance do algoritmo. Também utilizam técnicas de *benchmarks* para a validação do estudo no que diz respeito ao desempenho do algoritmo. De acordo com u autor é possível verificar que por não permitir um balanceamento de carga entre os dispositivos, o desempenho do OpenACC frente ao CUDA não é satisfatório.

Em outro trabalho estudado (PINTO, 2011) é feita uma abordagem conceitual sobre os sistemas paralelos e são apresentados ambientes de execução que oferecem suporte para arquiteturas multicore híbridas (CPU e GPU). Nesses ambientes há um *framework* para políticas de escalonamento que especifica como as tarefas serão distribuídas e redistribuídas entre os recursos de processamento. Em (ROSA, 2013) os autores abordam a temática de paralelismo híbrido entre CPU e GPU e como teste de validação implementam um algoritmo para calcular produto escalar. Também realizam o balanceamento de carga visando obter uma melhor performance do algoritmo híbrido. Em outro trabalho (LASTOVETSKY; ZHONG; RYCHKOV, 2012) é feito um estudo que visa analisar e otimizar o uso de *softwares*

científicos em arquiteturas híbridas que possuam multicore e GPUs. Segundo os autores, foi possível observar que em *softwares* que possuem uma grande quantidade de dados a serem processados, a comunicação entre as memórias da CPU e da GPU tomam uma grande quantidade de tempo. Além disso, verificou-se a necessidade de realizar o balanceamento de carga proporcional a potência de computação dos dispositivos, o que gerou uma otimização no algoritmo híbrido desenvolvido pelos referidos autores.

3.3 Modelos matemáticos aplicados

Outro grupo de trabalho pertence a área de pesquisas aplicadas em modelagem matemática, onde partindo de um problema real, associado a um conjunto de hipóteses se chega a um problema matemático que é usado para uma posterior resolução do problema real, como pode ser visto em (RAFIKOV; LIMEIRA, 2012; FILHO, 2012; BORATTO et al., 2013). A partir do seu estudo é possível obter e realizar explicações e interpretações dos fenômenos estudados realizando previsões e apontando tendências. Esse grupo de trabalho é caracterizado por necessitar de um alto poder computacional para que seja possível representar grandes quantidades de dados computacionais com um nível satisfatório de informações. O autor Sodré (2007) faz uma conceitualização sobre a modelagem matemática e afirma que existe uma forma matemática unificada capaz de tratar várias teorias científicas e matemáticas e que podem ser descritas como uma dinâmica geral, que vem sendo desenvolvida em áreas como Cálculo Diferencial e Equações Diferenciais. Ainda segundo (SODRÉ, 2007), podem existir dois tipos de modelos matemáticos, os reais, como proposto pelo autor (RAFIKOV; LIMEIRA, 2012) e os abstratos como o conjunto dos números naturais.

3.4 Comparativa entre os trabalhos relacionados

A Tabela 3.1 mostra de maneira esquemática as características de alguns dos trabalhos mais significativos anteriormente descritos, classificados por grupos, onde cada grupo contém um objetivo específico que se deseja alcançar. Desta forma, podemos encontrar três grupos:

Grupo 1 (G1)

Aqueles trabalhos que tem por objetivo usar a computação de alto desempenho para otimizar as tarefas de alto custo computacional.

Grupo 2 (G2)

Um grupo formado pelos trabalhos que utilizam equações matemáticas para representar um sistema real.

Grupo 3 (G3)

Por último, a proposta desse trabalho, que resume todos os objetivos a serem alcançados, seguindo a metodologia proposta e baseando-se no tempo de execução de um modelo matemático em um ambiente que possui diferentes arquiteturas.

Tabela 3.1: Comparação dos diferentes trabalhos.

Referências	Descrição	G1	G2	G3
(FILHO, 2012; MORAIS, 2012)	Aplicações Paralelas	•	—	•
(BORATTO et al., 2013; BORATTO et al., 2012b)	Modelagem Computacional	•	—	•
(SILVA, 2006; ROSA, 2013; CHATAIN, 2012)	Paralelismo Híbrido	•	•	•
(YANO, 2010; PINTO, 2011)	Ferramentas utilizadas	—	•	•

Fonte: [Próprio autor, 2014]

3.5 Justificativa e oportunidades de pesquisa

Alguns trabalhos que propuseram estudos na área de computação paralela híbrida destacaram a importância de soluções de adaptar a correta distribuição de carga de trabalho entre as diferentes arquiteturas. Porém, não foi discutido uma forma genérica de para a construção de um modelo para qualquer tipo de problema. Esta generalização torna-se bastante complicada de implementar, sendo necessário um esforço de pesquisa, validação e experimentação através de outros problemas aplicados, mas sendo essencial para obter a melhor solução independentemente do problema.

Assim o presente trabalho propõe otimizar a solução implementada e complementada pelos trabalhos relacionados, tendo como base um estudo de caso proposto por (BORATTO et al., 2013), e assim fornecer uma otimização ao mesmo.

Capítulo 4

Projeto experimental - polinômios matriciais

Nesta parte da monografia serão abordados os detalhes práticos do projeto: o planejamento e as etapas da construção do algoritmo híbrido que será utilizado nos experimentos.

4.1 Introdução

Modelos matemáticos são utilizados em muitos campos da atividade humana, como: Matemática, Física e a Química. Um modelo normalmente representa uma simplificação do mundo real ou alguma forma conveniente de trabalhar com este mundo, mas as principais características do mundo real devem ser representadas no modelo, de modo que o seu comportamento seja semelhante ao sistema modelado.

O estudos de polinômios matriciais começou a cerca de cinco séculos. Porém o estudo envolvendo equações polinomiais com matrizes e/ou polinômios com coeficientes matriciais é recente. Um polinômio matricial de grau m na variável z pode ser definido de acordo com a Equação 4.1

$$p(Z) = \alpha_d Z^d + \alpha_{d-1} Z^{d-1} + \cdots + \alpha_1 Z + \alpha_0 I \tag{4.1}$$

onde $\bar{\alpha} = [\alpha_i]_{i=0,\ldots,g}$ e $Z, I \in \mathcal{R}^{n \times n}$, onde I representa a matriz identidade.

Na física aplicada e na engenharia existem muitos problemas cuja solução depende da resolução de um conjunto de equações polinomiais. Como exemplo, podemos citar o problema de encontrar a frequência de vibração natural em muitos sistemas elétricos e mecânicos. Também são utilizados em sistemas robustos, onde a estabilidade do sistema é determinada pela localização das raízes de uma equação. Dessa forma, evidencia-se a importância das equações polinomiais (ZARPELON, 2004).

Um problema identificado para o cálculo de equações polinomiais matriciais foi o alto custo computacional, que está diretamente ligado ao tamanho da matriz (**Z**) e seu grau (**g**). Quando o tamanho da matriz e o seu grau tornam-se elevados, a demanda por processamento torna-se inestimável. Dessa forma, a tendência é que em um determinado momento, somente aumentar o poder de processamento da CPU não será proporcional à diminuição do tempo de execução do algoritmo. Uma forma de reduzir esse problema, é utilizar técnicas de programação paralela e *hardware* específico, para otimizar o algoritmo.

4.2 Modelo paralelo multi-GPUs

Para o cálculo dos polinômios matriciais, podem ser abordadas várias técnicas. Em (BORATTO et al., 2013) a técnica escolhida consistiu no uso da biblioteca CUBLAS (para o processamento em GPU) e a biblioteca LAPACK (para processamento em CPU). O Algoritmo 2 ilustra uma pseudo-solução que pode ser adaptada para uso em sistemas multi-GPUs e em sistemas híbridos onde *n* representa o grau, X representa o polinômio, d a dimensão e o $\bar{\alpha}$ representa o coeficiente inicial.

Em (BORATTO et al., 2013) foram usadas duas etapas para poder realizar o cálculo de polinômios matriciais:

- **Etapa 1**: Cálculo das potência da matriz.
- **Etapa 2**: Cálculo do polinômio matricial.

Algoritmo 2 Algoritmo para o cálculo de um polinômio matricial

1: **procedure** EVALUATEMATRIX(n, X, d, $\bar{\alpha}$)
2: $P \leftarrow \alpha_0 I$
3: $P \leftarrow P + \alpha_1 X$
4: $B \leftarrow X$
5: **para** $i \leftarrow 2, d \leftarrow$
6: **faça** $A \leftarrow B$
7: $B \leftarrow A \cdot X$
8: $P \leftarrow P + \alpha_i B$
9: **fim para**
10: **return** P
11: **fim procedure**

Com essa divisão em etapas, é possível realizar o cálculo paralelo de mais de um polinômio de mesmo grau, observando porém que o armazenamento do cálculo das potências poderia representar uma limitação. Para isso foi desenvolvido o Algoritmo 3, onde podem ser usadas até duas GPUs sendo que desta forma o cálculo das potências pares seria realizada em uma GPU (g=0) e o cálculo das potências impares em outra GPU (g=1).

Algoritmo 3 Algoritmo para calcular os coeficientes de uma matriz em duas GPUs.

1: **função** COMPUTE_POWERS(d, X) **devolve** (A, m)
2: `#pragma omp parallel for`
3: **para** $g \leftarrow 0, 1 \leftarrow$
4: **faça** $m = d/2 + g.mod(d, 2)$
5: $A(0) \leftarrow X$
6: $A(1) \leftarrow X\ A(0)$ ▷ $A(1) = X^2$
7: **se** $g = 0$ **então**
8: $A(0) \leftarrow A(1)$ ▷ $A(0) = X^2$ if $g = 0$
9: **fim se**
10: $A(2) \leftarrow A(0).A(1)$ ▷ $A(2) = \begin{cases} X^4 & \text{if } g = 0 \\ X^3 & \text{if } g = 1 \end{cases}$
11: **para** $i \leftarrow 3, m \leftarrow$
12: **faça** $A(i) \leftarrow A(1)A(i-1)$ ▷ $A(i) = X^2 \cdot A(i-1)$
13: **fim para**
14: **fim para**
15: **fim função**

As potências da matriz ficam armazenadas em um *array* de matrizes denominado (A). O valor de (A) para cada GPU após a execução do algoritmo será de acordo com a Equação 4.2.

$$A = \begin{cases} \begin{pmatrix} X^2 & X^2 & X^4 & X^6 & X^8 & \dots \end{pmatrix} & \text{se GPU0} \\ \begin{pmatrix} X & X^2 & X^3 & X^5 & X^7 & \dots \end{pmatrix} & \text{se GPU1} \end{cases} . \tag{4.2}$$

Os valores das matrizes de A(0) e A(1), tem uma maneira especial de utilização, já que foram necessárias algumas adequações para que o produto das matrizes fosse executado de maneira correta pelas rotinas CUBLAS, sendo isso representado no Algoritmo 3. Na linha 11 é realizado um laço *for*, onde para a posição A(i) recebe o valor da multiplicação da posição inicial A(1) pelo valor da posição A(i-1), como sugere a Equação 4.2.

Na etapa 2 é realizado o cálculo do polinômio matricial. Partindo de um laço *for* de 0 a N, onde N representa o número de GPUs. O Algoritmo 4 usa a idéia base do Algoritmo 3 que consiste em um laço *for*, executado por duas GPUs, utilizando o *array* de matrizes A, calculado na etapa anterior. Cada GPU realiza o cálculo referente a sua matriz A (linhas 8 e 11) e repassa ao *array* de matrizes *(representado pela variável hb)*, que está alocado no *host* e aceita até duas matrizes. A GPU0 calcula os termos pares e os coloca em uma matriz B, enquanto que a GPU1 calcula os termos ímpares e os coloca em outra matriz B (lembrando que cada GPU tem alocada em sua própria memória, uma matriz B). Na linha 13, cada GPU repassa o valor de B, para o *array* de matrizes *hb*. O último passo (linha 15) é realizado pelo *host*(CPU) para que assim seja possível obter o resultado final (Algoritmo 4).

Segundo Boratto et al. (2013), o modelo paralelo com duas GPUs é muito mais eficiente que o modelo executado somente nos multicores tanto que a partir de determinado momento se utiliza apena a comparação entre o modelo sendo executado em uma GPU e o modelo sendo executado em duas GPUs. Observou-se também que o ganho de desempenho do polinômio cresce se o grau ou o tamanho da matriz aumentar (embora neste último caso o incremento do *speedup* é menor sendo em alguns casos nulo). O modelo híbrido envolvendo duas GPUs se mostrou eficiente, já que em uma certa configuração (tamanho da matriz igual a 8000 e o grau igual a 20) conseguiu obter um desempenho 80% superior ao modelo sendo executado somente por uma GPU.

Algoritmo 4 Algoritmo para o cálculo de um polinômio matricial em duas GPUs.

1: **função** EVALUATE2(n, q, $\bar{\alpha}$, A) **devolve** P
2: `#pragma omp parallel for`
3: **para** $g \leftarrow 0, 1 \leftarrow$
4: **faça** $m = q/2 + g.mod(q, 2)$
5: **se** $g = 0$ **então**
6: $B \leftarrow \alpha_0 I$
7: **senão**
8: $B \leftarrow \alpha_1 A(0)$
9: **fim se**
10: **para** $i \leftarrow 1 + g, m \leftarrow$
11: **faça** $B \leftarrow B + \alpha_{2i-g} A(i)$
12: **fim para**
13: $hB(g) \leftarrow B$
14: **fim para**
15: (Host) $P \leftarrow hB(0) + hB(1)$
16: **fim função**

De acordo com Rosa (2013) computadores com mais de uma GPU com suporte a tecnologia GPGPU estão facilmente disponíveis, o que o levou a testar o desempenho de aplicações que utilizassem paralelismo híbrido entre GPU e CPU, tentando assim otimizar o desempenho do sistema. Na primeira abordagem a distribuição de carga do cálculo de um produto escalar foi igualitária entre CPU e GPU, porém os resultados esperados foram abaixo da expectativa, com *speedup* próximo a 4,1 vezes enquanto que no modelo executado somente pela GPU o *speedup* foi de 8,6 vezes, quando comparado ao modelo executado somente pela CPU, ou seja, o algoritmo híbrido foi aproximadamente 2 vezes mais lento que o algoritmo executado somente por 1 GPU. Analisando os resultados obtidos, decidiu-se por modificar a distribuição de carga entre CPU e GPU já que no ambiente de teste a capacidade de processamento da GPU era superior a da CPU. Após essa mudança o modelo paralelo híbrido para cálculo de produto escalar conseguiu um *speedup* de 9,1 vezes para a seguinte distribuição de carga (90% para a GPU e 10% para a CPU).

4.3 Aplicação do Modelo Paralelo Híbrido CPU+GPU

A próxima subseção irá apresenta como foi desenvolvido o algoritmo paralelo híbrido entre CPU e GPU para a otimização do cálculo dos polinômios matriciais.

4.3.1 Modelo Paralelo Híbrido

A aplicação híbrida do Modelo dos Polinômios Matriciais foi implementada na linguagem C com o uso do CUDA e da *API* OpenMP. Os dados primários de entrada são o tamanho e o grau das matrizes e duas variáveis de entrada, uma responsável por selecionar o dispositivo que será utilizado e a outra por executar ou não o modelo em um modo híbrido.

O modelo segue o método de construção proposto pelo autor (BORATTO et al., 2013) através do Algoritmo 2.

O código paralelo realiza a divisão das potências que deverão ser calculadas pelos recursos computacionais disponíveis no sistema. A Figura 4.1 ilustra como será a divisão das tarefas entre a CPU e a GPU, onde o *workload* GPU corresponde a parte da tarefa que foi repassada e será processada somente pela GPU, enquanto que as *threads* da CPU serão responsáveis por realizar os cálculos restantes que não foram repassados a GPU.

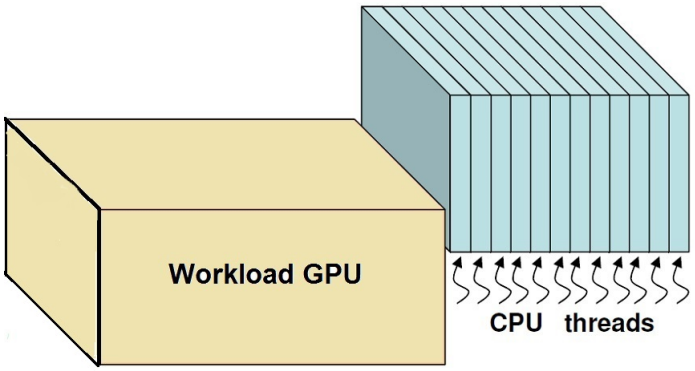

Figura 4.1: Divisão dos cálculos de potências entre GPU e CPU
Fonte: (BORATTO et al., 2012b)

Durante a execução do modelo, a parte do cálculo das potências é a mais custosa de todo o processo e como há uma dependência de dados, foi preciso adotar uma estratégia de divisão do modelo. A forma utilizada para poder realizar o cálculo simultâneo das potências, foi de dividir os coeficientes pares e ímpares entre as diferentes arquiteturas, como foi visto no Algoritmo 3. A principio, essa divisão foi feita de maneira equitativa entre os recursos computacionais disponíveis, ignorando o poder de processamento de cada um.

Como o modelo proposto atende a um ambiente híbrido que possui dois tipos de arquiteturas (multicores e GPU), foi feito um laço que realiza duas iterações paralelas em *threads* distintas, sem o compartilhamento de dados. Dessa forma foi utlizado o comando **#pragma omp threadprivate**. o qual tem a função de tornar particular as variáveis utilizadas por cada *threads*.

Seguindo o modelo proposto pelo autor (BORATTO et al., 2013), foi desenvolvido inicialmente o Algoritmo 5, onde inicialmente são alocadas as variáveis *mycols, ocols, mycol, ocol e myidx*, que são responsáveis pela divisão da quantidade de linhas e colunas que cada dispositivo ficará responsável por calcular (para o Algoritmo 5, cada dispositivo irá ficar com metade da quantidade de linhas e colunas da matriz).

Também são alocados os ponteiros que serão usados durante o processo para armazenar as matrizes, assim como são preenchidos os ponteiros A(0) e A(1) com a matriz inicial e o seu quadrado respectivamente. Nas linha 10 e 14 ocorre o preenchimento dos ponteiros A(0) e A(1), utilizando-se da função *dgemm*, pertencente a biblioteca MKL_BLAS e que tem por objetivo realizar a multiplicação entre duas matrizes quadráticas. Após isso, na linha 20 ocorre chamada da função *encaj_hibrido*, que é responsável por realizar o cálculo do polinômio.

O Algoritmo 6, que também representa a parte destinada somente aos multicore, mostra a construção do algoritmo recursivo para o cálculo do polinômio. A partir dessa função são calculados tanto as potências quanto os coeficientes restantes, até que se atinja o caso base, linha 3, onde o grau do polinômio (representado pela variável *degree*) é o menor possível (de acordo com o dispositivo que estiver sendo executado esse valor pode ser 0 ou 1). O caso geral consiste em decrementar o grau do polinômio com chamadas recursivas até que se atinja o caso base. Após essa etapa, é realizada a chamada da função *eval_hibrido_CPU* (linha 10), que tem a finalidade de multiplicar o coeficiente associado ao grau pela matriz correspondente. Quando finalizada essa etapa, é verificado na linha 11 se o grau que foi decrementado é zero ou diferente de zero. Caso seja zero será feita a multiplicação do termo independente do polinômio, através da função *daxpy*. Sendo diferente de zero é realizada a multiplicação entre a matriz inicial pela matriz da posição anterior, seguindo os passos da Equação 4.2.

Algoritmo 5 Algoritmo híbrido utilizando multicore para o cálculo de polinômios matriciais

```
1: else if (cpu_thread_id == 1) {
2
3: mycols = n/2+(n%2)*!(cpu_thread_id);
4: ocols = n-mycols;
5: mycol = ocols*(cpu_thread_id);
6: ocol = mycols*!(cpu_thread_id);
7: myidx = mycol*n;
8:
9: for( i=1; i<boxing; i++ ) {  /* X^(i) = X*X^(i-1) */
10:  dgemm( "N", "N", &n, &mycols, &n, &ONE, d_X, &n, d_A[i-1], &n,
11:                 &ZERO, d_A[i], &n );
12:
13: }
14:
15: dgemm( "N", "N", &n, &mycols, &n, &ONE, d_X,&n,
16:               d_A[boxing-1], &n, &ZERO, &(d_B[myidx]), &n );
17:
18: free(d_X);
19: d_X= (double*) malloc(n*mycols*sizeof(double));
20:
21: encaj_hibrido_cpu( n, degree, boxing, v );
22:
23: memmove( &(P[myidx]), d_Y, n*mycols*sizeof(double) );
24: } /* Fim hibrido multicore */
```

O Algoritmo 7 mostra a função *eval_hibrido_cpu*, que recebe como parâmetros, o tamanho da matriz, representado pela variável *n*, o grau que é representado pela variável *degree* e o *array v* que armazena todos os coeficientes do polinômio matricial. Na linha 8, verifica-se o atual grau de polinômio que se deseja calcular e caso seja zero, a função retornar para o Algoritmo 6. Na linha 12, a função *dscal* tem por objetivo ajustar a correta precisão da matriz que está na variável *d_X*. Já na linha 14 é realizada a multiplicação entre o coeficiente presente no *array* de coeficientes *v* e a matriz associada ao mesmo, enquanto que na linha 17 é realizada a multiplicação da matriz presente no *array d_A* em função do grau do polinômio.

Os Algoritmos 5, 6 e 7 foram desenvolvidos utilizando a biblioteca MKL_BLAS, fornecida pela INTEL. Vale ressaltar que também foram desenvolvidos outros algoritmos com essas mesma características, para realizar a parte dos cálculos destinadas somente a GPU, sendo este último construído utilizando a biblioteca CUBLAS, fornecida pela NVIDIA.

Algoritmo 6 Algoritmo recursivo para o cálculo de polinômios matriciais utilizando multicore.

```
1: void encaj_hibrido_cpu( int n, int degree,  int enc, double *coef ) {
2:
3:  if( degree<=enc ) {
4:    // Caso Base
5:    eval_hibrido( n, degree, coef );
6:   } else {
7:    // Caso geral
8:    degree_enc = degree-enc-1;
9:    encaj_hibrido( n, degree_enc, enc, coef );
10:    eval_hibrido_CPU( n, enc, &coef[degree-enc] );
11:    if( degree_enc > 0 ) {
12:       dgemm( "N", "N", &zy, &mycols, &zy, &ONE, d_B,
13:                          &zy, d_Y, &zy, &ONE,  d_X, &zy );
14:    }else {
15:      int n2 = n*mycols, inc = 1;
16:      daxpy(&n2, &coef[degree_enc], &(d_B[myidx]), &inc, d_X, &inc);
17:    }
18:  }
19:     memmove(d_Y,d_X,n*mycols*sizeof(double));
20:}
```

Algoritmo 7 Algoritmo híbrido utilizando multicore para o cálculo de polinômios matriciais .

```
void eval_hibrido_cpu( int n, int degree, double v[] ) {
1:
2:   int i = 0, ind = 0;
3:   int incx = 0;
4:   int incy = n+1;
5:   int inc  = 1;
6:   int size = n*mycols;
7:
8:  if( degree==0 ) {
9:       return;
10:  }
11:
12:  dscal( &size, &ZERO,d_X, &inc );
13:  ind = degree;
14:  daxpy( &mycols, &v[ind--], &ONE, &incx, &(d_X[mycol]), &incy );
15:
16:  for( i=0; i<degree; i++ ) {
17:      daxpy( &size, &v[ind--], d_A[i], &inc, d_X, &inc);
18:  }
19:}
```

De acordo com Rosa (2013) é importante que a percentagem de carga de trabalho repassada a cada arquitetura seja feita proporcionalmente a potência de computação de cada arquitetura, para que assim seja utilizada o máximo de recursos disponíveis. Porém no Algoritmo 5

o valor sempre é selecionado baseando-se numa divisão equitativa de tarefas, ignorando o poder de computação de cada dispositivo. No próximo capítulo serão discutidos os resultados e as implicações deste tipo de abordagem a um modelo híbrido.

4.4 Resumo do capítulo

Neste capítulo foi apresentado o estudo de caso que será utilizada para a validação desse trabalho. Também foi descrito o seu funcionamento e os passos necessários para a sua construção. Na parte final do capítulo são mostradas trechos do algoritmo híbrido, que foram construídos baseando-se em pseudo-códigos sugeridos pelo Autor (BORATTO et al., 2013).

Capítulo 5

Resultados experimentais

Serão descritos neste capítulo os resultados da implementação e execução de um modelo matemático aplicado a uma plataforma híbrida.

5.1 Introdução

No capítulo anterior foi apresentado o estudo de caso referente a resolução de Polinômios Matriciais que utiliza múltiplas GPUs. A partir deste estudo de caso, foi desenvolvida uma versão paralela híbrida entre multicores e a GPU e os resultados serão apresentados neste capítulo.

5.2 Análise dos Resultados

Para analisar o comportamento do modelo híbrido e servir de comparação, foi desenvolvida uma versão para o cálculo de polinômios matriciais utilizando uma arquitetura híbrida entre CPU e GPU.

Primeiramente, na Figura 5.1 é possível ver o tempo de execução (acima) e o *speedup* (abaixo) para o cálculo de um polinômio matricial no **Sistema 1**. Para os diferentes graus do polinômio (entre 4 e 20), onde a matriz X possuía um valor fixo $n = 4000$. Ambos os

gráficos mostram como se comporta o algoritmo sendo executado na GPU e na CPU. Dessa forma pode-se notar que quando executado o cálculo na GPU se consegue um *speedup* muito grande quando comparada a CPU. O termo ”híbrido 50/50” significa que 50% da computação será realizada pela GPU e os outros 50% pela CPU, ou seja, a distribuição da quantidade de computação que cada dispositivo irá calcular não foi feita de acordo com as suas potências relativas. Isso foi feito para poder analisar os impactos do balanceamento de carga na performance de um algoritmo híbrido.

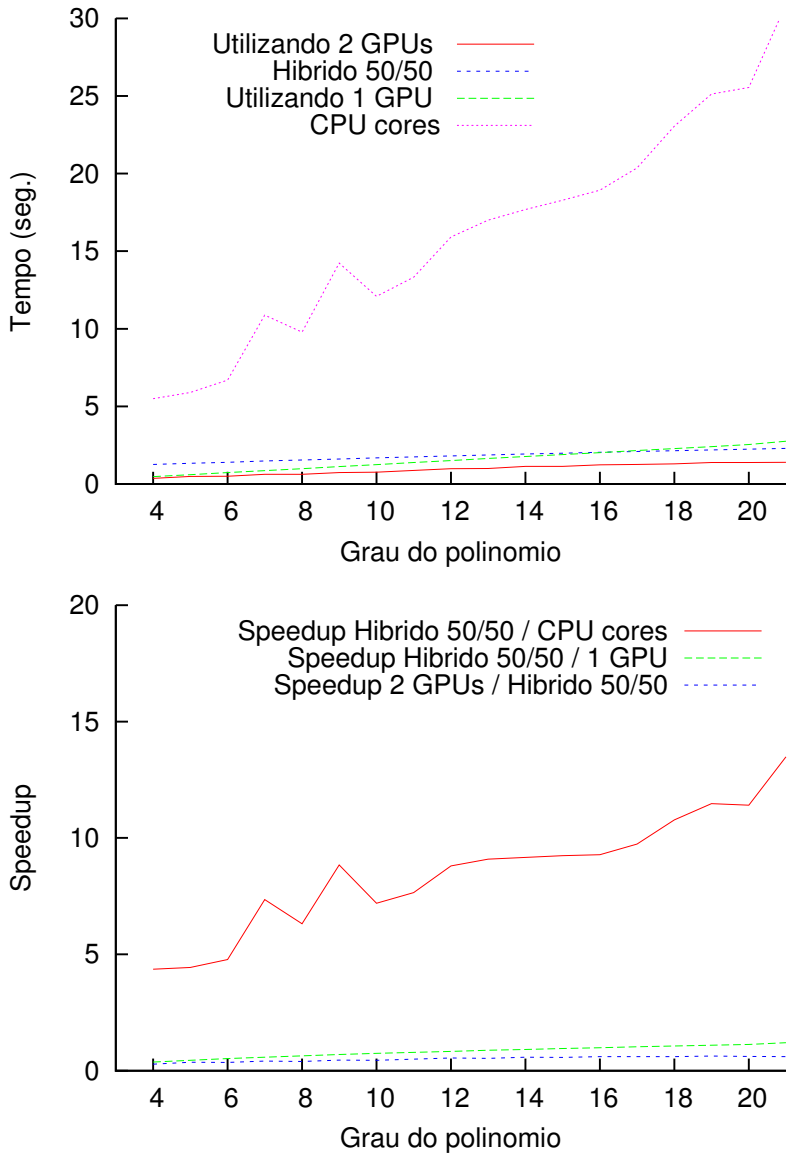

Figura 5.1: Tempo de execução e speedup para o cálculo de um polinômio matricial de tamanho $n = 4000$ no Sistema 1, variando o grau.

Fonte: [Próprio autor, 2014]

Também é possível observar o desempenho do algoritmo híbrido. Inicialmente obteve-se um desempenho inferior ao algoritmo executado somente por 1 GPU. Porém com o aumento do grau do polinômio a diferença de performance entre os 2 algoritmos foi reduzindo e a partir do grau 18 o algoritmo híbrido superou o algoritmo similar executado somente por 1 GPU. Isso ocorre devido ao aumento do tempo para cópia dos dados entre a CPU e a GPU. No modelo híbrido serão copiadas 50% a menos de dados quando comparado a cópia de dados que será feita pelo algoritmo executado somente por 1 GPU.

Na Tabela 5.1 é possível observar com precisão os tempos de execução para cada dispositivo no **Sistema 1**.

Grau do polinômio	Sequencial	1 GPU	Hibrido 50/50	2 GPUs
4	5,50	0,47	1,26	0,36
6	6,69	0,73	1,40	0,50
8	9,79	0,99	1,55	0,62
10	12,09	1,25	1,68	0,76
12	15,92	1,51	1,81	0,99
14	17,69	1,77	1,93	1,13
16	18,93	2,02	2,04	1,23
18	23,06	2,28	2,14	1,30
20	25,55	2,54	2,24	1,38

Tabela 5.1: Tempo de execução (em segundos) e speedup para o cálculo de um polinômio matricial de tamanho $n = 4000$ no Sistema 1.
Fonte: [Próprio autor, 2014]

5.2.1 Balanceamento de carga ótimo

Através da realização de testes de forma manual, para o **Sistema 1**, se obteve um melhor resultado realizando o seguinte balanceamento de carga para o algoritmo híbrido: 90% da computação a ser realizadas pela GPU e os 10% restantes sendo realizada pela CPU. Nessa configuração o algoritmo híbrido teve um desempenho muito próximo ao algoritmo executado por 2 GPUs, como pode ser visto na Figura 5.2 (tempo de execução (acima) e o *speedup* (abaixo)). É possível observar principalmente pelo *speedup* que apesar da melhora de performance do algoritmo híbrido com balanceamento de carga, este em nenhum

momento conseguiu superar a performance do algoritmo executado por 2 GPUS. Neste caso, o termo ”híbrido 90/10” ilustra esse ótimo balanceamento de carga. Outra observação importante é em relação aos algoritmos híbridos com e sem o correto balanceamento de carga, onde para grau 20 a diferença entre eles chega próximo a 40% (com vantagem para algoritmo com balanceamento de carga). Nota-se também que o algoritmo ”híbrido 50/50”, no espaço de amostragem do teste realizado, permanece a maior parte do tempo (desde o grau 4 até o grau 18) com o *speedup* menor que 1, o que indica que esse algoritmo está tendo uma performance inferior quando comparado a outro algoritmo (neste exemplo, o algoritmo que é executado por 1 GPU).

Dessa forma é possível observar a importância de realizar o correto balanceamento de carga de acordo com o poder computacional de cada dispositivo presente no sistema, já que esse balanceamento influi de maneira direta no tempo de execução do algoritmo.

A Tabela 5.2 mostra com uma maior precisão os tempos de execução do modelo híbrido com e sem o correto balanceamento de carga comparado com os outros algoritmos paralelos.

Grau do polinômio	1 GPU	Hibrido 50/50	Hibrido 90/10	2 GPUs
4	0,47	1,26	0,50	0,36
6	0,73	1,40	0,65	0,50
8	0,99	1,55	0,79	0,62
10	1,25	1,68	0,94	0,76
12	1,51	1,81	1,07	0,99
14	1,77	1,93	1,21	1,13
16	2,02	2,04	1,34	1,23
18	2,28	2,14	1,46	1,30
20	2,54	2,24	1,59	1,38

Tabela 5.2: Tempo de execução (em segundos) para o cálculo de um polinômio matricial de tamanho $n = 4000$ no Sistema 1.

Fonte: [Próprio autor, 2014]

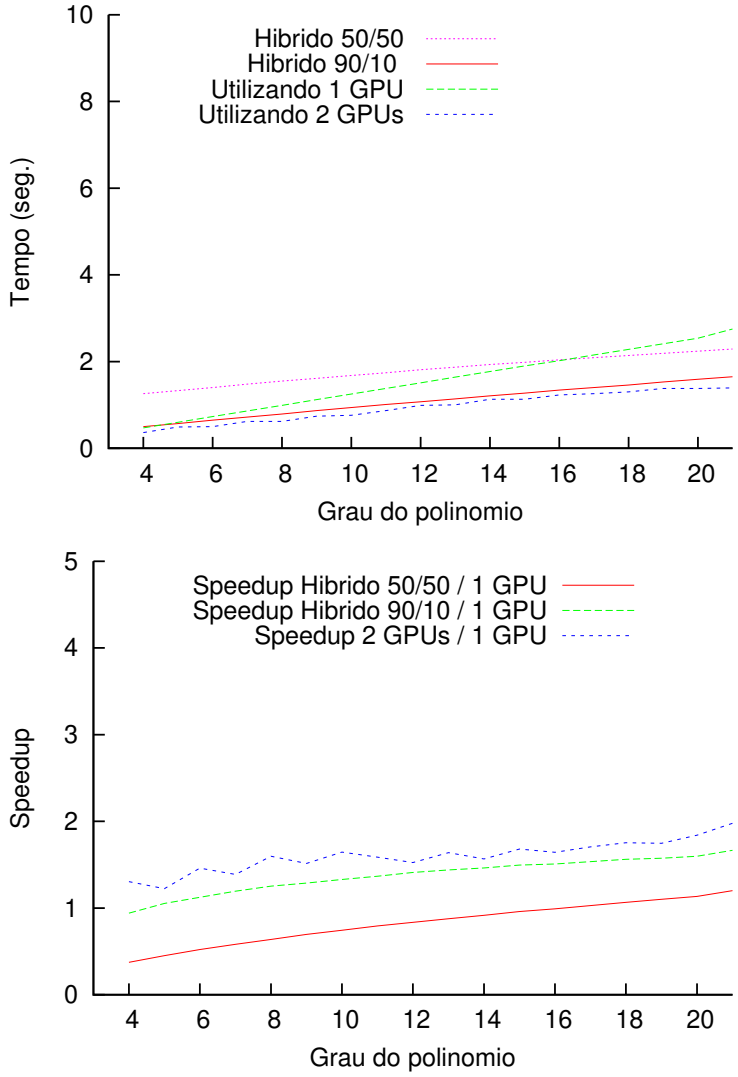

Figura 5.2: Tempo de execução para o cálculo de um polinômio matricial de tamanho $n = 4000$ no Sistema 1, variando o grau e aplicando o balanceamento de carga.
Fonte: [Próprio autor, 2014]

Capítulo 6

Conclusões e trabalhos futuros

A área de modelagem matemática se caracteriza por necessitar de um alto poder computacional para representar com uma maior fidelidade os resultados. Para que esses modelos sejam executados de uma maneira mais eficiente convém utilizar técnicas de computação paralela.

Uma das abordagens consiste em utilizar a programação paralela híbrida, buscando alcançar um melhor desempenho na execução do algoritmo através do uso dos recursos disponíveis.

Nesta pesquisa foi desenvolvida uma otimização sobre o modelo matemático dos Polinômios Matriciais, que utiliza a programação paralela híbrida entre as arquiteturas heterogêneas. Desta forma, buscou-se obter um desempenho para o algoritmo híbrido, superior ao algoritmo similar sendo executado somente por 1 GPU.

Para a correta validação, utilizou-se dispositivos com diferentes potências computacionais, afim de melhor avaliar os resultados obtidos. Também utilizou-se um balanceamento da carga de trabalho entre os diferentes dispositivos, afim de tornar o algoritmo híbrido ainda mais eficiente. Neste contexto, o algoritmo híbrido se mostrou mais eficiente quando se utiliza o correto balanceamento de carga, chegando a ser cerca de 40% mais rápido que o algoritmo executado por 1 GPU ao passo que não realizando o correto balanceamento de carga o algoritmo híbrido obteve um desempenho próximo ao algoritmo similar que foi executado por 1 GPU.

Por fim, para continuidade deste trabalho, sugere-se uma adaptação de outros modelos matemáticos para que seja possível executá-los em ambientes híbridos multi-GPUs. Outro importante linha de pesquisa baseada no presente trabalho seria a obtenção de um ótimo balanceamento de carga entre os dispositivos de maneira automática, sem a interferência do programador, já que como foi citado anteriormente o correto balanceamento de carga entre os dispositivos tem influência direta na performance de um algoritmo híbrido.

Bibliografia

BORATTO, M. et al. On the evaluation of matrix polynomials in gpus. *Universidade Estadual de Santa Cruz*, Junho 2013.

BORATTO, M. et al. Modelagem computacional da espacialização de variáveis meteorológicas na região agrícola do vale do são francisco. *Universidade Federal do Vale do São Francisco*, 2012.

BORATTO, M. et al. Modelagem computacional da espacialização do relevo na região ágrícola do sub-médio do vale do rio são francisco. *Universidade do Estado da Bahia*, 2012.

CALDAS, J.; SENA, M. *Tutorial OpenMP C/C++*. [S.l.], 2008.

CHATAIN, L. Hybrid parallel programming - evaluation of openacc. *Universidade Federal do Rio Grande do Sul*, 2012.

DUTRA, E.; VARINI, A.; CANAL, A. Paralelização de aplicações na arquitetura cuda: Um estudo sobre vetores. *Centro Universitário Franciscano*, 2012.

FERRÃO, L.; PLOTZE, R. Computação distribuída: o melhor aproveitamento de recursos computacionais. *Centro Universitário Claretiano de Batatais*, 2012.

FILHO, J. Espacialização da temperatura para o polo de desenvolvimento petrolina/juazeiro, utilizando computação de alto desempenho. *Universidade Federal do Vale do São Francisco*, 2012.

GRAMMELSBACHER, A.; MEDRADO, J. Comparação de desempenho entre gpgpu e sistemas paralelos. *Universidade Anhembi Morumbi*, 2009.

INTEL. *Intel Hyper-Threading*. 2001. Disponível em: <http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology>. Acesso em: 01 Jun. 2014.

KIRK, D.; HWU, W. *Programando para processadores paralelos - Uma abordagem prática a GPU*. [S.l.: s.n.], 2011.

LASTOVETSKY, A.; ZHONG, Z.; RYCHKOV, V. Data partitioning on heterogeneous multicore and multi-gpu systems using functional performance models of data-parallel applications. IEEE, p. 191–199, 2012.

MORAIS, S. *Computação Paralela em cluster de GPU aplicado a problema da engenharia nuclear*. Tese (Doutorado) — Instituto de Engenharia Nuclear do Rio de janeiro, 2012.

NVIDIA. *CUDA Architecture*. 2009. Disponível em: <http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview>. Acesso em: 05 Out. 2013.

NVIDIA. *CUDA Parallel Computing Platform*. 2013. Disponível em: <http://www.nvidia.com.br/object/cuda_home_new_br>. Acesso em: 07 Out. 2013.

PEREIRA, L. *Simulação estocástica e métodos heurísticos paralelizados para a resolução do problema de roteamento de veículos capacitados com base na estratégia cluster first route second*. Dissertação (Mestrado) — Universidade Tecnológica Federal do Paraná, 2012.

PEREIRA, S.; BONGANHA, C.; GUIGUER, N. Conceitos e fundamentos da modelagem matemática para gerenciamento de recursos hídricos subterrâneos. *Centro Universitário de Araraquara*, 2007.

PINTO, V. *Ambientes de programação híbridos*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, Dezembro 2011.

RAFIKOV, M.; LIMEIRA, E. Modelagem matemática do controle biológico de pragas da cana-de-açúcar. *Universidade Federal do ABC*, v. 01, 2012.

RIBEIRO, N. *Desenvolvimento de programas paralelos para máquinas numa: conceitos e ferramentas*. Dissertação (Mestrado) — Universidade Católica do Rio Grande do Sul, 2009.

ROCHA, K.; FILHO, L. Introdução ao cuda utilizando métodos numéricos. *Centro Universitário Vila Velha*, 2010.

ROSA, D. Programação paralela híbrida em cpu e gpu: Uma alternativa por busca de desempenho. *Universidade Regional Integrada*, Novembro 2013.

SANTANA, R. et al. Computação paralela. *Universidade de São Paulo*, 1997.

SILVA, L. Modelo híbrido de programação paralela para uma aplicação de elasticidade linear baseada no método dos elementos finitos. *Universidade de Brasília*, 2006.

SODRÉ, U. Modelos matemáticos. *Universidade Estadual de Londrina*, 2007.

YANO, L. Avaliação e comparação de desempenho utilizando tecnologia cuda. *Universidade Estadual Paulista*, 2010.

ZARPELON, E. Equações polinomiais matriciais. *Universidade Federal de Santa Catarina*, 2004.