Luis Felipe Sant'Ana

Balanceamento de Carga Dinâmico em Aglomerados de GPUs

Santo André
2014

Luis Felipe Sant'Ana

Balanceamento de Carga Dinâmico em Aglomerados de GPUs

Dissertação Apresentada ao Centro de Matemática, Computação e Cognição da Universidade Federal do ABC, para a obtenção do Título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Raphael Y. de Camargo
Co-orientador: Dr. Daniel Cordeiro

Santo André
2014

Sant Ana, Luis F.
Balanceamento de Carga Dinâmico em Aglomerados de GPUs
51 páginas
Dissertação de Mestrado - Centro de Matemática, Computação e Cognição da Universidade Federal do ABC.

1. Programação Paralela
2. Balanceamento de Carga
3. GPGPU

I. Universidade Federal do ABC. Centro de Matemática, Computação e Cognição.

Comissão Julgadora:

Prof. Dr.
XX

Prof. Dr.
YY

Prof. Dr.
Raphael Yokoingawa de Camargo

Resumo

O uso de GPUs (Graphic Processor Unit) em aplicações científicas está cada vez mais difundido, com aplicações em áreas como física, química e bioinformática. Mesmo com o ganho de desempenho obtido com o uso de uma GPU, algumas aplicações ainda requerem elevados tempos de execução. Para esses problemas a utilização de *clusters* de GPUs surgem como uma possível solução. Mas é comum termos máquinas contendo GPUs com variadas capacidades e de diferentes gerações, resultando em *clusters* heterogêneos. Neste cenário, um dos pontos fundamentais é o balanceamento de carga entre as diferentes GPUs, com o objetivo de maximizar a utilização das GPUs e minimizar o tempo de execução da aplicação. Este projeto tem como objetivo o desenvolvimento de um algoritmo de balanceamento de cargas dinâmico para aglomerados de GPUs. Implementamos o algoritmo no *framework* StarPu, um arcabouço para o desenvolvimento de aplicações para aglomerados e comparamos seu desempenho com outros algoritmos de balanceamento de carga.

Palavras-chave: Programação Paralela, Balanceamento de Carga, GPGPU.

Abstract

The use of GPUs for scientific applications is becoming more widespread, with applications in fields such as physics, chemistry and bioinformatics. Even with the performance gain obtained by using a GPU, some applications require even higher execution times. For these problems the use of GPU clusters arise as a possible solution. But it is common to machines containing GPUs with different capabilities and different generations, resulting in heterogeneous clusters. In this scenario, one of the key points is the load balancing between different GPUs, with the goal of maximizing the use of GPUs and minimize the execution time of the application. This project aims to develop an algorithm for dynamic load balancing among heterogeneous GPUs. We implement the algorithm in framework StarPu, a framework to development of applications for clusters and compare their performance with other load balancing algorithms.

Keywords: Parallel Programming, Load Balancing, CUDA, GPGPU.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

1.1 Introdução ao balanceamento de carga

A computação paralela vem sendo amplamente utilizada para atender necessidades de aplicações que exigem muito poder computacional. A cada dia surgem propostas de novas máquinas e modelos de arquiteturas paralelas, com diferentes quantidades de processadores, topologias e redes de interconexão. O universo de arquiteturas existentes é grande e elas são normalmente agrupadas como: SIMD e MIMD. Máquinas SIMD possuem um único fluxo de instrução aplicados a múltiplos dados e normalmente são específicas para um tipo de aplicação. As MIMD possuem multiplos fluxos de instrução sendo executados em paralelo. Atualmente as máquinas MIMD são as mais utilizadas e podem ser divididas como: Multi Processadores Simétricos (SMP), Vetoriais, Massivamente paralelas (MPP), agregados de computadores *(Clusters)* e Grades computacionais *(Grids)*.

As máquinas SMP, Vetoriais e MPP normalmente fornecem ambientes para programação e uso eficiente, porém são arquiteturas proprietárias e com alto custo de montagem e manutenção. Já os clusters e os grids consistem basicamente de diversas unidades de processamento, que podem ser independentes, interconectadas por uma rede de comunicação. As principais vantagens nesse modelo são custos baixos e alta escalabilidade, isto é, a possibilidade de facilmente alterar, ao longo do tempo, a quantidade de unidades de processamento da arquitetura. No entanto, a computação é baseada em memória distribuída, ou seja, cada unidade de processamento apresenta a sua própria memória e necessita programação da troca de informações entre os processadores, o que pode ser uma tarefa muito complexa.

As GPUs vêm se mostrando uma excelente alternativa na área de computação de alto-desempenho para aplicações que exigem um alto grau de paralelismo [Owens et al., 2008]. GPUs modernas podem possuir milhares de núcleos, cada um deles bastante simples, mas que

utilizados em paralelo geram um alto poder computacional [CUDA Development Core Team, 2012].

Muitas aplicações já utilizam o poder de processamento das GPUs, como mecânica dos fluidos [Riegel et al., 2009], visualização científica [Camargo et al., 2011], aprendizado de máquina [Li et al., 2011], bioinformática [Borelli et al., 2012] e redes neurais [de Camargo, 2011]. Devido ao relativo baixo custo de placas gráficas contendo GPUs, sua utilização é uma ótima alternativa para pesquisadores pertencentes a instituições com poucos recursos financeiros e com grande necessidade de recursos computacionais.

Em diversos casos, o uso de GPUs ocasionam ganhos superiores a 100 vezes, quando comparado às CPUs tradicionais [Vouzis and Sahinidis, 2011]. No entanto, para muitos problemas, o uso de uma única GPU ainda limita o tamanho e complexidade do problema que pode ser resolvido. Neste caso, é possível utilizar múltiplas GPUs localizados em diferentes computadores, como em *clusters* de GPUs [de Camargo, 2012; Fan et al., 2004].

Os *clusters* de GPUs são normalmente homogêneos, no qual todas as máquinas apresentam a mesma configuração de hardware. O desenvolvimento de um aplicativo para executar em um *cluster* de GPU é um desafio, porque o programador tem de empregar uma biblioteca de programação paralela, como MPI, além de usar a plataforma CUDA. Existem alguns esforços para oferecer alguns modelos de programação [Wang et al., 2011a,b] e bibliotecas [Kim et al., 2012; Miyoshi et al., 2012] para o desenvolvimento de aplicações para *clusters* de GPUs. No entanto, a forma mais utilizada para programação para *clusters* de GPUs é combinando CUDA e MPI.

Já no caso de *clusters* heterogêneos é necessário distribuir a carga computacional entre as GPUs. Desenvolver um mecanismo que funcione de modo eficiente para qualquer aplicação é difícil, mas pode-se restringir o mecanismo para alguns tipos específicos de aplicações. Por exemplo, podemos considerar aplicações que possuem um conjunto de dados que podem ser particionados entre as GPUs, utilizando o método de decomposição de domínio [Mandel, 1993]. Neste caso, cada GPU fica responsável por uma parte dos dados da aplicação. Muitas aplicações científicas se encaixam neste grupo, incluindo bioinformática [Borelli et al., 2012], redes neurais [de Camargo, 2011], química [Anastas and Warner, 2000], física [Delpech et al., 2009] e ciência dos materiais [da Silva et al., 2007].

Para este tipo de aplicação, a distribuição da carga em *clusters* homogêneos de GPUs é frequentemente obtida com uma divisão simples dos dados de entre as GPUs. No entanto, com grupos heterogêneos de GPUs, esta situação é mais difícil. Existem atualmente várias arquiteturas existentes, como a Tesla, Fermi e Kepler. Essas arquiteturas apresentam diferentes organizações de núcleos e multiprocessadores, quantidades de memória compartilhada por núcleo e velocidades de memória. Devido a essas diferenças, uma divisão da carga com base

em simples heurísticas, tal como o número de núcleos na GPU não é eficaz. Na verdade, isso pode levar a um maior tempo de execução [de Camargo, 2012] comparado a simples ideia de dividir a carga igualmente entre as GPUs heterogêneas. Em um cenário mais geral, uma aplicação pode ter vários *kernels*, cada um com dependências não-lineares entre o tempo de execução e o tamanho de entrada.

Apesar da importância do tema, poucos trabalhos apresentam algoritmos de balanceamento de carga em sistemas heterogêneos baseados em GPUs. Os existentes são *frameworks* que requerem a reescrita da aplicação para que seja possível a execução nestes *clusters*.

[Acosta et al., 2012] propuseram um algoritmo de balanceamento de carga dinâmico que interativamente tenta encontrar uma boa distribuição do trabalho entre GPUs durante o execução da aplicação. Uma biblioteca se concentra nas diferenças de tempos de códigos paralelos, com base em um esquema iterativo. É um esquema descentralizado em que as sincronizações são feitas a cada iteração para determinar se há uma necessidade de reequilibrar a carga. Cada processador executa uma redistribuição da carga de trabalho de acordo com o tamanho da tarefa atribuída e as capacidades do processador atribuído. O balanceamento de carga é obtido através da comparação do tempo de execução da tarefa atual para cada processador com a redistribuição da tarefa subseqüente, se o tempo gasto para precessar a tarefa atingir um limite a redistribuição de carga é feito.

Um algoritmo estático que determina a distribuição antes de iniciar a execução da aplicação, utilizando os perfis estáticos de execuções anteriores, também foi proposta [de Camargo, 2012]. O algoritmo foi avaliado usando uma simulação de rede neural. O algoritmo encontra a distribuição de dados que minimiza o tempo de execução da aplicação, com base em perfis de execuções anteriores, assegurando que todas as GPUs para passar mesma quantidade de tempo que executa o processamento.

O (HDSS) [Belviranli et al., 2013] fornece um algoritmo de balanceamento de carga dinâmico, dividido em duas fases. O primeiro é a fase adaptativo, que determina pesos que refletem a velocidade de cada processador. Esses pesos são determinados com base em ajustes de curvas logarítmicas em gráficos de velocidade de processamento. Estas ponderações são utilizadas durante a fase de acabamento, onde se divide nas iterações restantes entre as GPUs com base nos pesos relativos de cada GPU.

1.2 Objetivos e Contribuições

O presente trabalho tem como objetivo o desenvolvimento de um algoritmo de balanceamento de carga para aglomerados de GPUs heterogêneos, ou seja, todo o planejamento e respaldo para a criação de um algoritmo eficiente que minimize o tempo total de processamento.

O algoritmo será implementando usando o *framework* denominado StarPu, que é voltado para aglomerados. A estrutura do arcabouço facilitará a implementação do algoritmo.

Por fim, serão feitos testes de comparação de desempenho do algoritmo desenvolvido com os atuais algoritmos para a execução das mesmas tarefas.

1.3 Organização da Dissertação

Esta dissertação se divide em seis capítulos. O Capítulo 2 apresenta a fundamentação utilizada neste trabalho. Neste capítulo são discutidos os conceitos, classes e técnicas de escalonamento. O Capítulo 3 descreve os trabalhos relacionados ao que tange o escalonamento de tarefas e mais especificadamente ao balanceamento de carga em aglomerados heterogêneos. O Capítulo 4, apresenta o detalhadamento do algoritmo proposto. O capitulo 5, mostra detalhes de implementação, e os problemas testados. O Capítulo 6 apresenta os resultados e conclusões desta dissertação, assim como sugestões de trabalhos futuros. Por fim o capítulo 7 apresenta o plano de trabalho, das atividades já desenvolvidas e as próximas atividades a serem desenvolvidas.

Capítulo 2

Conceitos Computacionais

2.1 Escalonamento

Escalonamento é um processo de tomada de decisão que é usado como base em muitas indústrias manufatureiras e serviços industriais. Ele lida com a alocação de recursos para tarefas a fim de fornecer períodos de tempo a cada tarefa e o fim é otimizar um ou mais objetivos [Pinedo, 2012].

Encontrar um escalonamento significa encontrar, para cada tarefa, uma alocação de um ou mais intervalos de tempo, em uma ou mais máquinas. O problema de escalonamento correspondente é encontrar um escalonamento que satisfaça um determinado conjunto de restrições.

Em um ambiente genérico de fabricação, o papel do escalonamento das tarefas é destacado nas ordens de serviço que são lançadas na configuração da fabricação, em forma de tarefas com datas de entrega associadas. Essas tarefas frequentemente devem ser processadas em máquinas em uma dada ordem ou sequência. Os processamentos das tarefas podem atrasar, se certas máquinas estiverem ocupadas. Eventos imprevistos no chão-de-fábrica, tais como quebra de máquinas ou tempos de processamento maiores que os previstos, também devem ser levados em consideração, desde que esses eventos venham a impactar diretamente o escalonamento das tarefas. Neste ambiente, o desenvolvimento de um escalonador de tarefas detalhado ajuda a manter a eficiência e o controle das operações.

O chão-de-fábrica não é a única parte da organização que impacta o processo de escalonamento. O escalonador também é afetado pelo processo de planejamento da produção que lida com o planejamento a médio e a longo prazos para toda a organização. Esse processo tenta otimizar toda linha de produtos da empresa e a alocação de recursos baseados em seus níveis de estoque, previsões de demanda e necessidades de recursos. As decisões tomadas neste nível mais alto de planejamento podem impactar o processo de escalonamento diretamente.

Um exemplo de escalonamento é na indústria de semicondutores no qual o objetivo é maximizar o tempo de utilização dos equipamentos e minimizar o tempo ocioso e de configuração dos mesmos [Pinedo, 2012].

Na computação o escalonamento está associado a utilização dos recursos da máquina como por exemplo a CPU. Existem vários algortimos de escalonamento entre eles [Tanenbaum and Woodhull, 2006]:

- FIFO *(First In, First Out)*. A primeira tarefa a chegar será a primeira a ser executada.
- SJF *(Shortest Job First)*. A tarefa mais curta tem ganhará o recurso, e uma fila em ordem de tamanhos da menor para a maior é feita atrás da menor.
- Round Robin. Todas as tarefas recebem um tempo de processamento, esse tempo é chamado de quantum.

2.1.1 Notação dos problemas de escalonamento

Nesta seção são apresentados uma breve introdução a notação associada aos problemas de escalonamento:

- Tempo de execução (p_{ij}) tempo necessário para a execução da tarefa t_i na máquina m_j. Denotado apenas por p_i se todas as máquinas forem idênticas
- Data de disponibilidade (r_i) instante em que a tarefa t_i se torna disponível para ser executada
- Prazo (d_i) instante de tempo no qual a tarefa t_i deve estar pronta
- Peso (w_i) normalmente indica um fator de prioridade da tarefa t_i

A notação utilizada para representar os problemas de escalonamento, é representada pela tripla:

$\alpha|\beta|\gamma$

α que descreve os recursos disponíveis

β que descreve as tarefas a serem executadas

γ que descreve o critério de otimização

α pode ser representado por:

- 1 uma única máquina

- P ou P_m m máquinas paralelas idênticas cada tarefa t_i pode ser processada em qualquer uma das m máquinas por p_i unidades de tempo, se uma tarefa só puder ser executada em um subconjunto das máquinas, a restrição será indicada no campo
- $P\infty$ ou $\overline{P}$ é o número ilimitado de máquinas idênticas

β pode ser representado por:

- Q_m máquinas paralelas uniformes no qual m máquinas operam com velocidades iguais (a máquina m_j opera com velocidade v_j) $p_{ij} = p_i = v_j$
- R_m máquinas paralelas não-relacionadas, m máquinas diferentes em paralelo
- $p_{ij} = p_i = v_{ij}$, onde v_{ij} é a velocidade da tarefa t_i na máquina m_j

2.1.2 Classes de Escalonamento

Problemas de escalonamento podem ser divididos em classes, tais como escalonamento sem atrasos, escalonamento ativo e escalonamento semi-ativo. Essas classes agrupam os modelos de escalonamento e facilitam a forma de analisar o problema.

Escalonamento sem atrasos

Um escalonamento válido é dito sem atraso, se nenhuma máquina fica inativa, ou seja, ociosa, quando existem tarefas disponíveis para serem executadas.

Escalonamento ativo

Os algorirmos de escalonamento podem ser preemptivos e não preemptivos. Preemptivo significa que o algoritmo permite que um processo seja interrompido durante a sua execução, para posteriormente ser retomado.

Um escalonamento é ativo se não for possível, apenas trocando a ordem das tarefas/operações em uma máquina, construir um outro escalonamento onde ao menos uma tarefa termine mais cedo e nenhuma outra tarefa seja atrasada.

Escalonamento semi-ativo

Um escalonamento é dito semi-ativo se nenhuma tarefa/operação pode terminar mais cedo sem que a ordem do processamento das tarefas de alguma máquina seja mudada.

2.1.3 Classificação dos problemas de escalonamento

Os problemas de programação de operações em máquinas vêm sendo caracterizados por diversos autores em diferentes formas, dentre eles: Baker, 1974; Blazewicz et al., 1996; Conway et al., 1967; French, 1982; Graves, 1981 e Pinedo, 2008.

Em situações de escalonar tarefas nas máquinas disponíveis surgem problemas complexos. Pois, as restrições tecnológicas e a medida de desempenho do escalonador devem ser especificadas. As restrições tecnológicas são determinadas principalmente pelo fluxo das tarefas nas máquinas.

Neste contexto, Maccarthy e Liu (1993) classificam os problemas de programação de operações da seguinte forma:

- **Máquina única** - existe somente uma única máquina disponível para a execução das tarefas;
- **Flow shop** - em que todas as tarefas possuem o mesmo fluxo de processamento em todas as máquinas;
- **Job shop** - em que todas as tarefas possuem um roteiro específico de processamento, determinado para cada tarefa;
- **Open shop** - em que não existem roteiros de processamento preestabelecidos para as tarefas;

O problema do sequenciamento em uma única máquina quase sempre parte de um problema de programação complexo. Segundo Pinedo (2008), os problemas do sequenciamento em uma única máquina muitas vezes têm propriedades que os de em máquinas em paralelo ou em série não possuem. Os resultados que podem ser obtidos para os problemas do sequenciamento em uma única máquina não só fornecem o conhecimento para o ambiente de uma única máquina, como também fornecem base para heurísticas aplicáveis a ambientes mais complexos.

Na prática, os problemas de escalonamento em ambientes mais complicados são frequentemente decompostos em subproblemas de uma única máquina.

Por exemplo, um ambiente complexo, com um único gargalo, pode dar origem a um modelo de sequenciamento em uma única máquina. Dessa forma, o problema do sequenciamento em uma única máquina é importante por diversas razões, dentre elas pode-se citar:

- O processo de aprendizado, já que o problema de escalonamento em uma única máquina pode ilustrar uma variedade de tópicos de escalonamento tornando modelos tratáveis.

Esse problema fornece um contexto para que se investigue muitas medidas de desempenho e técnicas de solução. Além disso, é uma base para o entendimento de conceitos de escalonamento úteis para modelar sistemas mais complexos.

- Para entender completamente o comportamento de um sistema complexo, é vital entender como funciona cada um de seus componentes e muito frequentemente o problema de uma única máquina aparece como componente elementar em um problema de escalonamento maior.

- Algumas vezes é possível resolver o problema de escalonamento em uma única máquina independentemente e então incorporar o resultado em um problema maior. Por exemplo, em um processo com múltiplas operações, frequentemente existe uma operação gargalo e o tratamento dessa operação gargalo, vista como uma análise de um problema de uma única máquina, determina as propriedades de todo o escalonamento.

2.1.4 Modelo de escalonamento para uma única máquina

Ao lidar com os atributos para o modelo de uma única máquina, é útil distinguir entre informações conhecidas previamente e informações que são geradas como resultados de decisões de escalonamento. A informação que é conhecida previamente serve como parâmetro de entrada para função de escalonamento e é usualmente conveniente usar letras minúsculas para denotar esse tipo de informação, são elas $\alpha|\beta|\gamma$.

Os prazos podem não ser pertinentes em certos problemas, mas estabelecer os prazos (deadlines) é um problema comum na indústria e o problema básico pode auxiliar na determinação do prazo de entrega. É conveniente usar letras maiúsculas para denotar as informações resultantes do escalonamento.

- Tempo de Conclusão (C_j). O tempo no qual o processamento do trabalho j é terminado.

  Os critérios quantitativos para escolher uma sequência são geralmente funções dos tempos de conclusão. Duas funções importantes são:

- Tempo de Fluxo (Flowtime) – F_j – Tempo total que o trabalho j fica no sistema:

  $F_j = C_j – r_j$

- Defasagem (Lateness) – L_j – Diferença entre a data de conclusão e o prazo do trabalho j, podendo assumir valores positivos ou negativos

  É importante notar que a defasagem terá valor negativo quando uma tarefa é finalizada antecipadamente. Defasagens negativas podem representar serviços melhores do que

solicitados, enquanto atrasos positivos representam, quase sempre, serviços piores do que requisitados. Em muitas situações, penalidades distintas e outros custos serão associados para defasagens positivas e para defasagens negativas. Dessa forma, tem-se as definições de Atraso (Tardiness) e Antecipação (Earliness):

- Atraso (Tardiness) – T_j – é o quanto o trabalho j atrasou em relação ao seu prazo, caso contrário será considerado zero:

  $T_j = max\{0, L_j\}$

- Antecipação (Earliness) – E_j – é o quanto o trabalho j é antecipado em relação ao prazo, caso contrário será considerado zero:

  $E_j = max\{d_j – C_j, 0\}$

- Makespan – C_{max} – é definido como o maior dentre as datas de conclusão $(C_1, ..., C_n)$, ou seja, ao tempo de conclusão do último trabalho a sair do sistema $(C[n])$. Minimizar o makespan usualmente implica em uma boa utilização dos recursos.

2.2 Balancemento de Carga

O problema de balanceamento de carga é estudado por pesquisadores da área de Teoria de Escalonamento há mais de 50 anos. Em 1966, [Graham, 1966] estudou o que chamou de ”anomalias” na execução de tarefas em processadores de velocidades diferentes. Ele mostrou, por exemplo, que a adição de novos processadores ou a troca de alguns processadores por outros de maior velocidade podem propiciar o desbalanceamento de carga e, portanto, não necessariamente implicam em um ganho no desempenho. Logo ficou clara a necessidade de algoritmos mais sofisticados para lidar com esse problema.

O balanceamento de carga é uma técnica aplicada para distribuir a carga de trabalho entre dois ou mais servidores, enlaces de rede, CPU, ou outros recursos; a fim de otimizar a utilização destes recursos, maximizar o desempenho e evitar sobrecarga. Em geral o balanceamento de carga consiste em três fases [Devarakonda and Iyer, 1989]. A primeira, consiste na coleta de informações. A segunda, busca determinar qual seria a distribuição ótima para o estado em que o sistema se encontra. E por fim a terceira fase, que a ação de balanceamento é executada. O balanceador de carga pode ter uma das seguintes classificações:

- *Estático:* A regra de balanceamento é definida uma única vez, baseada em informações estáticas do sistema ou aplicações.

- *Dinâmico:* A regra de balanceamento é modificada em resposta ao estado atual do sistema ou das aplicações. O estado do sistema será constantemente atualizado e as decisões tomadas são baseadas nas informações atuais e possivelmente, dos estados anteriores do sistema.

Ao balancear a carga, tenta-se evitar que no sistema, existam simultaneamente máquinas com recursos subutilizados e máquinas com recursos super utilizados. Na maioria dos casos, realizar balanceamento de carga ótimo é impraticável, devido a complexidade computacional para resolver o problema. O problema de balanceamento de carga é similar a problemas de alta complexidade computacional [Singh et al., 2008], NP-Completo que não podem ser resolvido de maneira exata em tempo polinomial. Para contornar esta limitação, heurísticas ou algoritmos de aproximação, são usados para apresentarem soluções aproximadas do balanceamento de carga ótimo.

Distribuição de carga estática, também conhecida como planejamento determinístico, atribui um determinado trabalho a processador fixado. Toda vez que o sistema for reiniciado, a mesma tarefa no processador de ligação (atribuição de uma tarefa ao mesmo processador) é usada sem considerar as mudanças que podem ocorrer durante a vida útil do sistema. Além disso, a distribuição de carga estática pode também caracterizar a estratégia utilizada durante a execução, no sentido de que ele pode não resultar na mesma atribuição de tarefas no processador, mas atribui os postos de trabalho recém-chegados de uma forma sequencial ou fixa. Por exemplo, utilizando uma estratégia estática simples, os trabalhos podem ser atribuídos aos nós de uma maneira round-robin de modo que cada processador execute aproximadamente o mesmo número de tarefas.

Balanceamento de carga dinâmico leva em conta que os parâmetros do sistema não podem ser conhecidos com antecedência e, por conseguinte, utilizando um esquema fixo ou estático irão eventualmente produzir resultados ruins. A estratégia dinâmica geralmente é executado várias vezes e poderá realocar um trabalho previamente escalonado para um novo nó com base na dinâmica atual do ambiente de sistema.

Os problemas tratados neste trabalho se restringem aos problemas que não apresentam dependência de dados entre si. A técnica ultilizada é chamada de decomposição de domínio. Que consiste em dividir um conjunto inicial em subconjuntos. O problema restringe a determinar a quantidade de dados que devem ser enviados a cada processador de forma a minimizar o tempo total de processamento em um ambiente de processadores heterogêneos.

Balanceamento de carga Distribuído e Centralizado

Essa divisão geralmente cai sob o esquema de balanceamento de carga dinâmico, onde uma questão natural surge sobre o local onde a decisão é tomada. Políticas centralizadas armazenam informações globais em um local central e usa essas informações para fazer o escaonamento de decisões tomadas pelos que utilizam os recursos de armazenamento de um ou mais processadores. Este esquema é o mais adequado para sistemas em que as informações de estado de um processador individual podem ser facilmente coletados por uma estação central a baixo custo, e novos postos de trabalho que chegam a este local centralizado serão depois encaminhados para nós subseqüentes. A principal desvantagem deste sistema é que ele tem um ponto único de falha.

No balanceamento de carga distribuído, as informações de estado são distribuídas entre os nós que são responsáveis na gestão dos seus próprios recursos ou distribuiem as tarefas que residem em suas filas a outros processadores. Em alguns casos, o sistema permite que os processadores ociosos atribuam tarefas a si mesmos em tempo de execução, acessando uma fila global compartilhada. Note-se que as falhas ocorrem em um determinado nó irão permanecer localizado e não pode prejudicar o funcionamento global do sistema.

Em um escalonamento de balanceamento de carga local, cada processador pesquisa outros processadores em sua vizinhança e usa esta informação local para decidir sobre a transferência de carga. Esta vizinhança é geralmente indicada como o espaço de migração. O principal objetivo é minimizar a comunicação remota, bem como de forma eficiente equilibrar a carga sobre os processadores. No entanto, em um esquema de balanceamento global, informação global ou parte da informação do sistema é utilizado para iniciar o balanceamento de carga. Este sistema exige uma quantidade considerável de informações a serem trocadas no sistema o que pode afetar sua capacidade de expansão.

No âmbito da escalonamento distribuído global dinâmico, dois mecanismos podem ser distinguidos, envolvendo o nível de cooperação entre as diferentes partes do sistema. No esquema não cooperativo ou autônomos, cada nó tem autonomia sobre o seu próprio escalonamento de recursos. Isto é, as decisões são tomadas independentemente do resto do sistema e, portanto, o nó pode migrar ou afetar funções baseadas no desempenho local. Por outro lado, na programação cooperativa, os processos trabalham em conjunto para um balanço global de todo o sistema. Decisões de escalonamento são feitas após considerar seus efeitos sobre algumas medidas efetivas globais (por exemplo, o tempo global de conclusão).

Esquemas adaptativos e não-adaptativos são parte das políticas de balanceamento de carga dinâmico. Em um esquema adaptativo, as decisões escalonadas levam em consideração o passado e o desempenho atual do sistema e são afetados pelas decisões anteriores ou alterações no

ambiente. Se um (ou mais parâmetros) não se correlacionam com o desempenho do programa, que é ponderado menos próxima vez. No regime não-adaptativo, os parâmetros utilizados no escalonamento continuam os mesmos, independente do comportamento passado.

Confusões podem surgir entre a distinção entre escalonamento dinâmico e escalonamento adaptativo. Uma solução dinâmica leva em conta informações ambientais na sua decisão, uma solução adaptativa (que também é dinâmica) leva em conta estímulos ambientais em consideração para modificar a própria política de escalonamento.

2.3 GPUs

As GPUs vêm se mostrando uma excelente alternativa na área de computação de alto-desempenho para aplicações que exigem um alto grau de paralelismo [Owens et al., 2008]. GPUs modernas possuem dezenas de núcleos, cada um deles bastante simples, mas que utilizados em paralelo geram um alto poder computacional [CUDA Development Core Team, 2012].

Muitas aplicações já utilizam o poder de processamento das GPUs, exemplos são mecânica dos fluidos [Riegel et al., 2009], visualização científica [Camargo et al., 2011], aprendizado de máquina [Li et al., 2011], entre outras. Devido ao relativo baixo custo de placas gráficas contendo GPUs, sua utilização é uma ótima alternativa para pesquisadores pertencentes a instituições com poucos recursos financeiros e com grande necessidade de recursos computacionais.

2.3.1 CUDA

Compute Unified Device Architecture (CUDA) [CUDA Development Core Team, 2012] é uma arquitetura de computação paralela desenvolvida pela Nvidia para processamento gráfico. Com esta tecnologia é possível desenvolver não só aplicações convencionais de processamento gráfico, mas é possivel fazer programas de propósito geral, esta abordagem de se desenvolver programas de propósito geral em GPU é denominado GPGPU (*General-Purpose Computing on Graphics Processor Units*) [Merrill and Grimshaw, 2010].

Através de linguagens de programação amplamente utilizadas na indústria é possível desenvolver variadas aplicações que se beneficiam do alto poder de processamento das placas gráficas. A principal linguagem utilizada para o desenvolvimento em CUDA é a linguagem de programação C, que mundialmente é a mais amplamente utilizada. O principal mérito do CUDA é a facilidade que se tem para acessar todos recursos da placa gráfica.

CUDA foi lançado pela NVIDIA em novembro de 2006 [CUDA Development Core Team, 2012]. Sua proposta foi criar uma arquitetura de computação paralela de propósito geral. A

justificativa para isso foi aproveitar a plataforma de computação paralela das placas gráficas para resolver diversos algoritmos computacionais eficientemente. O aplicativo produzido sobre a API CUDA pode ser um algoritmo em linguagem C, OpenCL, Fortran, C++ e DX11, não havendo limitações para o suporte de novas linguagens.

Figura 2.1: Linguagens Suportadas(adaptado de [CUDA Development Core Team, 2012])

A arquitetura NVIDIA CUDA se baseia tanto em componentes de hardware como de software [CUDA Development Core Team, 2012]. A parte que entendemos por software é a executada na CPU, formada por um algoritmo seqüencial escrito comumente na linguagem C ou alguma outra que é suportada por CUDA. A parte do hardware é formada pelo código compilado por CUDA para torná-lo um kernel. *Kernel* são blocos que especificam parte do algoritmo em que se deseja a paralelização. Ele também pode ser tratado como um código que pode ser utilizado pela GPU para chamar outros kernels nela ou em outra GPU.

A GPU contém milhares de threads. CUDA faz a formatação do código de uma maneira que threads sejam alocadas paralelamente e o usuário usufrua dessa característica, podendo fazer chamadas de kernel que serão independentes entre si, o que permite a GPU rodar vários algoritmos.

Podemos ver na figura 2.3 a diferença de potencial que uma GPU, com seu paralelismo, possui em relação a um processador (CPU):

Na figura 2.2 gráfico temos informações sobre o pico de operações por ponto flutuante. A GPU alcança a casa de TFlops/s, e a CPU apresenta um desempenho baixo mesmo comparado a arquiteturas de GPUs anteriores que não são compatíveis com CUDA.

A GPU é voltada para computação intensiva, com alta paralelização, que são características do problema de renderização gráfica. Além disso, sua arquitetura é especializada no processamento deste tipo de dados e não no controle de fluxo. Podemos visualizar essa diferença

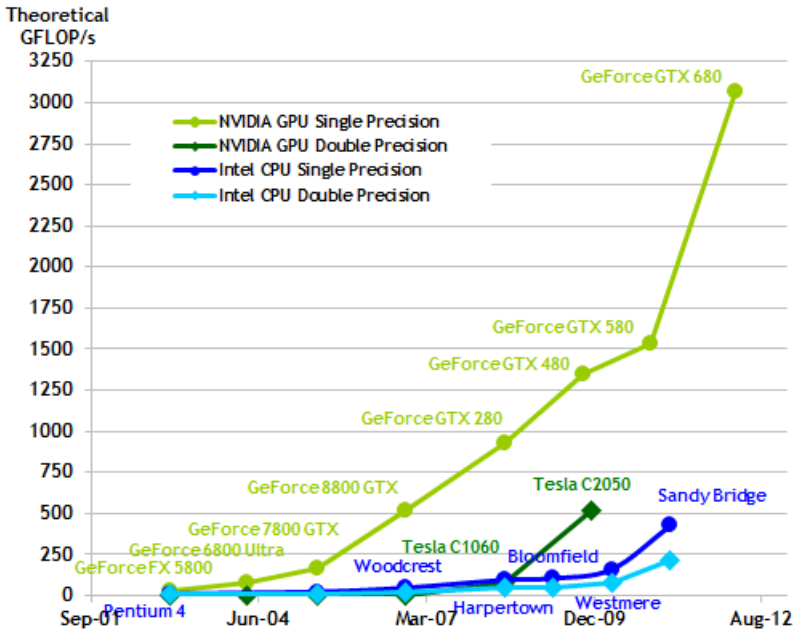

Figura 2.2: Operações de Ponto Flutuante por segundo para CPU e GPU(adaptado de [CUDA Development Core Team, 2012])

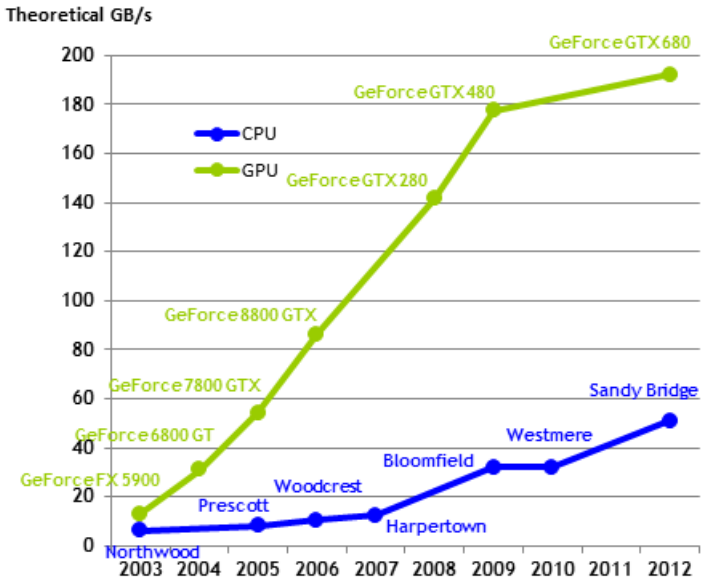

Figura 2.3: Tamanho de banda de memória para CPU e GPU(adaptado de [CUDA Development Core Team, 2012])

quando observamos a figura 2.4.

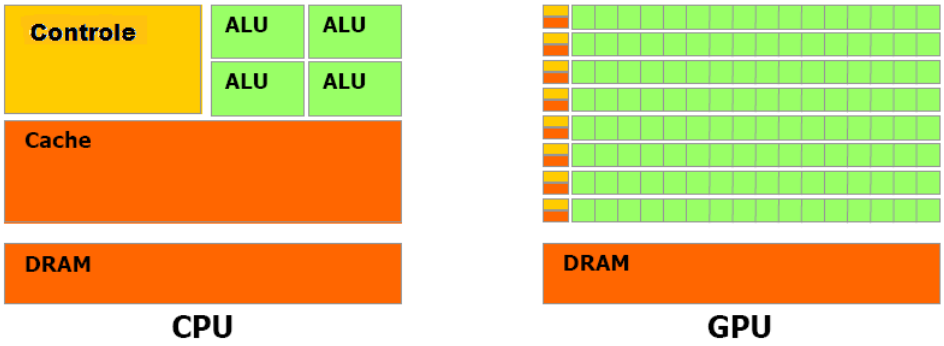

Figura 2.4: Quantidade de transistores para processamento de dados(adaptado de [CUDA Development Core Team, 2012])

A quantidade de transistores utilizados para fazer processamento (ALU - *Arithmetic Logic Unit*) é bem maior na GPU que na CPU. A arquitetura da CPU ainda está dividida em enormes blocos que contêm um componente de controle de fluxo e um de memória cache. No caso da GPU, o mesmo programa é executado em muitos elementos em paralelo. Dessa maneira, não possui um sofisticado componente de controle de fluxo, porém, oferece uma grande intensidade de computação aritmética. Além disso, a cache se torna pequena na GPU diminuindo a latência para execução de cada grande bloco de ALU's, diferente da CPU que contém uma cache enorme, obtendo uma latência alta para acesso. Possuindo uma unidade de controle, um componente de cache e um conjunto enorme de ALU's, é possível obter a estrutura de vários grupos de cores. Podemos assim certificar que a GPU é um manycore, apresentando de 32 até 128 cores dependendo do modelo. Essa subdivisão de cores na GPU é chamada de *Stream Processing*.

Um ponto importante a ser lembrado é que, para a execução na GPU, o algoritmo paralelizado não pode ter muita dependência de dados em cada passo de computação, pois a GPU não faz paralelização de tarefas e sim de processamento de dados. Caso essa otimização não seja feita no algoritmo, a GPU tem seu desempenho bastante afetado.

Com relação à hierarquia de compilação para o CUDA, a figura 2.5 ilustra a arquitetura da sua pilha de software. Essa pilha de software mostra a hierarquia de execução de CUDA. A API de CUDA fornece suporte a diversas funções matemáticas, bibliotecas, suporte ao runtime e ao driver.

O CUDA *runtime* [CUDA Development Core Team, 2012] é a camada de alto nível de programação, enquanto a camada do driver é a camada baixa para manipulação de dados. O CUDA driver gerencia e otimiza os recursos relacionados diretamente à GPU. CUDA é

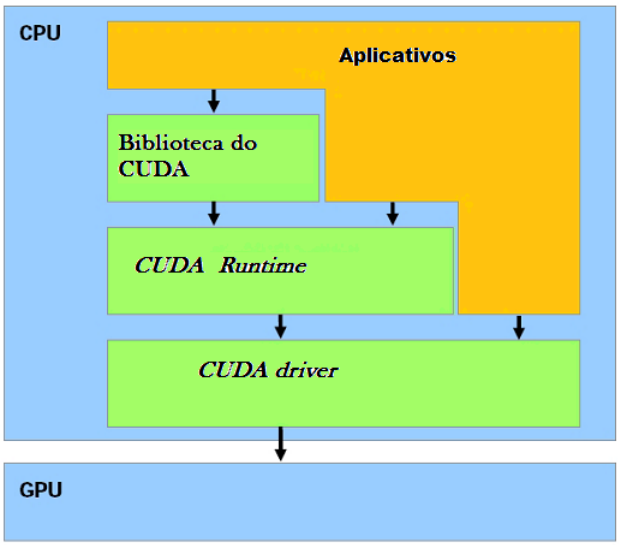

Figura 2.5: Pilha de Software(adaptado de [CUDA Development Core Team, 2012])

baseada em programação paralela, permitindo milhares de threads executando uma mesma tarefa. Neste caso, a GPU funciona como um co-processador da CPU, a qual chamamos de *HOST* e a GPU de *DEVICE*.

CUDA usa como base a linguagem C [CUDA Development Core Team, 2012], que permite uma curva rápida de aprendizado. Nela deve-se criar funções desejadas que CUDA otimiza e paraleliza na GPU. Essas funções são chamadas de kernel. A seguir, observamos um exemplo de um código simples, em seguida, uma explanação sobre cada aspecto inerente ao entendimento seqüencial desse código.

```
__global__ void matAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}


int main()
{
    // Kernel invocation
```

```
    dim3 dimBlock(N, N);
    matAdd<<<1, dimBlock>>>(A, B, C);
}
```

Nesse código temos uma função matAdd que apresenta uma tag *__global__* antes da definição da função. Essa tag define ao compilador que esse bloco será paralelizado na GPU. Esta função recebe três matrizes de tamanho N por N, onde cada thread nos eixos x e y serão responsáveis por somarem as matrizes A e B, sendo atribuido o resultado da soma na matriz C. Caso fosse desenvolvido um código similar no modo convencional, teria que aninhar dois laços de forma a percorrer todos os indices das matrizes. É claro, a diferença na forma de implementar e tratar o problema.

A função *main* faz a invocação do método. Ele se localiza no HOST e o código é executado na GPU (DEVICE). Em um computador podemos encontrar mais que uma GPU, neste caso elas serão chamadas de DEVICE 0, DEVICE 1, etc. O kernel é acionado colocando uma *tag* de configuração antes da declaração dos parâmetros, "<<<>>>". Há três parâmetros possíveis na configuração: a configuração do tamanho das *grids* (A), a configuração do tamanho dos blocos (B) e a quantidade de memória compartilhada que se deseja utilizar no algoritmo (Ns). Isso gera uma configuração genérica "$<<< A,B,Ns >>>$".

No main, antes da invocação, declaramos um tipo de variável definida pelo CUDA chamada "dim3". A variável criada dimBlock terá três dimensões e cada uma dessas dimensões representará a dimensão de um bloco. No exemplo, teremos um bloco com tamanho N para dimensão x, tamanho N para dimensão y e a dimensão z que foi omitida tem tamanho 1. Desta forma, seguindo a lógica de desenvolvimento da aplicação é possível construir algoritmos paralelizados na GPU.

A arquitetura CUDA é baseada em *arrays* escaláveis de multithreads SMs, ou thread Batching. Ele é um *batch* de threads que o kernel executa e organiza em grids de threads block. Uma thread block é um batch de threads que cooperam entre si para compartilhar eficientemente as informações que serão processadas. Essa eficiência vai desde o compartilhamento (cópia da DRAM GPU para a memória compartilhada e vice-versa) dessa informação através da memória compartilhada, até a sincronização de execução para o acesso a essa memória.

Cada thread é identificada através de ID única, chamada thread ID. Há um número limitado de threads por bloco. Porém, um bloco de mesma dimensão e tamanho pode ser executado por um mesmo kernel fazendo um grid de blocos, aumentando ainda mais a quantidade de threads sendo executados em paralelo. Vale salientar que as threads de blocos diferentes não podem se comunicar. Cada bloco é identificado por um block ID, e tem sua ID única em relação ao grid.

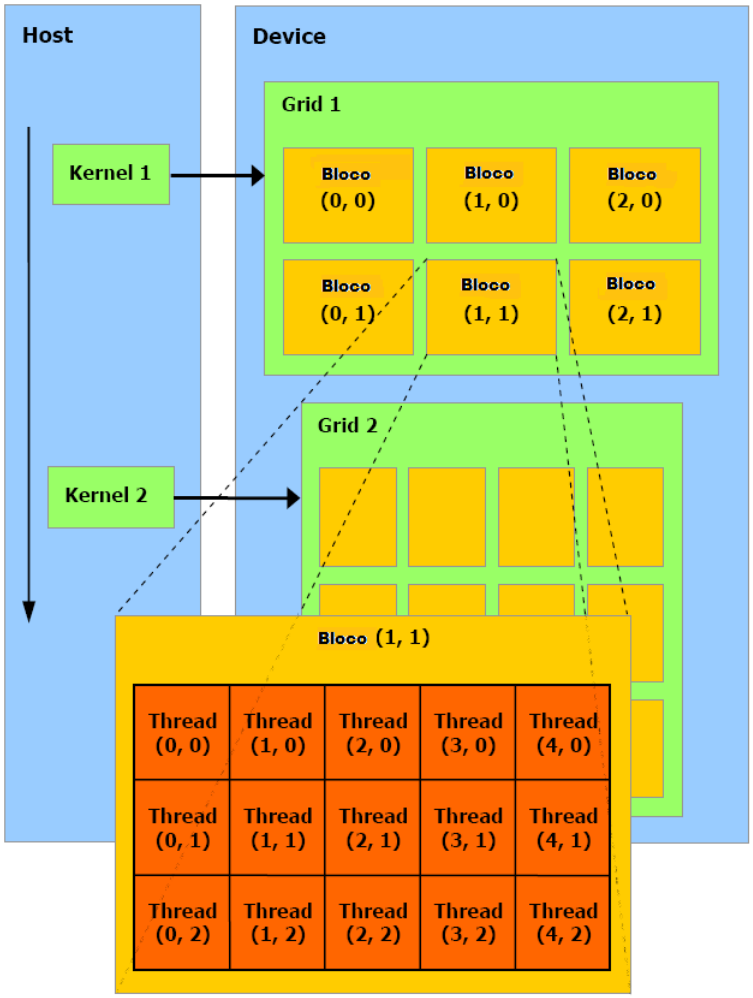

Figura 2.6: Batching de Threads(adaptado de [CUDA Development Core Team, 2012])

Um multiprocessador é um nome formalizado para threads. Ele consiste de oito cores de processadores escalares, duas unidades de funções especiais transcendentais, uma unidade de controle de instruções multithread e uma memória compartilhada on-chip.

O multiprocessador cria, gerencia, e executa threads concorrentes com custo nulo de hardware no âmbito de gerenciamento de processos. Para gerenciar centenas de threads rodando em diferentes programas, o multiprocessador nas últimas versões de CUDA emprega o SIMT. O multiprocessador mapeia cada thread a um core de processador escalar, que executa independentemente suas próprias instruções e tem seu próprio estado de registradores.

Esse multiprocessador SIMT cria, gerencia, escalona e executa grupos de 32 threads para-

lelas chamadas de warps. Threads individuais que compõem o SIMT *warp* podem começar sua execução juntas no mesmo programa, mas estão livres para ramificarem e executarem de forma autônoma. Contudo, se alguma thread ultrapassar a dependência condicional da ramificação, ela será desabilitada.

A arquitetura SIMD(*Single Instruction Multiple Data*) é implementada como um conjunto de múltiplos processadores que tem a capacidade de, com uma única instrução, em um mesmo ciclo de clock, processar várias informações.

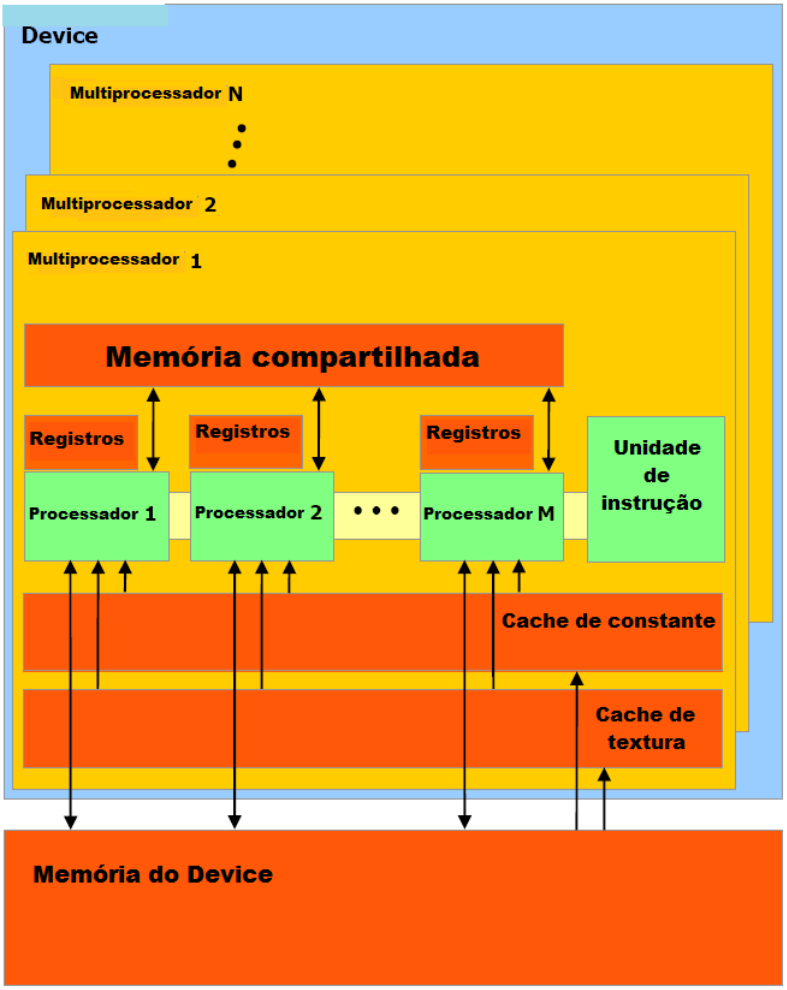

Figura 2.7: Conjunto de Multiprocessadores e Organização das Memórias(adaptado de [CUDA Development Core Team, 2012])

Como podemos observar cada multiprocessador possui: um conjunto de registradores de 32-bit, cache paralelo ou memória compartilhada que é partilhada com os outros processos,

uma cache constante e outra de textura apenas para leitura. A memória local e global não é ”cacheada”e pode ser escrita e lida pelos processos. Um ponto forte de CUDA é que há uma separação na memória DRAM entre o HOST e o DEVICE. A API de CUDA fornece uma forma de transmissão de alta performance para transferir os dados entre esses dispositivos usando *High Performance* (DMA), ou DMA de alto desempenho.

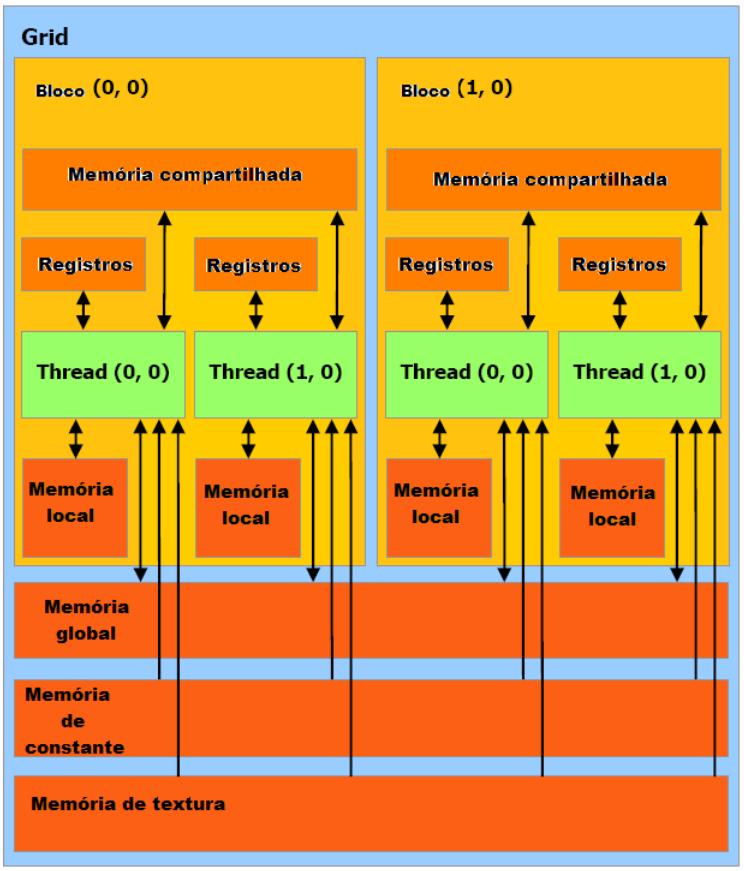

Figura 2.8: Hierarquia de memórias da GPU(adaptado de [CUDA Development Core Team, 2012])

2.4 Suporte ao Paralelismo de Tarefas

Diversas ferramentas de programação suportam paralelismo de tarefas em CPUs multicore. Recentemente, alguns trabalhos em andamento têm buscado oferecer suporte a esse paradigma também em sistemas híbridos compostos por CPUs e GPUs.

2.4.1 StarPU

StarPU [Augonnet et al., 2010] é um *framework* para o escalonamento de tarefas em plataformas heterogêneas, onde informações sobre o modelo de desempenho das tarefas podem ser passadas para orientar as políticas de escalonamento.

Os dois princípios básicos da StarPU são (i) as tarefas podem ter várias implementações, para alguns ou para cada uma das várias unidades de processamento heterogêneas disponíveis na máquina, e (ii) a transferência de dados para estas unidades de processamento são tratadas de forma transparente pelo StarPU.

O StarPU escalona dinamicamente tarefas em todas as unidades de processamento, assim evita que transferências de dados desnecessárias entre aceleradores aconteça. Também aceita tarefas que podem ter várias implementações, dependendo da plataforma. Este *framework* fornece uma camada alto nível de gerenciamento de dados, ou seja, quando uma tarefa é submetida, pela primeira vez, a tarefa entra em um ”pool”de ”Tarefas congeladas”, até que todas as dependências sejam atendidas. Em seguida, a tarefa é ”empurrada”para o escalonador. O StarPU apresenta algumas politicas de escalonamento, entre elas greedy, no-prio, ws, w-rand, heft-tm.

A estratégia *greedy*, *no-prior* e *ws* são estratégias gulosas, o qual assim que um recurso é liberado o escalonador submete uma tarefa. A *greedy* se diferencia da *no-prior*, pois a *greedy* tem suporte a prioridades, então pode ponderar pesos de prioridade a tarefas enquanto na *no-prior* não é possível. E a *ws* é uma estratégia gulosa baseada no ”work-stealing”que é uma estratégia que consiste em as unidades de processamento terem uma pilha de tarefas e se houver o término das tarefas pode haver o ”roubo”da tarefa de outras unidades.

Na estratégia *w-rand*, cada unidade de processamento está associada com uma relação que pode ser considerada como um fator de aceleração. Cada vez que uma tarefa é submetida, uma das unidades de processamento é selecionada com probabilidade proporcional à sua relação. Essa relação pode, por exemplo, ser definida pelo programador, ou ser medida com benchmarks de referência. A estratégia *w-rand* é normalmente adequado para tarefas independentes de igual tamanho.

A tarefa de distribuir uniformemente sobre as unidades de processamento baseado na velocidade não significa necessariamente fazer sentido para as tarefas que não são igualmente custosas, ou quando existem dependências entre eles. Assim, os programadores podem especificar um modelo de custo para cada tarefa. Este modelo pode por exemplo representar a quantidade de trabalho e ser usado em conjunto com a velocidade relativa de cada um das unidades de processamento. É também possível modelar diretamente o tempo de execução de cada uma das arquiteturas. Usando esses modelos, implementa-se a estratégia heft-tm. Dada

a sua duração esperada nas várias arquiteturas, é atribuída uma tarefa para as unidades de processamento que minimiza o tempo de término, no que diz respeito à quantidade de trabalho já atribuída a esta unidade de processamento.

O StarPU propõe uma abordagem de tarefas independente da arquitetura base. São definidos codelets como uma abstração de uma tarefa que pode ser executada em um núcleo de uma CPU multicore ou submetido a um acelerador. Cada codelet pode ter múltiplas implementações, uma para cada arquitetura em que o codelet pode ser executado, utilizando as linguagens ou bibliotecas específicas para a arquitetura alvo. Uma aplicação StarPU é descrita como um conjunto de codelets com suas dependências de dados.

StarPU é um sistema de execução unificado que consiste em um software e uma API de tempo de execução que tem como objetivo permitir que os programadores de aplicações com alto custo computacional explorem mais facilmente o poder de dispositivos disponíveis, com suporte a CPUs e GPUs. Submissões de tarefas são manipuladas pelo escalonador do StarPU, e a consistência dos dados é garantida através de uma biblioteca de gerenciamento de dados. No entanto, uma das principais vantagens perante a outras bibliotecas é que o StarPU tenta reduzir o custo da transferência de dados. Isso é feito usando as informações do histórico de cada tarefa e, de acordo com as decisões do escalonador, que de forma assíncrona prepara as dependências de dados, enquanto o sistema está ocupado realizando a computação de outras tarefas.

A API do StarPU apresenta uma certa terminologia bem definida, descrita a seguir:

- Manipulador de Dados (Data Handle): Referência a blocos de memória. A alocação do espaço requerido, e a entrega de informações sobre a transferência de dados para cada dispositivo.

- Codelet: Descreve uma função computacional que pode ser implementada em um ou mais arqquiteturas, tais como CPUs, CUDA ou OpenCL. Também armazena informação sobre a quantidade e tipos de dados em buffers que podem ser recebidos.

- Tarefa (Task): É definido como a associação entre um codelet e um conjutno de data handles.

- Partição (Partition): A subdivisão de um data handle em pequenos pedaços, de acordo com a função de divisão, que pode ser definida pelo usuário.

- Trabalhador (Worker): Um elemento processador, tal como um core CPU, gerenciado pelo StarPU para excutar Tasks;

- Escalonador (Scheduler): A biblioteca responsável pela atribuição das tarefas aos trabalhadores, com base em uma política de escalonamento bem definida.

Escalonador de tarefas

O *framework* StarPU emprega um modelo de programação baseado em tarefas. Kernels computacionais devem ser encapsuladas dentro de uma tarefa. StarPU vai lidar com a decisão de onde e quando a tarefa deve ser executada, com base em uma política de escalonamento de tarefas, e as implementações disponíveis para cada tarefa. Uma tarefa pode ser implementada de várias formas, tais como a CPU ou CUDA. Várias implementações para o mesmo tipo de dispositivo podem também ser utilizadas. Isso permite que StarPU automaticamente selecione a implementação apropriada, mesmo entre diferentes arquiteturas de CPU. Ao enviar uma tarefa, StarPU vai usar o escalonador escolhido para selecionar qual das implementações disponíveis serão utilizadas. A decisão varia de escalonador para escalonador, mas pode levar em conta informações como a atual disponibilidade de cada recurso, o modelo de desempenho já obtido para essa tarefa, e as estimativas sobre as transferências de dados necessários para resolver as dependências. A vantagem do uso de StarPU perante as outras ferramentas é a capcacidade e facildade de alteração do escalonador. A API do StarPU apresenta uma grande variedade de funções que permitem agilizar a implementação de um novo escalonador.

Os dados manipulados por uma tarefa são transferidos automaticamente quando necessário entre os vários dispositivos, assegurando a consistência de memória e liberando o programador de lidar com questões de escalonamento, transferência de dados e outros requisitos associados.

Dependências

StarPU cria automaticamente um grafo de dependência de todas as tarefas apresentadas, e os mantém em um conjunto de ”tarefas congeladas”, passando-as para o escalonador uma vez preenchidas todas as dependências.

Dependências podem ser implicitamente dadas pelos dados manipulados pela tarefa. Cada tarefa recebe um conjunto de buffers, cada uma correspondendo a uma parte dos dados geridos pela biblioteca de gerenciamento de dados do StarPU, e irá esperar até que todos os buffers estiverem prontos para ler.

Isso inclui as possíveis transferências de dados que são necessárias para atender as dependências, neste caso diferentes tarefas que dependem dos mesmos dados estão programados para rodar em diferentes nós computacionais. StarPU se certificará automaticamente das transferências de dados necessárias entre cada execução da tarefa para garantir a consistência dos dados.

Além de dependências de dados implícitos, outras dependências pode ser dados explicitamente a fim de forçar explicitamente a ordem de execução de um determinado conjunto de tarefas.

Modo de acesso aos dados

Cada dependência de dados que é explicitamente definida em uma tarefa, pode ter um modo de acesso diferente. Os dados podem ser usados em somente leitura, somente escrita ou no modo leitura e escrita. Este modelo descreve um tipo de padrão de mútua exclusão, onde um bloco de dados pode ser acessado simultaneamente por qualquer número de leitores, mas deve ser acessado exclusivamente por um escritor.

StarPU usa esse conceito para otimizar ainda mais os cálculos de dependência de dados. Se várias tarefas escalonadas dependem dos mesmos dados, então apenas com o acesso a leitura de dados, a dependência não deve bloquear as múltiplas tarefas de executar simultaneamente. Cópias temporárias de dados podem ser criadas, possivelmente em diferentes unidades de computação, e depois descartada, uma vez que um buffer de apenas leitura é assumido para permanecer inalterado no final de uma tarefa.

Memória virtual compartilhada

O objetivo é gerenciar automaticamente as alocações de memória e transferências em todos os dispositivos. Isto não só libera o programador do trabalho de gerenciamento manual de memória entre as tarefas, mas também tem o potencial de reduzir o custo de tais operações. StarPU gerencia a memória, forçando o usuário a declarar manipulador de dados para os seus dados. Esses manipuladores são usados como argumentos para as tarefas, permitindo que o escalonador aloque e transfira todos os buffers de dados necessários para a unidade de computação correta antes da execução da tarefa.

2.4.2 Charm++

Charm++ é uma linguagem de programação paralela orientada a objetos baseada em C++ e desenvolvida no Laboratório de Programação Paralela na Universidade de Illinois. Charme++ foi concebido com o objetivo de aumentar a produtividade do programador, fornecendo uma abstração de alto nível de um programa paralelo e ao mesmo tempo um bom desempenho em uma ampla variedade de plataformas de hardware. Programas escritos em Charme++ são decompostos em uma série de objetos controlados por mensagem chamados Chares. Quando um programador invoca um método em um objeto, o sistema de execução do Charme++ envia uma mensagem para o objeto chamado, que pode estar em um processador local ou em um

processador remoto. A mensagem inicia a execução de código dentro do chare que aguarda a mensagem de forma assíncrona.

Chares podem ser organizados em matrizes indexadas chamadas *Chare arrays* e mensagens podem ser enviadas para Chares individuais dentro de uma matriz chare ou para toda a matriz chare simultaneamente.

Os Chares em um programa são mapeados para processadores físicos por um sistema de execução adaptativo. O mapeamento de Chares para os processadores é transparente para o programador, e essa transparência permite que o sistema de execução altere dinamicamente a atribuição de Chares para processadores durante a execução do programa para conseguir atender algumas características, tais como balanceamento de carga baseada na medição, tolerância a falhas, pontos de verificação automática, e a capacidade de reduzir e expandir o conjunto de processadores utilizados por um programa paralelo.

No Charm++ [Kunzman et al., 2006] as tarefas são representadas pelos chares, que são objetos paralelos e representam unidades locais de trabalho. Cada chare possui dados locais, métodos para tratamento de mensagens e possibilidade de criar novos chares, assim como processos MPI-2. Existe ainda um tipo especial de chare, chamado branch-office, que possui uma ramificação em cada processador e um único nome global.

No Charm++ a sincronização pode ser feita através de *futures*, objetos de comunicação, replicados ou compartilhados. O *future* é uma estrutura que serve para armazenar um valor que pode ser acessado no futuro por outro Chare. A utilização de objetos de comunicação permite que um Chare se comunique por troca de mensagens, tornando a comunicação semelhante à realizada com MPI.

O Charm foi uma das primeiras implementações do conceito de Atores, que são objetos concorrentes que se comunicam apenas por troca de mensagens. Charm++ é baseado no Charm e suporta diferentes modos de compartilhamento de informações. Ele reúne recursos propostos em outros ambientes, como objetos sequenciais e paralelos, comunicação por troca de mensagens e futures.

O balanceamento de carga é feito através da migração de objetos em tempo de execução, através de medidas de tempo de processamento e comunicação. Os objetos encapsulam os dados e a função a ser executada sobre os dados, e através de troca de mensagens é possivel conhecer o estado de cada processador. Se um dados processador está demorando para executar uma dada função, o objeto migra para um outro processador, mais rapido ou que esteja ocioso.

2.4.3 Kaapi

O KAAPI (Kernel for Adaptative, Asynchronous Parallel and Interactive programming) é uma ferramenta para computação paralela em CPUs multicore e clusters. A ferramenta é baseada no algoritmo de roubo de tarefas, ou seja, cada processador recebe uma fila de tarefas a serem processdas, se um processador processa mais rápido do que se esperava a princípio, o processador começa a roubar tarefas da fila de outros processadores mais sobrecarregados.

O XKaapi [Gautier et al., 2013] é uma reimplementação do KAAPI com suporte a paralelismo de tarefas. . A implementação atual do XKaapi oferece suporte a arquiteturas multicore e extensões para suporte eficiente a GPUs que foram propostas em [Hermann et al., 2010; Lima et al., 2012]. O XKaapi é composto por um conjunto de APIs (Application Programming Interfaces) e pelo kernel, um ambiente de execução para as APIs oferece escalonamento baseado em roubo de tarefas. O Kaapi++ é a interface do XKaapi baseada em um grafo de fluxo de dados para C++ e é dividida em três componentes: a assinatura da tarefa (task signature) onde são definidos o número e as características dos parâmetros da tarefa; a implementação da tarefa (task implementation) que especifica a implementação da tarefa para uma arquitetura e a criação da tarefa (task creation) que submete a tarefa para a pilha de execução.

2.4.4 Cilk

Cilk é uma linguagem de programação de propósito geral projetada para computação parelela, é baseado na linguagem C, então é compatível com os compiladores gcc e Microsoft C++.

A ferramenta para programação paralela Cilk [Blumofe et al., 1995; Blumofe and Lisiecki, 1997] permite a submissão, execução e sincronização de tarefas paralelas. A implementação de Cilk é baseada no algoritmo de escalonamento dinâmico por meio de roubo de tarefas, como o KAAPI. Cilk define tarefas como funções individuais que podem submeter novas tarefas dinamicamente. A sincronização é feita permitindo que as tarefas esperem pelas tarefas filhas, ou seja, tarefas que foram submetidas pela tarefa original.

2.4.5 Intel TBB

O Intel TBB (Threading Building Blocks) [Kim and Voss, 2011] é uma biblioteca baseada em *templates* para programação paralela em C++ que faz uso de threads. Essa biblioteca permite expressar o paralelismo em diversos paradigmas de programação paralela como o paralelismo de dados, de laços ou de tarefas. As unidades de trabalho paralelas resultantes são escalonadas em threads por meio de um algoritmo de roubo de tarefas inspirado no ambiente Cilk.

2.4.6 OpenMP

OpenMP (Open Multi-Processing) [Quinn, 2003] é uma ferramenta para programação paralela baseada na adição de diretivas de compilação em códigos C, C++ e Fortran. As diretivas OpenMP são utilizadas para indicar a paralelização de laços ou trechos de código. A partir da versão 3.0 (OpenMP ARB, 2008) foi inserido o conceito de tarefas, o que permite a utilização do paralelismo de tarefas através do uso de diretivas para delimitação de trechos de código como unidades de trabalho paralelas. A especificação OpenMP não define uma política para o escalonamento das tarefas, ficando essa escolha a cargo de cada implementação.

2.4.7 Merge

Algoritmos de distribuição de carga são frequentemente baseados no conceito de tarefas. Neste caso, as tarefas das aplicações podem ser dinamicamente distribuídas entre os SMs (*Streaming Multiprocessor*) das GPUs [Chen et al., 2010a], ou entre diferentes GPUs [Augonnet et al., 2010] para realizar a distribuição da carga. Neste contexto surge o modelo de programação de uso geral para sistemas multi-core heterogêneos chamado *Merge* [Linderman et al., 2008]. O objetivo do *framework* é substituir as atuais abordagens *ad hoc* para programação paralela em plataformas heterogêneas, com uma metodologia baseada em uma biblioteca que pode distribuir automaticamente o cálculo através de núcleos heterogêneos. O balanceamento de carga e processamento é feita através do conceito de *map-reduce*, que é um modelo de programação dividido em dois procedimentos, o map, que organiza as informações e o reduce que realiza a operação de síntese.

Capítulo 3

Trabalhos Relacionados

A crescente demanda de processamento de informação levantou especial interesse no campo de balanceamento de carga. Muitos trabalhos foram feitos neste campo buscando o aprimoramento de técnicas e o tratamento específico para os problemas de escalonamento e balanceamento mais recorrentes, em especial em arquiteturas heterogêneas. Neste capítulo são abordados os trabalhos que serviram como base para esta dissertação ou que exploraram algum assunto diretamente relacionado ao tema aqui apresentado.

As novas possibilidades trazidas com as novas arquiteturas de GPU estão instigando pesquisadores a rever o problema do balanceamento de carga neste novo contexto. Apesar do balanceamento de carga ser crítico para o desempenho de aplicações paralelas, como no caso de aplicações gráficas [Foley and Sugerman, 2005; M. Mller et al., 2007], ainda não há na literatura muitos estudos sobre balanceamento de carga em GPUs.

[Cederman and Tsigas, 2008] analisaram várias estratégias de balanceamento de carga estática e dinâmica para um problema de partição de octree em GPUs. O balanceamento de carga é realizado internamente em uma única GPU, sem interação com a CPU. Neste trabalho foram testados os seguintes métodos de balanceamento de carga: (1) lista de tarefas estática, que consiste em dividir os dados em lista de tarefas, no qual cada unidade processamento retira uma tarefa da lista e executa-a, o trabalho termina quando a lista estiver vazia e o controle é retornado a CPU. (2) Lista de tarefas dinâmica bloqueante, que consiste em uma lista de tarefas que em tempo de execução pode receber novas tarefas. É bloqueante pois apenas um bloco de threads pode acessar a lista em um dado tempo. (3) Lista de tarefas dinâmica lock-free: consiste em fazer com seja controlado o acesso na inserção e remoção de tarefas na lista, dois blocos de threads podem acessar a lista, um para inserir e outro para remover tarefas, melhorando o paralelismo. (4) Roubo de tarefas: cada unidade de processamento recebe um conjunto de tarefas e quando completa as tarefas do conjunto tenta roubar uma tarefa de outra unidade de

processamento que ainda não concluiu suas tarefas atribuídas. Se uma unidade cria uma nova tarefa, a unidade acrescenta a seu próprio conjunto local de tarefas. Este estudo se diferencia do apresentado, pois é analisado apenas o comportamento em GPUs, não adicionando CPUs e clusters.

Em outro estudo [Guevara et al., 2009] com balanceamento de carga em GPUs, os autores propõem uma fila que mescla as cargas de trabalho que são subutilizados na GPU podendo ser executadas simultaneamente em uma GPU, através da fusão em um *super-kernel*. No entanto, esta fusão precisa ser realizada estaticamente, e assim o equilíbrio dinâmico de carga não pode ser sempre garantido. O trabalho foca somente em GPUs, apesar de apresentar a característica dinâmica, de modificar o escalonamento em tempo de execução.

[Acosta et al., 2012] propôs um algoritmo de balanceamento de carga dinâmico, que busca, interativamente, uma boa distribuição de trabalho entre as GPUs durante a execução da aplicação. O algoritmo de balanceamento consiste em obrigar todos os processadores a realizarem uma chamada a uma função chamada ULL_MPI_calibrate(). Nesta chamada todos os processadores realizam as mesmas operações de balanceamento, que são as seguintes. (1) Cada processador precisa determinar o tempo que levou para processar uma certa quantia de dados em cada iteração. (2) Testar se o tempo para processar é maior que um certo limiar, se sim, balancear a carga entre os processadores. (3) Calcular o que é chamado de *relative power*, que corresponde ao relacionamento entre o tempo gasto para processar um certo tamanho de dados em função do tamanho do problema. (4) Por fim, calcular a quantidade de dados atribuida a cada processador, assim todos os processadores realizam o próprio balanceamento de carga. Os autores avaliaram os problemas de multiplicação de matrizes e alocação de recursos. O método iterativo pode demorar para atingir uma distribuição de carga adequada, reduzindo o desempenho da aplicação. Além disso, não é possível utilizar informações relativas a tempos de execução anteriores para gerar um perfil de execução para as futuras execuções da aplicação. O trabalho é similar ao nosso, pois apresenta o conceito de limiar que determina se deve ser feito o balanceamento da carga.

Em outro trabalho, [de Camargo, 2012] propõe um algoritmo que determina a distribuição, antes do início da execução da aplicação, usando perfis estáticos obtidos em execuções anteriores. Antes de iniciar o processamento da carga o algoritmo fornece entradas de tamanhos diferentes às GPUs, e baseado nesses valores de tempos, encontra um tamanho de entrada que faz com que o tempo gasto por todos as GPUs seja o mesmo. O algoritmo foi avaliado para uma aplicação de redes neurais de grande escala. O algoritmo encontra a distribuição de dados que minimiza o tempo de execução da aplicação, garantindo que todas as GPUs gastem a mesma quantia de tempo realizando o processamento de *kernels*. Esta abordagem, entretanto, não permite alterações dinâmicas na distribuição dos dados.

Em [Clarke et al., 2012] os autores fizeram um balanceamento de carga dinâmico heterogêneo baseado na partição de matrizes, ou seja, apenas serve para problemas que envolvem matrizes. Há uma divisão da matriz entre as unidades, que se subdivide entre as subunidades e assim por diante.

No trabalho [Chen et al., 2010b], os autores propõe uma solução baseada em tarefas para o balanceamento de carga para sistema com várias GPUs. Atualmente no paradigma de programação em CUDA, para executar multiplas tarefas, o processo na CPU tem que lançar sequencialemente múltiplos *kernels* e o hardware é responsável por arranjar como os kernel rodarão na GPU, isto faz com que se execute um kernel, o controle retorne para a CPU, que submete um novo kernel e assim por diante. No balanceamento de carga propoto neste trabalho, a ideia é manter os blocos de threads sempre ativos e fazer com que uma fila de tarefas seja criada. Para cada GPU uma fila de tarefas é criada e assim multiplas GPUs realizam o processamento da carga. O trabalho é diferente do nosso, pois é voltado para o conceito de tarefas, em vez de decomposição de domínio, além do que no nosso trabalho é considerado CPUs como elementos de processamento, também de uma forma mais geral tem o foco no algoritmo de balanceamento de carga, não em características da GPU.

Por fim, [Belviranli et al., 2013] desenvolveram o que eles chamaram de Heterogeneous Dynamic Self-Scheduler (HDSS) que fornece um algoritmo de balanceamento de carga dinâmico, dividido em duas fases. A primeira fase é a fase adaptativa, que determina pesos que refletem a velocidade de cada processador. Esses pesos são determinados com base em ajustes de curvas logarítmicas em gráficos de velocidade de processamento. Estas ponderações são utilizadas durante a fase de acabamento, que é a segunda fase, onde é dividido os dados a serem processados nas iterações restantes entre as GPUs, com base nos pesos relativos de cada GPU. As principais diferenças com o nosso algoritmo é que eles restringem os modelos de execução a curvas logaritmicas, que são apropriados para GPUs, mas não são apropriadas para CPUs na faixa de interesse. Além disso, nosso algoritmo pode realizar ajustes até o fim da execução.

Capítulo 4

Algoritmo Proposto

Esta seção demonstra detalhes de implementação do algoritmo proposto para o balanceamento de carga dinâmico em aglomerados de GPUs.

Em uma típica aplicação paralela, os dados da aplicação são divididos entre as threads em um processo chamado decomposição de domínio. As threads então simultaneamente processam parte dos dados. Depois de terminarem, mesclam os resultados processados e terminam a aplicação ou continuam para a próxima fase de computação. A tarefa do algoritmo proposto é determinar o tamanho do bloco de dados atribuído a cada GPU e CPU no sistema. O termo processador significará uma única CPU ou GPU. Os processadores são diferentes entre si, o que torna o ambiente heterogêneo.

O algoritmo apresenta três partes, que são: (1) modelo de desempenho do processador, onde um modelo de execução é determinado durante a execução da aplicação; (2) seleção do tamanho ótimo do bloco, onde baseado no modelo de performance no qual o algoritmo seleciona a melhor distribuição de tamanho de blocos entre os processadores; e (3) o rebalancemaento onde o algoritmo recalcula o tamanho do bloco ótimo quando a diferença no tempo de execução nos diferentes processadores é maior que um certo limiar.

O algoritmo inicialmente gera um modelo de desempenho do processador. Este modelo consiste em gerar curvas baseadas no desempenho do processador para uma dada tarefa. A curva consiste em fornecer um pedaço do problema a cada processador e determinar o tempo que o processador gastou para processar aquele pedaço do problema. Para cada processador uma curva é gerada, que representa o modelo de desempenho para aquele dado problema. O algoritmo Inicialmente determina a curva $P_p[x]$, que fornece a velocidade de processamento para um dado bloco de tamanho x no processador p. Para construir estas curvas, um bloco de tamanho $initialBlockSize$, definido pelo usuário é atribuído para cada processador. Este tamanho inicial de bloco é arbitrário, e fica a cargo do usuário determina-lo.

Com o bloco de dados inicial igual para todos os processadores, é realizada a contabilização de todos os tempos. Este primeiro valor é o primeiro ponto utilizado para gerar a curva $P_p[x]$. O próximo ponto é determinado, elegendo o processador que realizou a tarefa em menor tempo, e dobrando o tamanho do bloco inicial para este processador. O restante dos processadores é realizado uma atribuição proporcional ao desempenho obtido na primeira atribuição. Supondo que o tempo do processador mais rápido seja t_f, dobra-se o tamanho do bloco para este processador. Para os outros processadores, com o tempo de término t_i, onde i é o tempo gasto para cada processador, seleciona-se o tamanho do bloco proporcional a razão t_i/t_f.

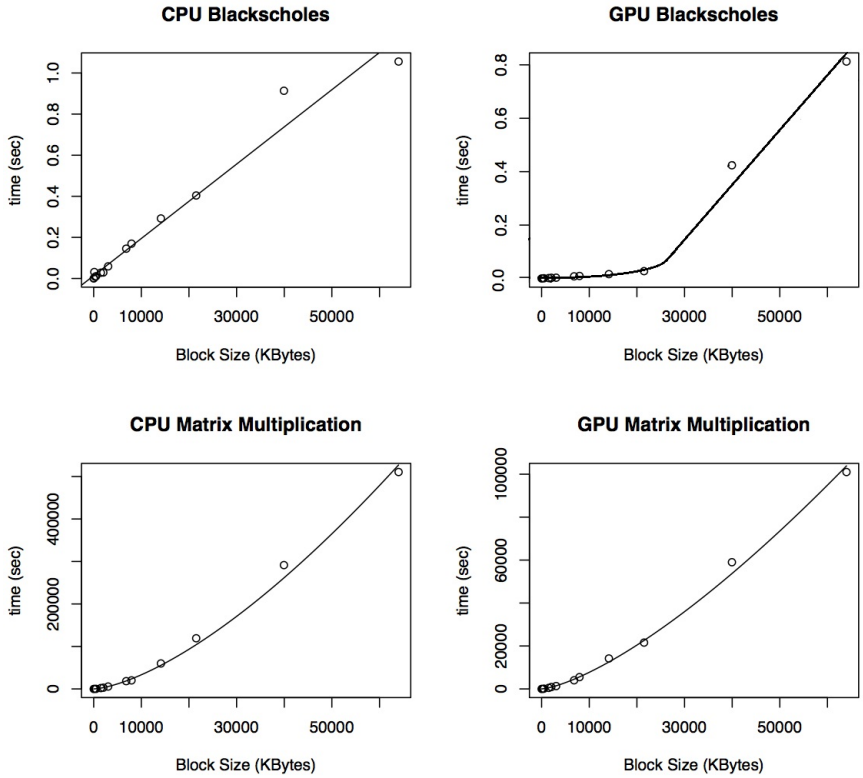

Figura 4.1: Curvas para GPU x CPU

A figura 4.1 mostra um exemplo de média de tempo para uma GPU e uma CPU para diferentes tamanhos de bloco. Pode-se notar que as curvas podem ser aproximadas por funções. Nestas curvas encontrou-se curvas que se ajustam através do método dos mínimos quadrados, usando uma função da forma $y = a_1 f_1(x) + a_2 f_2(x) + a_3 f_3(x) + ... + a_n f_n(x)$. A curva é ajustada após a geração de poucos pontos, por exemplo quatro, gerando um modelo de tempo de execução para cada processador.

A figura 4.1 mostra um exemplo de medida de tempo de processamento para uma GPU e uma CPU para diferentes tamanhos de bloco em duas aplicações. Pode-se notar que as curvas para a aplicação blackscholes pode ser aproximada por funções lineares, enquanto para a multiplicação de matrizes usa-se uma função exponencial. Encontrou-se o melhor ajuste das curvas utilizando o método dos mínimos quadrados:

$$y(x) = a_1 f_1(x) + a_2 f_2(x) + ... + a_n f_n(x) \tag{4.1}$$

onde $f_i(x)$ corresponde ao seguinte conjunto de funções x, x^2, x^3, e^x, lnx e seguintes combinações xe^x, $xlnx$. Se o modelo não se enquadrar em umas das funções descritas, é utilizada a função que aproxima o modelo. Os valores de a_i são dados pelo próprio método dos mínimos quadrados, como valor de ajuste. O método dos mínimos quadrados testa para cada função e a que apresentar menor erro é a selecionada.

Algorithm 1 Modelo de desempenho do processador

```
function determineModel()
blockSizeList ← initialBlockSize;
fitValues = determineCurveProcessor();
while iteration ≠ 4 do
    finishTimes = executeTasks(blockSizeList);
    synchronize();
    blockSizeList = evaluateNextBlockSizes(finishTimes);
    fitValues = determineCurveProcessor();
    iteration + +;
end while
return fitValues;
```

Os passos do algoritmo são mostrados no algoritmo 1. A variável $blockSizeList$ contem o tamanhos dos blocos atribuídos a cada processador e é inicializada com $initialBlockSize$, que é definida pelo usuário. A variável $fitValues$ contém o resultado do ajuste por mínimos quadrados, incluindo o erro no ajuste, que é inicializado como $+\infty$.

O laço é limitado a 4 pontos para gerar a curva. A função envia um pedaço de dados para cada processador e obtêm o tempo que demorou para realizar o processamento em cada processador. Depois de esperar, que todos os processadores terminem, é determinado o tamanho do bloco para a próxima iteração, baseado no tempo de término de cada processador. Por fim, o algoritmo tenta ajustar um modelo de curva para cada processador e recebe o resultado do ajuste.

Seleção do tamanho de bloco ótimo: O algoritmo determina o tamanho do bloco ótimo para cada processador com o objetivo de minimizar o tempo total da aplicação. Considere que

existem n processadores e o tamanho da entrada seja Z. O algoritmo atribui um pedaço de dados de tamanho x^g para cada processador $g = 1,...,n$, correspondendo a uma fração dos dados de entrada, tal que $\sum_{g=1}^{n} x^g = Z$. Denota-se como $E^g(x^g)$ o tempo de execução da tarefa E no processador g, para cada entrada de tamanho x^g. Para distribuir o trabalho entre os processadores, encontra-se um conjunto de valores:

$$X = \{x^g \in \mathbb{R} : [0,1] / \sum_{g=1}^{n} x^g = Z\} \tag{4.2}$$

que minimiza $E^1(x^1)$ enquanto satisfaz a restrição:

$$E^1 = E^2 = ... = E^n \tag{4.3}$$

que representa que todos os processadores gastariam uma mesma quantidade de tempo realizando o processamento. Para determinar o conjunto de valores de x, resolve-se o sistema de equações das curvas ajustadas para todos os processadores, dados por:

$$\begin{cases} E_1 = Y^1(x_1) \\ E_2 = Y^2(x_2) \\ E_n = Y^n(x_n) \end{cases} \tag{4.4}$$

O sistema de equações é resolvido aplicando *interior point line search filter method*. O algoritmo busca minimizar as funções no espaço de busca. Para cada função ele alcança uma solução ótima percorrendo o interior de uma solução viável. O objetivo é determinar o valor de x tal que o tempo seja mínimo. A complexidade do método é sempre polinomial.

Rebalanceamento: Depois de resolver o sistema de equações o escalonador mantem enviando tarefas de tamanho x_i para cada processador i, logo que o processador termina a tarefa anterior. Também monitora o tempo de término de cada tarefa. Se a diferença no tempo de término entre x_i e x_j de qualquer duas tarefas i e j for maior que um limiar, o processo de balanceamento é reexecutado. Neste caso, o algoritmo aplica o modelo de desempenho e a rotina de determinação do tamanho do bloco ótimo novamente. O escalonador então sincroniza as tarefas e inicia usando o novo valor de x_i. Ou seja,xs todo o processo de determinação dos modelos de desempenho e reolução do sistema de equações é reexecutado. O limiar (α) precisa ser determinado empiricamente através de alguns testes, apenas roda-se a aplicação com valores entre 0.1 e 1. Se o limiar é um valor muito pequeno, ocorre balanceamentos desnecessários, que aumentará o tempo total da aplicação. Se o limiar é muito grande balanceamento necessários podem não ocorrer, o que pode fazer com que threads fiquem ociosas.

A figura 4.2 apresenta um diagrama de execução para quatro threads e uma thread des-

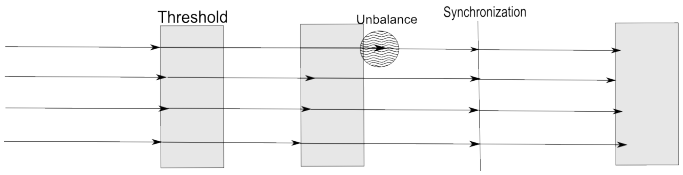

Figura 4.2: Execução das Threads com limiar

balanceada. As caixas representam os limiares, as linhas as threads e os círculos o desbalanceamento. As threads iniciam sincronizadas, recebem carga, mas em um segundo passo uma thread leva mais tempo que o valor de limiar, isto causa o desbalanceamento. Detectado o desbalanceamento, todas as threads são sincronizadas, e as partes são recalculadas para cada thread, fazendo com que as threads voltem ao estado de término dentro do valor de limiar estabelecido.

Algorithm 2 Algoritmo Dinâmico Completo

```
function dynamic()
fitValues = determineModel()
X = solveEquationSystem(fitValues);
while there is data do
    finishTimes = executeTasks(X);
    if maxDifference(finishTimes)≥ threshold then
        fitValues = determineModel();
        X = solveEquationSystem(fitValues);
        synchronize();
    end if
end while
```

Algoritmo Completo: Algoritmo 2 apresenta o pseudocódigo do algoritmo de escalonamento. A função $determineModel$ mostrada no algoritmo 1 retorna o modelo de desempenho para cada processador. Então resolve o sistema de equações para determinar a melhor distribuição de pedaços de dados para cada processador. Optou-se por não calcular a todo instante o modelo, para não aumentar o tempo gasto com o balanceamento de carga.

O laço inicial repete enquanto há dados ainda para serem processados. É distribuído pedaços de dados para cada processador do sistema, obtendo o tempo de término para cada execução de tarefa. É checado se a diferença máxima entre o término das tarefas está acima de um limar, obtido empiricamente. Se o limiar é alcançado, o algoritmo ajusta um novo modelo de curvas e resolve o sistema de equações para estas novas curvas para determinar uma nova distribuição de tamanhos pedaços de dados.

Capítulo 5

Implementação

5.0.8 StarPU

A aplicação foi feita na linguagem C, com o *framework* StarPU [Augonnet et al., 2010]. StarPU como brevemente citada no capitulo 2, é uma ferramenta para programação paralela que suporta arquiteturas híbridas como CPUs multicore e aceleradores. StarPU propõe uma abordagem de tarefas independentes baseada na arquitetura, para agilizar e facilitar o desenvolvimento de aplicações.

Para a implementação do algoritmo de balanceamento de carga, o StarPU apresenta uma variedade de ferramentas para a implementação da própria política de escalonamento. A principal estrutura utilizada para a implementação da política de escalonamento é a chamada ”starpu_sched_policy”, que é uma estrutura que contém métodos que implementam políticas de escalonamento. A forma de implementação da política de escalonamento consistiu em modificar as estruturas de dados pertencentes ao StarPU, simplificando e adicionando recursos.

A transferencia de dados consistiu em encapsular dentro de uma tarefa, os dados a serem processados por cada processador. Como utilizamos o conceito de decomposição de domínio, ou seja, os dados são independentes, as tarefas apresentam mesmo grau de prioridade. Assim foi possível utilizar a estrutura do StarPU, que é tem a execução baseada em grafos de tarefas, em um problema de decomposição de domínio.

Para avaliar o nosso algoritmo de balanceamento de carga, nós modificamos o algoritmo de escalonamento padrão do StarPU. A modificação do algoritmo de balanceamento de carga é feito alterando a variável STARPU_SCHED e classes internas do StarPU. O *framework* StarPU tem uma API que permite modificar as políticas de escalonamento. Existem estruturas de dados e funções que aceleram o processo de desenvolvimento. Por exemplo, a função ”double starpu_timing_now (void)”que retorna a data atual em micro segundos, o que torna mais fácil para a determinação de medidas de tempo de execução.

5.0.9 Multiplicação de Matrizes

A multiplicação de matrizes ou produto de matrizes é um problema clássico e foi escolhido por ter várias aplicações nos campos da engenharia e ciência, sendo de fácil comparação com os trabalhos da área. De forma visual a multiplicação de matrizes consiste em multiplicar as linhas de uma matriz pelas colunas da segunda matriz e somar, como na figura 5.1.

A multiplicação de matrizes é uma importante operação da álgebra linear. Um grande número de aplicações científicas e da engenharia incluem esta operação. Devido a sua fundamental importância, muitos esforços tem sido devotados ao estudo e implementação da multiplicação de matrizes. A multiplicação de A e B, onde A é uma matriz de N linhas por P colunas e a matriz B é de P linhas por M colunas. O resultado é a matriz C de N linhas e M colunas. No StarPU, foram implementadas duas versões uma para CPU e outra para GPU. Para GPU foi utilizado o kernel da NVidia com uso de memória compartilhada. Nesta implementação, cada bloco de threads é responsável por computar uma submatriz quadrada C_sub de C e cada thread dentro do bloco é responsável por um elemento de computação C_sub. C_sub é igual ao produto de duas matrizes retangulares: a submatriz de A de dimensão (A.largura, tamanho_bloco) que tem os mesmos índices das linhas de C_sub, e a submatriz de B de dimensão (block_size, A.largura), que tem os mesmos índices de coluna de C_sub. As duas matrizes retangulares são divididas em tantas matrizes quadradas de dimensão block_size quanto for necessário e C_sub é computada como a soma dos produtos destas matrizes quadradas. Com o StarPU, o conceito foi fornecer a cada processador as partes correspondentes das matrizes, para que seja respeitada a execução do kernel. Para a CPU, cada core foi responsável por computar uma linha de A e coluna de B, escrevendo o resultado em C.

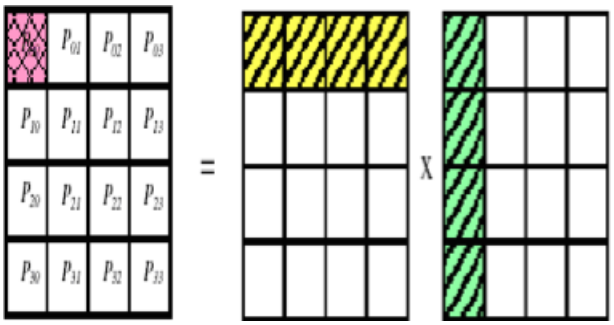

Figura 5.1: Multiplicação de Matrizes

Suponha que existem as matrizes A de N linhas por P colunas e B de P linhas e M colunas o resultado é uma matriz C de N por M colunas. De forma matemática a multiplicação pode ser definida como:

$$C_{ij} = \sum_{k=1}^{n_p} A_{ik} Bkj \tag{5.1}$$

A paralelização do código se dá dividindo partes das matrizes A e B entre os processadores, seguindo um padrão mestre-escravo. (1) O processo mestre envia trechos da matriz A (2) Cada processo escravo recebe uma matriz de todo o trabalho e recebe um determinado número de linhas da matriz B. Os nós computam as linhas da matriz de resultado para o número de linhas que recebeu e envia de volta para o mestre. (3) O processo mestre agora recolhe as linhas da matriz de resultados do escravo, juntamente com o seu deslocamento na matriz resultado.

O algoritmo utiliza um processo principal e uma série de tarefas escravas independentes. A implementação varia dependendo do modelo GPU, CPU, mas o conceito de divisão dos dados é o mesmo. Não foi utilizado um algoritmo mais eficaz para a multiplicação de matrizes, as únicas otimizações se referem a melhor utilização do hardware, como por exemplo o uso de memória compartilhada, e a divisão dos dados entre os blocos de threads, de forma a utilizar todas as threads de cada bloco. A divisão depende do modelo de placa, mas são configurações simples de serem feitas e não afetam a usabilidade do algoritmo de balanceamento de carga.

5.0.10 Blackscholes

O modelo Blackcholes é um modelo matemático de mercado financeiro contendo certo instrumentos de investimento. Do modelo, o que pode ser deduzido é a fórmula Blackscholes, que nos fornece a estimativa teórica dos preços de opções no estilo europeu. Em finanças, uma opção é um contrato que dá ao comprador (o proprietário) o direito, mas não a obrigação, de comprar ou vender um ativo ou instrumento subjacente, a um preço determinado antes de uma data especificada. O vendedor tem a obrigação de cumprir a operação, que é a de vender ou comprar. Uma opção que transmite ao titular o direito de comprar algo a um preço específico é referida como uma chamada; uma opção que transmite o direito do proprietário para vender algo a um preço específico é referida como uma opção de venda. Ambos são comumente negociadas, mas para maior clareza, a opção de compra é mais freqüentemente discutidos. A fórmula levou a um ”boom”na negociação de opções e legitimou cientificamente as atividades da Chicago Board Options Exchange e outros mercados de opções ao redor do mundo. Muitos testes empíricos têm demonstrado que o preço de Black-Scholes é ”bastante próximo”aos preços observados, embora haja discrepâncias bem conhecidas.

O modelo de Blackscholes foi publicado por Fischer Black e Myron Scholes em 1973, em um artigo entitulado *”The Pricing of Options and Corporate Liabilities”*, publicado no *Journal of Political Economy*. Eles derivaram uma equação diferencial parcial, agora chamada

de equação de Black-Scholes, que estima o preço da opção ao longo do tempo.

Robert C. Merton foi o primeiro a publicar um artigo ampliando a compreensão matemática do modelo de precificação de opções, e cunhou o termo ”Black-Scholes de precificação de opções”. Merton e Scholes receberam o Prêmio Nobel de Economia em 1997 (O Prêmio Sveriges Riksbank em Ciências Econômicas em Memória de Alfred Nobel) por seu trabalho. Embora inelegível para o prêmio por causa de sua morte, em 1995, Black foi mencionado como um contribuinte pela Academia Sueca.

Pressupostos do modelo foram relaxadas e generalizados em muitas vertentes, levando a uma infinidade de modelos que são atualmente utilizados em precificação de derivativos e gestão de riscos. É os insights do modelo, como exemplificado na fórmula Black-Scholes, que são frequentemente utilizados pelos participantes do mercado, como distinguir entre os preços reais. A equação de Black-Scholes, uma equação diferencial parcial que governa o preço da opção, também é importante, pois permite colocar preços quando uma fórmula explícita não é possível.

A fórmula Black-Scholes tem apenas um parâmetro que não pode ser observada no mercado: a volatilidade futura média do ativo. Uma vez que a fórmula é o aumento neste parâmetro, ela pode ser invertida de modo a produzir uma superfície de volatilidade que é então utilizado para calibrar outros modelos. A equação do Blackscholes é a seguinte:

$$\frac{\delta V}{\delta t} + \frac{1}{2}\sigma^2 S^2 \frac{\delta^2 V}{\delta S^2} + rS\frac{\delta V}{\delta S} - rV = 0 \tag{5.2}$$

onde V é o preço da opção em função do preço das ações S e t é o tempo, r é a taxa de juros livre de risco, e σ é a volatilidade das ações.

O principal problema da implementação em CUDA da fórmula Blackscholes é escolher o melhor *layout* de armazenamento de dados. A implementação do *kernel* utilizada foi a fornecida pela API do Cuda. A implementação consiste em atribuir a cada thread um índice específico dos dados de entrada. A divisão dos dados na StarPU basicamente, é dividir as opções entre os processadores de forma direta.

5.0.11 Explicação dos algoritmos

Três outros algoritmos foram implementados para comparação: o guloso (StarPU), o estático e o HDSS. O guloso consistiu em dividir o conjunto de entrada em pedaços e atribuir cada pedaço da entrada a qualquer processador ocioso, sem qualquer atribuição de prioridade. O estático [de Camargo, 2012], mede as velocidades de processamento antes da execução e atribui um conjunto de blocos estático para cada processador, no início da execução, com os tamanhos do bloco proporcional à velocidade do processador. Por fim, o HDSS [Belviranli

et al., 2013] utiliza a estimativa dos mínimos quadrados para estimar a curva logarítmica e divide em duas fases: fase de adaptação e fase de conclusão.

HDSS - Heterogeneous Dynamic Self-Scheduler

O algoritmo HDSS [Belviranli et al., 2013] determina o tamanho do bloco por processador usando uma abordagem em duas fases. A fase inicial, chamada de fase de adaptação, é responsável por encontrar os pesos computacionais que refletem as velocidades relativas de cada processador. Esta fase processa uma quantidade relativamente pequena de dados, cujo limite superior é definido como uma percentagem fixa do total de dados. A fase final, chamada fase de conclusão, processa o restante dos dados, usando os pesos calculados na fase adaptativa. Esta fase começa com os maiores blocos possíveis para otimizar o tempo de execução. O tamanho do bloco diminui progressivamente à medida que a computação trabalha no sentido final para que todas as unidades de execução completem, quase ao mesmo tempo. O conceito do algoritmo é em um primeiro momento encontrar pesos para serem usados posteriormente na fase de conclusão, esses pesos darão a forma com que os dados devem ser divididos entre os processadores. Os pesos são determinados com base em ajustes em gráficos logaritmicos de processamento. As principais diferenças do algoritmo proposto o nosso modelo não se restringe a curvas logarítimocas, além de se ajustar até o fim da execução.

5.0.12 IPOPT

A biblioteca utilizada para resolver o sistema de equações é a IPOPT [Nocedal et al., 2009]. IPOPT *(Interior Point Optimizer)* é um pacote de software de código aberto para otimização não-linear em grande escala, roda em Windows, Linux e Mac-OS. Ele pode ser utilizado para resolver problemas de programação lineares gerais, da forma:

$$\begin{aligned} \min_{x \in \mathbb{R}^n} f(x) \\ s.a.\ g^L \leq g(x) \leq g^U \\ x^L \leq x \leq x^U, \end{aligned} \tag{5.3}$$

onde x, $\in$, $\Re^n$ são as variáveis de otimização, $f : R^n$ em R é a função objetivo, e $g : R^n$ em R^M são as restrições não lineares. A função $f(x)$ e $g(x)$ podem ser linear ou não linear.

Para resolver um problema de otimização precisa-se criar um IpoptProblem com a função CreateIpoptProblem, que posteriormente precisa ser passado para a função IpoptSolve. O IpoptProblem criado por CreateIpoptProblem contem as dimensões do problema, as variáveis

e os limites das restrições. A execução da resolução torna-se apenas um processo de correta montagem do problema a ser tratado. O algoritmo leva certo tempo para resolver o problema que é na ordem dos milissegundos.

Capítulo 6

Resultados

Nesta seção são apresentados os resultados obtidos utilizando o algoritmo de balanceamento de carga dinâmico em um *clusters* heterogêneo. Foram realizados experimentos para duas aplicações, a multiplicação de matrizes e para a aplicação Blackscholes.

6.1 Resultados dos experimentos

Foram usadas três diferentes máquinas para testar o algoritmo, apresentadas na tabela 6.1. A máquina A tem uma CPU i7 a20, com 4 cores com velocidade de clock de 2,67 GHz, 8192 MB de cache, 8 GB de RAM and 2 nVidia GTX 295, cada GTX 295 tem 280 cores por GPU, clock da memória de 999 MHz e tamanho da banda de memória de 223,8 GB/segundos com 2 GPUs em cada.

A máquina B tem um processador i7 4930K CPU, com 6 cores, clock de 3,4 GHz, memoria cache de 12.288 KB, 32 GB de RAM e 2 nVidia GTX 680, cada GTX 680 tem 1546 cores, 6 Gbps clock de memória e tamanho da banda de memória de 192,2 MHz.

A máquina C tem um processador CPU i7 3939K, com 6 cores, velocidade de clock de 3,2 GHz, memória cache de 122.88KB, 32 GB de RAM e 1 nVidia GTX Titan, a GTX Titan tem 2688 cores, velocidade da memória de 6 Gbps e tamanho da banda de memória de 223,8 GB/segundo.

Foram considerados os cenários com apenas a máquina A, com a máquina A e B, e com a três máquinas (A, B e C). Os computadores estão conectados por uma rede Gigabit Ethernet. Foi usado o sistema operacional Ubuntu 12.04 e CUDA 5.5.

Para usar todos os n multiprocessadores da GPU, é necessário criar ao menos n blocos. Mais que isso, cada multiprocessador simultaneamente executa grupos (chamados de warps) de m threads de um único bloco. e muitos warps estão presentes em cada GPU para o uso

Tabela 6.1: Configuração das máquinas

Máquina	Modelo CPU	Número de GPUs	Memória da GPU	Modelo de GPU
A	Intel i7 a20	4 x GTX200b	896MB	GTX 295
B	Intel i7 4930K	2 x GK104	2GB	GTX 680
C	Intel i7 3930K	2 x GK110	6GB	GTX Titan

eficiente dos processadores. Warp é um grupo de threads presentes em um bloco da GPU.

Em todos os testes, usa-se todos os multiprocessadores das GPUs lançando kernels com k blocos com 1024 threads por bloco, onde k é o numero de processadores na GPU. Para as GPUs usadas, k é 14, 8 e 30 na GTX Titan, GTX 680 e GTX 295 respectivamente. Para as CPUs, foram usados todos os cores das CPUs, lançando uma tarefa por core.

6.2 Multiplicação de Matrizes

O algoritmo foi testado para a multiplicação de matrizes usando uma, duas e três máquinas e quatro diferentes algoritmos de escalonamento: (1) algoritmo proposto, (2) estático, (3) HDSS, e (4) StarPU (guloso).

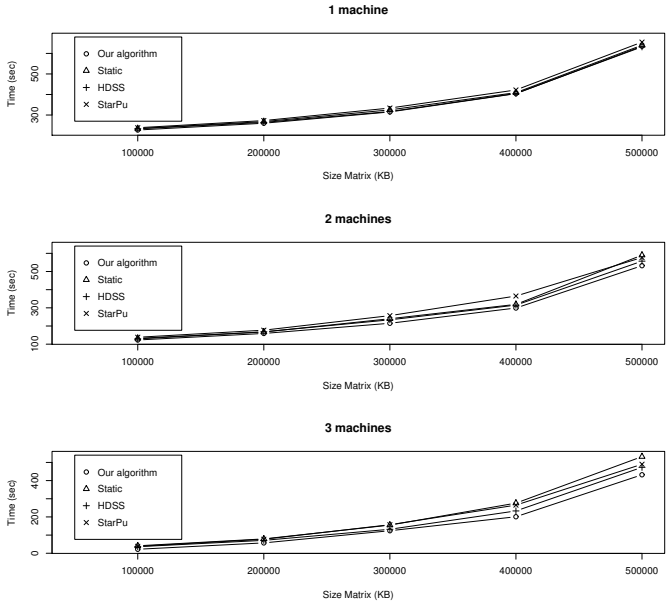

Figura 6.1: Diferença de execução com diferentes tamanhos de matrizes, para a multiplicação de matrizes

A figure 6.1 apresenta os resultados para matrizes com tamanhos 10.000 x 10.000 a 50.000 x 50.000 elementos. Em todos os cenários, o algoritmo de escalonamento proposto obteve

melhores resultados, com o HDSS em segundo. O estático e o StarPU foram claramente inferiores.

Com uma máquina a diferença foi pequena porque há poucos tipos de dispositivos para o escalonador selecionar. Com duas máquinas nosso algoritmo começa a ter melhor desempenho, especialemente para casos de matrizes grandes, que é explicado pelo fato de que o tempo de execução da maultiplicação de matrizes aumenta rapidamente como aumenta-se o tamanho da matriz. Com três máquinas obteve-se o ambiente mais heterogêneo e o ganho de desempenho usando o algoritmo proposto é o maior. Como nos outros cenários, aumentando-se o tamanho da matriz também aumenta-se o ganho de desempenho como o algoritmo proposto. Para matrizes 10.000 x 10.000, o algoritmo proposto gastou 20 segundos enquanto o HDSS gastou 36, que resulta em 22,2% mais rápido. E para matrizes 50.000 x 50.000, o algoritmo proposto gastou 433 segundos enquanto o HDSS 473, que resulta em 8,49% mais rápido, o resumos está na tabela 6.2

Tabela 6.2: Comparativo: HDSS x Algoritmo Proposto

	Tamanho Matriz 10,000 x 10,000 KB			tamanho Matriz 50,000 x 50,000		
Num. Máquinas	HDSS	Algoritmo Proposto	Differ. (%)	HDSS	Algoritmo Proposto	Differ. (%)
1 Machine	233.42	227.01	2.75	632.13	635.32	-0.50
2 Machines	129.43	123.62	4.49	557.92	532.11	4.63
3 Machines	36.85	22.43	39.13	473.64	433.41	8.49

Os resultados obtidos mostram que em ambientes mais heterogêneos o uso do algoritmo proposto apresenta mais vantagem. Esta vantagem é devido melhor distribuição dos dados ao longo da execução. A figura 6.2 mostra a diferença de tempo entre a primeira e a última thread, no cenário com três máquinas.

O algoritmo guloso do StarPU teve um bom desempenho, considerando que ele não tem diretamente a informação sobre a velocidade de processamento dos dispositivos. Mas ele usa estas informações indiretamente, desde que dispositivos mais rápidos que terminam suas tarefas mais cedo e, consequentemente, recebem mais tarefas. O algoritmo estático teve o pior desempenho, porque deixa a thread ociosa por muito tempo ver figura 6.2. Há grandes diferenças entre os tempos de término das threads. O algoritmo estático não realiza rebalanceamentos durante a execução, a curva obtida no inicio da execução é obtida, através da estimativa dos pontos e o balanceamento obtido é mantido até o fim da execução.

HDSS utiliza o início da execução para estimar o melhor tamanho de bloco e usa essa distribuição até o fim da execução. Além disso, grandes blocos são usados no início da execução, causando um atraso maior devido ao desbalanceamento quando usa diferentes tipos de dispositivos, como GPUs e CPUs. Por fim, HDSS utiliza uma aproximação bruta para a capacidade do dispositivo.

Uma das medidas realizadas foi de tempo utilizada para por exemplo o HDSS realizar o

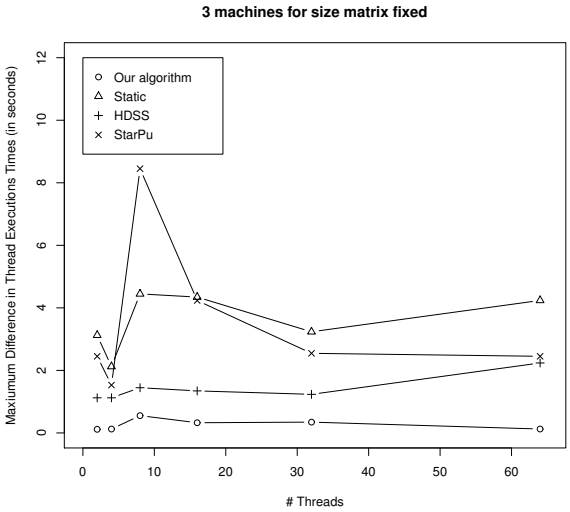

Figura 6.2: Diferença de tempo entre Threads

balanceamento, comparado ao algoritmo proposto. No HDSS, o tempo de balanceamento é de 0,2 segundo, enquanto para o algoritmo proposto é de 0,6 segundos. Diferença considerável, mas a melhor precisão do balanceamento compensa, o tempo gasto em cada balanceamento.

O algoritmo proposto tem bastante precisão na estimativa, a quantidade de dados que devem ser fornecida para cada unidade de processamento, a solução de um problema de optimização com o restrição de que todas as unidades devem terminar a execução das tarefas ao mesmo tempo. Quando a diferença entre terminar a execução de threads para certos partição de dados excede um certo limite, o algoritmo reequilibra os dados distribuição.

6.3 Blackscholes

Os experimentos para o blackscholes seguiram a mesma linha apresentados na multiplicação de matrizes. O algoritmo Blackscholes tem complexidade linear no tempo, diferente da multiplicação de matrizes que apresenta complexidade $O(n^3/2)$.

A transferência de dados é feita dividindo as opções que devem ser calculadas entre os diferentes processadores. O número de opções atribuidas a cada processador depende do algoritmo de balanceamento de carga.

Foram realizados testes de tempo de execução para a aplicação chamada blackscholes usando diferentes configuraçõs de máquinas. A figura 6.3 apresenta os tempo de execução, onde variou-se o número de opõs em cada execução e obteve-se o tempo de execução. Simi-

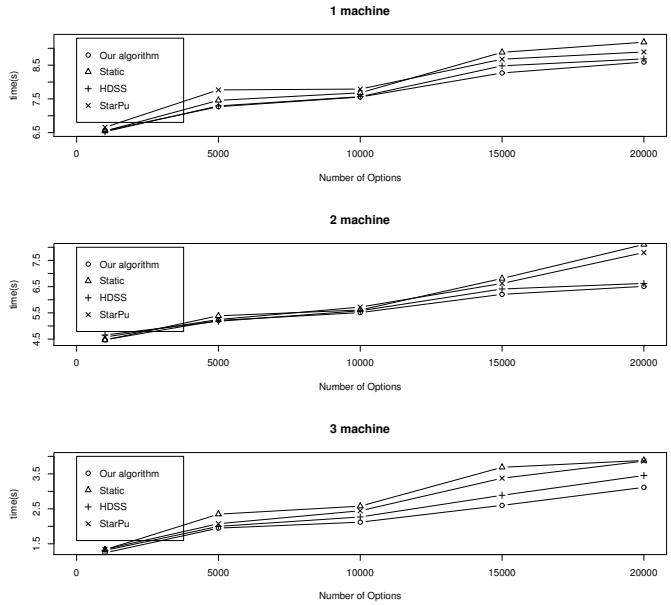

Figura 6.3: Diferença no tempo de execução para diferentes números de opções

h

Tabela 6.3: Comparativo: HDSS x Algoritmo Proposto

	1000 Opções			20000 Opções		
Num. Máquinas	HDSS (s)	Algoritmo (s)	Diff. (%)	HDSS (s)	Algoritimo (s)	Diff. (%)
1	6.52	6.55	-0.45	8.68	8.59	1,03
2	4.65	4.47	3.87	6.62	6.51	1.66
3	1.31	1.24	5.34	3.45	3.11	9.81

larmente aos experimentos de multiplicação de matrizes, o melhor desempenho foi alcançado nos problemas maiores, com maior números de opções e em ambientes mais heterogêneos. Os resultados podem ser explicados com os mesmos argumentos. Blackscholes tem complexidade linear com o número de opções, mostrando que o algoritmo proposto é também útil para esta classe de problemas. A tabela 6.3 mostra os valores extremos com 1000 e 20000 opções, o melhor resultado foi com três máquinas, e com o maior número de opções.

Também, considerou-se que a aplicação termina em menos que 4 segundos, para o cenário com 3 máquinas, é possível notar que o *overhead* imposto pelo solucionador do sistema de equações para determinar a melhor distribuição é pequeno e o ganho obtido ultrapassa o custo dos cálculos. Para as aplicações testadas, em poucas iterações, foi possível obter a solução do sistema de equações, o que resultou em poucos milissegundos, na ordem de 0.4 s.

A figura 6.4 confirma o resultado de que a diferença entre a primeira a terminar e a última, no cenário com três máquinas é sempre menor para o algoritmo proposto.

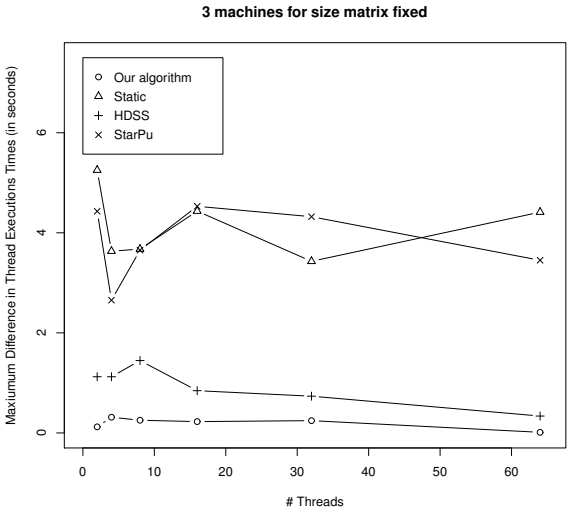

Figura 6.4: Diferença de tempo entre a primeira e a última thread a terminarem o trabalho

6.4 Conclusões

Foi proposto um algoritmo de escalonamento de tarefas para problemas de decomposição de domínio que executam em *clusters* de CPUs e GPUs hetererogêneas. O algoritmo proposto supera outros algoritmos similares devido a estimativa online da curva de desempenho para cada processador e a seleção da melhor distribuição para os dispositivos. foi apresentado para duas aplicações que o algoritmo proposto fornece maiores ganhos para problemas grandes e em ambientes mais heterogêneos.

Embora usou-se clusters dedicados, pode-se também considerar o uso em nuvem públicas, onde o usuário pode requisitar um numero de recursos alocados me máquinas virtuais de máquinas compartilhadas. Neste caso, a qualidade do serviço pode alterar durante a execução, e a adição de um limiar permite ajustar a distribuição de dados. Pode-se também considerar o cenário com a adição de tolerância a falhas, onde máquinas podem ser tornar indisponíveis durante a execução. Neste cenário, uma simples redistribuição dos dados entre os dispositivos restantes permitiria que a aplicação se readapte a este novo cenário.

Capítulo 7

Plano de Trabalho

Os detalhes do plano de tabalho são apresentados na tabela 7.1. O aluno ingressou no programa em fevereiro de 2013 e tem previsão de término em dezembro de 2014.

Etapa	2013											2014												2015		
	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
Disciplinas do Mestrado	X	X	X	X	X	X	X	X	X	X	X															
Levantamento Bibliográfico				X	X	X	X	X	X	X					X	X	X	X	X	X						
Desenvolvimento do Algoritmo					X	X	X	X	X	X	X	X	X	X												
Implementação do Algoritmo							X	X	X	X	X	X	X	X	X	X	X	X								
Implementação da Comunicação																		X	X	X	X					
Experimentos													X	X	X	X	X	X	X	X	X	X				
Escrita da Qualificação																		X	X							
Escrita de Artigo de Conferência																		X	X	X	X					
Escrita de Artigo para Revista																						X	X	X		
Escrita da Dissertação																							X	X	X	

Tabela 7.1: Cronograma das atividades realizadas e previstas

O plano de trabalho foi dividido em várias etapas. Nas quais foram realizadas todas as disciplinas necessárias, feito um levantamento bibliográfico inicial, que direcionou a ideia inicial do algoritmo. Foi feito o desenvolvimento do algoritmo, que vem sendo realizado desde agosto de 2013, com vários estudos e tutoriais principalmente relacionados a StarPU. Foi implementado os algoritmos de comparação, que apresentam certa complexidade de implementação. E inicialmente foi gerado um algoritmo baseado no HDSS, que era uma modificação do algoritmo original. Após a familizarização com o StarPU foi possível o total desenvolvimento do algoritmo e implementação do algoritmo de escalonamento.

Os próximos passos são a implementação do modelo de comunicação, que está sendo feito. Que vai levar em consideração a quantidade de dados transmitida entre os processadores.

Também será adicionada como aplicação teste uma rede de inferencia regulatória gênica.

Referências Bibliográficas

Acosta, A., Blanco, V., and Almeida, F. (2012). Towards the dynamic load balancing on heterogeneous multi-gpu systems. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 646–653.

Anastas, P. T. and Warner, J. C. (2000). *Green chemistry: theory and practice*. Oxford University Press.

Augonnet, C., Thibault, S., and Namyst, R. (2010). Starpu: a runtime system for scheduling tasks over accelerator-based multicore machines. Rapport de recherche RR-7240, Laboratoire Bordelais de Recherche en Informatique - LaBRI, RUNTIME - INRIA Bordeaux - Sud-Ouest, http://hal.inria.fr/inria-00467677/PDF/RR-7240.pdf.

Belviranli, M. E., Bhuyan, L. N., and Gupta, R. (2013). A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA. ACM.

Blumofe, R. D. and Lisiecki, P. A. (1997). Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '97, pages 10–10, Berkeley, CA, USA. USENIX Association.

Borelli, F., Camargo, R., Martins, D., Stransky, B., and Rozante, L. (2012). Accelerating gene regulatory networks inference through gpu/cuda programming. In *Computational Advances in Bio and Medical Sciences (ICCABS), 2012 IEEE 2nd International Conference on*, pages 1–6.

Camargo, E., Kostin, S., and Pinto, R. (2011). A tool for scientific visualization based on particle tracing algorithm on graphics processing units. In *Sistemas Computacionais (WSCAD-SSC), 2011 Simposio em*, pages 10–10.

Cederman, D. and Tsigas, P. (2008). On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 57–64, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Chen, L., Villa, O., Krishnamoorthy, S., and Gao, G. (2010a). Dynamic load balancing on single- and multi-gpu systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12.

Chen, L., Villa, O., Krishnamoorthy, S., and Gao, G. R. (2010b). Dynamic load balancing on single-and multi-gpu systems. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE.

Clarke, D., Ilic, A., Lastovetsky, A., and Sousa, L. (2012). Hierarchical partitioning algorithm for scientific computing on highly heterogeneous cpu + gpu clusters. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 489–501, Berlin, Heidelberg. Springer-Verlag.

CUDA Development Core Team (2012). CUDA 4.1 Programming Guide.

da Silva, M. A., Boaventura, J. S., de Alencar, M. G., Cerqueira, C. P., de Energia, G.-G., and dos Materiais, C. (2007). Desenvolvimento de protótipo de células a combustível do tipo óxido sólido com reforma direta. *Revista Matéria*, 12(1):99–110.

de Camargo, R. (2011). A multi-gpu algorithm for communication in neuronal network simulations. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–10.

de Camargo, R. (2012). A load distribution algorithm based on profiling for heterogeneous gpu clusters. In *Applications for Multi-Core Architectures (WAMCA), 2012 Third Workshop on*, pages 1–6.

Delpech, S., Merle-Lucotte, E., Heuer, D., Allibert, M., Ghetta, V., Le-Brun, C., Doligez, X., and Picard, G. (2009). Reactor physic and reprocessing scheme for innovative molten salt reactor system. *Journal of fluorine chemistry*, 130(1):11–17.

Devarakonda, M. and Iyer, R. (1989). Predictability of process resource usage: a measurement-based study on unix. *Software Engineering, IEEE Transactions on*, 15(12):1579–1586.

Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S. (2004). Gpu cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pages 47–, Washington, DC, USA. IEEE Computer Society.

Foley, T. and Sugerman, J. (2005). Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA. ACM.

Gautier, T., Ferreira Lima, J. V., Maillard, N., and Raffin, B. (2013). XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Boston, Massachusetts, États-Unis.

Graham, R. L. (1966). Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581.

Guevara, M., Gregg, C., Hazelwood, K., and Skadron, K. (2009). Enabling task parallelism in the cuda scheduler. In *Workshop on Programming Models for Emerging Architectures*, PMEA, pages 69–76, Raleigh, NC.

Hermann, E., Raffin, B., Faure, F., Gautier, T., and Allard, J. (2010). Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In D'Ambra, P., Guarracino, M. R., and Talia, D., editors, *Europar 2010 - 16th International Euro-Par Conference on Parallel Processing*, volume 6272 of *Lecture Notes in Computer Science*, pages 235–246, Ischia-Naples, Italie. Springer.

Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., and Lee, J. (2012). Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 341–352, New York, NY, USA. ACM.

Kim, W. and Voss, M. (2011). Multicore desktop programming with intel threading building blocks. *IEEE Softw.*, 28(1):23–31.

Kunzman, D., Zhang, G., Bohm, E., and Kale, L. V. (2006). Charm++, offload api, and the cell processor. *Urbana*, 51:61801.

Li, Q., Salman, R., Test, E., Strack, R., and Kecman, V. (2011). Gpusvm: a comprehensive cuda based support vector machine package. *Central European Journal of Computer Science*, 1(4):387–405.

Lima, J. V., Gautier, T., Maillard, N., and Danjean, V. (2012). Exploiting concurrent gpu operations for efficient work stealing on multi-gpus. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 75–82. IEEE.

Linderman, M. D., Collins, J. D., Wang, H., and Meng, T. H. (2008). Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43(3):287–296.

M. Mller, C., Strengert, M., and Ertl, T. (2007). Adaptive load balancing for raycasting of non-uniformly bricked volumes. *Parallel Comput.*, 33(6):406–419.

Mandel, J. (1993). Balancing domain decomposition. *Communications in Numerical Methods in Engineering*, 9(3):233–241.

Merrill, D. G. and Grimshaw, A. S. (2010). Revisiting sorting for gpgpu stream architectures. In *PACT '10 Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 545–546. University of Virginia, Charlottesville, VA, USA.

Miyoshi, T., Irie, H., Shima, K., Honda, H., Kondo, M., and Yoshinaga, T. (2012). Flat: a gpu programming framework to provide embedded mpi. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 20–29, New York, NY, USA. ACM.

Nocedal, J., Wächter, A., and Waltz, R. (2009). Adaptive barrier update strategies for nonlinear interior methods. *SIAM Journal on Optimization*, 19(4):1674–1693.

Owens, D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE*, 96(1):879–899.

Pinedo, M. L. (2012). *Scheduling: theory, algorithms, and systems*. Springer.

Quinn, M. J. (2003). *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group.

Riegel, E., Indinger, T., and Adams, N. A. (2009). Implementation of a lattice–boltzmann method for numerical fluid mechanics using the nvidia cuda technology. *Computer Science - Research and Development*, 23(3-4):241–247.

Singh, A., Korupolu, M., and Mohapatra, D. (2008). Server-storage virtualization: Integration and load balancing in data centers. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12.

Tanenbaum, A. and Woodhull, A. (2006). *Sistemas Operacionais: Projetos e Implementação*. Bookman Companhia ed.

Vouzis, P. D. and Sahinidis, N. V. (2011). Gpu-blast: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–188.

Wang, L., Huang, M., Narayana, V. K., and El-Ghazawi, T. (2011a). Scaling scientific applications on clusters of hybrid multicore/gpu nodes. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 6:1–6:10, New York, NY, USA. ACM.

Wang, L., Jia, W., Chi, X., Wu, Y., Gao, W., and Wang, L.-W. (2011b). Large scale plane wave pseudopotential density functional theory calculations on gpu clusters. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–10.