UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

UM COMPONENTE PARA EXPLORAÇÃO DA CAPACIDADE DE PROCESSAMENTO DE GPUS EM GRADES COMPUTACIONAIS

DISSERTAÇÃO DE MESTRADO

Guilherme Linck

Santa Maria, RS, Brasil

2010

UM COMPONENTE PARA EXPLORAÇÃO DA CAPACIDADE DE PROCESSAMENTO DE GPUS EM GRADES COMPUTACIONAIS

por

Guilherme Linck

Dissertação apresentada ao Programa de Pós-Graduação em Informática
da Universidade Federal de Santa Maria (UFSM, RS), como requisito
parcial para a obtenção do grau de
Mestre em Computação

Orientador: Prof. Dr. Benhur de Oliveira Stein (UFSM)

Dissertação de Mestrado Nº 11
Santa Maria, RS, Brasil

2010

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós-Graduação em Informática

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

UM COMPONENTE PARA EXPLORAÇÃO DA CAPACIDADE DE PROCESSAMENTO DE GPUS EM GRADES COMPUTACIONAIS

elaborada por
Guilherme Linck

como requisito parcial para obtenção do grau de
Mestre em Computação

COMISSÃO EXAMINADORA:

Prof. Dr. Benhur de Oliveira Stein (UFSM)
(Presidente/Orientador)

Prof. Dr. Nicolas Maillard (UFRGS)

Prof. Dr. Cesar Tadeu Pozzer (UFSM)

Santa Maria, 24 de Setembro de 2010.

AGRADECIMENTOS

Quero primeiramente de agradecer a Deus, por ter me dado saúde e sabedoria para conduzir este trabalho até o final. Quero agradecer também ao meu orientador, prof. Dr. Benhur Stein, pela paciência e aconselhamentos em determinados momentos, e também por ter acreditado que eu conseguiria terminar este trabalho. Não poderia deixar de agradecer aos integrantes do colegiado do PPGI, por terem aceitado meu pedido de prorrogação, o que tornou possível a conclusão deste trabalho.

Também quero agradecer à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pelas bolsas de estudo recebidas durante o mestrado. Aos familiares e amigos, pela força e pela compreensão da minha ausência em determinadas ocasiões. Agradeço também ao prof. Dr. Cesar Pozzer, por ter cedido algumas horas do seu tempo, inclusive nas férias, para que eu pudesse realizar a avaliação deste trabalho. Por fim, meus mais sinceros agradecimentos a todos os demais que tiveram participação na realização deste trabalho.

“Mas não basta pra ser livre
Ser forte, aguerrido e bravo
Povo que não tem virtude
Acaba por ser escravo”

SEGUNDA ESTROFE DO HINO RIO-GRANDENSE

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

UM COMPONENTE PARA EXPLORAÇÃO DA CAPACIDADE DE PROCESSAMENTO DE GPUS EM GRADES COMPUTACIONAIS

Autor: Guilherme Linck
Orientador: Prof. Dr. Benhur de Oliveira Stein (UFSM)
Local e data da defesa: Santa Maria, 24 de Setembro de 2010.

Grades de computadores surgiram na década de 90 com o objetivo de utilizar computadores geograficamente dispersos para computação de alto desempenho. Através destas grades, pode-se chegar ao poder computacional de um supercomputador de uma forma simples, eficiente e barata. Tais benefícios fizeram com que pesquisas em grades de computadores obtivessem destaques no ramo da computação.

Recentemente, surgiram no mercado placas adaptadoras gráficas cujo poder computacional supera, e com larga vantagem, mesmo os mais modernos processadores de uso geral. Isso deu origem a pesquisas que resultaram em técnicas de programação relativamente fáceis de aprender e que simplificam a programação de aplicações para estes processadores. Estas técnicas efetivamente introduziram estes processadores no ramo de computação de alto desempenho. A utilização destas técnicas deu origem à programação de propósito geral em unidades de processamento gráfico (*General-Purpose computation on Graphical Processor Units*-GPGPU).

Aplicações de grades são geralmente programadas através de um *framework* de computação em grade. *TUXUR* é um destes *frameworks* e encontra-se em desenvolvimento por mestrandos do Programa de Pós-graduação em Informática da Universidade Federal de Santa Maria. Este trabalho aborda o desenvolvimento de uma funcionalidade prevista no *TUXUR*. Tal funcionalidade permite que a grade de computadores gerenciada por *TUXUR* usufrua dos benefícios de aplicações GPGPU, sobretudo no que diz respeito à melhor utilização do *hardware* dos nós que a compõem. O reflexo imediato desta sinergia é o aumento significativo da capacidade computacional da grade sem o acréscimo de novos computadores.

Os resultados encontrados na avaliação evidenciam a importância do uso de GPGPU nas tarefas que se beneficiam desta técnica de programação, mesmo quando executadas em uma grade.

Palavras-chave: GPGPU, computação em grade, otimização dos recursos da grade.

ABSTRACT

Master's Dissertation
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

DEVELOPMENT OF A MODULE TO EXPLORE GPGPU CAPABLE COMPUTERS IN A GRID COMPUTING
Author: Guilherme Linck
Advisor: Prof. Dr. Benhur de Oliveira Stein (UFSM)

Computer grids emerged in the 90's with the goal of using geographically dispersed computers for high performance computing. Through grids, computational power of a supercomputer can be reached in a simple, efficient and inexpensive way. Such benefits led to highlights in researchs of computer grids.

Recently, appeared on market graphics adapter cards whose computational power overcomes, and by a wide margin, even the most modern processors commonly used. This led to researchs that resulted in programming techniques relatively easy to learn and did simplify application programming for these processors. These techniques effectively introduced the processors in the business of high performance computing. The use of these techniques gave rise to General Purpose computing on Graphic Processing Units (GPGPU).

Grids applications are generally programmed through a grid computing *framework*. *TUXUR* is one of those *frameworks* and is under development by Master's Program Graduate at the Federal University of Santa Maria. This dissertation discusses the development of a *TUXUR*'s foreseen feature. Such feature allows the computer grid managed by *TUXUR* to enjoy the benefits of GPGPU applications, particularly regarding to the best use of the nodes's *hardware* that comprises it. The immediate impact of this synergy is the significant increase in grid computational capacity without adding new computers.

The findings of the evaluation highlights the importance of using GPGPU tasks that take advantage of this programming technique, even when performed in a grid.

Keywords: GPGPU, grid computing, grid resources optimization.

LISTA DE FIGURAS

Figura 2.1 – GFLOP/s: CPUs *versus* GPUs. Fonte: (NVIDIA, 2009). 24
Figura 2.2 – Largura de banda: CPU *versus* GPU. Fonte: (NVIDIA, 2009). 24
Figura 2.3 – Utilização de transistores: CPU *versus* GPU. Fonte: (NVIDIA, 2009). 24
Figura 2.4 – Estrutura interna de um *kernel*. Fonte: (NVIDIA, 2009). 28
Figura 2.5 – Modelo de memória em CUDA. Fonte: (NVIDIA, 2007) 29
Figura 2.6 – Modelo de programação simplificado do ATI *Stream Computing*. Fonte: (AMD, 2009). 32
Figura 2.7 – Poder computacional dos clientes do projeto *Folding@home*. Fonte: (STANFORD, 2010b). 34

Figura 3.1 – Arquitetura geral de Tuxur. 40

Figura 4.1 – Presença do módulo na hierarquia de computadores gerenciada por *TUXUR*. 44
Figura 4.2 – Arquitetura de um trabalhador. 45
Figura 4.3 – Exemplo de uma possível grade formada pelo lançador. 49

Figura 5.1 – Representação de um Bloco. 54
Figura 5.2 – Código da Função FDCT. 54
Figura 5.3 – Relação das quantidades de trabalho dos *Jobs* recebidos no Cenário 1. 56
Figura 5.4 – Tempo de resolução dos *Jobs* no Cenário 1. 57
Figura 5.5 – Capacidades de trabalho dos *Trabalhadores* no Cenário 1. 58
Figura 5.6 – Tamanho em *bytes* das respostas dos *Jobs* do Cenário 1. 58
Figura 5.7 – Tamanhos EM *bytes* dos *Jobs* do Cenário 1. 59
Figura 5.8 – Ociosidade dos trabalhadores no Cenário 1. 59
Figura 5.9 – Porcentagem do tempo total em estado ocioso dos trabalhadores no Cenário 1. 60
Figura 5.10 – Relação das quantidades de trabalho dos *Jobs* recebidas no Cenário 2. 61
Figura 5.11 – Tempo de resolução dos *Jobs* no Cenário 2. 62
Figura 5.12 – Capacidades de trabalho dos *Trabalhadores* no Cenário 2. 63
Figura 5.13 – Tamanhos em *bytes* dos *Jobs* do Cenário 2. 63
Figura 5.14 – Tamanho em *bytes* das respostas dos *Jobs* do Cenário 2. 63
Figura 5.15 – Ociosidade dos trabalhadores no Cenário 2. 64
Figura 5.16 – Porcentagem do tempo total em estado ocioso dos trabalhadores no Cenário 2. 64
Figura 5.17 – Relação das quantidades de trabalho dos *Jobs* recebidas no Cenário 3. 65
Figura 5.18 – Tempo de resolução dos *Jobs* no Cenário 3. 66
Figura 5.19 – Capacidades de trabalho dos *Trabalhadores* no Cenário 3. 66

Figura 5.20 – Tamanho em *bytes* das respostas dos *Jobs* do Cenário 3. 67
Figura 5.21 – Tamanhos em *bytes* dos *Jobs* do Cenário 3. 67
Figura 5.22 – Ociosidade dos trabalhadores no Cenário 3. 68
Figura 5.23 – Porcentagem do tempo total em estado ocioso dos trabalhadores no Cenário 3. 68

LISTA DE TABELAS

Tabela 5.1 – Descrição do *hardware* dos computadores utilizados na avaliação. 55
Tabela 5.2 – Localização dos trabalhadores. 56

LISTA DE ABREVIATURAS E SIGLAS

ALU	Arithmetic and Logic Unit (Unidade Lógica e Aritmética)
AMD	Advanced Micro Devices
BMP	Acrônimo de BitMaP – mapa de bits
BOT	Bag of Tasks (Saco de Tarefas)
CEA	Comissão de Energia Atômica
CPU	Central Processing Unit (Unidade Central de Processamento)
CUDA	Compute Unified Device Architecture
FDCT	Forward Discrete Cosine Transform (Transformada Discreta do Cosseno Direta)
GFLOPS	Giga Floating point Operations Per Second (Bilhões de Operações de Ponto Flutuante por Segundo)
GPGPU	General Purpose computing on Graphics Processing Units (Computação de Propósito Geral em Unidades de Processamento Gráfico)
GPU	Graphics Processing Unit (Unidade de Processamento Gráfico)
IBM	International Business Machines
JPEG	Joint of Photographic Experts Group (União do Grupo de Especilialistas em Fotografia)
LSC	Laboratório de Sistemas de Computação
Mbps	Milhões de bits por segundo
OpenCL	Open Computing Language (Linguagem de Computação Aberta)
SETI	Search for Extraterrestrial Intelligence (Busca por Inteligência Extraterrestre)
TFLOPS	Tera Floating point Operations Per Second (Trilhões de Operações de Ponto-flutuante por Segundo)
UFSM	Universidade Federal de Santa Maria
ULA	Unidade Lógica e Aritmética

SUMÁRIO

1 INTRODUÇÃO 14
1.1 Contexto e Motivação 14
1.2 Objetivos e Contribuição 15
1.3 Organização do Texto 16

2 REVISÃO BIBLIOGRÁFICA 17
2.1 Grades Computacionais 17
2.1.1 Principais Conceitos 17
2.1.2 Problemas Envolvidos e Pesquisas Atuais 19
2.2 GPGPU 22
2.2.1 Computação de Propósito Geral em GPUs 23
2.2.2 GPGPU: Benefícios 23
2.2.3 GPGPU: Desafios 26
2.3 Técnicas de Programação para GPGPU 27
2.3.1 Nvidia CUDA 27
2.3.2 AMD Stream Computing 32
2.3.3 OpenCL 32
2.4 Utilização de GPGPU em Grades 33
2.4.1 *Folding@home* 33
2.4.2 SETI@*home* 35

3 FRAMEWORK TUXUR 36
3.1 Objetivos e Contribuição 36
3.2 Tipos de Tarefas Suportadas por Tuxur 37
3.3 Características Importantes 37
3.3.1 Divisão Dinâmica de Tarefas 38
3.3.2 Heterogeneidade de Ambiente 38
3.4 Principais Componentes e Suas Funções 39
3.4.1 Gerente 40
3.4.2 Trabalhador 40
3.4.3 Interface de Comunicação 41
3.4.4 Lançador 41
3.4.5 *Job* 42
3.4.6 Interface do Usuário 42

4 MÓDULO GPGPU 43
4.1 Objetivos 43
4.2 Relação do módulo com o TUXUR 43
4.3 Componentes Implementados 44
4.3.1 Trabalhador 44
4.3.2 Interface de Comunicação 46
4.3.3 *Job* 47
4.4 Integração deste Módulo ao Tuxur 48

5 AVALIAÇÃO 50
5.1 Metodologia 50
5.2 Abordagem do Problema 51
5.2.1 Função *Forward Discrete Cosine Transform* 52
5.2.2 Etapas Transformadas em *Job* 52
5.2.3 Descrição do *Job* 53
5.3 Resultados Obtidos 55
5.3.1 Cenário 1 56
5.3.2 Cenário 2 60
5.3.3 Cenário 3 65

6 CONSIDERAÇÕES FINAIS 69

REFERÊNCIAS 71

APÊNDICE A FUNÇÕES PRÉ-DEFINIDAS 80
A.1 Executadas pelo *Gerente* 80
A.1.1 job breakJob() 80
A.1.2 byte[] description() 80
A.1.3 void setSubResult() 81
A.1.4 boolean isSolved() 81
A.2 Executas pelos *Trabalhadores* 81
A.2.1 void solve() 81
A.2.2 int workload() 81
A.3 Executadas por Ambos 82
A.3.1 job newJob() 82
A.3.2 byte[] resultDescription() 82

APÊNDICE B FUNCIONALIDADES DO GERENTE NO TUXUR 83
B.1 Divisão e Delegação de Tarefas 83
B.2 Recepção e Formação da Resposta Final 85

APÊNDICE C VERSÃO IMPLEMENTADA DO TUXUR 86
C.1 Componentes Implementados 86
C.1.1 Lançador 86
C.1.2 Gerente 87
C.1.3 Trabalhador 87
C.1.4 *Job* 87

1 INTRODUÇÃO

1.1 Contexto e Motivação

Aplicações científicas costumam envolver computação intensa sobre um enorme volume de dados. Devido à quantidade de cálculos a serem realizados por estas aplicações, um único computador, mesmo um supercomputador, poderia levar uma grande quantidade de tempo para concluir uma tarefa. Geralmente, o tempo total decorrido pode tornar inviável a sua espera. Uma das alternativas para se melhorar o desempenho na execução destas aplicações está no emprego de grades de computadores. O uso de grades de computadores é uma das formas mais simples e baratas de se alcançar poder computacional de um supercomputador (SINGH et al., 2008).

Pesquisas na área de computação de alto desempenho em meados da década de 90 deram origem às grades de computadores (FOSTER, 2001). Estas grades são formadas por diversos computadores, local ou geograficamente distribuídos e interligados em rede (FOSTER; KESSELMAN, 1999). O baixo custo para a execução de programas distribuídos, quando comparado a um supercomputador, associado à escala de paralelização possível de ser alcançada e até então impensável para um supercomputador, fez com que as grades computacionais ganhassem destaque no ramo da computação (CIRNE; SANTOS-NETO, 2005).

Há poucos anos apareceram no mercado processadores gráficos cujos custos e poder computacional chamaram a atenção da comunidade científica. Isso abriu caminho para utilizá-los não somente para gerar imagens e exibi-las na tela do monitor, mas utilizá-los para computação de propósito geral. Avanços neste sentido deram origem à computação de propósito geral em unidades de processamento gráfico (*Graphic Processing Unit - GPU*), mais conhecida pela sigla GPGPU (*General-Purpose computation on Graphical Processor Units*) (OWENS et al., 2007).

A lei de Moore afirma que o poder de processamento dos computadores dobra a cada 2 anos (MOLLICK, 2006). Esta lei não se aplica às GPUs, cujo poder computacional supera esta cifra a cada ano (WU; LIU, 2008; LI et al., 2009). Este descolamento em relação à lei de Moore leva a crer que seu uso seja cada vez mais vantajoso.

Os dois principais fabricantes de placas adaptadoras gráficas, Nvidia (NVIDIA, 2010a) e AMD (AMD, 2010a), viram no ramo do GPGPU um valioso nicho de mercado. Embora já fosse possível utilizar uma GPU para computação dos mais diversos tipos de problemas, a programação destas aplicações exigia considerável conhecimento da arquitetura do processador gráfico (WU; LIU, 2008). Estas empresas então criaram técnicas de programação que simplificaram e facilitaram a programação de aplicações GPGPU. A Nvidia criou o CUDA (*Compute Unified Device Architecture*) (NVIDIA, 2009) e a AMD desenvolveu o *Stream Computing* (AMD, 2010b). Estes ambientes facilitaram o ingresso dos produtos destas empresas no ramo de computação de alto desempenho. A maioria das novas GPUs produzidas por estas empresas suportam seus respectivos ambientes. Por possuírem supremacia neste mercado, a probabilidade de encontrar uma de suas GPUs instalada em um computador é relativamente alta.

Tornar possível a execução de aplicações GPGPU em uma grade de computadores é a principal motivação para o desenvolvimento deste trabalho.

1.2 Objetivos e Contribuição

A aglomeração de recursos dispersos de uma grade computacional com algumas centenas de computadores resulta em uma considerável capacidade computacional. O poder computacional de uma grade pode ser aumentado através da inserção de novos computadores, ou pela atualização (*upgrade*) de *hardware* dos existentes. Esta atualização pode incluir a simples substituição da placa adaptadora gráfica. O trabalho aqui apresentado objetiva colocar estes processadores à disposição das aplicações de uma grade computacional.

TUXUR, atualmente em desenvolvimento no Laboratório de Sistemas de Computação (LSC) da Universidade Federal de Santa Maria (UFSM), permitiu que esse objetivo fosse alcançado. Assim, computadores que possuam *hardware* compatível com GPGPU podem receber tarefas especialmente codificadas, fazendo uso destes valiosos recursos que do contrário seriam praticamente ignorados. O uso concomitante da GPU e CPU des-

tes computadores para a computação de tarefas melhora significativamente a utilização do *hardware* destes computadores. O desenvolvimento desta funcionalidade, como um módulo do *TUXUR*, é a contribuição deste trabalho.

1.3 Organização do Texto

A organização desta dissertação respeita a seguinte estrutura de capítulos. No capítulo dois é apresentada a revisão bibliográfica sobre grades de computadores, GPGPU e seu uso em grades. No capítulo três discorremos sobre *TUXUR*, o *framework* de computação em grade onde o módulo GPGPU, razão desta dissertação, está inserido. O capítulo quatro aborda o módulo desenvolvido. A avaliação do módulo encontra-se no capítulo cinco. As considerações finais e trabalhos futuros relacionados a este trabalho estão inseridas no capítulo seis.

2 REVISÃO BIBLIOGRÁFICA

2.1 Grades Computacionais

2.1.1 Principais Conceitos

Grades de computadores surgiram na década de 90, como proposta de uma infraestrutura de computação distribuída que permitisse o compartilhamento de recursos entre projetos de colaboração científica (FOSTER; KESSELMAN, 2003). Inicialmente, grades de computadores foram definidas por FOSTER; KESSELMAN (1999) como sendo uma infraestrutura de *software* e *hardware*, que fornece acesso a uma grande quantidade de recursos computacionais de forma segura, consistente, abrangente e barata.

Atualmente, FOSTER; KESSELMAN (2003) definem grade como sendo "um sistema que coordena recursos distribuídos utilizando protocolos de propósito geral, padronizados, abertos e interfaces que entregam qualidade de serviço não trivial". OGSA (*Open Grid Services Architecture*) (FOSTER; KESSELMAN; TUECKE, 2003; FOSTER et al., 2006) é no momento uma das arquiteturas para grades mais influentes no mundo (LU; MA, 2009).

Grades de computadores tem sido adotadas nos mais diversos domínios de aplicação. Em (FOSTER; KESSELMAN, 2003) são abordadas doze importantes aplicações, que englobam problemas enfrentados pela indústria, ciência, laboratórios, universidades, grandes corporações, entre outras.

Aplicações em grades podem ser agrupadas em cinco grandes classes (FOSTER; KESSELMAN, 1999, 2003).

- *Distributed Supercomputing*: estão inseridas nesta classe as aplicações que necessitam de uma enorme quantidade de recursos computacionais para serem concluídas. Estas aplicações são fortemente acopladas. Muitas vezes os requisitos computacionais não podem ser atendidos por um único supercomputador. Em vista disso,

recursos de vários supercomputadores podem ser agregados. Atualmente esta classe de aplicação é também conhecida como metacomputação.

- *High-Throughput Computing*: esta classe de aplicações compreende aplicações independentes ou fracamente acopladas. Normalmente suas execuções exploram ciclos não utilizados pelo processador. Aqui, recursos computacionais de diversos computadores são agrupados para se obter alta vazão na conclusão de tarefas. Esta classe leva este nome pois o desempenho das aplicações não se dá pela tradicional métrica de operações de ponto-flutuante realizadas por segundo, mas sim pela velocidade em que computadores devolvem os resultados pretendidos.

- *On-Damand Computing*: aplicações inseridas nesta classe necessitam temporariamente de recursos computacionais que não podem ser obtidos localmente. seja pelo custo ou mesmo por conveniência. Para estas aplicações, o custo envolvido é mais importante do que o desempenho obtido.

- *Data-Intensive Computing*: são aplicações que necessitam armazenar uma enorme quantidade de dados. Geralmente este volume é grande demais para ser armazenado localmente. Grades fornecem um meio de armazenar e acessar estes dados em repositórios geograficamente distribuídos. Normalmente os dados destas aplicações são processados antes de serem efetivamente armazenados, o que as torna computacionalmente intensivas também. É necessária uma boa infraestrutura de comunicação para acesso remoto às bases de dados distribuídas.

- *Collaborative Computing*: são aplicações que tornam possível a interação e compartilhamento de recursos entre pessoas geograficamente distribuídas, normalmente em tempo real.

As aplicações contidas nestas classes utilizam a grade para obter os recursos necessários para atingirem seus objetivos. SUCIU; POTOLEA (2008) proporam uma taxonomia para aplicações em grade, com base nas suas características estruturais, dados que utilizam como entrada e gerados como saída. A sintaxe das siglas por eles utilizada é semelhante à utilizada na taxonomia de Flynn.

As características das aplicações executadas por uma grade também geram uma classificação para estas grades (KRAUTER; BUYYA; MAHESWARAN, 2001; BALI et al., 2009).

- *Computational Grid*: é o tipo de grade que agrega recursos computacionais de um grande número de computadores, com o propósito de oferecer uma gigantesca capacidade computacional para execução de tarefas. As classes de aplicações *Distributed* e *High-Throughput Computing* são executadas nestas grades.

- *Data Grid*: são grades que oferecem uma estrutura especializada para acesso e armazenamento distribuído de informações. Aplicações classificadas como *Data-Intensive Computing* fazem uso deste tipo de grade.

- *Service Grid*: basicamente, este é o tipo de grade que oferece serviços que não podem ser fornecidos por um único computador. As classes *On-Damand* e *Collaborative Computing* executam nesta classificação de grade.

- *Equipment Grid*: são grades onde os principais recursos são instrumentos que podem ser controlados e acessados remotamente. Os dados coletados por estes instrumentos são processados pela grade que os suportam.

2.1.2 Problemas Envolvidos e Pesquisas Atuais

A promessa de alocar uma grande quantidade de recursos para uma aplicação paralela, necessitando de baixo investimento financeiro, despertou o interesse pelas grades de computadores (COSTA; CIRNE; FIREMAN, 2005). No entanto, para um bom aproveitamento dos recursos colocados à disposição, muitos desafios que envolvem a utilização de recursos presentes na grade devem ser superados.

De acordo com BUYYA; ABRAMSON; GIDDY (2001), o gerenciamento e o escalonamento dos recursos é uma tarefa complexa, pois: os recursos estão distribuídos geograficamente; são heterogêneos; pertencem a diferentes indivíduos ou organizações com políticas próprias; possuem diferentes modelos de custos e acesso; a disponibilidade e carga dos recursos é dinamicamente variável. Gerenciar e providenciar os recursos que satisfaçam as solicitações dos usuários são os principais objetivos dos sistemas de gerenciamento de recursos(CHEN; LU, 2008).

2.1.2.1 Descoberta de Recursos

É fato que não se consegue gerenciar o que não se conhece. Para contornar este problema, grades devem possuir mecanismos de descoberta de recursos. Estes mecanismos são fundamentais para grades de computadores, pois auxiliam no gerenciamento de

recursos e escalonamento de tarefas (RANJAN et al., 2007). O objetivo básico destes mecanismos é descobrir recursos presentes na grade, que se adequem à descrição dos recursos feita pelos usuários e necessários para a execução de suas tarefas.

Diversos trabalhos implementam mecanismos de descoberta de recursos utilizando abordagens centralizadas ou hierárquicas. Estas abordagens tendem a apresentar problemas de escalabilidade e tolerância a falhas na medida em que o tamanho da grade aumenta (TRUNFIO et al., 2007; COKUSLU; HAMEURLAIN; ERCIYES, 2009). Abordagens centralizadas possuem um ponto único de falha e sofrem problemas de sobrecarga no servidor central. Já a hierárquica elimina o problema de sobrecarga, mas a falha em um ponto da hierarquia prejudica o acesso a recursos em níveis inferiores.

Em (COKUSLU; HAMEURLAIN; ERCIYES, 2009) pode ser encontrada uma avaliação de mecanismos de descoberta de recursos atuais, baseados em *Web Services* (WS). Estão presentes abordagens centralizadas e hierárquicas. Com base nos trabalhos avaliados, os autores chegaram à conclusão de que a utilização de WS para esta tarefa só é adequada para grades pequenas e que não apresentam grande dinamicidade no número de nós.

Em vista das deficiências das abordagens mencionadas, soluções distribuídas que empregam características de sistemas ponto-a-ponto (*Peer-to-Peer* - P2P) se mostram promissoras. SCHMIDT; PARASHAR (2003) citam descentralização, auto-organização e tolerância a falhas como características que tornam estes sistemas escaláveis e atrativos para serem utilizados em grades. Trabalhos que utilizam soluções P2P podem ser encontrados em (IAMNITCHI; FOSTER, 2004; SCHMIDT; PARASHAR, 2003; RANJAN et al., 2007; FATTAHI; CHARKARI, 2009; MA; SUN; GUO, 2010). TRUNFIO et al. (2007) apresentam uma interessante fonte de consulta sobre sistemas P2P e discutem outros projetos bem sucedidos que o empregam em serviços de descoberta de recursos.

2.1.2.2 Gerenciamento e Alocação de Recursos

Estabelecer acordos entre consumidores de recursos e fornecedores de recursos, onde os fornecedores concordam em fornecer as capacidades necessárias para a execução de tarefas dos consumidores, é o principal propósito de sistemas de gerenciamento de recursos (CZAJKOWSKI; FOSTER; KESSELMAN, 2003). Devido às características das grades de computadores, a alocação de recursos para execução de tarefas é um dos pro-

blemas mais difíceis e complicados de serem superados (MINGBIAO et al., 2007).

BUYYA; ABRAMSON; GIDDY (2000a) propuseram a utilização de modelos econômicos para guiar o gerenciamento de recursos em grades computacionais. Nesta abordagem, os usuários devem pagar pela utilização dos recursos da grade, cujos preços são guiados pela lei da oferta e da demanda. De acordo com os pesquisadores, a precificação dos recursos é uma das melhores formas de controlar e regular o acesso aos recursos.

Em grades que empregam modelos econômicos para gerenciar os recursos, existem dois participantes principais: consumidores e fornecedores de recursos (BUYYA; ABRAMSON; GIDDY, 2001). De um modo geral, consumidores objetivam executar suas tarefas com o menor custo possível e no tempo planejado, enquanto os fornecedores objetivam maximizar os lucros e utilização de seus recursos, utilizando o preço como artifício. Desta forma, dentre os fornecedores disponíveis, os consumidores têm a opção de escolher o melhor fornecedor de recursos para execução das suas tarefas.

Eles constaram que o Globus *Toolkit*[1] (FOSTER; KESSELMAN, 1997) não possuía serviços que suportassem a negociação dinâmica de recursos. Para contornar essa deficiência eles desenvolveram uma infraestrutura de *middleware*, que pode ser facilmente combinada a outros *middlewares* e assim expandir as funcionalidades neles existentes, como o Globus por exemplo. Eles a batizaram de arquitetura de grade para economia computacional (*GRid Architecture for Computational Economy - GRACE*), a qual foi inicialmente descrita em (BUYYA; ABRAMSON; GIDDY, 2000a) e em (BUYYA; ABRAMSON; GIDDY, 2001) é apresentado um caso de uso desta arquitetura. Basicamente, GRACE oferece recursos que permitem a negociação dinâmica de recursos para suportar a economia computacional.

GRACE pode ser então utilizada por sistemas de gerenciamento e escalonamento de recursos em grades de computadores que suportem a negociação dinâmica de recursos. Como exemplo, podemos citar Nimrod/G (BUYYA; ABRAMSON; GIDDY, 2000b; ABRAMSON; GIDDY; KOTLER, 2000), construído utilizando serviços do Globus. Nimrod/G suporta mecanismos de escalonamento baseados em custos e em prazo para término da execução da tarefa (*deadline*). Entende-se por custos o orçamento financeiro disponível para execução das tarefas.

Desde seu lançamento, GRACE tem recebido diversos aprimoramentos. LIU; XU

[1]Conjunto de serviços que facilitam a criação de grades computacionais e o desenvolvimento de aplicações para computação em grade.

(2007) constaram que GRACE não possuía um módulo que controlasse a flutuação de preços. A não flutuação dos preços resultaria em menores lucros para os fornecedores de recursos e em uma menor taxa de finalização de tarefas. Eles então implementaram um módulo de flutuação de preços para a GRACE, chamando-o de gerente de precificação flutuante dos recursos (*Resources Pricing Fluctuation Manager* - RPFM). Este módulo foi em seguida aperfeiçoado e reapresentado em (MA et al., 2009). Foram inseridos mecanismos de gerenciamento de confiança, onde a reputação dos integrantes passou a ser considerada no processo de escalonamento de recursos.

Em um ambiente onde os usuários pagam para executar suas tarefas, surge a necessidade de se oferecer uma boa qualidade de serviço (*Quality of Service* - QoS). Um módulo que dá suporte à QoS garantida por acordo de nível de serviço (*Service Level Agreement* SLA) foi desenvolvido por WANG et al. (2009) e acrescentado à GRACE.

Alguns modelos econômicos que podem ser utilizados na negociação de recursos, bem como estratégias de precificação, são abordados em (BUYYA et al., 2002). Dentre os modelos discutidos, *Double Auction* é o mais promissor para ambientes de grade segundo os autores. Este modelo é caracterizado por leilões, onde consumidores e fornecedores efetuam lances. Quando lances destes dois participantes são compatíveis, o negócio é fechado.

KANT; GROSU (2005) avaliaram três protocolos de alocação de recursos que utilizam *Double Auction*. Dentre os protocolos avaliados, chegaram a conclusão de que o *Continuous Double Auction* (CDA) é o melhor para consumidores e produtores, assim como permitiu a maior utilização dos recursos. TAN; GURD (2007) aperfeiçoaram o CDA, apresentando uma variação deste método chamado *Stable Continuous Double Auction*. As mudanças implementadas levaram a uma melhor eficiência econômica e de escalonamento de recursos.

OGSA

2.2 GPGPU

Esta seção aborda inicialmente os avanços das GPUs que acabaram por dar origem à GPGPU. Em seguida, comenta-se sobre os principais ambientes de desenvolvimento que facilitam a programação de aplicações GPGPU e os benefícios em utilizar esta técnica de programação. A parte final deste capítulo faz um breve relato sobre os projetos

Folding@home e SETI@*home*, os quais obtiveram significativo sucesso no emprego de GPUs na computação em grade.

2.2.1 Computação de Propósito Geral em GPUs

A rápida evolução das GPUs observada nos últimos anos (WU; LIU, 2008), motivada principalmente pelo mercado altamente competitivo e dinâmico de jogos de computadores (SINGH et al., 2008), deu origem a modernas unidades de processamento gráfico. Atualmente, estas unidades (GPUs) são computadores paralelos completamente programáveis e capazes de escalonar milhares de *threads* (ZHENG et al., 2008). Tais características permitem às GPUs atingir um pico de desempenho cuja magnitude é maior do que a de uma CPU (SINGH et al., 2008). Essas mesmas características despertaram o interesse da comunidade científica para a utilização desses processadores em computação de propósito geral (WU; LIU, 2008), dando origem à GPGPU.

A codificação de aplicações não gráficas nas primeiras GPUs era uma árdua tarefa. Para tal, era necessário conhecimento sobre a arquitetura da GPU e a codificação se dava em *assembly*, fato que só mudou a partir de 2001 quando as GPUs passaram a suportar programação (WU; LIU, 2008). A programação das GPUs permitiu aos desenvolvedores explorarem as suas unidades lógicas e aritméticas (ULA) num campo além do processamento gráfico (OWENS et al., 2007). O desenvolvimento de novas técnicas de programação para GPUs foi fundamental para que isso acontecesse. Na subseção a seguir são abordados alguns benefícios da utilização de GPGPU.

2.2.2 GPGPU: Benefícios

GPUs recentes reúnem em um único chip a capacidade de processamento correspondente a dezenas de processadores de propósito geral. O gráfico da figura 2.1 evidencia o progresso em termos de GFLOP/s entre CPUs e GPUs em apenas cinco anos de evolução.

A vantagem das GPUs frente às CPUs em termos de GFLOP/s impressiona. O mesmo ocorre quando analisamos a largura da banda de acesso à memória, cujo gráfico pode ser observado na figura 2.2. A razão desta superioridade se deve ao fato de que os processadores gráficos são especializados para a computação intensiva e paralela, concentrando o uso de transistores em unidades de execução. Ao contrário das CPUs tradicionais, onde se concentram em regiões de *cache* e controle de fluxo (NVIDIA, 2009; OWENS et al., 2007), como pode ser observado na figura 2.3.

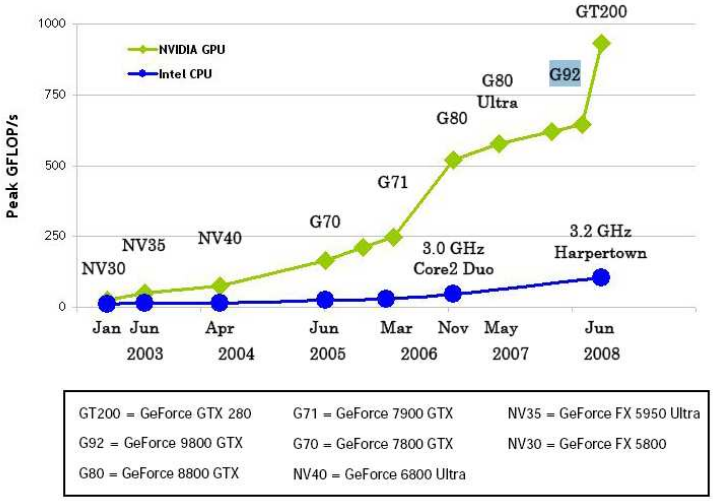

Figura 2.1: GFLOP/s: CPUs *versus* GPUs. Fonte: (NVIDIA, 2009).

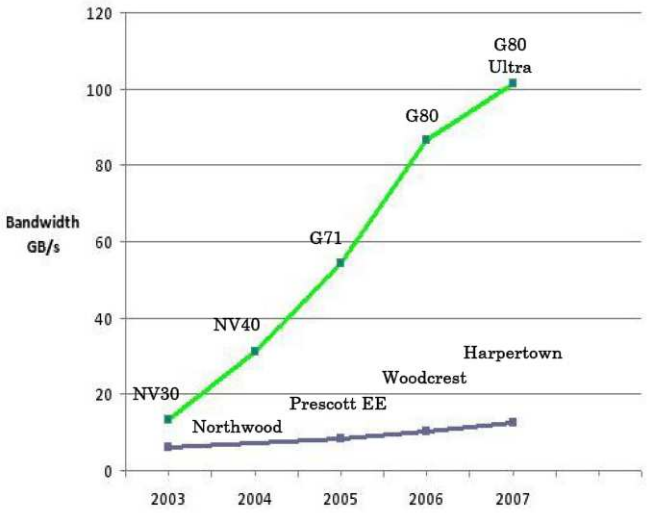

Figura 2.2: Largura de banda: CPU *versus* GPU. Fonte: (NVIDIA, 2009).

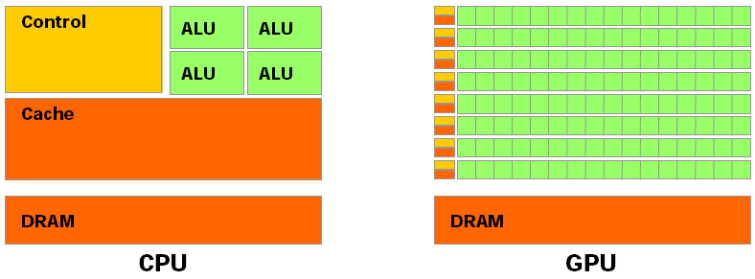

Figura 2.3: Utilização de transistores: CPU *versus* GPU. Fonte: (NVIDIA, 2009).

GPUs são especializadas para computação intensiva e altamente paralela, o que é necessário para renderização gráfica (NVIDIA, 2009) e por isso mais transistores são empregados em unidades de computação (OWENS et al., 2007). Já as CPUs foram proje-

tadas para atingirem um alto desempenho em código sequencial. Esta é a razão pela qual emprega-se mais transistores na unidade de controle de execução das CPUs (NVIDIA, 2009).

CPUs também necessitam de bastante memória *cache* para compensar a latência de acesso à memória, latência esta que é "escondida" através da computação intensa e paralela ao invés do uso de *cache* nas GPUs (NVIDIA, 2009). Por serem processadores com características SIMD, GPUs também não necessitam de unidades de controle complexas e isso justifica as pequenas regiões em amarelo na figura. Isso é exatamente o contrário do que ocorre nas CPUs, onde métodos de extração de paralelização em nível de instrução são implementados com técnicas de predição de desvios e execução fora de ordem (OWENS et al., 2007).

A partir do momento em que as GPUs passaram a suportar programação, abriu-se caminho para o seu emprego em diversas aplicações (SINGH et al., 2008). Owens (2007) afirma que a programação GPGPU não é apenas uma questão de aprender uma nova linguagem de programação, necessita-se compreender o modelo de programação para GPUs. Esse mesmo trabalho afirma que a dificuldade na programação para GPUs, devido a esse modelo diferenciado, não é o bastante para ignorar os seus benefícios.

Para que uma aplicação possa usufruir do poder computacional de uma GPU, é necessário que esta possa se adequar ao modelo de computação SIMD (SINGH et al., 2008). Desta forma, aplicações que efetuam as mesmas operações sobre uma grande quantidade de dados podem ser facilmente codificadas utilizando uma das técnicas de programação citadas na seção 2.3. A computação de uma tarefa pela GPU significa que a CPU pode ser utilizada simultaneamente. Assim, rotinas computacionais possíveis de serem computadas em SIMD podem ser executadas na GPU, enquanto as demais têm sua computação realizada pela CPU.

DATTA et al. (2008) afirma que vários problemas reais foram solucionados através de GPGPU com um desempenho superior aos últimos processadores com vários núcleos, superando inclusive o processador IBM Cell. Salienta-se que esse desempenho não se deu unicamente em mais operações realizadas por segundo, mas também em operações por segundo por dólar e em operações por *watt* consumido.

2.2.3 GPGPU: Desafios

Embora as técnicas de programação descritas anteriormente facilitem a codificação de aplicações que executarão nas GPUs, obter explorar com eficiência esses processadores não é uma tarefa fácil. Nem todas as aplicações que desempenham bem em uma CPU repetirão este feito em uma GPU.

Um dos requisitos fundamentais para que se obtenha um bom desempenho nestas arquiteturas e a possibilidade da aplicação ser codificadas empregando rotinas SIMD. No entanto, isso ainda não garante que estas aplicações terão um desempenho excepcional. Em CUDA, por exemplo, um dos maiores desafios do programador é conseguir explorar ao máximo a memória de vídeo disponível para as GPUs.

Isso é alcançado quando um conjunto de *threads* conseguem acessar endereços de memória de forma aglutinada (*coalesced*). Basicamente, *threads* devem acessar endereços de memória, onde cada endereço esteja localizado em um *bank* de memória diferente. Os requisitos para que o acesso à memória seja feito de forma aglutinada varia entre as versões do CUDA. Em (NVIDIA, 2009) este problema é abordado com mais detalhes, onde também são apresentadas abordagens para cada versão do CUDA.

Para auxiliar o programador a localizar trechos de códigos que afetam o desempenho da aplicação, ferramentas de *profiling* são bastante úteis. Estas ferramentas fornecem uma série de contadores sobre diversos processos ocorridos durante a execução do programa. Como exemplo podemos citar uma ferramenta brasileira desenvolvida por COUTINHO et al. (2009) e o NVIDIA CUDA *Visual Profiler* (NVIDIA, 2010b).

Um outro desafio relacionado à utilização das GPUs é a confiabilidade dos resultados por elas apresentados. Conforme mencionamos anteriormente, a existência de modernas GPUs é fruto do mercado de jogos de computadores. Em vista disso, o foco do desenvolvimento das GPUs é o mercado de *Desktops*, onde os requerimentos de segurança são baixos (SHEAFFER; LUEBKE; SKADRON, 2006).

GPUs podem apresentar dois tipos de erros: *soft* e *hard*. A corrupção transiente de um único *bit* em um circuito é chamado de *soft error*. Estes diferem de *hard errors* pois são aleatórios, temporários e é impossível prever quando irão acontecer (SHEAFFER; LUEBKE; SKADRON, 2006).

Gerações atuais de GPUs não oferecem suporte via *hardware* para verificação desses erros, sejam lógicos ou em células de armazenamento de memória. Consequentemente

os resultados podem ser silenciosamente corrompidos (DIMITROV; MANTOR; ZHOU, 2009).

De acordo com SHEAFFER; LUEBKE; SKADRON (2006), falhas em um único quadro não são tão preocupantes pois podem ser corrigidos no quadro seguinte. No entanto, uma falha em uma aplicação científica ou comercial invalida a computação inteira. Em vista disso, DIMITROV; MANTOR; ZHOU (2009) propuseram algumas estratégias via *software* para aumentar a confiabilidade de aplicações GPGPU.

2.3 Técnicas de Programação para GPGPU

Facilitar o desenvolvimento de aplicações GPGPU foi fundamental para promover a sua utilização. Esta seção contém uma breve abordagem sobre as três principais técnicas de programação GPGPU.

2.3.1 Nvidia CUDA

CUDA é uma arquitetura de computação paralela de propósito geral, lançada em 2006, que alavancou a utilização das GPUs Nvidia na resolução de problemas de forma mais eficiente que nas CPUs (NVIDIA, 2009). A curva de aprendizado para programação através desta arquitetura é suave para programadores familiarizados com a linguagem C. Atente que isso se refere à escrita do código, não na facilidade de se criar aplicações que explorem as GPUs ao máximo.

Através de mecanismos de abstração de *hardware*, CUDA esconde dos programadores os detalhes relacionados à GPU. Basicamente, CUDA oferece ao programador um conjunto de bibliotecas e um compilador C (nvcc) que gera código alvo tanto para o processador principal como para a GPU. Para isso, é necessário explicitar dentro do código-fonte as porções de código que devem executar na GPU, CPU ou em ambas. Esta classificação tornou necessário adicionar algumas extensões à linguagem C padrão.

2.3.1.1 Kernel

O processamento no lado da GPU é executado por um núcleo composto por blocos de *threads*, chamado *kernel*. A figura 2.4 exibe a estrutura de um *kernel*. Um bloco é composto por uma ou várias *threads*. GPUs atuais suportam até 1024 *threads* por bloco. Cada bloco é escalonado em um dos multiprocessadores das GPUs. Um multiprocessador cria, gerencia e executa de modo concorrente todas as *threads* de um bloco, com custo

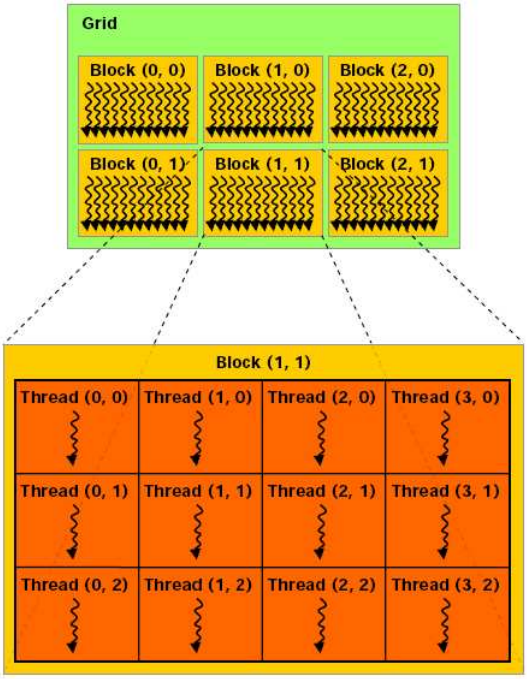

Figura 2.4: Estrutura interna de um *kernel*. Fonte: (NVIDIA, 2009).

(*overhead*) zero de escalonamento. Novos blocos são escalonados para os multiprocessadores na medida em que estes terminam a computação dos blocos que receberam.

Uma GPU pode possuir vários multiprocessadores, e cada multiprocessador possui oito processadores em seu interior. Os multiprocessadores criam grupos de 32 *threads*, chamados de *warps*. Estas *threads* são escalonadas e executadas nos oito processadores do multiprocessador. O melhor desempenho é obtido quando todas as *threads* de um *warp* percorrem o mesmo caminho de execução. Quando *threads* seguem caminhos diferentes, devido a desvios condicionais, por exemplo. Quando estes ocorrem, serializam-se as *threads* que seguem um determinado caminho e paralizam-se as outras. Isso se repete até que todas voltem a ter um caminho de execução comum.

Fica então evidente que códigos contendo muitos desvios condicionais tendem a apresentar um fraco desempenho nas GPUs. É necessário bastante esforço por parte dos usuários em escrever códigos onde as *threads* não divirjam em seus fluxos de execução.

O usuário não necessita saber quantos multiprocessadores uma GPU possui para ajudá-lo a definir o número de blocos e *threads* por bloco. É o sistema quem lida com o escalonamento dos blocos nos multiprocessadores da GPU disponível, completamente transparente para o usuário.

Assim, é possível atribuir a cada *thread* uma pequena quantidade de dados e uma

função para operar sobre esses dados. O trabalho de um *kernel* termina quando todas as *threads* de todos os blocos tiverem sido executadas, só então é permitida a execução de um novo *kernel*.

Enquanto um *kernel* está sendo executado, a CPU fica ociosa até a sua conclusão. Isto abre espaço para computação heterogênea, ou seja, enquanto a GPU estiver computando, o processador pode ser utilizado para computação de tarefas que não se adequam ao modelo de programação para GPUs.

2.3.1.2 Organização da Memória

A figura 2.5 exibe o modelo de memória em CUDA.

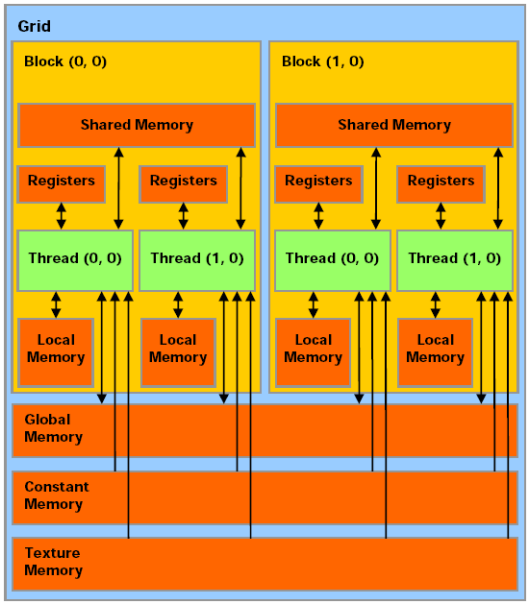

Figura 2.5: Modelo de memória em CUDA. Fonte: (NVIDIA, 2007)

Como pode ser observado, há seis tipos de memória, os quais são brevemente descritos a seguir.

- *Registers*: São os registradores do processador que podem ser lidos e escritos pelas *threads*. Cada *thread* possui um conjunto próprio de registradores. Possuem alta largura de banda e baixa latência. Para um melhor desempenho, encoraja-se que a quantidade de *threads* por bloco seja um múltiplo de 64.

- *Shared Memory*: Pode ser lida e escrita pelas *threads* de um mesmo bloco. É o tipo de memória que permite que as *threads* de um mesmo bloco cooperem na resolução de um problema. Esta memória é dividida em módulos de tamanhos

iguais, chamados *banks*. Quando são feitos acessos a endereços que não pertencem a um mesmo *bank*, a largura de banda obtida é igual ao número de *banks* acessados multiplicado pela largura de banda de cada um. Nessa situação ela é tão rápida quanto os registradores.

- *Constant Memory*: Quantidade de memória do dispositivo que só permite leitura, sendo otimizada para esta operação. Dados são colocados em uma *cache* após serem acessados pela primeira vez, o que acelera leituras futuras. Pode ser lida por todas as *threads*.

- *Global Memory*: Porção de memória acessível por todas as threads de todos os blocos de um kernel. Possui baixa largura de banda e alta latência pois os dados não são colocados em uma cache.

- *Local Memory*: Memória local de uma *thread*. Possui propriedades semelhantes a Global Memory.

- *Texture Memory*: Memória semelhante a *Constant*. Operações de leitura nessa memória podem ser mais vantajosas do que se forem realizadas nas memórias *Global* e *Constant*.

Texture, Global e *Constant Memory* podem ser lidas e escritas pelo *host* e perduram até o final da execução da aplicação.

O número de blocos que um multiprocessador pode processar por vez depende da quantidade de registradores utilizados por cada *thread* e de quanta memória do tipo *shared* cada bloco utiliza. Isso porque os registradores e a memória *shared* são divididos entre todas as *threads* dos blocos. Imagine uma situação hipotética onde há duzentos registradores disponíveis e um total de 100KB de memória *shared*. Considere ainda que os blocos possuam dez *threads* cada um. As *threads* necessitam de cinco registradores cada uma e cada bloco ocupa 50KB de memória *shared*. Neste caso, cada bloco ocupa cinquenta registradores. Se for considerado somente o total de registradores disponíveis, seria possível executar quatro blocos simultaneamente. No entanto, somente dois podem ser executados em paralelo, pois cada bloco ocupa 50KB de memória *shared* e há apenas 100KB disponíveis.

Em uma situação real, um multiprocessador pode executar até oito blocos em paralelo. Importante ressaltar que caso um bloco necessite mais recursos do que os disponíveis, o lançamento do *kernel* falhará.

2.3.1.3 Versões do CUDA

Nesta seção são abordadas algumas características das versões existentes do CUDA(NVIDIA, 2009).

- Versão 1.0
  - Máximo de 512 threads por bloco.
  - 8192 registradores por multiprocessador[2].
  - 16KB de *shared memory* para cada multiprocessador.
  - 64KB de *constant memory*.
  - Máximo de 8 blocos ativos por multiprocessador.
  - Máximo de 768 threads ativas por multiprocessador.
  - Máximo de 24 *warps* ativos por multiprocessador.
- Versão 1.1
  - Características da versão anterior.
  - Suporte a operações atômicas em dados de 32 bits da *global memory*.
- Versão 1.2
  - Características da versão anterior.
  - Suporte a operações atômicas em dados de 64 bits da *constant memory* e *shared memory*.
  - 16384 registradores por multiprocessador.
  - Máximo de 1024 *threads* ativas por multiprocessador.
- Versão 1.3

[2]Um multiprocessador possui 8 processadores.

- Características da versão anterior.
- Máximo de 32 *warps* ativos por multiprocessador.
- Suporte a variáveis de dupla precisão (double).

2.3.2 AMD Stream Computing

Assim como o Nvidia CUDA, esta plataforma também possui mecanismos que abstraem os detalhes da GPU para o programador. Os programas em execução nos *Stream Cores* são denominados *Stream Kernels*, onde a linguagem de programação Brook+ (AMD, 2009), semelhante à linguagem C, é utilizada para a sua codificação.

O modelo de programação do AMD *Stream Computing* é exibido na figura 2.6. Nesta arquitetura, cada instância de um *Stream Kernel* sendo executado em um *Thread Processor* é denominada *Thread*. Um vetor de *Threads* compõe o domínio de execução, onde estas são escalonadas para execução pelo *Stream Processor* em seus *Thread Processors*. Um novo *Stream Kernel* só será executado quando todas as *Threads* do domínio de execução tiverem sua execução terminada.

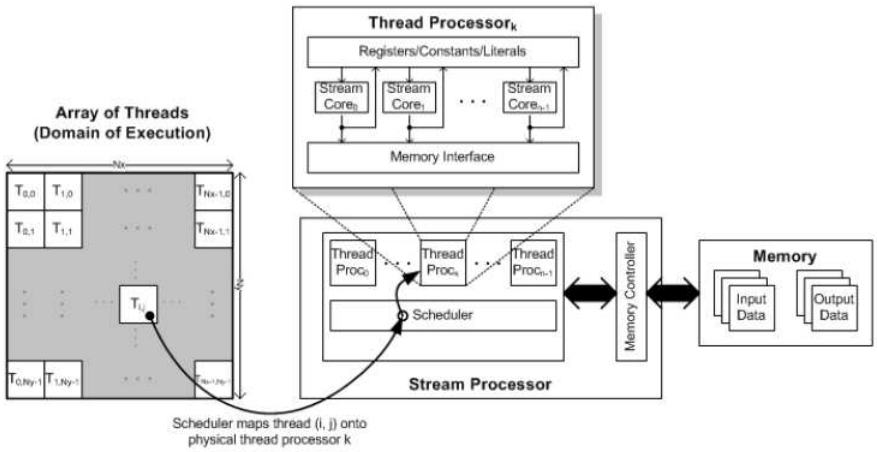

Figura 2.6: Modelo de programação simplificado do ATI *Stream Computing*. Fonte: (AMD, 2009).

2.3.3 OpenCL

Linguagem de Computação Aberta (*Open Computing Language*- OpenCL) foi criada para permitir a programação de aplicações GPGPU, de forma que possam executar nas GPUs de diferentes fabricantes. OpenCL foi o primeiro padrão aberto, livre de *royalties*, de programação de propósito geral para sistemas heterogêneos (NVIDIA, 2010c).

A unidade de execução das aplicações codificadas utilizando OpenCL é chamada de *kernel*. GPUs da Nvidia e da ATI[3] suportam a sua execução nas GPUs que fabricam. A utilização deste padrão não está restrito à geração de código para as GPUs, mas para CPUs de vários núcleos também (AMD, 2010c). Esta flexibilidade é obtida na especificação do tipo de *kernel* criado. Um *kernel* orientado à paralelização dos dados é melhor executado nas GPUs, enquanto um *kernel* projetado para paralelização de tarefas se adequa bem às CPUs (AMD, 2010c). Isso dá liberdade ao desenvolvedor para escolher a melhor arquitetura para resolver um determinado problema (HPCWIRE, 2008).

2.4 Utilização de GPGPU em Grades

Grades computacionais formadas por computadores de voluntários[4] é uma das formas mais simples, baratas e eficientes de se obter o poder computacional de um supercomputador. Como modernas GPUs estão presentes em praticamente todos os computadores comprados recentemente, tarefas de computação de alto desempenho que envolva GPGPU podem ser executadas praticamente em qualquer computador (SINGH et al., 2008).

As tarefas podem ser classificadas de acordo com a sua granularidade. Estas podem ser de grão fino ou grão grosso. Tarefas de grão fino apresentam pouco processamento a cada intervalo de comunicação pela rede, já nas de grão grosso há uma diferença maior entre o tempo gasto computando e as comunicações. SINGH et al. (2008) proporam uma arquitetura de grade computacional híbrida, onde as tarefas de grão grosso são processadas nas GPUs e as de grão fino são executadas nas CPUs dos computadores dos voluntários. As tarefas processadas envolvem comparação de genomas.

Dois grandes projetos em computação paralela e distribuída, cuja grade é formada principalmente por computadores de voluntários, viram nas GPUs uma excelente plataforma para execução dos problemas a que se proporam resolver. Esta seção faz uma breve descrição de cada um deles.

2.4.1 *Folding@home*

O poder computacional das GPUs está sendo utilizado pelo maior projeto em computação distribuída do Mundo, o *Folding@home* (STANFORD, 2010a). Esse projeto, nascido em meados do ano 2000 dentro da Universidade de Stanford, procura descobrir

[3]Marca controlada pela AMD desde 2006.

[4]Usuários que voluntariamente colocam seus computadores à disposição da grade através da Internet.

como as proteínas do corpo humano se formam, a fim de encontrar uma ligação entre as más formações destas proteínas e as doenças por elas causadas (BEBERG et al., 2009).

Qualquer pessoa pode ingressar nesse projeto e colocar o seu computador à disposição. Basta acessar o sítio do projeto e descarregar uma aplicação cliente. Há aplicações cliente especialmente desenvolvidas em Nvidia CUDA e ATI *Stream Computing* disponíveis para *download*. Quando executados, esses clientes acessam um servidor e o mesmo lhes designa alguma tarefa para computar. O cliente visa não prejudicar o desempenho do computador enquanto alguém estiver operando, só entrando em funcionamento em períodos onde ociosidade é detectada.

Nesse sítio também é possível visualizar a superioridade computacional mensurada em Teraflops (TFLOPS) dos poucos clientes GPGPU se comparados à enorme quantidade de clientes CPUs cadastrados e ativos. A figura 2.7 exibe uma tabela[5] comparativa desses clientes.

OS Type	Native TFLOPS*	x86 TFLOPS*	Active CPUs	Total CPUs
Windows	298	298	313780	3128566
Mac OS X/PowerPC	4	4	4915	134324
Mac OS X/Intel	29	29	9368	112752
Linux	82	82	48015	473801
ATI GPU	1211	1278	11874	120568
NVIDIA GPU	1171	2471	9839	171353
PLAYSTATION®3	1065	2247	37765	928604
Total	3860	6409	435556	5069968

Figura 2.7: Poder computacional dos clientes do projeto *Folding@home*. Fonte: (STANFORD, 2010b).

Atente para as colunas *Native TFLOPS* e *x86 TFLOPS*. A primeira coluna informa quantos TFLOPS possuem os clientes, a segunda adequa esse valor a *TFLOPS x86* por motivos explicados em (STANFORD, 2010c). Como exemplo da superioridade computacional das GPUs, observe as linhas *Nvidia GPU* e *Windows*. Podemos notar que há 9839 GPUs da Nvidia, cujo poder computacional combinado é de 2471 TFLOPS x86. Por outro lado, temos 313780 CPUs executando clientes no sistema operacional Windows e os mesmos oferecem apenas 298 TFLOPS.

[5]Esta tabela é continuamente atualizada e podem haver diferenças em consultas futuras. Versão obtida em 22/2/2010.

2.4.2 SETI@*home*

ver esse artigo (ANDERSON et al., 2002). A invenção do rádio despertou o interesse científico em captar sinais de vida extra-terreste. O surgimento dos primeiros rádio-telescópios e avanços na tecnologia dos rádios permitiu aos cientistas aperfeiçoar a busca por sinais de rádio de civilização alienígenas. Contudo, não se conhecem os parâmetros desses sinais e a sensibilidade de uma busca exige um significativo poder computacional, conforme descrito em (KORPELA et al., 2001).

SETI (*Search for Extraterrestrial Intelligence*) (BERKLEY, 2010) foi o primeiro projeto de computação distribuída em larga escala a entrar em operação (KORPELA et al., 2001). Tal projeto, lançado em 1999, permitiu que qualquer pessoa proprietária de um computador com conexão à Internet contribuísse para a identificação de ondas de rádio extraterrestres (BANSAL, 2001).

Assim como o *Folding@home*, SETI@home também suporta a execução de tarefas nas GPUs. No sítio do projeto pode ser descarregado um cliente específico para GPUs. Porém, somente GPUs da Nvidia são suportadas até o momento.

3 FRAMEWORK TUXUR

TUXUR é um *framework* de computação em grade que está sendo desenvolvido no Laboratório de Sistemas de Computação (LSC), localizado na Universidade Federal de Santa Maria (UFSM). Há uma dissertação em andamento descrevendo-o por completo. O trabalho a que se refere esta dissertação contribui para o seu desenvolvimento, implementando tal funcionalidade prevista em seu projeto.

Neste capítulo são abordados primeiramente os objetivos e a contribuição do *TUXUR* para a área de processamento paralelo e distribuído. Os tipos de tarefas suportados por *TUXUR* são encontrados a seguir. Dando sequência, são abordadas duas características importantes deste *framework*. As descrições sobre os principais componentes do *TUXUR* encerram este capítulo.

3.1 Objetivos e Contribuição

TUXUR foi idealizado para permitir a utilização de recursos disponíveis em uma grade computacional de forma simples, transparente e eficiente. Os pontos chave atacados para lograr êxito nestes objetivos são as tarefas que serão gerenciadas por este *framework*, e quais nós as executarão. Para isso, as aplicações de computação paralela e distribuída gerenciadas por este *framework* são automática e dinamicamente divididas em aplicações menores, de acordo com a disponibilidade de recursos e poder computacional dos computadores que compõem a grade.

Esta característica de dinamicidade em *TUXUR* não se restringe unicamente à divisão das tarefas, mas também no gerenciamento do número de computadores que integram a grade de computadores. Isso tem por objetivo permitir a inclusão/remoção de computadores a qualquer momento, seja antes ou durante a computação de alguma tarefa.

Para melhorar a eficiência no aproveitamento dos recursos de uma grade computaci-

onal, é fundamental que a diversidade de *hardware* existente nestes computadores seja explorada. *TUXUR* procura explorar a heterogeneidade dos ambientes computacionais que compõem a sua grade de computadores a fim de otimizar a sua utilização.

Estas características do *TUXUR* pretendem contribuir para as pesquisas em *frameworks* de grades de computadores. Nas subseções seguintes são discutidos os tipos de tarefas suportadas, a importância em dividi-las dinamicamente e porque explorar ambientes heterogêneos para sua execução.

3.2 Tipos de Tarefas Suportadas por Tuxur

Para que *TUXUR* logre êxito em seus objetivos é necessário definir que tipos de tarefas permitem alcançá-los. Dentre os tipos de tarefas compatíveis com estes objetivos, optou-se por incluir suporte às tarefas do tipo *Bag-of-Tasks* (saco de tarefas) e maleáveis. Características destes dois tipos de tarefas são brevemente descritas em seguida. Maiores detalhes sobre outros tipos de tarefas podem ser obtidos em FEITELSON; RUDOLPH (1996).

As tarefas classificadas como *Bag-of-Tasks* também podem ser classificadas como tarefas de carga divisível, mesmo que esta divisão não possa ser arbitrária (LIN et al., 2010). As tarefas que se enquadram nesta categoria não necessitam se comunicar enquanto executadas e não dependem uma das outras (COSTA; CIRNE; FIREMAN, 2005).

Tarefas maleáveis podem ser consideradas as mais importantes para o *TUXUR*. Entende-se por maleável uma tarefa capaz de originar subtarefas a partir dela. Isso abre a possibilidade de se criar uma subtarefa menor a partir de uma tarefa sabidamente demorada para ser resolvida. Em outras palavras, pode-se personalizar as tarefas para adequarem-se à capacidade computacional do computador escolhido para sua resolução. Foi este tipo de tarefa que motivou o desenvolvimento do *TUXUR*.

3.3 Características Importantes

Nesta sessão são abordadas duas das características importantes do *TUXUR*: Divisão Dinâmica de Tarefas e Heterogeneidade de Ambientes. O módulo implementado é beneficiado pela primeira e é fundamental para a segunda. As subsessões a seguir oferecem uma abordagem mais detalhada destas duas características.

3.3.1 Divisão Dinâmica de Tarefas

Em *TUXUR*, a expressão "Divisão Dinâmica" não se refere a uma técnica de escalonamento de tarefas, mas sim na divisão propriamente dita de uma tarefa em subtarefas. O objetivo primário desta divisão é assegurar que tarefas que demandem elevada quantidade de processamento não sobrecarreguem computadores com baixo poder computacional.

Este objetivo é satisfeito quando se criam subtarefas a partir de uma tarefa sabidamente demorada para ser computada. *TUXUR* cria uma subtarefa cujo tempo de processamento necessário para sua conclusão (quantidade de trabalho) seja compatível com o poder computacional do computador que a executará. Esta quantidade de trabalho é calculada em função de um tempo alvo de execução que toda subtarefa deve levar.

Para a primeira tarefa a ser delegada a um nó, faz-se uso de uma estimativa inicial sobre a capacidade de trabalho deste nó. Com base nesta estimativa, uma determinada tarefa é dividida e dá origem a uma subtarefa, cuja quantidade de trabalho é compatível com esta estimativa. Esta divisão não é obrigatória, pois existe a possibilidade de existirem tarefas de um tamanho adequado.

A capacidade de trabalho de um nó é recalculada a cada tarefa concluída. Esta calibragem é realizada na medida em que as mensagens de estado de um nó forem sendo recebidas. Uma das informações contidas nestas mensagens é a real capacidade de trabalho computada localmente pelo nó, após a conclusão de uma tarefa.

Desta forma, equilibra-se a quantidade de trabalho das tarefas entre os nós, objetivando um melhor aproveitamento dos computadores da grade. Outro benefício desta divisão diz respeito ao tempo final para conclusão de todas as tarefas. Ao delegar uma tarefa cuja quantidade de trabalho seja compatível com a capacidade de trabalho de um nó, contorna-se o problema de que o tempo final de processamento seja impactado pelos nós mais fracos da grade.

3.3.2 Heterogeneidade de Ambiente

O *framework* aqui discutido busca explorar novas unidades de processamento, mais especificamente unidades de processamento gráfico com capacidades GPGPU. A razão pela qual se deseja explorar estes dispositivos se deve ao fato de que grades computacionais são geralmente compostas por computadores heterogêneos. Se relacionarmos isso ao baixo custo do *hardware* gráfico compatível com GPGPU, existe uma boa probabilidade

de que alguns destes computadores possuam placas adaptadoras gráficas compatíveis com um dos ambientes mencionados na seção 2.3. Sendo assim, torna-se interessante e desejável a existência de uma ferramenta capaz de identificar e utilizar estes novos ambientes de execução, para que seus benefícios, citados na seção 2.2.1, possam ser utilizados em uma grade computacional. É neste aspecto que o módulo desenvolvido, assunto principal desta dissertação, complementa o *framework TUXUR*.

Desta forma, objetiva-se utilizar estes processadores gráficos, permitindo que as tarefas sejam executadas concomitantemente em CPUs tradicionais e nas GPUs. O aspecto da heterogeneidade de ambiente é importante pois desta forma podem ser exploradas as melhores características de cada um desses processadores na resolução de problemas. Outro benefício é que isso permite uma melhor utilização do *hardware* existente nos computadores da grade computacional.

3.4 Principais Componentes e Suas Funções

TUXUR possui cinco componentes principais: *Lançador*, *Gerente*, *Trabalhador*, *Job* e *Interface do Usuário*. Uma breve explanação sobre todos é feita a seguir.

O componente *Lançador* é o responsável pela formação da grade. Basicamente, estabelece a hierarquia de nós que hospedam o componente *Gerente* e nós que hospedam o componente *Trabalhador*. Ao final deste processo, a hierarquia formada está organizada na forma de uma árvore. Ao componente *Gerente* cabe a função de gerenciar a delegação/divisão de tarefas entre os *Trabalhadores* que possui. Já o componente *Trabalhador* simplesmente recebe e resolve as tarefas enviadas pelo seu *Gerente*, enviando-lhe os resultados encontrados e seu estado atual. O componente *Job* implementa a tarefa a ser resolvida e contém algumas rotinas computacionais pré-definidas que devem ser implementadas pelo usuário. A *Interface do Usuário* permite ao usuário realizar consultas e disparar ações na grade.

A figura 3.1 fornece uma visão geral da relação entre os componentes. Na figura apresentada, as setas indicam comunicação entre estes componentes. Embora não seja efetivamente um componente, a *Interface de Comunicação* merece destaque neste trabalho. Esta interface e os demais componentes mencionados são mais detalhadamente explanados a seguir.

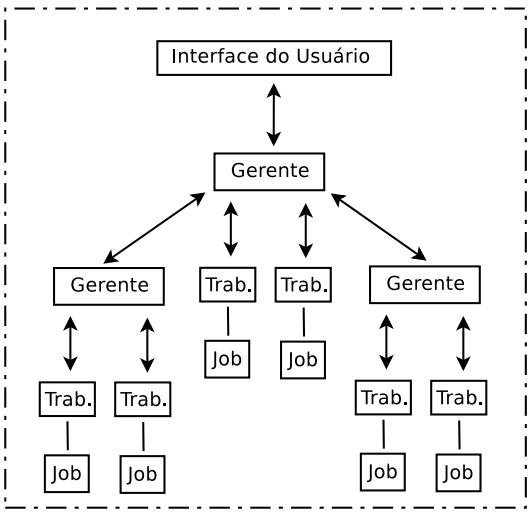

Figura 3.1: Arquitetura geral de Tuxur.

3.4.1 Gerente

O *Gerente* pode ser considerado o componente mais importante na arquitetura de *TUXUR*. Dentre as responsabilidades de um *Gerente*, podemos mencionar: dividir e delegar tarefas; compor o resultado final das tarefas; abastecer a fila de tarefas a computar de seus *Trabalhadores* e controlar seus estados. De uma forma geral, é o *Gerente* que permite ao *TUXUR* alcançar os objetivos descritos no início deste capítulo.

Os *Gerentes* implementam a mesma *Interface de Comunicação* que um *Trabalhador*, e não diferenciam seus subalternos entre *Gerentes* e *Trabalhadores*. Um *Gerente* é visto por seu superior como um *Trabalhador*, com a capacidade de trabalho agregada de todos que estão abaixo dele. Maiores detalhes sobre as funcionalidades do *Gerente* são abordados no apêndice B.

3.4.2 Trabalhador

Conforme mencionado, um *Trabalhador* é o componente de *TUXUR* que efetua a resolução das tarefas nos computadores que compõem a grade. Sua função se resume a resolver as tarefas enviadas pelo seu *Gerente* e retornar o resultado encontrado. Seu estado também é enviado a seu *Gerente*.

Mensagens de estado são enviadas por um *Trabalhador* após alguns eventos. Um evento que origina uma mensagem de estado é o término da computação de um *Job*. A capacidade de trabalho é uma das informações contidas na mensagem de estado de um *Trabalhador*. Outra informação anexada nesta mensagem é a quantidade de trabalho que

ele ainda tem por resolver. Juntas, estas duas informações correspondem ao estado de um *Trabalhador*. Com base neste estado, um *Gerente* sabe quando e qual quantidade de trabalho ele deve enviar para um determinado *Trabalhador* para que este não fique sobrecarregado ou ocioso.

Um segundo evento que origina uma mensagem de estado é a recepção de novos *Trabalhadores* por um *Gerente*. Conforme mencionado anteriormente, o superior de um determinado *Gerente* o reconhece como sendo um de seus subalternos. Ao receber um novo *Trabalhador*, a capacidade de trabalho deste *Gerente* é aumentada. O superior deste *Gerente* deve ser informado deste aumento para que possa lhe enviar mais trabalho.

Todas as tarefas esperando para serem resolvidas dentro de um *Trabalhador* são armazenadas em uma fila. É desta forma que se inibe a ociosidade de um *Trabalhador*, pois seu *Gerente* sabe a quantidade de trabalho que ele ainda tem e pode decidir qual é o melhor momento para destinar-lhe mais.

3.4.3 Interface de Comunicação

Esta interface implementa os métodos necessários para o envio e recebimento de mensagens. Toda a comunicação em *TUXUR* se dá através de vetores de *bytes*. Uma outra possibilidade de implementar a troca de mensagens seria através de métodos de serialização disponíveis em Java (JAVA, 2010), haja visto que esta é a linguagem de programação utilizada na codificação do *TUXUR*. No entanto, seria mais difícil tratar essas mensagens em componentes que não utilizem Java em sua programação. A utilização de vetores de *bytes* objetiva descomplicar a troca de mensagens entre os componentes.

3.4.4 Lançador

O *Lançador* é o componente responsável por criar a topologia dos nós que irão compor a grade de computadores. Chamaremos o processo de criação desta topologia de *lançamento* no decorrer deste trabalho. O intento de *TUXUR* é organizar os computadores em uma topologia hierárquica, em forma de árvore, onde os nós inferiores são controlados/gerenciados pelos nós superiores.

Existe uma implementação simplificada e centralizada do *Lançador*, que sozinho realiza o *lançamento* e dá origem à grade de computadores. No futuro existirá um programa distribuído ocupando seu lugar.

O *Lançador* também deverá reconhecer a forma de comunicação entre os nós. Quando

necessário, o conhecimento destas formas de comunicação permitirá a inserção de *proxies* que realizarão a tradução de protocolos.

O *Job* a ser computado está situado no primeiro *Gerente* da grade. O processamento deste *Job* tem início tão logo este *Gerente* possua um *Trabalhador*. Assim, não é necessário que toda a formação esteja concluída para que a computação de uma tarefa seja iniciada. Desta forma, o processamento de um *Job* e a inserção de novos *Trabalhadores* e *Gerentes* pelo *Lançador* ocorrem paralelamente, permitindo que o tempo para a formação da grade também seja aproveitado para a resolução das tarefas. O componente *Job* é apresentado a seguir.

3.4.5 *Job*

Um *Job* é essencialmente a representação da tarefa a ser resolvida. Basicamente, ele é composto pelas rotinas que o computam e pelos dados que são processados nestas rotinas.

A programação das rotinas computacionais relacionadas à resolução de um *Job* são de responsabilidade do usuário. Basicamente, este usuário deve programar algumas rotinas computacionais pré-definidas que permitam a um *Job* ser: descrito; inicializado, calculado, dividido em *Jobs* menores e outra que verifique se um *Job* já está completamente resolvido. Além disso, rotinas que descrevem um resultado e componham o resultado final também devem ser implementadas. Maiores detalhes sobre estas rotinas podem ser consultados no Apêndice A.

3.4.6 Interface do Usuário

Este componente é responsável pelo relacionamento com os usuários do *framework*. Ela permite que o usuário realize consultas e envie ordens. Dentre as consultas, constam: informações sobre andamento das tarefas, tarefas terminadas, soluções já encontradas e o tempo total de processamento, quantidade de nós que integram a grade, etc. Quanto às ordens, podemos citar: submissão de tarefas, inclusão/remoção de computadores e a destruição da rede de nós.

Todas as consultas são submetidas ao primeiro *Gerente*. Ordens e inserção e remoção de nós são dirigidas ao *Lançador*.

4 MÓDULO GPGPU

Este capítulo aborda assuntos relacionados ao módulo que foi acoplado ao *TUXUR*. Primeiramente comenta-se sobre os objetivos deste módulo. Em seguida aborda-se o posicionamento do módulo no *TUXUR*. Os componentes implementados são descritos em seguida. As alterações necessárias ao *TUXUR* para o seu acoplamento encerram este capítulo.

4.1 Objetivos

De maneira geral, o módulo implementado visa permitir que as tarefas a serem executadas pela grade computacional controlada pelo *TUXUR* possam utilizar GPGPU para sua resolução. Conforme mencionado na seção 2.2.1, muitos são os benefícios da utilização das GPUs em computação de propósito geral. Pretende-se então usufruir destes benefícios e do poder computacional das GPUs na área de computação em grade, com os propósitos de melhorar a utilização do *hardware* dos computadores que a compõem e agilizar a computação de tarefas.

Através deste módulo é possível multiplicar o poder computacional destes computadores sem que seja necessário um investimento financeiro para adquirir *hardware*. Placas adaptadoras gráficas lançadas nos últimos anos já contam com suporte às técnicas de programação GPGPU mencionadas na seção 2.2.1. Isto permite que computadores adquiridos recentemente usufruam destes benefícios sem custos adicionais.

4.2 Relação do módulo com o TUXUR

A figura 4.1 permite obter uma visão geral da presença do módulo implementado na hierarquia de computadores gerenciada por *TUXUR*. A função do módulo implementado é computar as tarefas delegadas por um *Gerente* e retornar o resultado encontrado. Na

visão do *Gerente*, tal módulo nada mais é do que um *Trabalhador* a sua disposição, assim como são os trabalhadores da figura 3.1. A diferença deste *Trabalhador* para os demais *Trabalhadores* da grade, é que este resolve as tarefas utilizando a GPU e a CPU, ao passo que o primeiro utiliza somente a CPU. Chamaremos este *Trabalhador* de *Trabalhador* GPGPU.

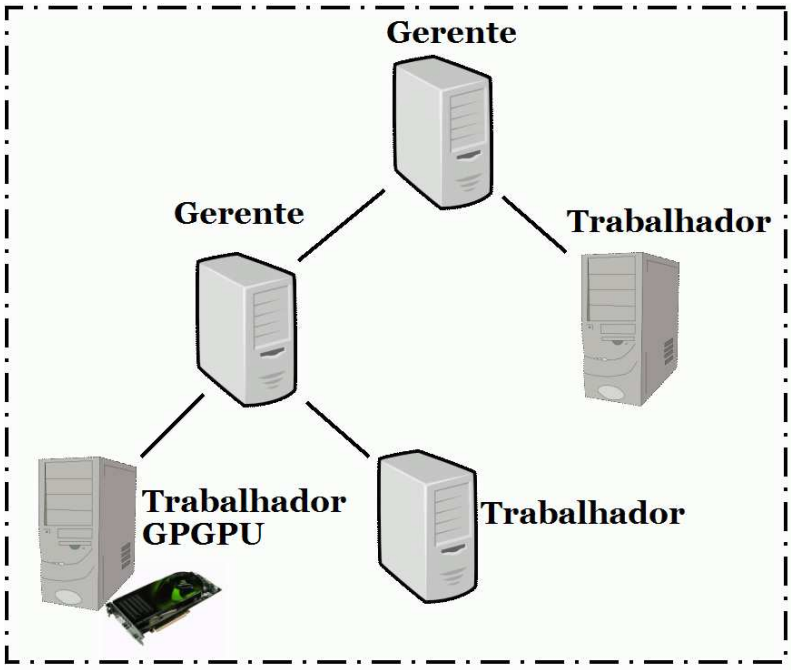

Figura 4.1: Presença do módulo na hierarquia de computadores gerenciada por *TUXUR*.

Para inserir esta nova funcionalidade no *TUXUR*, foi necessária a codificação de novos componentes em uma linguagem de programação que é suportada pelas técnicas de programação GPGPU. Escolhemos a linguagem de programação C para esta tarefa. Em relação aos componentes e rotinas implementados, teve-se que implementar todos os que envolvem a computação do *Job*, sendo portanto: o *Job* propriamente dito e suas rotinas, conforme descrito na seção 3.4.5; um *Trabalhador* completo, conforme descrito na seção 3.4.2. Uma abordagem mais detalhada destes componentes é apresentada a seguir.

4.3 Componentes Implementados

4.3.1 Trabalhador

O *Trabalhador* é o responsável pela resolução de um *Job*. Em síntese, ele retira um *Job* da sua fila de *Jobs* para resolver e envia o resultado para seu *Gerente*. Este processo se repete sucessivamente até que esta fila fique vazia, quando bloqueia à espera de novos

Jobs.

Por se tratar de um *Trabalhador* cuja única diferença para um *Trabalhador* nativo do *TUXUR* é que este resolve as tarefas executando-as na GPU, suas arquiteturas são idênticas. A figura 4.2 exibe esta arquitetura.

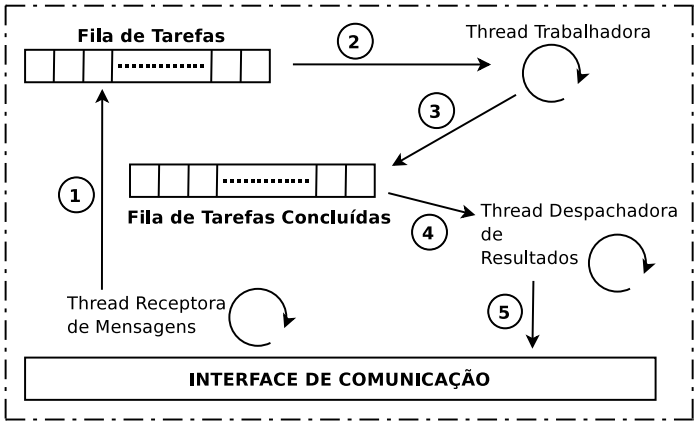

Figura 4.2: Arquitetura de um trabalhador.

As funções internas ao *Trabalhador* são executadas pelas *threads* identificadas na figura 4.2, para permitir que suas atividades sejam independentes. A *Fila de Tarefas* normalmente possui tarefas em seu interior, de forma que sempre existam tarefas para manter a *Thread Trabalhadora* ativa. Para uma melhor compreensão do funcionamento do *Trabalhador*, enumeramos as atividades na sequência em que ocorrem quando ele estiver em funcionamento. Primeiramente, uma tarefa é recebida através da *Interface de Comunicação* e inserida pela *Thread Receptora de Mensagens* na *Fila de Tarefas* (**1**). A *Thread Trabalhadora* fica bloqueada à espera de uma tarefa nesta fila. Quando houver uma tarefa, esta *Thread* desbloqueia e retira a primeira tarefa da fila (**2**). Como seu nome sugere, é esta *Thread* que efetivamente computa uma tarefa. Ao terminá-la, insere-a na *Fila das Tarefas Concluídas* (**3**). Feito isso, tenta retirar uma nova tarefa da *Fila de Tarefas*. Se houver uma tarefa nesta fila a mesma é resolvida, do contrário ela bloqueia até que uma nova tarefa seja inserida na fila.

Existe uma outra *Thread*, referenciada por *Thread Despachadora de Resultados* na figura em discussão, que fica bloqueada aguardando por tarefas na *Fila das Tarefas Concluídas*. Quando houver alguma, esta a retira (**4**) para posterior envio ao *Gerente* deste *Trabalhador* através da *Interface de Comunicação* (**5**).

A comunicação entre o *Gerente* do *TUXUR* e o *Trabalhador* GPGPU é simples de implementar pois esta se dá através de vetores de *bytes*. Estes vetores são recebidos pela

Interface de Comunicação e tratados pela *Thread Receptora de Mensagens*. O mesmo ocorre quando há o envio de mensagens ao *Gerente* por esta mesma *thread*. Uma breve descrição da *Interface de Comunicação* é apresentada a seguir.

4.3.2 Interface de Comunicação

A *Interface de Comunicação* possui conhecimento limitado sobre o conteúdo das mensagens que recebe. Basicamente, entende o suficiente para saber o que fazer quando recebe uma determinada mensagem de acordo com seu tipo.

Um *Trabalhador* está programado para receber dois tipos de mensagens e enviar outros três. Mensagens recebidas e não reconhecidas pela *Interface de Comunicação* são automaticamente descartadas. As mensagens passíveis de serem aceitas ou enviadas estão classificadas da seguinte forma:

- Recepção:
  - *BossContact*: Mensagem enviada por um *Gerente* a um *Trabalhador* avisando-o que ele é seu *Gerente*;
  - *NewJob*: Mensagem enviada por um *Gerente* contendo um *Job* a ser resolvido;
- Envio:
  - *NodeStarted*: Mensagem enviada pelo *Trabalhador* ao *Lançador* avisando-o sobre sua inicialização;
  - *JobResult*: Mensagem enviada pelo *Trabalhador* a seu *Gerente* contendo a resposta de um *Job* que foi resolvido;
  - *NodeStatus*: Mensagem enviada pelo *Trabalhador* a seu *Gerente* contendo os dados referentes ao seu estado;

Ao identificar uma mensagem como sendo do tipo *BossContact*, a *Interface de Comunicação* configura o canal de comunicação como sendo do *Gerente* deste *Trabalhador*. Desta maneira, os *Jobs* podem ser recebidos, resolvidos e seus resultados podem ser corretamente remetidos para este *Gerente*.

Jobs são recebidos através das mensagens do tipo *NewJob*. Quando mensagens deste tipo chegam, uma função pré-definida que inicializa um *Job* com a descrição do *Job*

contida na mensagem é automaticamente executada. Em seguida o *Job* é inserido na fila de tarefas do *Trabalhador*.

Em relação ao envio de mensagens, um *Trabalhador* pode enviar três tipos. A mensagem *NodeStarted* é utilizada para avisar o *Lançador* que este *Trabalhador* foi lançado com sucesso. Nesta mensagem constam a identificação do *Trabalhador* e a descrição de um canal de comunicação que foi aberto. O futuro *Gerente* deste *Trabalhador* estabelecerá uma conexão com este canal.

A mensagem do tipo *JobResult* contém a resposta de um *Job* resolvido pelo *Trabalhador*. A *Interface de Comunicação* disponibiliza uma função específica para envio de um resultado para o *Gerente* deste *Trabalhador*. Imediatamente após o envio desta mensagem, uma mensagem do tipo *NodeStatus* é enviada para informar o estado de um *Trabalhador* ao seu *Gerente*.

As atividades relacionadas ao envio e recebimento de mensagens são realizadas por *threads*. Isso permite ao *Trabalhador* continuar computando novos *Jobs* enquanto o recebimento de novos *Jobs* e/ou envio dos resultados encontrados acontece. Beneficiam-se desta divisão de funções principalmente os processadores de múltiplos núcleos, onde a execução de uma determinada *thread* não necessita ser interrompida para a execução de outra.

4.3.3 *Job*

Conforme descrito na seção 3.4.5, um *Job* representa a tarefa a ser computada por um *Trabalhador*. *TUXUR* suporta nativamente *Jobs* codificados em Java, haja visto que essa é a linguagem em que ele está sendo completamente implementado.

Atualmente existe um projeto chamado *jCuda* que oferece suporte Java ao Nvidia CUDA (JCUDA, 2010). Porém, no momento em que o trabalho aqui descrito foi iniciado tal projeto ainda não existia. Devido a isso, a linguagem C foi utilizada para codificação do componente *Job*.

Neste caso, há uma série de funções pré-definidas à espera de codificação por parte do usuário. Por se tratar do componente que implementa a tarefa a ser computada, cabe ao usuário a codificação das rotinas relacionadas à resolução do *Job*. Algumas executadas pelos *Gerentes*, outras pelos *Trabalhadores* e algumas são executadas por ambos. Quando o usuário tiver interesse em utilizar o módulo implementado, deve-se implementar dois

Jobs. O primeiro deve ser codificado em Java, para que seja suportado por *TUXUR*; o segundo deve implementar todas as rotinas utilizando uma das técnicas de programação citadas na seção 2.3, com excessão das que são executadas pelo *Gerente*. Estas rotinas são descritas no Apêndice A.

4.4 Integração deste Módulo ao Tuxur

Para inserir este módulo ao *TUXUR*, algumas modificações devem ser realizadas em seu código. Tais mudanças não devem interferir nas características que norteiam seu desenvolvimento. Dentre estas características, destacam-se prover uma plataforma de computação em grade, que faça uso dos recursos disponíveis de forma simples, transparente e eficiente.

Para tanto, o usuário não deve interferir e nem se preocupar sobre como e onde a tarefa que pretende resolver vai ser executada. O mesmo se aplica ao *Gerente*, já que para ele todos os seus *Trabalhadores* são iguais, esforçando-se apenas para atender as suas capacidades de trabalho.

O modo encontrado para inserir este módulo de forma a tornar transparente a utilização das GPUs resume-se a modificações no componente *Lançador*. A justificativa para isso é simples, ele é o componente responsável por lançar os computadores e conectar a hierarquia dos componentes do *TUXUR* sobre uma grade. A descrição dos computadores atualmente se encontra em um arquivo. Desta forma, modificou-se este arquivo, inserindo informações sobre quais computadores possuem suporte à GPGPU e sua respectiva técnica de programação. Também foi modificado o código do *Lançador* para reconhecer estas mudanças. Desta forma, ele pode lançar um binário específico para estes computadores, contendo um *Trabalhador* que está apto a computar tarefas GPGPU de uma determinada técnica de programação.

Consequentemente, é dado ao usuário o direito de escolha se deseja ou não ativar estes *Trabalhadores*. Tal escolha está diretamente ligada à existência de *Jobs* que estes *Trabalhadores* são especializados para calcular. Esta escolha se dá através de parâmetros passados para o *Lançador*. Basicamente, estes parâmetros especificam quais tipos de *Trabalhadores* devem ser lançados e o *Lançador* encarrega-se de ativá-los. O *lançamento* é então efetuado e os *Gerentes* tratam de distribuir as tarefas entre os computadores que integram a grade.

A figura 4.3 exibe uma configuração possível desta grade. Como pode ser observado, há *Trabalhadores* nativos do *TUXUR* e *Trabalhadores* que resolvem tarefas utilizando as GPUs.

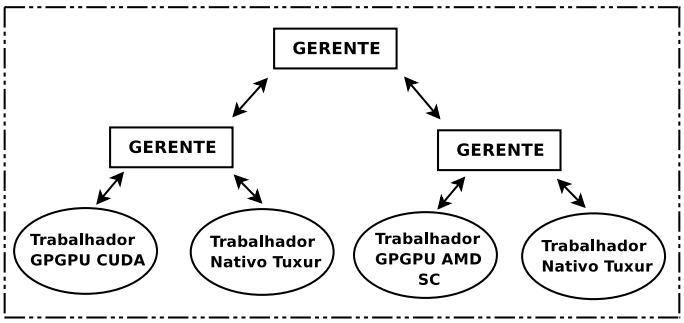

Figura 4.3: Exemplo de uma possível grade formada pelo lançador.

A avaliação do módulo é abordada no próximo capítulo.

5 AVALIAÇÃO

Este capítulo aborda assuntos relacionados à avaliação do módulo implementado. Inicialmente discorre-se sobre a metodologia empregada para a realização da avaliação, onde são definidos como e quais objetivos espera-se alcançar. Dando continuidade ao capítulo são discutidos assuntos relacionados ao *Job* desenvolvido. A discussão sobre os resultados obtidos finaliza este capítulo.

5.1 Metodologia

Para a realização da avaliação do módulo implementado, *TUXUR* já devia estar operacional. No entanto, *TUXUR* ainda não se encontra em um estágio de desenvolvimento que permitisse a sua utilização. Para contornar este problema foi implementada uma versão simplificada do *TUXUR*, a qual é descrita no Apêndice B.

No contexto do módulo descrito anteriormente, deve-se encontrar uma forma de avaliar o papel do *Trabalhador* implementado na grade gerenciada por *TUXUR*. De um modo geral, o papel deste *Trabalhador* é receber *Jobs* de um *Gerente* e enviar os resultados encontrados para ele. O que deve ser avaliado então é o quão bem este trabalhador desempenha sua função, quando comparado aos *Trabalhadores* nativos do *TUXUR*. Entende-se por bem, neste caso, a velocidade com que ele computa um *Job* que lhe foi delegado.

Desta forma, optamos por coletar algumas informações durante o processo de resolução dos *Jobs*. Dentre estas informações, constam: a quantidade de trabalho do *Job*, tempo de resolução, tempo de envio da resposta encontrada pela rede, tempo que o *Job* permaneceu na fila até ter sua resolução iniciada. Coletou-se também o tempo em que o *Trabalhador* ficou ocioso à espera de novos *Jobs* para computar.

De posse destes dados, é possível fazer uma representação gráfica das informações obtidas na resolução deste *Job*. Espera-se que tal representação ilustre a superioridade

dos *Trabalhadores* GPGPU frente aos *Trabalhadores* nativos do *TUXUR*.

Definida esta etapa, resta escolher e programar o problema a ser representado pelo *Job*. Esta escolha deve ser bem feita, pois o tempo gasto para se transmitir o *Job* e *Subjobs* gerados a partir dele pela rede, deve ser compensado pelo tempo gasto nas suas resoluções. Isto representa o grão de paralelização do *Job*. Quanto maior o tempo gasto na computação, quando comparado ao tempo gasto de gerar e enviar um *Subjob*, melhor terá sido esta escolha.

Para uma melhor avaliação, foram planejados três cenários. O primeiro envolve unicamente os *Trabalhadores* nativos do *TUXUR* na computação do *Job*. Já no segundo há o emprego dos *Trabalhadores* que computam a tarefa utilizando GPGPU. Por fim, o terceiro cenário previsto emprega unicamente *Trabalhadores* GPGPU.

Para a avaliação do módulo envolvendo estes cenários, decidimos resolver um problema que envolva processamento de imagens. Processamento de imagens geralmente envolve um grande número de operações sobre uma vasta quantidade de dados. Optamos por um problema simples, onde as mesmas instruções são executadas sobre um grande número de variáveis. O problema escolhido é a conversão de imagens no formato BMP para o formato JPEG.

5.2 Abordagem do Problema

A conversão de imagens no formato BMP para o formato JPEG exige uma grande quantidade de computação. O processo de obtenção da imagem comprimida envolve cinco etapas(WALLACE, 1992).

1. Blocos 8x8: Deve-se agrupar os *pixels* da imagem BMP em porções de 8x8 *pixels*.

2. Aplicação da FDCT: Aplicar a Transformada Bidimensional Discreta do Cosseno direta (*Forward Discrete Cosine Transform*– FDCT) em cada um dos blocos gerados anteriormente. Esta operação dá origem a novos blocos, cujos valores são os coeficientes gerados pela FDCT;

3. Quantização: Representa-se os coeficientes obtidos com um menor número de bits, obtendo assim uma compressão;

4. Codificação de Entropia: Compacta-se os coeficientes através de um algoritmo de codificação de entropia. *Huffman coding*(HUFFMAN, 1962) e *Arithmetic co-*

ding(PENNEBAKER et al., 1988) são exemplos de algoritmos que podem ser utilizados nesta etapa;

5. Imagem comprimida: Gera-se o cabeçalho da imagem e toda a informação que foi comprimida para posterior obtenção da imagem;

Não é objetivo do presente trabalho detalhar todos os processos associados às etapas citadas. Contudo, vamos nos aprofundar um pouco sobre a aplicação da FDCT.

5.2.1 Função *Forward Discrete Cosine Transform*

A maior parcela do tempo gasto para a conversão entre os formatos se deve à aplicação da FDCT. Seu objetivo consiste em converter os sinais do domínio espaço/tempo para o domínio da frequência. As informações obtidas nessa subseção foram obtidas em WALLACE (1992).

Uma imagem é um sinal no domínio do espaço. Quando aplica-se esta transformada sobre uma imagem, gera-se uma imagem no domínio da frequência. Esta imagem é caracterizada pela variação do tom entre os *pixels* vizinhos, sendo que quanto maior a intensidade desta variação, maior é a frequência da imagem.

O formato JPEG foi concebido para compressão de fotos. Fotos normalmente não apresentam uma variação brusca dos tons entre os *pixels*, ou seja, existe uma grande correlação entre os *pixels* vizinhos. Ao contrário de um texto digitalizado, onde a variação do preto (letra) para o branco (cor do papel) é frequente.

Desta forma, os *pixels* de uma foto apresentam uma baixa frequência, ou seja, não há uma grande variação entre os *pixels*. O que a FDCT faz é concentrar a energia nos coeficientes iniciais, de baixa frequência espacial, e assim obter compactação. Algoritmos que implementam esta função para aplicação em imagens geram blocos de tamanho fixo, normalmente 8x8. Neste caso, deve-se aplicar a FDCT bidimensional.

5.2.2 Etapas Transformadas em *Job*

A etapa mais dispendiosa do processo de compressão da imagem BMP para o formato JPEG é a aplicação da FDCT. Não utilizamos uma implementação otimizada desta função. Basicamente, formam-se porções de dimensões 8x8 *pixels* da imagem a ser comprimida, extraem-se os valores dos componentes RGB de cada *pixel* destas porções, e agrupam-se os valores do mesmo componente de modo a gerarem matrizes quadradas de ordem

oito. Ou seja, temos três matrizes 8x8, uma para cada componente RGB da imagem 8x8. Chamaremos de bloco o agrupamento destas três matrizes. A FDCT é aplicada nestes blocos.

O cálculo de cada bloco da imagem constitui um trabalho independente, e pode-se facilmente controlar o tamanho e número total de *Jobs* escolhendo-se o tamanho das imagens a tratar. Por exemplo, uma imagem de dimensões 2048 x 1536 pixels tem 147.456 blocos.[1]

É fácil chegar à conclusão de que para imagens com pelo menos o dobro do tamanho, o número de blocos em que a FDCT será aplicada é muito maior. Este foi o principal motivo pelo qual se decidiu por transformar esta etapa em uma aplicação paralela e que faz parte do *Job* a ser resolvido.

A quantização se resume em dividir o valor dos coeficientes gerados pela FDCT por um determinado número. O resultado é um valor menor que o inicial e que pode ser representado por um número reduzido de *bits*. Por ser aplicada em cada coeficiente, decidiu-se por acrescentar esta etapa ao *Job* implementado.

Para a geração da imagem JPEG, fizemos algumas alteração no conversor de imagens BMP para JPEG desenvolvido por Cristi Cuturicu. Este conversor pode ser obtido em CUTURICU (1999). As principais modificações realizadas no código disponibilizado foram a substituição da função FDCT e a geração de novos valores para a realização da quantização. A função FDCT foi substituída pela versão recodificada em linguagem C da implementação em Java presente em (POZZER, 2008).

5.2.3 Descrição do *Job*

O problema a ser resolvido consiste em gerar uma imagem JPEG a partir de uma imagem BMP, cujas dimensões são 10000 x 7504 *pixels*. Uma imagem com estas dimensões pode ser dividida em 1.172.500[2] imagens de dimensões 8x8 *pixels*. Este é o número de blocos a serem processados pela FDCT. Como cada bloco possui três matrizes, o total de matrizes a serem processadas nesta imagem é de 3.517.500. Para facilitar a compreensão, um bloco é representada pelo código exibido na figura 5.1, codificado em linguagem C.

Um conjunto destes blocos é a quantidade de trabalho de um determinado *Job*. O tamanho em *bytes* de cada bloco é igual a 192. Assim, basta multiplicar a quantidade de

[1] Resultado obtido através da operação $\frac{2.048 \times 1.536}{64} = 147.456$.
[2] Número obtido através da operação $\frac{10000 \times 7.504}{64} = 1.172.500$.

```
typedef struct {
    signed char R[64];
    signed char G[64];
    signed char B[64];
}bloco;
```

Figura 5.1: Representação de um Bloco.

blocos a serem enviados para se conhecer o tamanho em *bytes* dos dados que representam um *Job*. O código fonte da função FDCT pode ser visualizado na figura 5.2.

```
void fdct(signed char *b)
{
    int u, v, x, y;
    double pix;
    float df[64];
    int pos = 0;
    int idx = 0;
    for(u = 0; u < 64; u++) {
        df[u] = b[u];
    }
    for(u = 0; u < 8; u++) {
        for(v = 0; v < 8; v++, idx++) {
            pix = 0.0;
            pos = 0;
            for(x = 0; x < 8; x++) {
                for(y = 0; y < 8; y++, pos++) {
                    pix += df[pos] * cos(((2.0 * x + 1.0) *M_PI *u)
                    / 16) * cos(((2.0 * y + 1.0) *M_PI * v) / 16);
                }
            }
            mat[idx] = (pix / 4) * muv[u][v];
        }
    }
}
```

Figura 5.2: Código da Função FDCT.

A quantidade de trabalho inicial enviada para um *Trabalhador* é de 8000 blocos. Espera-se que a aplicação da FDCT sobre todos os estes blocos termine em três segundos. Em testes realizados em um processador AMD Athlon 64 X2 4400+ 2.3Ghz, constatou-se que a aplicação da FDCT por duas *threads* conseguia processar em torno de 4000 blocos em três segundos. Os processadores que são utilizados no teste possuem o dobro de núcleos e são mais avançados. Nestes processadores, quatro *threads* aplicarão a FDCT. Por conta disso, dobramos a quantidade de trabalho inicial a ser enviada para estes computadores. O mecanismo que adapta a quantidade de trabalho de acordo com a capacidade de trabalho de cada computador balanceará as futuras quantidades de trabalho a serem

enviadas.

A função FDCT recebe “n” blocos, cada bloco contendo três conjuntos de 64 valores de 1 *byte* cada como entrada (figura 5.1). Ao término da sua execução, são gerados 64 valores de 4 *bytes* para cada um dos três conjuntos. Estes então são quantizados e como resultado são gerados três conjuntos de 64 valores de 2 *bytes* cada. Na quantização são gerados muitos valores iguais a zero. Para otimizar o envio das respostas e reduzir o tráfego na rede, enviam-se apenas os “n” primeiros valores, onde a posição “n” contém o último elemento diferente de zero do conjunto de 64 valores. Estes valores formam a resposta do *Job* que será remetida ao *Gerente*. Ao chegar no *Gerente*, o mesmo insere os zeros faltantes na resposta, com base no número de valores que recebeu.

O *Job* GPGPU foi codificado utilizando o Nvidia CUDA. Sua execução exige a configuração dos parâmetros do *kernel*. Utilizando o *Cuda Occupancy Calculator* (NVIDIA, 2010d), ferramenta disponibilizada pela NVIDIA para auxiliar o programador a definir as configurações do *kernel*, chegou-se a um valor de 64 *threads* por bloco. Este foi o melhor valor encontrado para GPUs da família G80. Esta família de GPUs foi uma das primeiras a suportar GPGPU. Um *kernel* que desempenha bem nesta GPU tende a ter um desempenho superior nas famílias que a sucederam. Definiu-se então 64 *threads* por bloco e um total de “n” blocos de *threads*, onde “n” é o maior inteiro resultante da divisão do número de unidades de trabalho por 64.

O *Job* que será executado pelo *Trabalhador* nativo do *TUXUR* foi codificado utilizando a linguagem C. Os resultados obtidos, conforme a metodologia planejada e em seus respectivos cenários, são apresentados na seção a seguir.

5.3 Resultados Obtidos

Para a realização da avaliação, foi construído um pequeno *cluster* de computadores. Para isso, foram empregados cinco computadores, interligados por uma rede de 100Mbps, cujas características de *hardware* podem ser observadas na tabela 5.2.

ID	CPU	Memória	Placa de Vídeo
Gerente	AMD Turion 64 2.3Ghz	1GB DDR2 800MHz	não relevante
Desktop 1	Intel C2Q 2.4GHz	4GB DDR2 800MHz *Dual Channel*	GeForce 8800GT
Desktop 2	Intel C2Q 2.4GHz	4GB DDR2 800MHz *Dual Channel*	GeForce 220 GT
Desktop 3	Intel C2Q 2.4GHz	4GB DDR2 800MHz *Dual Channel*	GeForce 220 GT
Desktop 4	Intel C2Q 2.4GHz	4GB DDR2 800MHz *Dual Channel*	GeForce 8800 GT

Tabela 5.1: Descrição do *hardware* dos computadores utilizados na avaliação.

ID	ID dos Trabalhadores
Gerente	Nenhum
Desktop 1	CPU0; GPU0
Desktop 2	CPU1; GPU1
Desktop 3	CPU2; GPU2
Desktop 4	CPU3; GPU3

Tabela 5.2: Localização dos trabalhadores.

Todos os computadores executaram o sistema operacional Ubuntu Linux, através do *Live* CD Dotsch/UX (DOTSCH, 2010). Conforme mencionado anteriormente, a avaliação foi realizada em três cenários. Uma descrição sobre eles e os resultados encontrados são discutidos a seguir.

5.3.1 Cenário 1

O cenário 1 compreende o ambiente onde somente *Trabalhadores* nativos do *TUXUR* foram empregados na computação do *Job*. Em síntese, optou-se por criar este cenário para que os resultados aqui obtidos sirvam de base de comparação para os resultados dos demais cenários.

Por se tratarem de processadores Intel *Core 2 Quad*, a aplicação da FDCT é realizada por quatro *threads*, cada uma sendo responsável por uma fração da quantidade de trabalho recebida. Optamos então por exibir somente os resultados dos *Trabalhadores* dos computadores *Desktop 2* e *Desktop 3*. A figura 5.3 exibe o gráfico com as quantidades de trabalho dos *Jobs* delegados aos *Trabalhadores*.

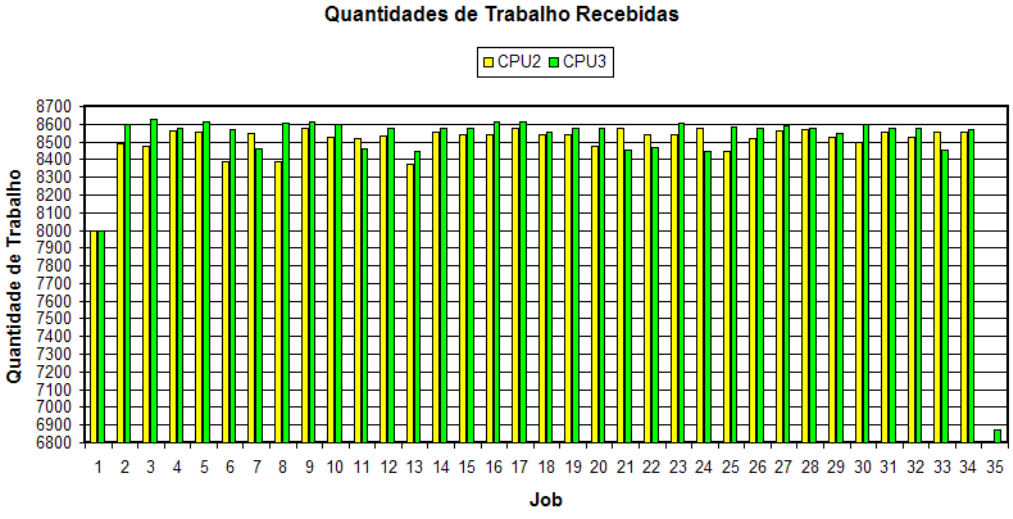

Figura 5.3: Relação das quantidades de trabalho dos *Jobs* recebidos no Cenário 1.

O primeiro *Job* de cada *Trabalhador* possui 8000 blocos, quantidade inicial de traba-

lho previamente definida e comentada anteriormente. Após a conclusão desta quantidade de trabalho, as quantidades seguintes concentraram-se na faixa de 8400 à 8600 blocos. Estas quantidades são reflexo das capacidades de trabalho dos *Trabalhadores*. A capacidade de trabalho está fortemente ligada ao tempo que um *Trabalhador* necessitou para computar um *Job* de um determinado tamanho. Estes tempos podem ser visualizados no gráfico da figura 5.4.

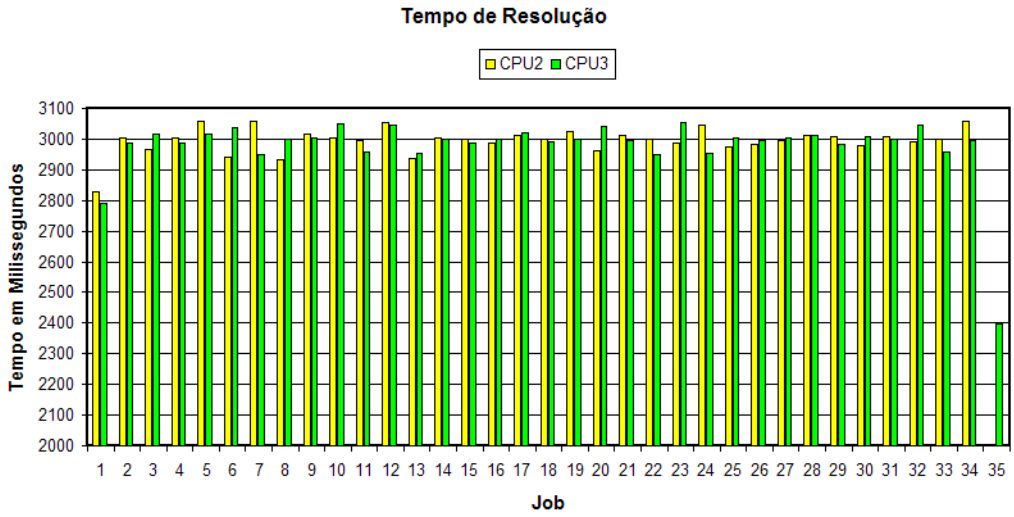

Figura 5.4: Tempo de resolução dos *Jobs* no Cenário 1.

A última coluna apresenta o menor tempo entre todos outros. Isso porque o trigésimo quinto *Job* resolvido pelo *Trabalhador* CPU3 continha os últimos blocos a serem computados. Prova disso é a sua pequena quantidade de trabalho no gráfico da figura 5.3. Por terem sido relativamente poucos blocos processados, o tempo decorrido para sua computação também foi pequeno. Neste gráfico, percebe-se uma grande concentração de colunas cujos tamanhos estão próximos da região de 3000 milissegundos. O tempo alvo para o término da execução dos *Jobs* está configurado para três segundos, razão pela qual ocorre tal concentração nesta região. Também é a justificativa para o acréscimo nas quantidades de trabalho enviadas, haja visto que 8000 blocos não foram suficientes para manter os *Trabalhadores* ocupados por três segundos.

Essa variação na quantidade de trabalho ocorre porque o *Gerente* sempre ajusta a quantidade de trabalho a ser enviada a um *Trabalhador*, com o objetivo de atender a capacidade de trabalho por ele ofertada. A capacidade de trabalho de um *Trabalhador* é o resultado da divisão da quantidade de trabalho recebida pelo tempo que seu processamento levou. Não é objetivo desta dissertação encontrar um bom método para o cálculo

deste valor, razão pela qual a capacidade de um *Trabalhador* sempre se refere ao último *Job* resolvido. As capacidades de trabalho dos *Trabalhadores* podem ser visualizadas no gráfico da figura 5.5.

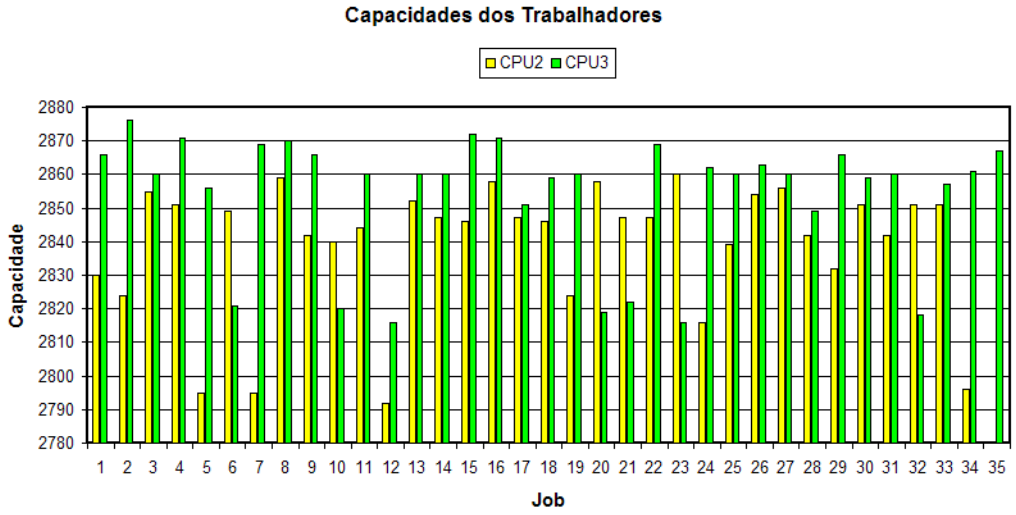

Figura 5.5: Capacidades de trabalho dos *Trabalhadores* no Cenário 1.

O tamanho em *bytes* das respostas dos *Jobs* que serão enviadas ao *Gerente* estão representados nas colunas do gráfico da figura 5.6. A explicação sobre o que origina este número de *bytes* foi apresentada na subseção 5.2.3. A figura 5.6 exibe os tamanhos em *bytes* dos dados que representam os *Jobs* recebidos. Se não tivesse sido feita a otimização para envio das respostas, estas teriam o dobro do valor de *bytes* dos *Jobs* que as originaram.

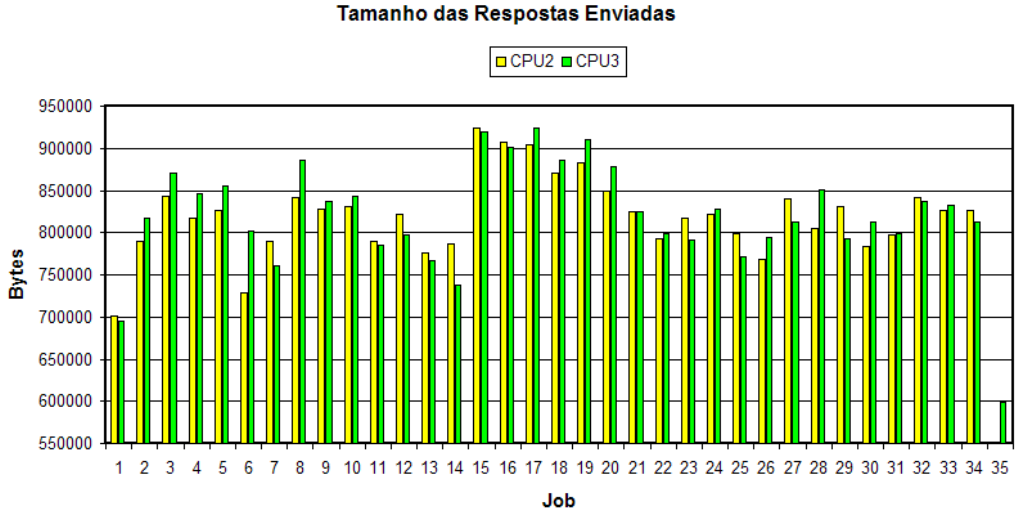

Figura 5.6: Tamanho em *bytes* das respostas dos *Jobs* do Cenário 1.

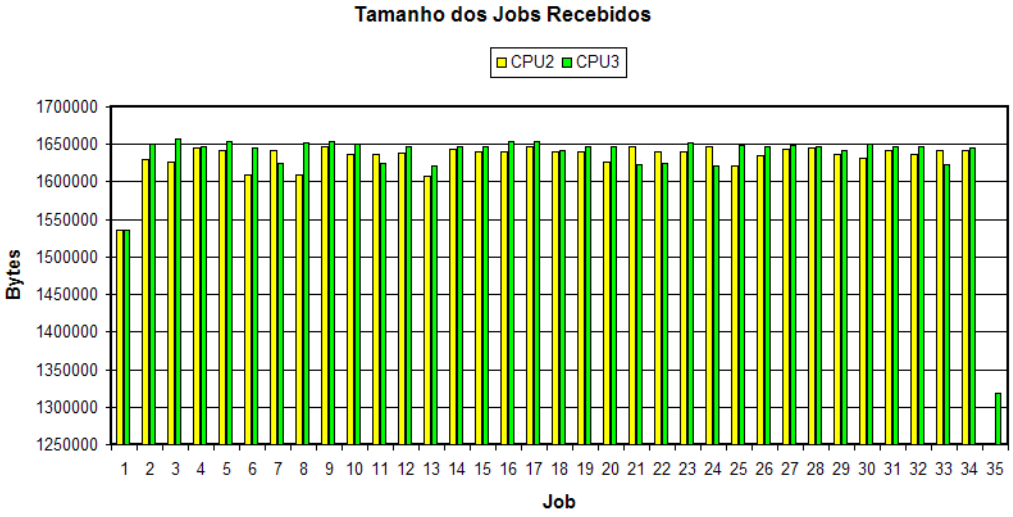

Figura 5.7: Tamanhos EM *bytes* dos *Jobs* do Cenário 1.

O *Gerente* implementado não procura manter seus *Trabalhadores* continuamente ocupados. Ou seja, um *Trabalhador* resolve um *Job*, envia o resultado pela rede e permanece ocioso até que um novo *Job* seja recebido. Como consequência desta deficiência por parte do *Gerente*, os *Trabalhadores* tornam-se frequentemente ociosos. A representação gráfica das vezes em que um *Trabalhador* ficou ocioso e o tempo que assim permaneceu está inserida na figura 5.8.

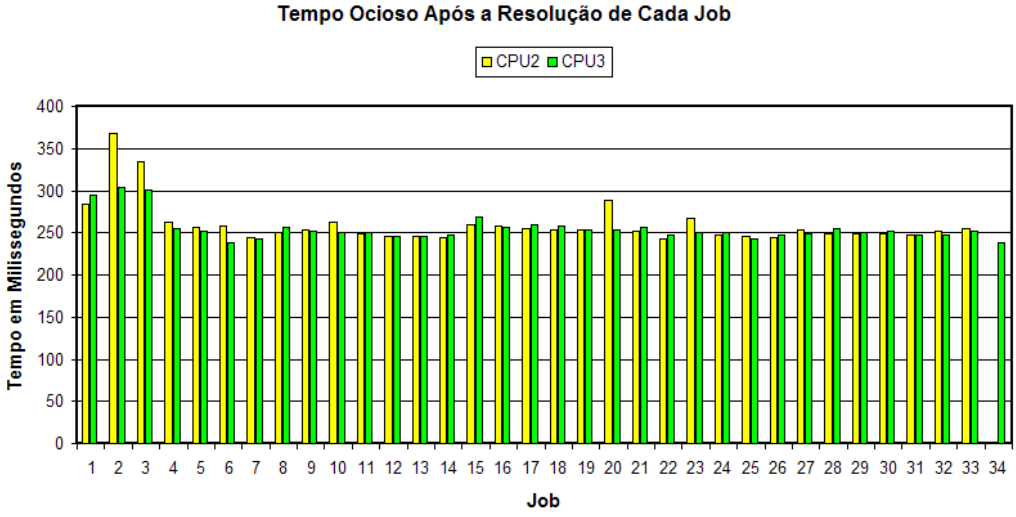

Figura 5.8: Ociosidade dos trabalhadores no Cenário 1.

Um *Trabalhador* torna-se ocioso a partir do momento em que termina a resolução de um *Job*. A ociosidade termina quando um novo *Job* tem sua resolução iniciada. Nesta avaliação não consideramos como ociosidade a espera do primeiro *Job* a ser computado.

Esta é a razão pela qual o número de vezes que um *Trabalhador* permaneceu ocioso é igual ao número de *Jobs* resolvidos menos 1. No gráfico apresentado, observa-se que o *Trabalhador* CPU3 permaneceu ocioso por quase 300 milissegundos após a resolução do primeiro *Job*.

O tempo de envio de uma resposta pela rede contribui para o aumento do tempo em que um *Trabalhador* permanecerá ocioso. Contudo, o tempo decorrido entre a chegada de uma resposta ao *Gerente* e o envio de um novo *Job* para o *Trabalhador* que a remeteu é muito mais significativo. Um *Gerente* é capaz de receber as respostas em paralelo, mas o envio de tarefas é sequencial. A ordem dos *Trabalhadores* que receberão um novo *Job* é igual à ordem dos *Trabalhadores* cujas respostas foram recebidas. Ou seja, o primeiro a ter uma resposta recebida será o primeiro a receber um novo *Job*. Quanto maior o tráfego na rede e pior a colocação do *Trabalhador* na ordem de envio de *Jobs* pelo *Gerente*, maior será o tempo ocioso deste *Trabalhador*. O gráfico da figura 5.9 exibe a porcentagem do tempo total em que os *Trabalhadores* permaneceram ociosos.

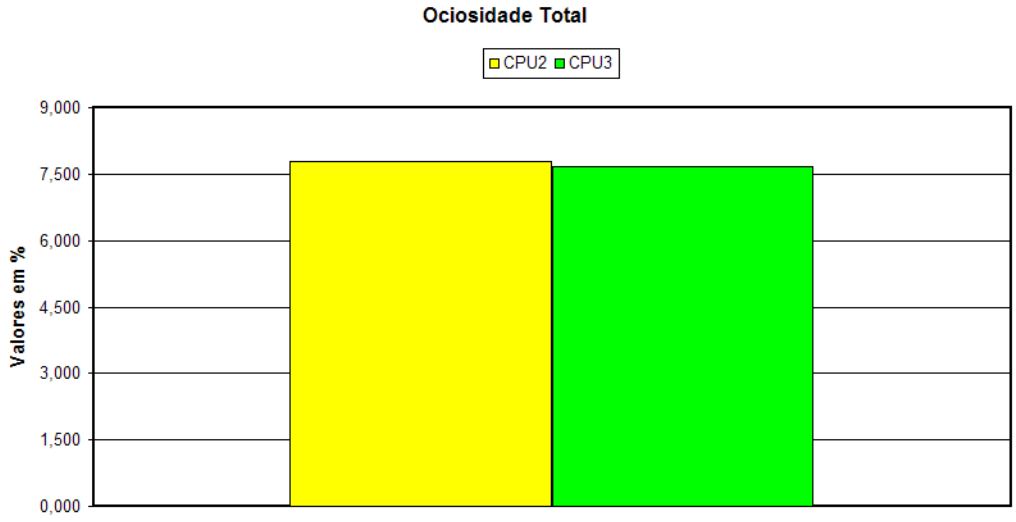

Figura 5.9: Porcentagem do tempo total em estado ocioso dos trabalhadores no Cenário 1.

Neste cenário, desde o envio do primeiro *Job* pelo *Gerente* até a recepção da última resposta, passaram-se 113.46 segundos. A imagem JPEG foi obtida 119.9 segundos após o envio do primeiro *Job*.

5.3.2 Cenário 2

A diferença básica deste cenário para o cenário 1 é a inserção do *Trabalhador* GPGPU nos computadores. Desta forma, cada computador executa um *Trabalhador* nativo do

TUXUR e um *Trabalhador* GPGPU.

A figura 5.10 exibe o gráfico com as quantidades de trabalho dos *Jobs* delegados aos *Trabalhadores* neste cenário. Atente para a escala logarítmica no gráfico. A quantidade

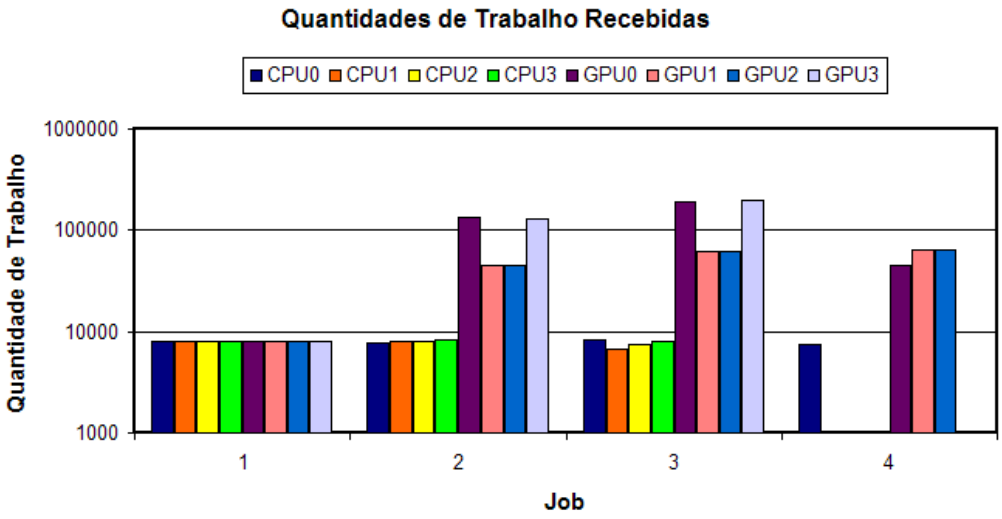

Figura 5.10: Relação das quantidades de trabalho dos *Jobs* recebidas no Cenário 2.

de trabalho do *Job* inicial foi igual para todos os *Trabalhadores*, 8000 blocos. Essa quantidade variou pouco nos *Trabalhadores* CPU, no entanto, para os *Trabalhadores* GPU estas aumentaram consideravelmente. O quarto *Job* resolvido pelo *Trabalhador* GPU3 continha os blocos restantes para serem computados, razão pela qual ouve a redução na quantidade de trabalho enviada a este *Trabalhador* quando comparada ao *Job* anterior. A diferença entre as quantidades de trabalho recebidas pelos *Trabalhadores* CPU e o *Trabalhadores* GPU surpreende. Isso evidencia a maior capacidade de processamento das GPUs frente às CPUs na resolução destes *Jobs*.

Os tempos de resolução dos *Jobs* podem ser visualizados no gráfico da figura 5.11. Os primeiros *Jobs* foram resolvidos pelos *Trabalhadores* GPU em um tempo muito menor quando comparado ao tempo que os *Trabalhadores* CPU necessitaram. Isso fez com que o *Gerente* enviasse muito mais trabalho no *Job* seguinte, objetivando alcançar o tempo alvo de execução de três segundos. Nota-se através do gráfico que o *Gerente* alcançou esse objetivo, pois os tempos convergiram para a região dos 3000 milissegundos. É possível visualizar no gráfico que o *Trabalhador* CPU0 necessitou de mais de três segundos para resolver seu terceiro *Job*, o que foi prontamente corrigido pelo *Gerente*, enviando menos trabalho para o quarto *Job*, como pode ser observado no gráfico da figura 5.10.

Um detalhe importante no gráfico apresentado são os tempos que os *Trabalhadores*

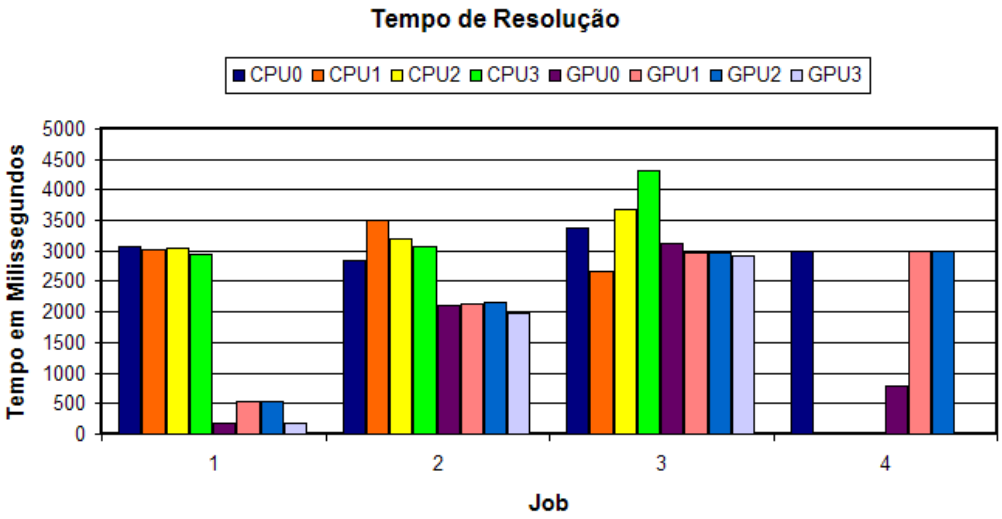

Figura 5.11: Tempo de resolução dos *Jobs* no Cenário 2.

CPU necessitaram para calcular o primeiro *Job*. Estes sofreram um acréscimo frente aos tempos do Cenário 1 (figura 5.4), devido a dois motivos.

O primeiro está na recepção dos *Jobs* para os *Trabalhadores* GPU. Embora os cálculos dos *Jobs* nestes *Trabalhadores* ocorram na GPU, as tarefas que antecedem este cálculo são executadas pela CPU. Isso fez com que os cinco *Trabalhadores* disputassem os quatro núcleos do processador por um dado período de tempo. A consequência foi o aumento no tempo da resolução dos *Jobs* dos *Trabalhadores* CPU.

O segundo motivo é semelhante ao primeiro. Neste caso são as tarefas relacionadas ao envio da resposta encontrada pelo *Trabalhadores* GPU que ocupam a CPU por um determinado tempo. Os *Trabalhadores* GPU terminaram o cálculo do primeiro *Job* recebido muito antes do que os *Trabalhadores* CPU, razão pela qual houve esta segunda disputa pelos núcleos do processador enquanto os *Trabalhadores* CPU ainda resolviam seu primeiro *Job*.

As capacidades de trabalho dos *Trabalhadores* podem ser visualizadas no gráfico da figura 5.12. Novamente, atente para a escala logarítmica. Nesta figura fica evidente a superioridade em termos de capacidade de trabalho dos *Trabalhadores* GPU frente aos *Trabalhadores* CPU. Quanto maior a capacidade de trabalho de um *Trabalhador*, maior será a quantidade de trabalho dos *Jobs* a ele delegados. Como efeito imediato deste aumento na quantidade de trabalho, os tamanhos dos *Jobs* recebidos e das respostas por eles gerados também crescem. Estes tamanhos podem ser visualizados nos gráficos das figuras 5.13 e 5.14, respectivamente, ambos em escala logarítmica.

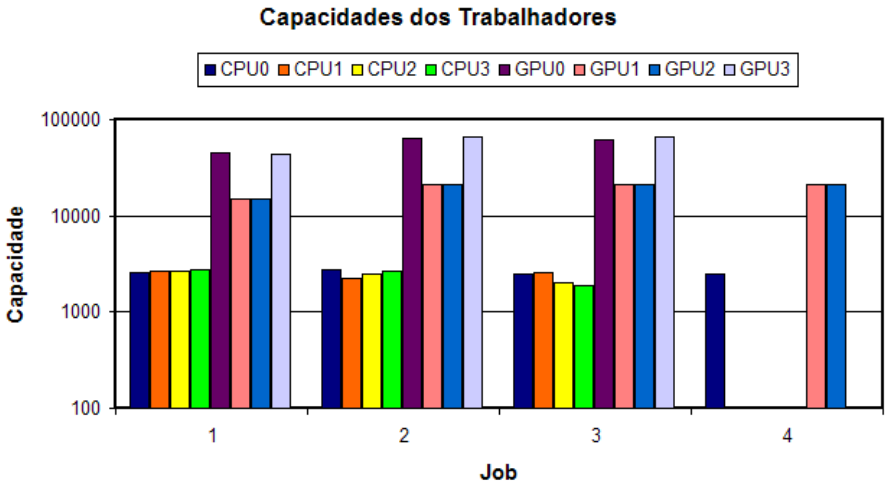

Figura 5.12: Capacidades de trabalho dos *Trabalhadores* no Cenário 2.

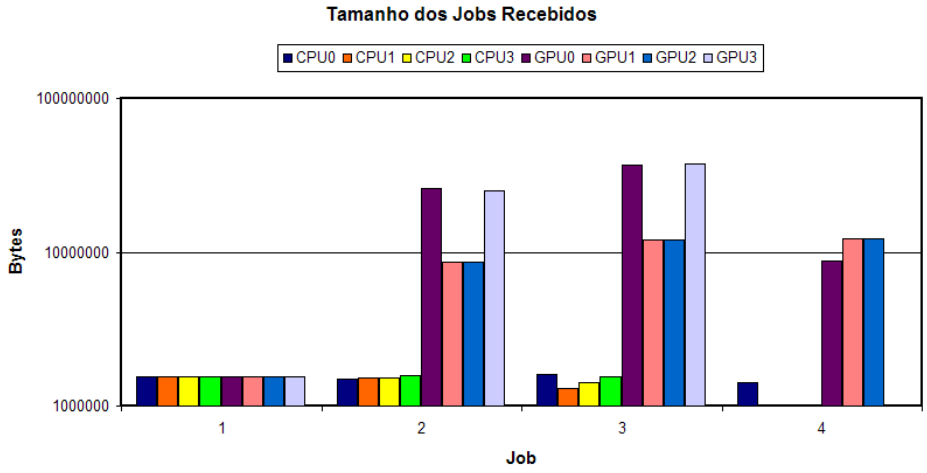

Figura 5.13: Tamanhos em *bytes* dos *Jobs* do Cenário 2.

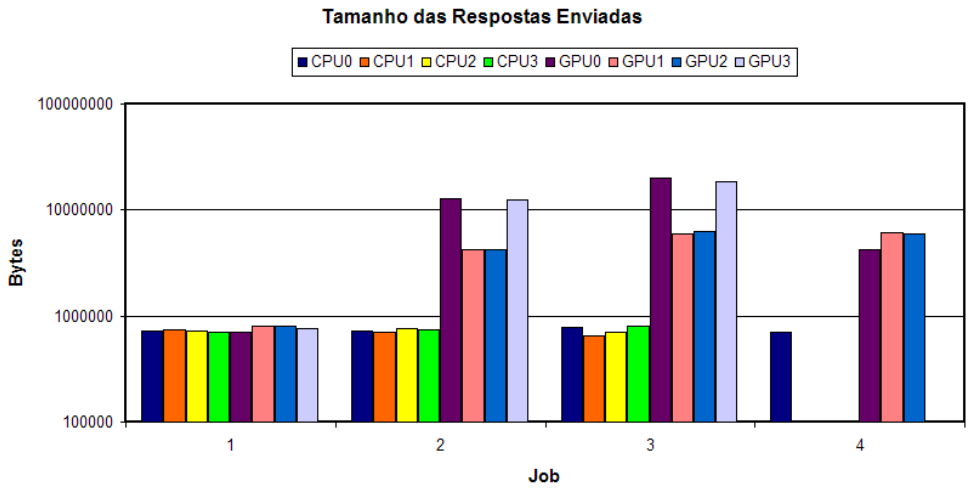

Figura 5.14: Tamanho em *bytes* das respostas dos *Jobs* do Cenário 2.

Como consequência dos aumentos dos tamanhos dos *Jobs* e das suas respostas, há um sensível aumento no tráfego da rede. O *Gerente* necessita então de mais tempo para transmitir um *Job* pela rede. Os *Trabalhadores* também necessitam de mais tempo para enviarem os resultados encontrados. Por esses motivos, a ociosidade dos *Trabalhadores* se agrava bastante. A representação gráfica das vezes que um *Trabalhador* se tornou-se ocioso e o tempo que assim permaneceu está inserida no gráfico da figura 5.15. O gráfico da figura 5.16 exibe a porcentagem do tempo total em que os *Trabalhadores* permaneceram ociosos.

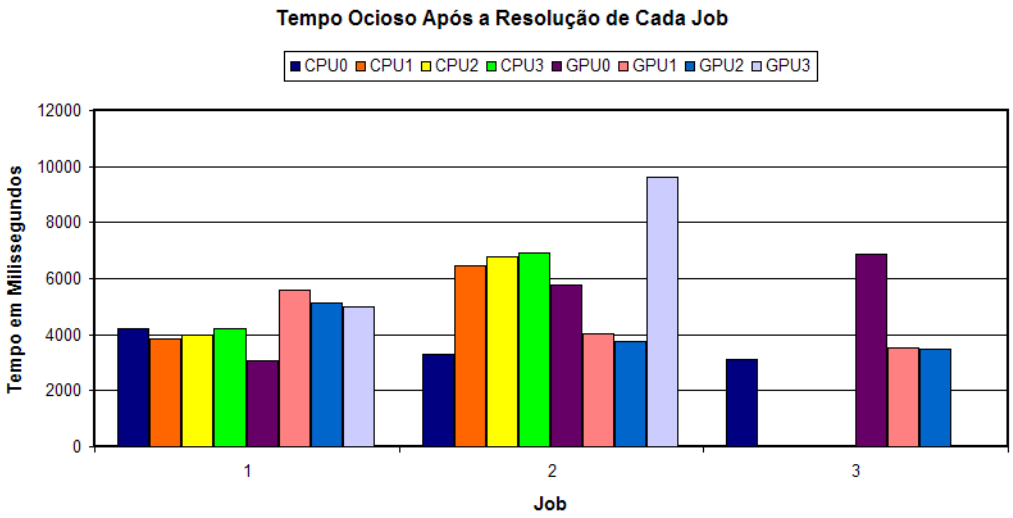

Figura 5.15: Ociosidade dos trabalhadores no Cenário 2.

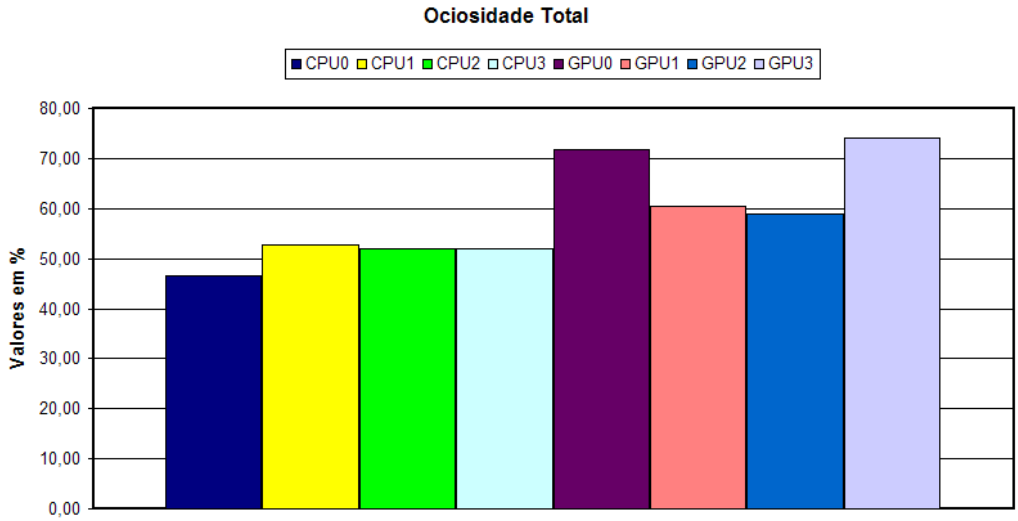

Figura 5.16: Porcentagem do tempo total em estado ocioso dos trabalhadores no Cenário 2.

Neste cenário, desde o envio do primeiro *Job* pelo *Gerente* até a recepção da última

resposta, passaram-se 23.84 segundos. A imagem JPEG foi obtida 29.03 segundos após o envio do primeiro *Job*. Comparado aos tempos obtidos no cenário 1, obtivemos uma redução de 79% para o primeiro tempo e de 75.8% para o segundo.

5.3.3 Cenário 3

O último cenário emprega unicamente *Trabalhadores* GPU, sendo um por computador.. A figura 5.17 exibe as quantidades de trabalho delegadas aos *Trabalhadores*.

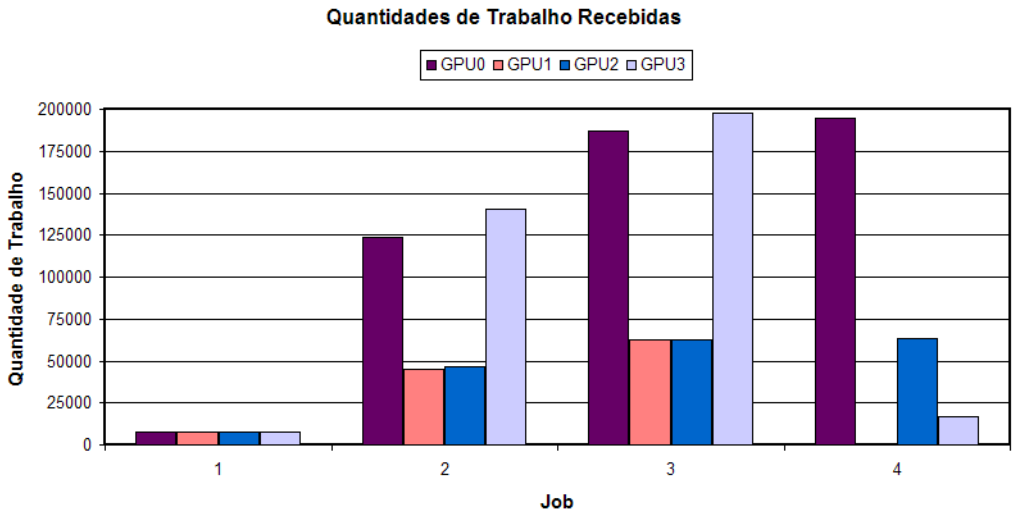

Figura 5.17: Relação das quantidades de trabalho dos *Jobs* recebidas no Cenário 3.

Os *Trabalhadores* GPU0 e GPU3 possuem a mesma placa de vídeo, motivo pelo qual recebem quantidades de trabalho semelhantes. O mesmo pode ser dito em relação aos *Trabalhadores* GPU1 e GPU2. Nesse gráfico pode ser observada uma expressiva redução na quantidade de trabalho do quarto *Job* resolvido pelo *Trabalhador* GPU3. Esse *Job* continha a quantidade de trabalho restante para ser computada.

Os tempos necessários para encontrar as respostas dos *Jobs* recebidos podem ser encontrados no gráfico da figura 5.18. As quantidades de trabalho dos *Jobs* resolvidos aumentaram para que o tempo alvo de 3000 milissegundos fosse alcançado. Ao final do terceiro *Job* resolvido todos os *Trabalhadores* se aproximaram desta marca. O tempo de resolução do quarto *Job* pelo *Trabalhador* GPU3 apresentou essa queda pois a quantidade de trabalho desse *Job* era bastante reduzida.

As quantidades de trabalho dos *Trabalhadores* GPU0 e GPU3 superaram com uma boa vantagem as quantidades recebidas pelos demais *Trabalhadores*. Isso permite concluir que as placas de vídeo utilizadas por estes *Trabalhadores* são superiores às utilizadas

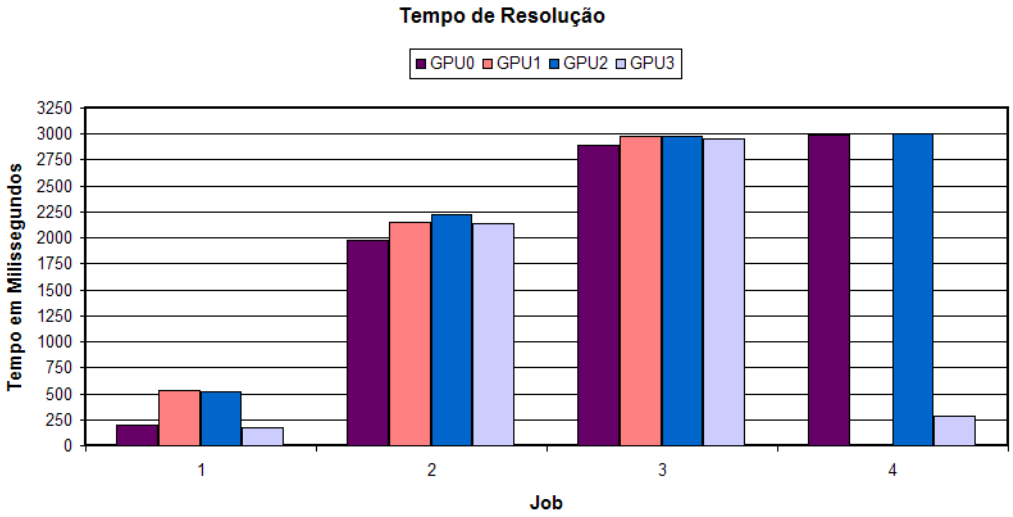

Figura 5.18: Tempo de resolução dos *Jobs* no Cenário 3.

pelos *Trabalhadores* GPU1 e GPU2. As capacidades de trabalho podem ser observadas no gráfico da figura 5.19.

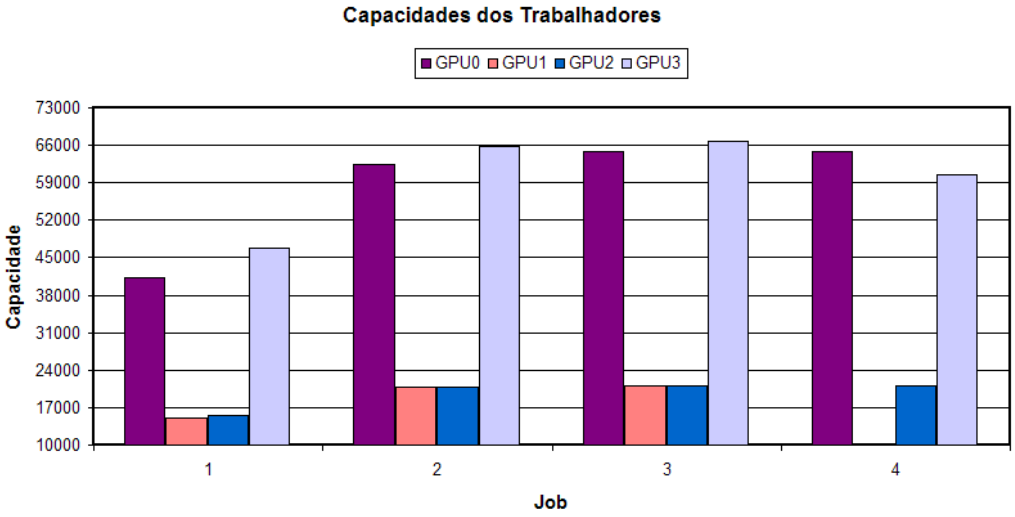

Figura 5.19: Capacidades de trabalho dos *Trabalhadores* no Cenário 3.

O gráfico da figura 5.20 exibe os tamanhos das respostas dos *Jobs* resolvidos pelos *Trabalhadores*. Os tamanhos dos *Jobs* que originaram essas respostas podem ser observados no gráfico da figura 5.20. Atente para a escala logarítmica.

Assim como ocorreu no Cenário 2, o tráfego da rede afeta sobremaneira o tempo que os *Trabalhadores* permanecem ociosos. O número de vezes que um *Trabalhador* se tornou ocioso e a duração desta ociosidade estão representados no gráfico da figura 5.22. O gráfico da figura 5.23 exibe a porcentagem do tempo total em que os *Trabalhadores*

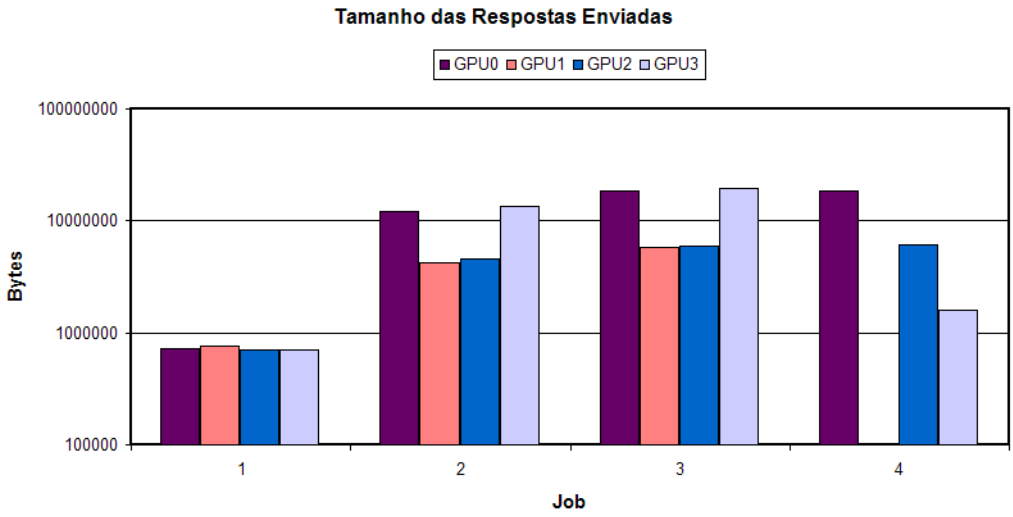

Figura 5.20: Tamanho em *bytes* das respostas dos *Jobs* do Cenário 3.

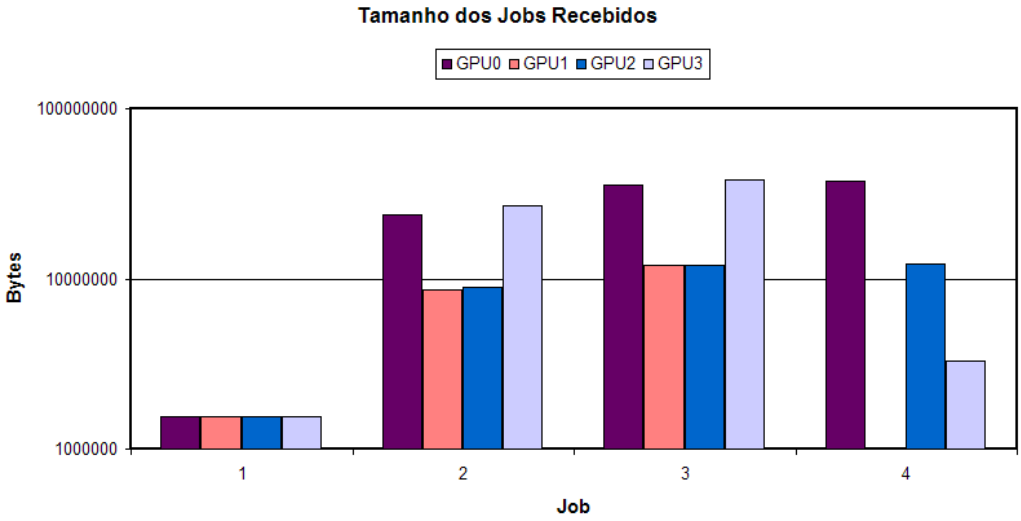

Figura 5.21: Tamanhos em *bytes* dos *Jobs* do Cenário 3.

permaneceram ociosos. Comparado ao gráfico 5.16 do Cenário 2, as porcentagens não apresentaram variações significativas.

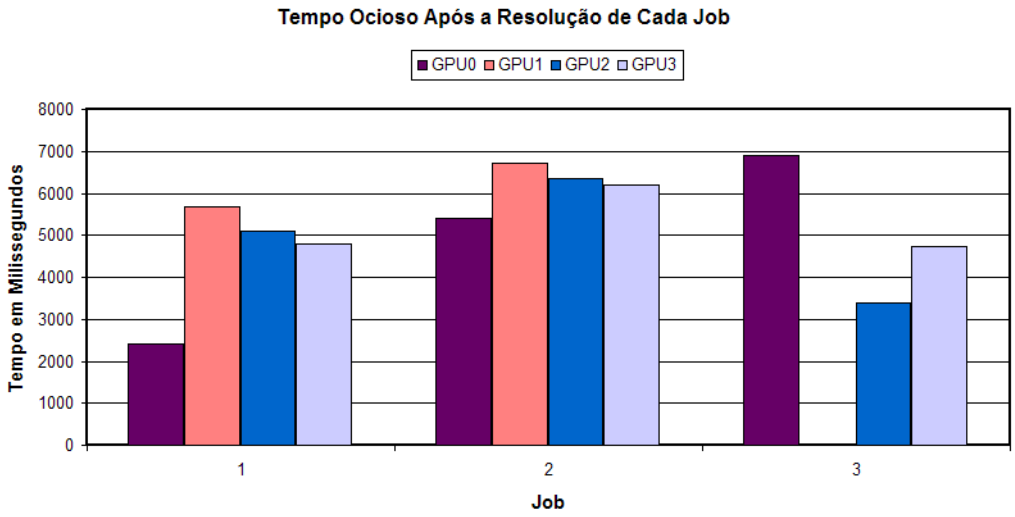

Figura 5.22: Ociosidade dos trabalhadores no Cenário 3.

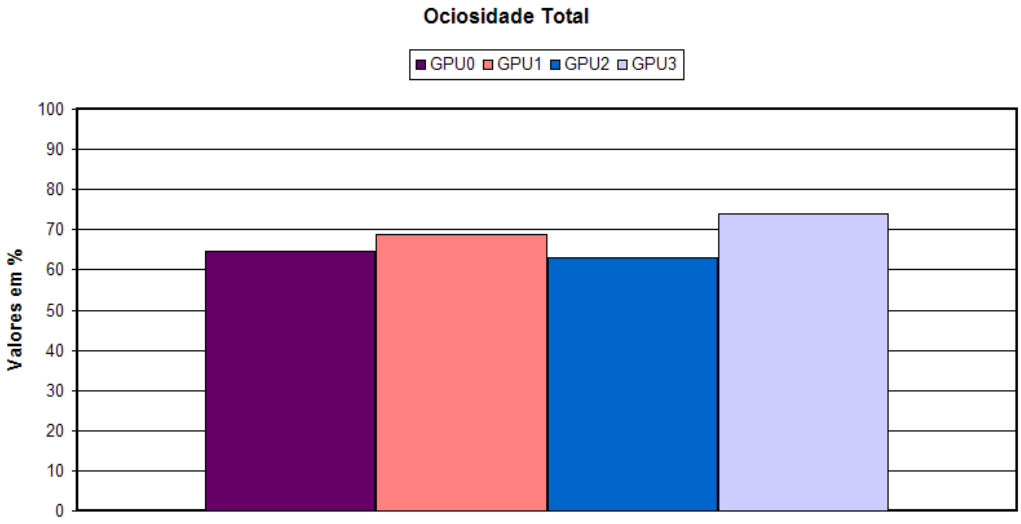

Figura 5.23: Porcentagem do tempo total em estado ocioso dos trabalhadores no Cenário 3.

Neste cenário, desde o envio do primeiro *Job* pelo *Gerente* até a recepção da última resposta, passaram-se 25.21 segundos, contra 23.84 segundos no cenário 2. A imagem JPEG foi obtida 31.03 segundos após o envio do primeiro *Job*, enquanto que no cenário 2 este tempo foi de 29.03 segundo. A utilização simultânea de *Trabalhadores* GPU e CPU obteve uma pequena vantagem na avaliação realizada.

Em todos os cenários apresentados os *Trabalhadores* apresentaram ociosidade. No cenário 1 a ociosidade foi pequena pois os *Jobs* e suas respostas possuíam poucos *bytes* de tamanho. Isso tornava rápido o envio de novos *Jobs* e ainda mais rápido o envio das respostas encontradas.

6 CONSIDERAÇÕES FINAIS

A utilização de grades de computadores na computação de tarefas científicas vem se intensificando desde o início deste século. Linguagens de programação que facilitem a codificação de aplicações que utilizarão as GPUs como processadores começaram a aparecer somente a partir de 2006. Atualmente estas linguagens já estão bem desenvolvidas e diversos trabalhos conseguiram notáveis ganhos de desempenho ao empregar a GPU em suas resoluções.

O presente trabalho buscou contribuir nesse sentido, apresentando um componente que permite a utilização de GPGPU em grades de computadores. Mais especificamente, objetiva dar suporte à execução de tarefas utilizando as GPUs dos computadores que formam a grade gerenciada pelo *TUXUR*.

O componente teve seu funcionamento validado nos dois últimos cenários de teste apresentados. Os resultados encontrados mostram a importância do uso de GPGPU em tarefas onde o modelo de programação SIMD pode ser aplicado.

O emprego dos *Trabalhadores* GPU reduziu em 75.8% o tempo total necessário para completar a tarefa. Essa melhora foi alcançada sem investimento financeiro na aquisição de *hardware*, pois as placas de vídeo já estavam disponíveis. O único investimento foi o tempo empregado na codificação das rotinas que resolvem os *Jobs* utilizando GPGPU. A expressiva redução do tempo de conclusão da tarefa compensou o esforço de programação.

Os trabalhos futuros estão relacionados às melhorias que serão futuramente implementadas no *Trabalhador* nativo do *TUXUR* e no *Gerente*. Dentre elas, pretende-se implementar segurança e autenticação na comunicação entre o *Trabalhador* e seu *Gerente*; tolerância a falhas, recepção de novos tipos de mensagens pela *Interface de Comunicação*, como por exemplo: mensagem que instrui o *Trabalhador* a ser desligado no exato

momento em que a recebe ou após ter computado todos os seus *Jobs*, troca de *Gerente* a quem deve responder, entre outras. A disponibilidade de um *Gerente* com todas as funcionalidades previstas no *TUXUR* permitirá o desenvolvimento mais completo do módulo de utilização de GPUs ora apresentado, principalmente no que diz respeito à redução da ociosidade dos computadores.

REFERÊNCIAS

ABRAMSON, D.; GIDDY, J.; KOTLER, L. High performance parametric modeling with Nimrod/G: killer application for the global grid? In: IPDPS 2000: PROCEEDINGS OF 14TH INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, 2000. **Anais...** IEEE, 2000. p.520–528.

AMD. **AMD Stream Computing Overview**. Disponível em <http://developer.amd.com/gpu_assets/Stream_Computing_User_Guide.pdf>. Acessado em: 22 de Fevereiro de 2010.

AMD. **Advanced Micro Devices Inc**. Disponível em <http://www.amd.com>. Acessado em: 20 de Julho de 2010.

AMD. **ATI Stream Computing Tecnical Overview**. Disponível em <http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf>. Acessado em: 22 de Julho de 2010.

AMD. **An Introduction to OpenCL™**. Disponível em <http://www.amd.com/us/products/technologies/stream-technology/opencl/pages/opencl-intro.aspx>. Acessado em: 20 de Julho de 2010.

ANDERSON, D. P.; COBB, J.; KORPELA, E.; LEBOFSKY, M.; WERTHIMER, D. SETI@home: an experiment in public-resource computing. **ACM Commun**, New York, NY, USA, v.45, p.56–61, November 2002.

BALI, V.; RATHORE, R.; SIROHI, A.; VERMA, P. Information Technology Architectures for Grid Computing and Applications. In: ICCGI '09: FOURTH INTERNATIONAL MULTI-CONFERENCE ON COMPUTING IN THE GLOBAL INFORMATION TECHNOLOGY, 2009. **Anais...** IEEE Computer Society, 2009. p.52–56.

BANSAL, R. ET or EC? [SETI@Home project]. **IEEE Antennas and Propagation Magazine**, [S.l.], v.43, n.4, p.118, Agosto 2001.

BEBERG, A.; ENSIGN, D.; JAYACHANDRAN, G.; KHALIQ, S.; PANDE, V. Folding@home: lessons from eight years of volunteer distributed computing. In: IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL DISTRIBUTED PROCESSING, 2009. **Anais...** IEEE, 2009. p.1–8.

BERKLEY. **SETI@Home Site Project**. Disponível em <http://setiathome.ssl.berkeley.edu/>. Acessado em: 20 de Julho de 2010.

BUYYA, R.; ABRAMSON, D.; GIDDY, J. An Economy Driven Resource Management Architecture for Global Computational Power Grids. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, 2000. **Anais...** [S.l.: s.n.], 2000.

BUYYA, R.; ABRAMSON, D.; GIDDY, J. Nimrod/G: an architecture for a resource management and scheduling system in a global computational grid. In: FOURTH INTERNATIONAL CONFERENCE/EXHIBITION ON HIGH PERFORMANCE COMPUTING IN THE ASIA-PACIFIC REGION, 2000, Beijing, China. **Anais...** IEEE, 2000. v.1, p.283–289.

BUYYA, R.; ABRAMSON, D.; GIDDY, J. A case for economy grid architecture for service oriented grid computing. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, 2001. **Anais...** IEEE, 2001. p.776–790.

BUYYA, R.; ABRAMSON, D.; GIDDY, J.; STOCKINGER, H. Economic models for resource management and scheduling in Grid computing. **Concurrency and Computation: Practice and Experience**, New York, NY, USA, v.14, n.13-15, p.1507–1542, 2002.

CHEN, J.; LU, B. Load Balancing Oriented Economic Grid Resource Scheduling. In: PACIIA '08: PACIFIC-ASIA WORKSHOP ON COMPUTATIONAL INTELLIGENCE AND INDUSTRIAL APPLICATION, 2008. **Anais...** IEEE Computer Society, 2008. v.2, p.813–817.

CIRNE, W.; SANTOS-NETO, E. **Grids Computacionais**: Da Computação de Alto De-

sempenho a Serviços sob Demanda. Simpósio Brasileiro de Redes de Computadores, 2005, Fortaleza/CE. Minucursos - SBRC 2005.

COKUSLU, D.; HAMEURLAIN, A.; ERCIYES, K. Grid resource discovery based on web services. In: ICITST 2009: PROCEEDINGS OF INTERNATIONAL CONFERENCE FOR INTERNET TECHNOLOGY AND SECURED TRANSACTIONS, 2009. **Anais...** IEEE, 2009. p.1–6.

COSTA, L.; CIRNE, W.; FIREMAN, D. Converting space shared resources into intermittent resources for use in bag-of-tasks grids. In: SBAC-PAD 2005: 17TH INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 2005. **Anais...** IEEE Computer Society, 2005. p.243–250.

COUTINHO, B.; TEODORO, G.; OLIVEIRA, R.; NETO, D.; FERREIRA, R. Profiling General Purpose GPU Applications. In: SBAC-PAD '09: 21ST INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 2009, São Paulo, SP, Brasil. **Anais...** IEEE Computer Society, 2009. p.11–18.

CUTURICU, C. **JPEG Encoder**. Disponível em: `<http://www.wotsit.org/list.asp?search=jpeg\&button=GO!>`. Selecionar o link Download da linha "JPEG Compression and the JPEG File format & sourcecode". Acessado em: 10 de Agosto de 2010.

CZAJKOWSKI, K.; FOSTER, I.; KESSELMAN, C. Resource and Service Management. In: FOSTER, I.; KESSELMAN, C. (Ed.). **The Grid 2**: blueprint for a new computing infrastructure. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. p.259–283.

DATTA, K.; MURPHY, M.; VOLKOV, V.; WILLIAMS, S.; CARTER, J.; OLIKER, L.; PATTERSON, D.; SHALF, J.; YELICK, K. Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In: SC 2008: INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2008. **Anais...** IEEE, 2008. p.1–12.

DIMITROV, M.; MANTOR, M.; ZHOU, H. Understanding software approaches for GPGPU reliability. In: GPGPU-2: PROCEEDINGS OF 2ND WORKSHOP ON GE-

NERAL PURPOSE PROCESSING ON GRAPHICS PROCESSING UNITS, 2009, New York, NY, USA. **Anais...** ACM, 2009. p.94–104.

DOTSCH. **Dotsch/UX - A USB/Diskless/Harddisk BOINC Ubuntu Linux Distribution**. Disponível em <http://www.dotsch.de/boinc/Dotsch_UX.html>. Acessado em: 20 de Julho de 2010.

FATTAHI, S.; CHARKARI, N. Distributed resource discovery in grid with efficient range query. In: CSICC: 14TH INTERNATIONAL CSI COMPUTER CONFERENCE, 2009. **Anais...** IEEE, 2009. p.335–340.

FEITELSON, D. G.; RUDOLPH, L. (Ed.). **Toward convergence in job schedulers for parallel supercomputers**. [S.l.]: Springer Berlin / Heidelberg, 1996. p.1–26. (Lecture Notes in Computer Science, v.1162).

FOSTER, I. The anatomy of the grid: enabling scalable virtual organizations. In: FIRST IEEE/ACM INTERNATIONAL SUPERCOMPUTER SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, 2001. **Anais...** IEEE, 2001. p.6–7.

FOSTER, I.; KESSELMAN, C. Globus: a metacomputing infrastructure toolkit. **International Journal of Supercomputer Applications**, [S.l.], v.11, p.115–128, 1997.

FOSTER, I.; KESSELMAN, C. (Ed.). **The Grid**: blueprint for a new computing infrastructure. [S.l.]: MORGAN-KAUFMANN, 1999. 259-278p.

FOSTER, I.; KESSELMAN, C. (Ed.). **The Grid 2**: blueprint for a new computing infrastructure. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Open Grid Services Architecture. In: FOSTER, I.; KESSELMAN, C. (Ed.). **The Grid 2**: blueprint for a new computing infrastructure. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. p.215–257.

FOSTER, I.; KISHIMOTO, H.; SAVVA, A.; BERRYM, D.; DJAOUI, A. et al. **The Open Grid Services Architecture, Version 1.5**. Disponível em: <http://www.ogf.org/documents/GFD.80.pdf>. Acessado em: 15 de Dezembro de 2010.

HPCWIRE. **AMD Adopts OpenCL 1.0 Specification**. Disponível em <http://www.hpcwire.com/topic/developertools/AMD-Adopts-OpenCL-10-/Specification-35773904.html>. Acessado em: 20 de Julho de 2010.

HUFFMAN, D. A Method for the Construction of Minimum-Redundancy Codes. **Proceedings of the IRE**, [S.l.], v.40, n.9, p.1098–1101, Sept 1962.

IAMNITCHI, A.; FOSTER, I. A peer-to-peer approach to resource location in Grid environments. In: NABRZYSKI, J.; SCHOPF, J. M.; WEGLARZ, J. (Ed.). **Grid resource management**. Norwell, MA, USA: Kluwer Academic Publishers, 2004. p.413–429.

JAVA. **Java Programming Language**. Disponível em <http://www.java.com>. Acessado em: 20 de Julho de 2010.

JCUDA. **Java bindings for CUDA**. Disponível em <http://www.jcuda.org>. Acessado em: 18 de Julho de 2010.

KANT, U.; GROSU, D. Double auction protocols for resource allocation in grids. In: ITCC 2005: INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY: CODING AND COMPUTING, 2005. **Anais...** IEEE Computer Society, 2005. v.1, p.366–371.

KORPELA, E.; WERTHIMER, D.; ANDERSON, D.; COBB, J.; LEBOISKY, M. SETI@home-massively distributed computing for SETI. **Computing in Science Engineering**, [S.l.], v.3, n.1, p.78–83, Janeiro-Fevereiro 2001.

KRAUTER, K.; BUYYA, R.; MAHESWARAN, M. A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing. **Software: Practice and Experience**, [S.l.], v.32, p.135–164, 2001.

LI, J.; DESHPANDE, A.; SRINIVASAN, J.; MA, X. Energy and performance impact of aggressive volunteer computing with multi-core computers. In: MASCOTS '09: IEEE INTERNATIONAL SYMPOSIUM ON MODELING, ANALYSIS SIMULATION OF COMPUTER AND TELECOMMUNICATION SYSTEMS, 2009. **Anais...** IEEE, 2009. p.1–10.

LIN, X.; MAMAT, A.; LU, Y.; DEOGUN, J.; GODDARD, S. Real-time scheduling of divisible loads in cluster computing environments. **Journal of Parallel and Distributed Computing**, [S.l.], v.70, n.3, p.296–308, 2010. Disponível em <http://www.sciencedirect.com/science/article/B6WKJ-4XVK3X1-1/2/d12cdfbe934c218abdad379ed876813b>. Acessado em: 22 de Fevereiro de 2010.

LIU, G.; XU, Y. A New Grid Economy Architecture with Resources Pricing Fluctuation Module. In: GCC 2007: SIXTH INTERNATIONAL CONFERENCE ON GRID AND COOPERATIVE COMPUTING, 2007. **Anais...** IEEE Computer Society, 2007. p.701–704.

LU, B.; MA, J. A Strategy for Resource Allocation and Pricing in Grid Environment Based on Economic Model. In: CINC '09: INTERNATIONAL CONFERENCE ON COMPUTATIONAL INTELLIGENCE AND NATURAL COMPUTING, 2009. **Anais...** IEEE Computer Society, 2009. v.1, p.221–224.

MA, S.; LIU, G.; XU, Y.; PAN, Z.; WANG, J.; JI, J. A New Resource Pricing Fluctuation Model with Trust System in Grid Economy. In: MUE '09: THIRD INTERNATIONAL CONFERENCE ON MULTIMEDIA AND UBIQUITOUS ENGINEERING, 2009. **Anais...** IEEE Computer Society, 2009. p.300–305.

MA, S.; SUN, X.; GUO, Z. A resource discovery mechanism integrating P2P and Grid. In: ICCSIT: 3RD IEEE INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE AND INFORMATION TECHNOLOGY, 2010. **Anais...** IEEE, 2010. v.7, p.336–339.

MINGBIAO, L.; XINMENG, C.; FENG, J.; ZHIJIAN, W. Optimization of Grid Resource Allocation Combining Fuzzy Theory with Generalized Assignment Problem. In: GCC 2007: SIXTH INTERNATIONAL CONFERENCE ON GRID AND COOPERATIVE COMPUTING, 2007. **Anais...** IEEE Computer Society, 2007. p.142–146.

MOLLICK, E. Establishing Moore's Law. **IEEE Annals of the History of Computing**, [S.l.], v.28, n.3, p.62–75, Julho-Setembro 2006.

NVIDIA. **CUDA 1.1 Programming Guide**. Disponível em <http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf>. Acessado em: 20 de Julho de 2010.

NVIDIA. **CUDA 2.3 Programming Guide**. Disponível em <http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf>. Acessado em: 20 de Julho de 2010.

NVIDIA. **Nvidia Corporation**. Disponível em <http://www.nvidia.com>. Acessado em: 20 de Julho de 2010.

NVIDIA. **Cuda Visual Profiler**. Disponível em <http://developer.nvidia.com/object/cuda_3_2_downloads.html>. Acessado em: 14 de Dezembro de 2010.

NVIDIA. **OpenCL™**: the open standard for parallel programming of gpus and multi-core cpus. Disponível em <http://www.amd.com/us/products/technologies/stream-technology/opencl/Pages/opencl.aspx>. Acessado em: 20 de Julho de 2010.

NVIDIA. **CUDA Occupancy Calculator**. Disponível em <http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls>. Acessado em: 20 de Julho de 2010.

OWENS, J. D.; LUEBKE, D.; GOVINDARAJU, N.; HARRIS, M.; KRüGER, J.; LEFOHN, A. E.; PURCELL, T. J. A Survey of General-Purpose Computation on Graphics Hardware. **Computer Graphics Forum**, [S.l.], v.26, n.1, p.80–113, 2007. Disponível em <http://www.blackwell-synergy.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x>. Acessado em: 20 de Julho de 2010.

PENNEBAKER, W.; MITCHELL; J.L. et al. Arithmetic Coding Articles. **IBM J. Res. Dev**, [S.l.], v.32, n.6, p.717–774, 1988.

POZZER, C. T. **Computação Gráfica - Imagem**. Material da Disciplina Computação Gráfica do curso de Ciência da Computação da UFSM. Disponível em: <http://www-usr.inf.ufsm.br/~pozzer/disciplinas/cg_3_imagem.pdf>. Acessado em: 8 de Agosto de 2010.

RANJAN, R.; CHAN, L.; HARWOOD, A.; KARUNASEKERA, S.; BUYYA, R. Decentralised Resource Discovery Service for Large Scale Federated Grids. In: IEEE IN-

TERNATIONAL CONFERENCE ON E-SCIENCE AND GRID COMPUTING, 2007. **Anais...** IEEE Computer Society, 2007. p.379–387.

SCHMIDT, C.; PARASHAR, M. Flexible Information Discovery in Decentralized Distributed Systems. In: HPDC '03: PROCEEDINGS OF THE 12TH IEEE INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, 2003, Washington, DC, USA. **Anais...** IEEE Computer Society, 2003. p.226.

SHEAFFER, J. W.; LUEBKE, D. P.; SKADRON, K. The visual vulnerability spectrum: characterizing architectural vulnerability for graphics hardware. In: ACM SIGGRAPH/EUROGRAPHICS SYMPOSIUM ON GRAPHICS HARDWARE, 21., 2006, New York, NY, USA. **Proceedings...** ACM, 2006. p.9–16.

SINGH, A.; CHEN, C.; LIU, W.; MITCHELL, W.; SCHMIDT, B. A Hybrid Computational Grid Architecture for Comparative Genomics. **IEEE Transactions on Information Technology in Biomedicine**, [S.l.], v.12, n.2, p.218–225, Março 2008.

STANFORD. **Folding@home Project**. Disponível em <http://folding.stanford.edu>. Acessado em: 22 de Fevereiro de 2010.

STANFORD. **Client statistics by OS**. Disponível em <http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats>. Acessado em: 22 de Fevereiro de 2010.

STANFORD. **Folding@home FLOP FAQ**. Disponível em <http://folding.stanford.edu/English/FAQ-flops>. Acessado em: 22 de Fevereiro de 2010.

SUCIU, A.; POTOLEA, R. A taxonomy for grid applications. In: AQTR 2008: IEEE INTERNATIONAL CONFERENCE ON AUTOMATION, QUALITY AND TESTING, ROBOTICS, 2008. **Anais...** IEEE, 2008. v.3, p.365–368.

TAN, Z.; GURD, J. Market-based grid resource allocation using a stable continuous double auction. In: IEEE/ACM INTERNATIONAL CONFERENCE ON GRID COMPUTING, 8., 2007. **Anais...** IEEE, 2007. p.283–290.

TRUNFIO, P.; TALIA, D.; PAPADAKIS, H.; FRAGOPOULOU, P.; MORDACCHINI, M.; PENNANEN, M.; POPOV, K.; VLASSOV, V.; HARIDI, S. Peer-to-Peer resource

discovery in Grids: models and systems. **Future Gener. Comput. Syst.**, Amsterdam, The Netherlands, v.23, n.7, p.864–878, Aug. 2007.

WALLACE, G. K. The JPEG still picture compression standard. **IEEE Transactions on Consumer Electronics**, [S.l.], v.38, n.1, p.18–34, Feb 1992.

WANG, X.; XU, Y.; LIU, G.; PAN, Z.; WANG, J. A New QoS Guaranteed Mechanism Based on SLA in Grid Market. In: PACCS '09: PACIFIC-ASIA CONFERENCE ON CIRCUITS, COMMUNICATIONS AND SYSTEMS, 2009. **Anais...** IEEE Computer Society, 2009. p.197–201.

WU, E.; LIU, Y. Emerging technology about GPGPU. In: APCCAS 2008: IEEE ASIA-PACIFIC CONFERENCE ON CIRCUITS AND SYSTEMS, 2008. **Anais...** IEEE, 2008. p.618–622.

ZHENG, J.; CHEN, W.; CHEN, Y.; ZHANG, Y.; ZHAO, Y.; ZHENG, W. Parallelization of spectral clustering algorithm on multi-core processors and GPGPU. In: ACSAC 2008: 13TH IEEE ASIA-PACIFIC COMPUTER SYSTEMS ARCHITECTURE CONFERENCE, 2008. **Anais...** IEEE, 2008. p.1–8.

APÊNDICE A FUNÇÕES PRÉ-DEFINIDAS

Este capítulo faz uma descrição das funções pré-definidas a serem implementadas pelo usuário. Todas estas funções são relacionadas ao *Job*. A codificações destas funções permite ao *Gerente* e ao *Trabalhador* desenvolverem suas atividades no *TUXUR*. Para facilitar a compreensão, estas foram classificadas de acordo com o componente que as executa.

A.1 Executadas pelo *Gerente*

A.1.1 job breakJob()

Função que divide um *Job*. Dividir um *Job* resume-se em originar um *Subjob*, cuja quantidade de trabalho deve ser próxima da quantidade de trabalho suportada pelo *Trabalhador* ao qual será transmitido. Normalmente cria-se um *Subjob* quando diminuímos a quantidade de dados utilizados na computação de um Job. Durante a resolução de um *Job*, geralmente ocorre intenso processamento sobre um conjunto de variáveis. Diminuindo a quantidade de variáveis, reduz-se a quantidade de trabalho necessária para sua computação.

Como argumentos, recebe uma estrutura de dados contendo o *Job* que se deseja dividir e a quantidade de trabalho (*workload*) pretendida para o *Subjob* a ser gerado. Uma nova estrutura de dados contendo o *Subjob* é retornada pela função. Este *Job* deve então ser transformado em um vetor *bytes* para ser transmitido pela rede. A função que realiza este procedimento é abordada a seguir.

A.1.2 byte[] description()

Esta função é responsável por retornar a descrição de um *Job*. Descrever um *Job* significa descrever os dados que serão utilizados pela rotina que o calcula. Entende-se

por descrever a inserção destes dados em um vetor de *bytes*. Este vetor é necessário pois conforme explanado anteriormente, toda a comunicação se dá através destes vetores.

A função recebe dois argumentos. O primeiro é uma estrutura de dados de um *Job* que se deseja obter a descrição. O segundo é o endereço de uma variável inteira que deve armazenar o tamanho em *bytes* da descrição gerada. Este inteiro é necessário, pois a função que envia deve saber quantos *bytes* devem ser enviados e quem recebe deve saber quantos *bytes* a resposta possui. O vetor de *bytes* gerado é enviado para um de seus subalternos. Um subalterno pode ser um *Gerente* ou um *Trabalhador*. Conforme mencionados na seção 3.4.1, um *Gerente* não diferencia seus subalternos.

Caso a descrição do *Job* seja recebida por um *Trabalhador*, o *Job* é reconstruído através de outra função pré-definida, para ser computado. No caso de ser recebido por um *Gerente*, a descrição é armazenado para ser posteriormente enviada a um dos subalternos desse *Gerente*.

A.1.3 void setSubResult()

Função que gera de forma incremental o resultado final de um *Job*. Como argumentos, recebe a descrição do resultado encontrado na forma de um vetor de *bytes* e um inteiro que indica o tamanho em *bytes* desta descrição.

A.1.4 boolean isSolved()

O retorno desta função é um valor *booleano*. Retorna 1 caso o *Job* já esteja resolvido, 0 do contrário. Como argumento recebe a estrutura de dados de *Job*.

A.2 Executas pelos *Trabalhadores*

A.2.1 void solve()

Esta é a rotina que calcula um *Job*. Deve conter a programação necessária para encontrar a resposta deste *Job*. O argumento desta rotina é o *Job* que fora inicializado.

A.2.2 int workload()

Função que retorna a quantidade de trabalho de um determinado *Job*, na forma de um valor inteiro. Uma estrutura de dados com o *Job* que se deseja obter a quantidade de trabalho é o argumento desta função.

A.3 Executadas por Ambos

Nesta subseção são listadas as funções que podem ser executadas por *Gerentes* e *Trabalhadores*.

A.3.1 job newJob()

Esta rotina cria (inicializa) um *Job* cuja descrição é passada como argumento. É a função chamada quando uma mensagem do tipo *NewJob* é recebida. Em síntese, cria-se uma estrutura de dados que contenha as variáveis que serão utilizadas na rotina que o calcula. A descrição do *Job* recebido foi gerada pela função *description()* do *Gerente*.

A.3.2 byte[] resultDescription()

Esta função é responsável por criar a descrição do resultado computado de um *Job*. Descrever um resultado é semelhante a descrever o *Job*. Neste caso, o resultado encontrado é que é inserido em um vetor de *bytes*. *Trabalhadores* utilizam esta função para descrever o resultado de um *Job* processado e enviá-lo a seu *Gerente*. *Gerentes* a utilizam quando todos os resultados das subtarefas originadas a partir de uma tarefa tiverem sido recebidos. Um *Gerente* descobre que todas as respostas foram recebidas quando a função *isSolved()* retornar *true*. Ele então constrói a resposta a partir de todas as respostas recebidas, obtém a descrição através desta função e a remete para seu *Gerente*.

Essa função recebe dois argumentos. O primeiro argumento recebido pela função é uma estrutura de dados contendo o *Job* que fora resolvido. O segundo é o endereço de uma variável inteira que deve armazenar o tamanho em *bytes* da descrição criada. A descrição do resultado gerado é retornada na forma de um vetor de *bytes*.

APÊNDICE B FUNCIONALIDADES DO GERENTE NO TUXUR

Neste capítulo são abordados com mais detalhes as principais tarefas do componente *Gerente* do *TUXUR*.

B.1 Divisão e Delegação de Tarefas

Conforme mencionado na seção 3.3.1, as tarefas são dinamicamente divididas. Esta divisão é realizada para que quantidade de trabalho resultante se adeque à capacidade de trabalho de um determinado *Trabalhador*. Esta pode ser somente a sua capacidade, quando tratar-se de um nó folha, ou a soma de todas as capacidades de trabalho dos nós que formam um determinado ramo da árvore.

Para uma melhor compreensão da definição realizada, exibimos uma possível configuração para a grade na figura B.1.

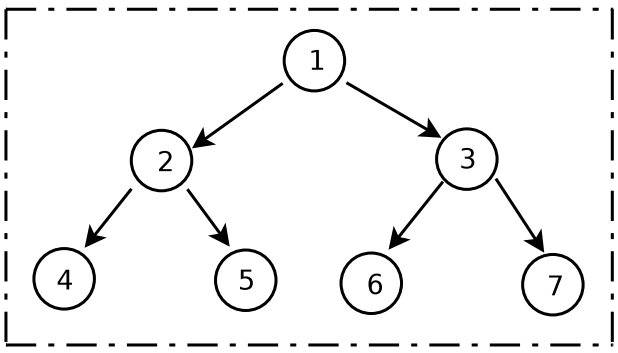

Figura B.1: Exemplo de organização hierárquica

Nesta figura, o computador representado por **1** é o *Gerente* dos *Trabalhadores* {**2**, **3** }. Na mesma linha de raciocínio, os computadores **2** e **3** são os *Gerentes* dos *Trabalhadores* {**4**, **5** } e {**6**, **7** }, respectivamente. A capacidade de trabalho do computador **1** corresponde à soma das capacidades de trabalho dos computadores {**1**, **2**, **3** }. Já a capacidade de

trabalho do computador **2**, por exemplo, corresponde ao somatório das capacidades de trabalho dos computadores {**2**, **4**, **5** }. A mesma lógica se aplica ao computador **3**.

Na visão do *Gerente*, a capacidade de trabalho de um de seus *Trabalhadores* é única e exclusivamente dele. Ou seja, mesmo que a capacidade de trabalho aferida corresponda ao somatório de vários outros *Trabalhadores*, para ele trata-se da capacidade de um único *Trabalhador* e lhe será delegada a quantidade de trabalho suficiente para atendê-la.

A capacidade de trabalho de um determinado nó não pode ser prevista com exatidão; ela necessita ser computada e constantemente atualizada. Isso representa um sério problema quando for enviada a primeira tarefa para um *Trabalhador*, pois não se sabe qual quantidade de trabalho enviar. Para resolver esse problema, antes de ordenar a resolução de toda tarefa através da grade, o usuário deve estimar a capacidade de trabalho de um computador cujo *hardware* seja conhecido. Uma tarefa com uma determinada quantidade de trabalho deve ser criada e enviada para este computador. Quando resolvida, ele informará a sua capacidade de trabalho. Desta forma, a capacidade de trabalho daquele perfil de *hardware* torna-se conhecido. Esse processo pode ser repetido até encontrar uma quantidade de trabalho suficiente para que uma determinada configuração de *hardware* a resolva no tempo alvo pretendido.

De posse da capacidade de trabalho deste computador, assume-se que todos os computadores tenham a mesma configuração que a dele. Com isso, envia-se uma tarefa com a melhor quantidade de trabalho encontrada na simulação. *Trabalhadores* informam a sua capacidade de trabalho ao seu *Gerente* a cada tarefa resolvida. Com esta informação, a quantidade de trabalho das tarefas futuras a serem enviadas para este *Trabalhador* pode ser facilmente obtida. Caso a capacidade de trabalho informada for maior do que a do computador utilizado para calcular a quantidade de trabalho inicial, implica que o trabalhador que a resolveu possui um poder computacional superior e assim pode-se enviar uma quantidade de trabalho maior para a próxima tarefa.

As tarefas são mantidas em uma fila dentro do *Gerente*. Um *Gerente* retira estas tarefas, divide-as em tarefas menores se for preciso, e as envia para seus *Trabalhadores*. A divisão de tarefas não é automática, tampouco existe uma solução que se aplica para todos os tipos de problemas. Embora seja realizada pelo *Gerente*, cabe ao usuário que criou a tarefa programar a rotina pré-definida que efetua a divisão de uma tarefa em subtarefas. Cada subtarefa originada contém uma quantidade de trabalho menor do que a da tarefa que

a originou. Portanto, delega-se tantas subtarefas quantas forem necessárias para atender a capacidade de trabalho de um *Trabalhador*. No caso de nem todas as subtarefas serem enviadas, estas são mantidas pelo *Gerente* e podem ser remetidas a outros *Trabalhadores*.

Do mesmo modo que são criadas e despachadas subtarefas, também são criados e recebidos sub-resultados. Estes resultados devem então ser recebidos por um *Gerente* e a partir deles compor o resultado final. Esta é outra responsabilidade do *Gerente*, a qual é abordada a seguir.

B.2 Recepção e Formação da Resposta Final

Quando ocorre a quebra de uma tarefa em tarefas menores, estas são resolvidas pelos *Trabalhadores*. Assim que terminarem seu trabalho, o resultado encontrado é transmitido para seu *Gerente*. Ele deve então, a partir de todos os sub-resultados encontrados, formar a resposta final da tarefa que originou as subtarefas.

Para conseguir formar esta resposta, um *Gerente* deve saber a qual subtarefa pertence um dado resultado. Há no *Gerente* uma estrutura de busca onde são armazenadas todas as tarefas que já foram designadas para algum *Trabalhador*. Ao receber uma resposta, localiza-se a tarefa a que a resposta pertence e atribui-se o seu resultado. Caso todos os resultados das subtarefas de uma determinada tarefa já tenham sido encontrados, computa-se o resultado final e o mesmo é enviado ao *Gerente* deste *Gerente*. Para facilitar a compreensão, utilizamos o termo *Gerente Remoto* para o *Gerente* de um *Gerente*. Todo o processo descrito aqui se repete no *Gerente Remoto*.

Quando não mais existir um *Gerente Remoto* a quem transmitir uma mensagem de resposta, implica que chegou-se ao resultado final da computação de uma tarefa que havia sofrido sucessivas divisões, ou então ao resultado final de uma tarefa do tipo *Bag-of-Tasks*. Semelhante ao que ocorre com um *Trabalhador*, cada mensagem contendo uma resposta enviada por um *Gerente* é imediatamente seguida de outra mensagem, informando o estado deste *Gerente*. Esta mensagem permite ao *Gerente Remoto* descobrir se um dado *Trabalhador* corre o risco de ficar sem trabalho. Se for o caso, mais trabalho lhe é enviado, seguindo a mesma lógica de envio de tarefas de um *Gerente* para um de seus *Trabalhadores*.

APÊNDICE C VERSÃO IMPLEMENTADA DO TUXUR

Para tornar possível a defesa do trabalho a que se refere esta dissertação de mestrado, uma versão simplificada do *TUXUR* foi implementada. Por razões de prazo, implementou-se as funcionalidades mínimas necessárias para que este atue como um *Framework* de computação em grade, respeitando-se as premissas que norteiam desenvolvimento do *TUXUR* e que já foram citadas neste trabalho. Foram então implementados os componentes *Lançador*, *Gerente*, *Trabalhador* e *Job*. A seguir é feita uma breve descrição de cada um deles.

C.1 Componentes Implementados

C.1.1 Lançador

De modo a simplificar este trabalho, o *Lançador* lança somente um *Gerente* e designa tantos *Trabalhadores* quantos existirem no arquivo de descrição como seus subordinados. Ao fazer o *lançamento* desta madeira, o código do *Lançador* é simplificado pois não há necessidade de se programar as rotinas que geram toda a hierarquia entre os nós da grade.

O arquivo de configuração contém todos os computadores que podem pertencer à grade de computadores gerenciada por *TUXUR*. Dentre os dados deste arquivo constam se um determinado computador possui suporte à GPGPU. Em caso positivo, é descrito qual técnica de programação GPGPU ele suporta, NVIDIA CUDA ou ATI *Stream Computing*.

Quando o *Lançador* é ativado, alguns parâmetros de configuração são enviados como argumento. Se há suporte à execução de um *Job* através de GPGPU, especifica-se qual das técnicas de programação GPGPU é suportada. Desta forma o *Lançador* ativa os *Trabalhadores* que oferecem suporte a esta técnica. No caso de ambas as técnicas serem suportadas, os respectivos *Trabalhadores* são ativados.

Na hipótese de um computador descrito neste arquivo não oferecer suporte à GPGPU

quando existir suporte de execução de *Jobs* GPGPU, o *Trabalhador* nativo é disparado. Na ausência de suporte à execução de *Jobs*, somente *Trabalhadores* nativos são ativados.

C.1.2 Gerente

O *Gerente* implementado visa dar todo o suporte à execução de um *Job*. Para isso foi necessário implementar algumas das funcionalidades do *Gerente*, as quais foram descritas na seção 3.4.1.

No entanto, por se tratar do componente mais complexo de *TUXUR*, algumas de suas funcionalidades foram implementadas e cumprem sua função, mas não da forma como oficialmente está planejado. Como exemplo podemos citar a delegação de tarefas. Não é levada em consideração a quantidade de trabalho que um *Trabalhador* ainda possui para então decidir se deve ou não enviar-lhe mais trabalho. O *Gerente* simplesmente envia um novo *Job* que atenda a capacidade de trabalho deste *Trabalhador*, tão logo receba uma mensagem que informa o seu estado.

As principais tarefas que um *Gerente* deve executar foram implementadas com sucesso. Estas são: reconhecer e gerenciar os *Trabalhadores* enviados pelo *Lançador*; gerar e gerenciar os *Jobs* cujas quantidades de trabalho sejam compatíveis com a capacidade de trabalho de um determinado *Trabalhador*; gerenciar as partes das respostas encontradas para originar a resposta final.

C.1.3 Trabalhador

Conforme mencionado na seção 3.4.2, um *Trabalhador* é o responsável por computar os *Jobs* enviados pelo seu *Gerente*. Tal responsabilidade reflete na sua arquitetura, qual também é explicada na referida seção.

Esta arquitetura foi completamente implementada. Em síntese, este *Trabalhador* realiza todas as funções que um *Trabalhador* nativo do *TUXUR* deve realizar.

C.1.4 *Job*

Job é a tarefa a ser encaminhada pelos *Gerentes* a seus *Trabalhadores*. Uma descrição mais aprimorada do *Job* foi realizada na seção 3.4.5. Por não ser um componente funcional, mas sim representar um problema a ser resolvido, não há uma programação única para ele.

O que existe são funções que são chamadas por um *Gerente* e/ou por um *Trabalhador*,

as quais foram descritas no Apêndice A. Assim, criaram-se os protótipos destas funções e documentou-se o que cada uma delas deve implementar.