UNIVERSIDADE DO SUL DE SANTA CATARINA

FERNANDO ARAÚJO DOS SANTOS

KAIO CESAR KOERICH

COMPUTAÇÃO HETEROGÊNEA – ROTEIRO DE DESENVOLVIMENTO DE UMA APLICAÇÃO UTILIZANDO UNIDADE DE PROCESSAMENTO GRÁFICO (GPU)

Palhoça

2013

FERNANDO ARAÚJO DOS SANTOS
KAIO CESAR KOERICH

COMPUTAÇÃO HETEROGÊNEA – ROTEIRO DE DESENVOLVIMENTO DE UMA APLICAÇÃO UTILIZANDO UNIDADE DE PROCESSAMENTO GRÁFICO (GPU)

Trabalho de Conclusão de Curso do curso de Ciência da Computação, da Universidade do Sul de Santa Catarina.

Orientador: Prof. Dr. Saulo Popov Zambiasi.
Co-Orientador: Prof. MSc. Osmar de Oliveira Braz Jr.

Palhoça
2013

FERNANDO ARAÚJO DOS SANTOS
KAIO CESAR KOERICH

COMPUTAÇÃO HETEROGÊNEA – ROTEIRO DE DESENVOLVIMENTO DE UMA APLICAÇÃO UTILIZANDO UNIDADE DE PROCESSAMENTO GRÁFICO (GPU)

Este Trabalho de Conclusão de Curso foi julgado adequado à obtenção do título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo Curso de Graduação em Ciência da Computação da Universidade do Sul de Santa Catarina.

Palhoça, 21 de Junho de 2013.

_____
Prof. Saulo Popov Zambiasi, Dr.
Universidade do Sul de Santa Catarina

_____
Prof. Osmar de Oliveira Braz Jr., MSc.
Universidade do Estado de Santa Catarina

_____
Prof. Fernanda Oviedo Bizarro, Esp.
Universidade do Sul de Santa Catarina

_____
Prof.ª Maria Inés Castineira, Dra.
Universidade do Sul de Santa Catarina

RESUMO

A alta demanda computacional vem exigindo cada vez mais desempenho dos computadores. Para tanto, é necessário que sejam aproveitados ao máximo os recursos disponíveis. Neste trabalho, em sua revisão bibliográfica, apontam-se diferentes soluções para uma melhor utilização destes recursos (relacionados a computação heterogênea). Com base na pesquisa realizada optou-se então pela utilização da *GPU* (*Graphics Processing Unit* – Unidade de processamento gráfico) como solução para melhora dos tempos de execução em uma aplicação de exemplo. É comum a utilização da *GPU* apenas em aplicações que exijam um alto nível de processamento gráfico, como por exemplo jogos e aplicativos de tratamento de imagem e vídeo. Tecnologias como o *CUDA* e *JCUDA* (bibliotecas externas) direcionam o processamento da *GPU* para execução de tarefas atípicas, transformando este hardware comumente ocioso em um potente processador auxiliar. Foi realizada a confecção de um roteiro para a configuração do ambiente de desenvolvimento de uma aplicação paralela utilizando a *GPU*, e a validação deste ambiente por via de um protótipo simples (aplicação exemplo), que executa o cálculo da matriz identidade (de uma matriz aleatória, com dados de ponto flutuante) através do método de Gauss-Jordan. Em dois cenários diferentes – sem aceleração da *GPU* e com aceleração provida pela *GPU* – foi possível verificar os ganhos significativos em relação aos tempos de execução da aplicação quando a *GPU* é utilizada. Em todas as situações, foi realizado um comparativo em que observa-se que a solução escolhida foi capaz de proporcionar ganhos superiores a 90%, relacionados a tempos de execução.

Palavras chave: *GPU*. *CUDA*. *JCUDA*. Roteiro. Desenvolvimento. Computação Heterogênea. Desempenho.

ABSTRACT

The high computational demand is increasingly demanding more performance from computers. To do so, the resources need to be exploited to their maximum. In this paper, in the literature review, it was pointed out different solutions for the better use of these resources (related to heterogeneous computing). The use of the GPU (Graphics Processing Unit) was adopted as a solution to improve execution times based on the research conducted. It is common to use the GPU only in applications that require a high level of graphics processing, such as games and image and video processing applications. Technologies such as CUDA and JCUDA (external libraries) directs the GPU processing to the execution of atypical tasks, transforming this commonly idle hardware in a potent auxiliary processor. A script was elaborated for configuring the development environment of a parallel application using the GPU, and the validation of this environment by a simple prototype to calculate the identity matrix (of a random matrix with floating point data) by the Gauss-Jordan method was performed in order to verify and evaluate the paper. In two different scenarios - no GPU acceleration and acceleration provided by GPU - it was possible to verify the significant gains in relation to the execution times of the application when the GPU is used. In all situations, a comparison has been carried out in which it is observed that the chosen solution was able to provide gains greater than 90% related to the execution times.

Keywords: GPU. CUDA. JCUDA. Script. Development. Heterogeneous Computing. Performance.

SUMÁRIO

1 INTRODUÇÃO

O processamento de aplicativos em um ambiente computacional (seja ele empresarial ou doméstico) pode não fazer uso de todos os recursos de hardware disponíveis. O melhor aproveitamento destes recursos pode ser dado através de computação heterogênea, que tem por objetivo otimizar a utilização do processamento de dados em um determinado nó computacional (STRINGHINI; GONÇALVES; GOLDMAN, 2012). Por nó computacional compreende-se cada ponto de interconexão com uma estrutura ou rede, independente da função do equipamento (computador) representado por este nó.

Os sistemas heterogêneos consistem em vários processadores que podem se comunicar uns com os outros, para que possam fazer melhor uso do hardware disponível. A programação das funções e atividades divididas entre os núcleos de processamento são de responsabilidade do desenvolvedor, que deve realizar a construção de seus aplicativos em um ambiente de desenvolvimento apto a suportar as tecnologias por ele selecionadas. Com o ambiente devidamente construído, o programador deve modelar e desenvolver aplicações que admitam comunicação entre os processadores, possibilitando ao sistema resultados de processamento mais efetivos.

Expandindo o conceito de heterogeneidade, encontram-se alguns fundamentos para modelar e desenvolver aplicações em ambientes multiprocessados. De acordo com Grune e outros (2001), podem-se observar os fundamentos de sistema paralelo, sistema distribuído, modelos de programação paralela, processos e encadeamentos, variáveis compartilhadas, passagem de mensagens, linguagens paralelas orientadas a objetos, espaço de tuplas, paralelização automática, além de conceitos de *grids* e *clusters*.

Para melhor entendimento da importância de sistemas heterogêneos serão apresentados os conceitos necessários e passíveis de explicação a fim de elucidar (por via de um roteiro) a configuração de um ambiente de desenvolvimento para aplicações paralela, bem como os benefícios deste modelo computacional. Como uma

alternativa de processador pode-se observar a *GPU* (*Graphics Processing Unit* – Unidade de processamento gráfico), que em grande parte do tempo de uso encontra-se ociosa.

A *GPU* pode ser apontada como um recurso acelerador, que tem por objetivo uma melhor performance em cálculos com pontos flutuantes por segundo (*FLOPS - Floating-point Operations Per Second*). A especialização das *GPU's* permite que elas executem essas operações específicas com muito mais eficiência Um processador i7 980-X *Extreme Edition* da Intel atinge até 20 Gflop/s enquanto uma GeForce 8800 Ultra atinge até 576 *Gflop/s* (KHRONOS GROUP, 2012).

As *GPUs* atuais já ultrapassam com facilidade a barreira do Tera Flop. Existem bibliotecas auxiliares responsáveis pelo direcionamento do uso da GPU como um recurso auxiliar de processamento. A *NVIDIA*, empresa responsável por desenvolver microchips de processamento gráfico, possui o *CUDA*. Em linhas gerais, para Nvidia (2012) o *CUDA* é "uma plataforma de computação paralela e um modelo de programação inventados pela NVIDIA, que permite aumentos significativos de performance computacional ao aproveitar a potência da unidade de processamento gráfico".

Outra biblioteca utilizada com o mesmo objetivo é o *OpenCL*. Stringhini, Gonçalves e Goldman (2012) colocam que esta é a primeira linguagem padrão para o desenvolvimento multiplataforma de aplicações em ambiente heterogêneo com o uso dos processadores gráficos atuais, desenvolvida pela também fabricante de processadores gráficos AMD em conjunto com a Apple Inc., Intel e Nvidia.

Em ambientes computacionais heterogêneos, que possuam uma *GPU* (aceleradoras especializadas na execução de operações com ponto flutuante, mais utilizadas na renderização de gráficos), é possível que uma aplicação distribua processos, *threads* ou trechos de código específicos com a *GPU*, melhorando sua performance através do processamento paralelo das mesmas.

1.1 PROBLEMA

A elevada demanda computacional das aplicações modernas exige cada vez mais que os computadores possuam alto poder de processamento. Comumente, alguns computadores já atendem tal necessidade, mas estes recursos encontram-se ociosos, mal gerenciados, e acabam não sendo utilizados de maneira correta pelas aplicações.

Com intuito de prover um melhor aproveitamento dos recursos computacionais disponíveis aos desenvolvedores, este trabalho apresenta o roteiro de desenvolvimento de uma aplicação paralela, utilizando a *GPU* como acelerador.

1.2 OBJETIVOS

A seguir são apresentados o objetivo geral e os objetivos específicos da monografia.

1.2.1 Objetivo Geral

O presente trabalho tem como objetivo a confecção de um roteiro do processo de montagem e configuração do ambiente de desenvolvimento para a criação de uma aplicação heterogênea paralela utilizando a *GPU*.

1.2.2 Objetivos Específicos

Para nortear este trabalho e alcançar o objetivo geral, são definidos os seguintes objetivos específicos:

a) Levantar material bibliográfico referente as tecnologias envolvidas para embasar e direcionar a solução do problema apresentado;
b) apresentar os conceitos de computação heterogênea;
c) pesquisar e configurar o ambiente de desenvolvimento de uma aplicação paralela;
d) montar um roteiro de configuração do ambiente de desenvolvimento de uma aplicação paralela;
e) apresentar uma aplicação paralela exemplo (protótipo);
f) apresentar um comparativo de desempenho entre a aplicação executada em um ambiente computacional não heterogêneo e em um ambiente computacional heterogêneo.

1.3 JUSTIFICATIVA

Comumente a *GPU*, um hardware extremamente potente, inicialmente designado apenas para processamento gráfico, encontra-se ocioso; entretanto, aplicações desenvolvidas com o auxílio de bibliotecas externas podem fazer uso dos processadores deste recurso acelerador, criando um ambiente computacional heterogêneo, no qual tarefas podem ser balanceadas entre *CPU* e *GPU*.

De acordo com Du e outros (2012), a *CPU* é responsável por executar o processo principal, em que se inicializam as *threads* que serão executadas na *GPU*; as *GPU's* por sua vez, possuem uma hierarquia própria de memória; seus dados transitam por via de um barramento *PCI Express*. As duas principais bibliotecas auxiliares utilizadas para ativar o paralelismo *CPU*/*GPU* são o *CUDA* – desenvolvido pela *NVIDIA* – e o *OpenCL* – um padrão de modelo aberto para computação

heterogênea que pode ser utilizado por diversas aceleradoras (não apenas as da *NVIDIA*, como o *CUDA*).

Machado e Maia (2002) apontam que a utilização de computação heterogênea proporciona ao utilizador deste artifício os seguintes benefícios:

- **Desempenho**: melhor desempenho quando processadores extra são adicionados ao ambiente computacional. No caso deste trabalho, apresentar-se-á como alternativa o uso do processador da *GPU*;
- **Relação custo/desempenho**: possibilidade de se oferecer um ambiente computacional de alto desempenho – com o auxílio de um recurso comumente ocioso em grande parte do tempo (a *GPU*) – sem aumento de custos;
- **Balanceamento de carga**: dividir as tarefas por todos os processadores disponíveis, de modo que todos estejam ocupados com processamento das mesmas, quando necessário, reduzindo então o tempo de processamento.

A modelagem e o desenvolvimento da aplicação exemplo (protótipo) são considerados de caráter essencial para entendimento da utilização dos paradigmas computacionais envolvidos no processo de construção do software paralelo; optou-se, então, por construir a aplicação e não fazer uso de alguma aplicação com funcionamento paralelo já existente.

1.4 ESTRUTURA DA MONOGRAFIA

Apresentam-se nesta monografia alguns dos conceitos de computação heterogênea, bem como todos os conceitos relacionados/necessários para seu entendimento. Está disposta a seguir a estrutura da monografia e uma breve descrição dos pontos abordados em cada um dos capítulos.

O capítulo 2 – Revisão Bibliográfica – está subdividido em tópicos que apresentam os conceitos de Computação Heterogênea (2.1) de uma forma geral, o Processamento de Alto Desempenho (2.2) é responsável por apontar especificações/vantagens do uso deste tipo de arquitetura; Estrutura do Software Paralelo (2.3) visa explanar os principais conceitos e paradigmas computacionais

relacionados ao funcionamento de uma aplicação heterogênea; o item A *GPU* (2.4) fala sobre a arquitetura da *GPU*.

O capitulo 3 – Método – explana o método e os artifícios utilizados para a confecção da monografia, tais como: Caracterização do Tipo de Pesquisa (3.1) no qual são colocadas a natureza da pesquisa, sua classificação, abordagem do problema; em Etapas (3.2), aponta-se o passo-a-passo planejado para que o objetivo seja atingido; a Proposta ou Arquitetura da Solução (3.3) apresenta a aplicação modelo e seu funcionamento e em Delimitações (3.4), está definido o escopo do trabalho.

No capítulo 4 – Desenvolvimento e Resultados – estão contidos os dados de modelagem (Requisitos de Software – 4.1 e Modelagem da Implementação – 4.2) definidos pelos autores para embasar e elucidar o desenvolvimento da aplicação exemplo; encontram-se também as tecnologias utilizadas no desenvolvimento da mesma (4.3 – Tecnologias Utilizadas), o roteiro de criação/configuração do ambiente de desenvolvimento (Seção 4.4), uma breve apresentação desta aplicação (Seção 4.5), a demonstração dos resultados obtidos com a aplicação exemplo (Seção 4.6), as principais dificuldades enfrentadas pelos autores ao construir o ambiente de desenvolvimento e o projeto (dificuldades técnicas associadas as tecnologias escolhidas, Seção 4.7).

No Capítulo 5 – Considerações Finais – encontra-se o fechamento das ideias a partir dos dados apresentados, aspectos conclusivos retirados da revisão bibliográfica e do estudo dos resultados obtidos a partir do tempo de execução do da aplicação exemplo, bem como possíveis ideias para trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

Os tópicos expostos a seguir representam informações necessárias reunidas com intuito de fundamentar as ferramentas e paradigmas que serão utilizados ao decorrer da monografia.

2.1 COMPUTAÇÃO HETEROGÊNEA

Os tópicos a seguir apresentam conceitos importantes para o entendimento do funcionamento de um ambiente computacional heterogêneo; serão apresentadas informações chave sobre os principais tipos de computação heterogênea – Paralelismo, Distribuição, Grid e Cluster.

2.1.1 Paralelismo

Para Tanenbaum (2001), os projetistas de arquitetura encontram-se constantemente provocados a obter melhoras no *hardware* responsáveis por aumentar o desempenho dos computadores; uma das maneiras para que isto seja feito é o aumento da velocidade de *clock*, fazendo com que os chips rodem cada vez mais rápidos.

Por maiores que sejam os avanços obtidos pelos arquitetos de *hardware*, há um limite tecnológico, dependente da arte tecnológica disponível na época do desenvolvimento do mesmo. O paralelismo surge então como uma forma de obter um desempenho maior de um (ou mais) *hardwares* para uma dada velocidade de *clock* (TANENBAUM, 2001).

Conceitualmente, o paralelismo pode ser subdividido em dois modelos: o paralelismo em nível de instruções e no nível de processador. No primeiro modelo,

um maior número de instruções (individuais) possíveis é executado em um espaço de tempo curto. No segundo caso, vários processadores trabalham em conjunto na solução do mesmo problema (TANENBAUM, 2001).

O conceito base de utilização de um sistema paralelo incide em resolver um problema o mais rápido possível, utilizado processadores múltiplos. A justificativa para uso de paralelismo em aplicações se dá por via destas levarem muito tempo para obtenção do resultado de execução. (GRUNE et al., 2001).

De acordo com Machado, Maia (2002), só haverá ganho de desempenho se a aplicação em questão (ou um conjunto de instruções da mesma) puder ser dividida em partes independentes para execução simultânea; para ilustrar a situação, o autor da obra propõe duas situações. Na primeira tem-se uma operação aritmética, passível de divisão em três partes, onde cada Unidade Central de Processamento (UCP) ficará responsável pela execução de uma multiplicação, em simultâneo.

*x := (a * b) + (c * a) + (d * b)*

Na segunda situação é colocada a questão de processamento vetorial, onde um vetor de 100 posições deve ser inicializado. Fazendo uso de paralelismo, a inicialização de cada posição do vetor ficaria por conta de um UCP diferente, tornando o processamento mais rápido do que se fosse executado de maneira sequencial (MACHADO; MAIA, 2002).

Normalmente, o processamento seria de maneira sequencial, da seguinte forma:

```
FOR i := 1 TO 100 DO
    Vetor [ i ] := 0;
```

Em um sistema com 100 processadores, ter-se-ia algo da seguinte maneira:

Vetor [1] : = 0; → Operação executada pelo processador 1.

Vetor [2] : = 0; → Operação executada pelo processador 2.

...

Vetor [99] : = 0; → Operação executada pelo processador 99.

Vetor [100] : = 0; → Operação executada pelo processador 100.

Um dos problemas que pode se observar nos sistemas que implementam processamento paralelo é determinação do que pode/não pode ser executado de maneira paralela. O paralelismo pode ser indicado pelo programador/desenvolvedor da aplicação (o que se conhece por paralelismo explícito), ou ainda pode ser detectado automaticamente pelos sistemas operacionais, que determinam quando instruções podem ser executadas em paralelo – paralelismo implícito (MACHADO; MAIA, 2002).

2.1.1.1 Paralelismo Implícito

No paralelismo implícito o sistema (ou componente de paralelização automática) subdivide as tarefas a serem executadas, particionando-as e alocando as mesmas aos processadores disponíveis de forma automática. O processo de sincronização de uma ou mais tarefas executadas em paralelo fica também a critério do dispositivo de paralelização automática. Este modelo de paralelismo liberta o desenvolvedor dos detalhes de execução paralela, proporcionando uma solução generalizada e flexível para uma determinada gama de problemas. Entretanto, é difícil que se consiga uma solução eficiente capaz de abranger todos os casos existentes (ROCHA, 2012).

2.1.1.2 Paralelismo Explícito

De acordo com Rocha (2012), no paralelismo explicito o particionamento das tarefas e da execução aplicação em questão fica sob o controle do desenvolvedor. Fica a critério deste, então: anotar a instrução (ou conjunto de instruções) que será executada paralelamente, além de estabelecer qual dos processadores irá executar as tarefas determinadas pelo programador.

Para tal, o desenvolvedor deve ter conhecimento sobre o hardware da máquina, obtendo sobre este a melhor performance possível – no que diz respeito ao aumento de localidade, diminuição de comunicação e etc. É perceptível então que, desenvolvedores com maior experiência serão capazes de produzir melhores soluções para problemas específicos. Aspectos negativos que podem ser levantados sobre esta metodologia são: baixa portabilidade entre diferentes arquiteturas e a maior responsabilidade do programador em gerenciar a execução das tarefas.

2.1.2 Distribuição

Sistema computacional distribuído consiste em um sistema composto de um conjunto de computadores interconectados por uma rede de comunicação. Os processadores comunicam-se trocando mensagens através da rede, sendo assim, o sistema distribuído considerado fracamente acoplado. Alguns dos nodos de processamento podem ser compostos de conjuntos de computadores; estes nodos podem se comunicar através de memória compartilhada. Duas características importantes dos sistemas distribuídos são destacadas pelos autores; são elas: a organização hierárquica e a heterogeneidade (SCHATZ; WANG, 1989).

Na concepção de Schatz e Wang (1989), o nodo de um sistema distribuído pode ser composto de vários computadores, com isso o sistema distribuído fica limitado a apenas um nível hierárquico. Já em uma arquitetura distribuída um nodo pode ser agregado de outros nodos. Por definição, a arquitetura distribuída é organizada em vários níveis/subníveis. Cada nível é composto de diversos nodos que utilizam mensagens para se comunicar. Nos níveis inferiores da hierarquia um conjunto de computadores formará um nodo; este conjunto de computadores também será considerado um nodo. Surgem assim, as seguintes definições:

a) **Arquitetura Distribuída:** Uma arquitetura distribuída consiste em uma organização computacional composta de nodos, as quais se comunicam através de mensagens trocadas por um meio de comunicação;
b) **Nodo:** Define-se como elemento organizacional da arquitetura distribuída;

c) **Nodo Elementar:** É constituído por um ou mais processadores fazendo uso de uma memória. Tem-se como exemplo de nodo elementar, um uniprocessador. Essa configuração é nomeada arquitetura elementar. Pode-se citar também como nodo elementar, uma arquitetura paralela baseada em compartilhamento.
d) **Nodo Composto:** É definido por um nodo composto por outros nodos, onde os mesmos podem ser elementares e/ou compostos;
e) **Nodo Homogêneo:** Define-se como sendo somente elementares ou somente compostos, ou seja, constitui em um nodo composto de nodos do mesmo tipo;
f) **Nodo Heterogêneo:** Um nodo heterogêneo é quando se tem um ou mais nodos elementares misturados com um ou mais nodos compostos, sendo assim, um nodo composto de nodos de tipos distintos;
g) **Nodo Básico ou Nodo Homogêneo Básico:** É quando um nodo é formado de nodos elementares;
h) **Nodo Avançado ou Nodo Homogêneo Avançado:** Tem-se como nodo avançado, um nodo composto de nodos compostos.

2.1.3 Grid

Segundo Foster (2012), *Grid* pode ser compreendido como um ambiente computacional distribuído que se utiliza do processamento de vários computadores conectados em uma rede, para melhorar o desempenho de aplicações. Suas principais características são a não centralização da coordenação dos recursos, o uso de protocolos de comunicação e interfaces padrões, abertos de uso geral e a entrega de serviços de qualidade não triviais.

Foster (2012), afirma que os *Grids* são utilizados em tarefas que não poderiam ser concluídas por uma única máquina devido a um fraco desempenho, relacionados com tempo de resposta, disponibilidade, vazão de dados, segurança e alocação de múltiplos recursos que satisfaçam a necessidade do usuário.

Algumas aplicações para fins de pesquisa científica utilizam esse tipo de recurso com o objetivo de diminuir a análise de grandes quantidades de dados de

muitos anos para alguns meses, como por exemplo, a análise dos dados dos aceleradores de partículas como o *LHC – Large Hadron Collider* (FOSTER, 2012).

2.1.4 Cluster

Define-se por cluster um conjunto formado por dois ou mais computadores (monoprocessáveis ou multiporcessáveis), que harmonizam seu processamento para trabalharem juntos, a fim de fornecer uma solução para um problema específico ou geral (MANOEL; FERREIRA, 2012). Para Sterling (2012), esta tecnologia é capaz de substituir supercomputadores em situações de processamento de determinados tipos de aplicações – por exemplo: cálculos científicos, simulações entre outras tarefas de origem complexa - que necessitem de alto poder computacional; porém seu custo é significativamente menor, pois faz uso (geralmente) de processadores/computadores com menor desempenho computacional individual e preços mais baixos.

Existem três tipos (mais comuns) de clusters encontrados atualmente. São eles: de Alto desempenho de computação (*High Performance Computing – HPC*), os de alta disponibilidade (*High Availability – HA*) e os de balanceamento de carga (*Horizontal Scaling – HS*) (MANOEL; FERREIRA, 2012).

2.2 PROCESSAMENTO DE ALTO DESEMPENHO

O processamento de alto desempenho consiste no melhor aproveitamento de recursos de um ambiente computacional; com intuito de atingir este objetivo, as melhorias se expandiram para domínios externos aos dos comumente utilizados. *Clusters* de computadores, combinados e interconectados por via de redes de alta velocidade, contando com processamento proveniente de multiprocessadores com memória compartilhada (*Symmetric multiprocessing – SMP*) agregam múltiplos

processadores em um único sistema de memória e os *Chip Multi-Processor* (*CMP*), constituem exemplos de máquinas que operam em paralelo (DU et al., 2012).

Para Du et al. (2012), observa-se recentemente uma nova maneira de utilização de recursos computacionais paralelos, voltados para utilização de componentes que anteriormente eram utilizados apenas para processar instruções específicas, de origem gráfica; o processamento paralelo com uso de *GPU*. Com intuito de aceleração de execução, passa-se então a existir uma divisão do que será executado na *CPU-GPU*. O Código pode ser dividido em porções menores (sequenciais ou não) aptas a serem escalonadas entre *CPU* e *GPU*.

2.2.1 Vantagens da Arquitetura de Processamento Heterogênea

Alguns são os benefícios provenientes dos sistemas com múltiplos processadores. Dentre eles estão:

a) **Desempenho**: o desempenho no sistema é melhor quando novos processadores são acrescentados à arquitetura de uma máquina. Contudo, o ganho real de desempenho não é linear ao número de processadores acrescentados (MACHADO; MAIA, 2002);

b) **Escalabilidade**: é descrito como escalável um sistema no qual se pode adicionar mais processadores e aumentar de forma correspondente a sua potência computacional (TANENBAUM, 2001). É possível reduzir os custos em relação à aquisição de um novo sistema com maior desempenho quando a capacidade de computação é ampliada apenas adicionando-se novos processadores;

c) **Relação custo/desempenho**: é possível oferecer sistemas de alto desempenho com custo acessível, devido aos sistemas com múltiplos processadores que admitem utilizar *CPU's* convencionais de baixo custo interligadas às unidades funcionais por meios de mecanismos de interconexão (MACHADO; MAIA, 2002);

d) **Tolerância a falhas e disponibilidade**: define-se como tolerância a falhas a capacidade de manter o sistema em operação mesmo quando há falhas de

algum componente. Assim, se ocorrer de algum dos processadores falhar, os demais podem assumir suas funções de modo transparente aos usuários e suas aplicações, porém com menos capacidade computacional. A disponibilidade é mensurada em número de minutos por ano que o sistema permanece funcionando de maneira ininterrupta, incluindo possíveis falhas de hardware e software, sustentações preventivas e corretivas (MACHADO; MAIA, 2002); e por fim

e) **Balanceamento de carga**: é quando todos os processadores são mantidos ocupados o maior tempo possível com a distribuição do processamento entre os diversos componentes da configuração do sistema. As tarefas são transferidas dos processadores sobrecarregados para aqueles que possuem menor carga de processamento (MACHADO; MAIA, 2002).

Na visão de Machado e Maia (2002), Os sistemas com múltiplos processadores apresentam suas vantagens, no entanto, devido ás suas características existem algumas desvantagens, podendo se destacar as relacionadas aos problemas de comunicação e sincronização entre os processadores que estejam acessando as mesmas posições de memória. As soluções para os problemas causados pelo acontecimento podem ser descritos de maneira implícita ou explícita, sendo este um fator crítico.

2.3 ESTRUTURA DO SOFTWARE PARALELO

Tanenbaum (2001) classifica em quatro as metodologias para a produção de softwares para ambientes paralelos: (i) acrescentando bibliotecas especiais para o tratamento de números em linguagens de programação sequencial; (ii) incorporando bibliotecas com primitivas de comunicação e controle; (iii) acrescentando estruturas especiais à linguagem capazes de efetuar processamento paralelo e (iv) desenvolvendo uma nova linguagem de programação que contemple o paralelismo.

Tanenbaum ainda fala que a análise sobre qual metodologia usar depende de cinco fatores que formam a estrutura de softwares paralelos: (i) modelo de controle,

(ii) granularidade do paralelismo, (iii) paradigmas computacionais, (iv) métodos de comunicação e (v) primitivas de sincronização.

2.3.1 Modelos de controle

Existem dois modelos de controle de *threads*. Num primeiro caso, há o suporte para apenas uma *thread*, a um único programa em um único PC, mas para vários conjuntos de dados. Conforme as instruções são geradas, ela é executada simultaneamente sobre todos os conjuntos de dados. No segundo caso, há o suporte para várias *threads*, cada uma com seu próprio PC, seus registradores e variáveis locais. Cada *thread* executa seu próprio programa, com seus próprios dados, se comunicando e sincronizando com outras *threads* conforme necessidade (TANEMBAUM, 2001).

2.3.2 *Thread*

Segundo Tanembaum (2003), as estruturas chamadas *threads* são úteis em sistemas com paralelismo real. As *threads* podem ser de usuário ou de *kernel* (nativas). Cada sistema operacional implementa o mapeamento das *threads* de usuário para *threads* de *kernel*. O modelo “um-para-um” comummente encontrado em sistemas operacionais como o Windows NT mapeiam cada *thread* de usuário para uma de *kernel*. Esses sistemas operacionais executam as *threads* conforme a disponibilidade dos núcleos de processamento.

O paralelismo é depende diretamente do sistema operacional e do modelo adotado pelo mesmo. O controle do escalonamento fica a cargo do sistema operacional, mesmo que o programa utilize funções que priorizem um determinado processador, isso não garante que ele seja escolhido no processo de escalonamento.

Segundo Silberschartz (2004), nos processadores *multicores,*que fornecem um ambiente homogêneo, com núcleos iguais, os processos podem ser escalonados para qualquer núcleo. Entretanto, alguns processadores podem ficar ociosos (após o término da execução de sua pilha de processos) enquanto outros podem estar sobrecarregados. Uma solução possível é o uso de uma única fila de processos em que o escalonador distribua os mesmo aos processadores disponíveis no momento. Mesmo essa abordagem pode causar problemas, pois mais de um processador pode acabar acessando uma estrutura de dados comum, causando uma possível inconsistência de dados.

O monitoramento, coordenação e sincronização desses dados é fundamental para acabar com o problema. Estas estruturas são compostas de variáveis que definam o estado da instância e funções que implementem essas operações (SILBERSCHATZ, 2004). O monitor garante que somente um processo por vez acessará um os dados que ele gerencia. Linguagens de programação como Java suportam esse conceito de monitor.

2.3.3 Granularidade do paralelismo

Os processos podem ser divididos em vários níveis, aumentando a granularidade do mesmo. Começando no nível mais baixo, onde são utilizadas as instruções de máquina indo até o nível mais alto, onde processos independentes são executados, passando pelo uso de *threads* nos processos.

Essas *threads* executam instruções independentes umas das outras. Determinados sistemas operacionais efetuam o controle de todas as *threads*, executando seu escalonamento; em outros sistema operacionais, porém, cada processo gerencia suas *threads* sem o conhecimento da cama de sistema em que o processamento está sendo executado. O uso de *threads* explícitas pela linguagem de programação é comum em linguagens de alto nível (como Java e C#).

2.3.4 Paradigmas computacionais

Considerando-se o nível de abstração, existem os seguintes tipos de paradigmas:

- **SPMD *(Single Program, Multiple Data)*:** O modelo *SPMD* foi definido por Darema (2001), e usa aplicações com paralelismo de dados e paralelismo de *threads*;
- **SIMT *(Single Instruction, Multiple Threads)*:** Este modelo é utilizado para gerenciar a execução de várias *threads* na arquitetura (termo utilizado pela NVIDIA para o *CUDA*);
- **MSIMD *(Multiple Single Instruction, Multiple Data)*:** São máquinas que contam com uma memória global ilimitada, e *n* processadores *SIMD* independentes (BRIDGES, 1990).

2.3.5 Métodos de comunicação

Segundo Tanembaum (2001), a comunicação entre processos executados em paralelo é fundamental e pode ser realizada pela troca de mensagens entre os processos ou pelo acesso de variáveis globais disponíveis aos processos. No primeiro caso, em que cada processo possui seu espaço de memória, primitivas *send* e *receive* são utilizadas para a comunicação, especificando o processo de destino que copia a informação para o seu espaço de endereçamento específico.

Esta abordagem é comumente utilizada em sistemas multicomputadores, como *grids* e *clusters*. Já no segundo caso, todos os processos leem e escrevem em uma espaço de memória comum, em que as variáveis podem ser compartilhadas entre vários processos, mais comum em sistemas multiprocessados.

2.3.6 Primitivas de sincronização

Segundo Tanembaum (2001), a sincronização de ações se torna fundamental quando se utiliza a comunicação entre processos paralelos. No momento em que uma variável é logicamente compartilhada se faz necessária a garantia de que não haja o acesso de um processo à variável enquanto outro estiver executando uma ação de escrita na mesma.

Portanto, se faz necessário o uso de um mecanismo de sincronização entre os processos ao acessar os dados. Através dessa sincronização, permite-se o acesso exclusivo aos recursos compartilhados por exclusão mútua dos processos. Essa exclusão mútua pode ser garantida pelo uso de semáforos, travamentos, *mutexes* e regiões críticas (TANEMBAUM, 2001).

Baseado nessas metodologias apresentadas, foram definidas como diretivas na construção do software (TANEMBAUM, 2001):

- Modelo de controle que suporte várias *threads*.
- Granularidade de paralelismo que contemple *threads*.
- Primitivas de comunicação que adotem o compartilhamento de variáveis em uma memória comum.
- Paradigma computacional que suporte *threads* e processos independentes.

2.3.7 Bibliotecas

No desenvolvimento da aplicação foi utilizada a biblioteca *CUDA*, com a camada auxiliar *JCUDA*; estas bibliotecas são responsáveis pela comunicação entre os processadores da *CPU* e da *GPU*, ou seja, a aplicação a ser desenvolvida utilizará destas ferramentas para conversação dos processadores e divisão das tarefas. Outra biblioteca que faz uso da *GPU* para computação heterogênea é o *OpenCL*, descrito no item 2.7.1.2 deste capítulo.

2.3.1.1 CUDA

Segundo Nvidia (2012), *CUDA* é uma plataforma (*framework*) de computação paralela e um modelo de programação criados pela própria empresa. Esta tecnologia tem objetivo de obter uma melhoria de performance ao aproveitar o alto desempenho da *GPU* na execução de tarefas que lhe são atípicas. O alto grau de difusão de *GPUs* habilitadas para *CUDA* demonstra que diversas aplicações científicas e de pesquisa já estão sendo feitas utilizando esta tecnologia, tais como: identificação de placas ocultas em artérias, análise de fluxo de trafego aéreo e visualização de moléculas.

Para o desenvolvimento em *CUDA* se utiliza essencialmente a linguagem de programação *C++* com a adição de alguns *plugins* para o suporte à tecnologia. A *NVIDIA* prove um Conjunto de Desenvolvimento de Software (*SDK*) contendo tanto uma *API* de baixo nível quanto uma de alto nível (NVIDIA, 2012).

Para Kirk e Hwu (2011), a estrutura de uma aplicação escrita com o *CUDA* se apresenta da seguinte maneira:

> Um programa CUDA consiste em uma ou mais fases que são executadas ou no host (CPU) ou em um device como uma GPU. As fases que exibem um pouco ou nenhum paralelismo de dados são implementadas no código. Já as que exibem uma rica quantidade de paralelismo de dados são implementadas no código do device. Um programa CUDA é um código fonte unificado compreendendo código do host e código do device. O compilador C da NVIDIA (nvcc) separa os dois durante o processo de compilação. O código do host é um código ANSI C simples; ele também é compilado com os compiladores C padrão do host e roda como um processo comum da CPU. O código do device é escrito usando ANSI C, estendido com palavras chave para rotular as funções com dados paralelos, chamadas kernels, e suas estruturas de dados associados. O código do device normalmente é compilado ainda pelo nvcc e executado em um device gpu. Em situações as quais nenhum device está disponível ou o kernel é executado de modo mais apropriado em uma CPU, também é possível executar os kernels em uma CPU usando os recursos de emulação no kit de desenvolvimento de software (SDK) CUDA, ou a ferramenta MCUDA.

Para que não fosse necessária a utilização direta da linguagem C, os autores desta monografia fazem uso de uma camada adicional, entre *CUDA* e *JAVA*, afim de simplificar a construção do protótipo – A biblioteca *JCUDA* (uma biblioteca destinada a interagir entre o *JAVA* e as bibliotecas de tempo de execução do *CUDA*)

2.3.1.2 OPENCL

OpenCL foi o primeiro padrão de programação paralela *cross-plataform* livre e de código aberto desenvolvido. Criado pela *AMD*, proprietária da marca, em parceria com *Apple*, *Intel* e *NVIDIA*, esta biblioteca fornece uma *API* de baixo nível para o desenvolvedor, que utiliza um subconjunto de instruções em *C* para a criação e execução de *kernels* (KHRONOS GROUP, 2012).

Kirk e Hwu (2011) definem o *OpenCL* da seguinte maneira:

> OpenCL é uma API padronizada, multiplataforma, para computação paralela baseada na linguagem C. Ela foi projetada para permitir o desenvolvimento de aplicações paralelas portáveis para sistemas com devices de computação heterogêneos. O desenvolvimento da OpenCL foi motivado pela necessidade de uma plataforma de desenvolvimento de aplicação de alto desempenho, padronizada, para a grande variedade de plataformas de computação paralela em rápido crescimento.

2.4 A GPU

Stringhini, Gonçalves e Goldman (2012), afirmam que o termo Computação Heterogênea tem sido utilizado para falar de diferentes arquiteturas em um mesmo nó (nodo) computacional; faz-se uso deste aparato para que se tenha um melhor desempenho das aplicações executadas em um determinado ambiente computacional, fazendo uso de recursos que comumente encontram-se ociosos. A *GPU* (um exemplo de recurso acelerador) é um componente importantíssimo quando se fala de heterogeneidade computacional; o processador gráfico, comumente, é um dos recursos de maior potencial que se encontra em grande parte do tempo, ocioso.

Uma situação em que se pode observar o uso de computação heterogênea é divisão de tarefas entre os recursos disponíveis; ilustrando: permitir que um código sendo executado na *CPU* possa enviar trabalho para um acelerador. Neste tipo de computação a heterogeneidade está em desenvolver aplicações que executem em determinados dispositivos da máquina. No entanto, vale lembrar que o modelo de programação dos aceleradores nem sempre é adequado para todo o tipo de algoritmo, sendo importante que o desenvolvedor saiba relacionar as tarefas mais adequadas

para cada tipo de acelerador e para uma *CPU* tradicional (STRINGHINI; GONÇALVES; GOLDMAN, 2012).

Tanenbaum (2001) aponta que análise sobre qual metodologia de desenvolvimento de software heterogêneo se deve usar depende de cinco fatores base (descritos anteriormente, com mais profundidade no item 2.3); os autores entendem como sendo alicerce para obtenção do conhecimento acerca de heterogeneidade computacional.

Du e outros (2012) explanam que, o maior exemplo de aceleradores são as *GPU's*. Inicialmente utilizadas somente para processamento gráfico, rapidamente seu potencial foi notado e versões para processamento genérico (*GPGPU – General Purpose GPU*) surgiram. As *GPUs* atuam em conjunto com a *CPU;* as duas principais fabricantes destes hardwares são a *NVIDIA* e a *AMD*. As *GPU's* podem atuar em conjunto com os processadores Intel e AMD.

O paralelismo utilizado por elas é o *SIMD*, em que milhares de *threads* são executados simultaneamente nas centenas de núcleos de uma *GPU*. O processo principal executado na *CPU* é o responsável por iniciar as *threads* na *GPU*. As *GPU*'s tem sua própria hierarquia de memória e os dados são transferidos através de um barramento *PCI express*. (Du et al., 2012).

2.4.1 Arquitetura da GPU

A arquitetura das aceleradoras da *NVIDIA* apresentada na série *GTX400* é a Fermi, que permite a execução concorrente de *kernels*; entretanto na nova série *GTX600* a arquitetura *Kernel* foi introduzida. As *GPUs* possuem centenas de núcleos simples que executam em centenas a milhares de *threads* concorrentes o mesmo código. Ao contrário do modelo tradicional de processo paralelo em que múltiplos núcleos completos executam diferentes *threads* e processos (NVIDIA, 2012).

De acordo com a Nvidia (2012), as *CPUs*, formadas de núcleos completos, conseguem executar instruções paralelas e fora de ordem, além de conterem uma melhor hierarquia de *cache*. As *GPUs*, por outro lado, contém unidades de controle e de execução mais simples, que distribuem apenas uma instrução para um conjunto

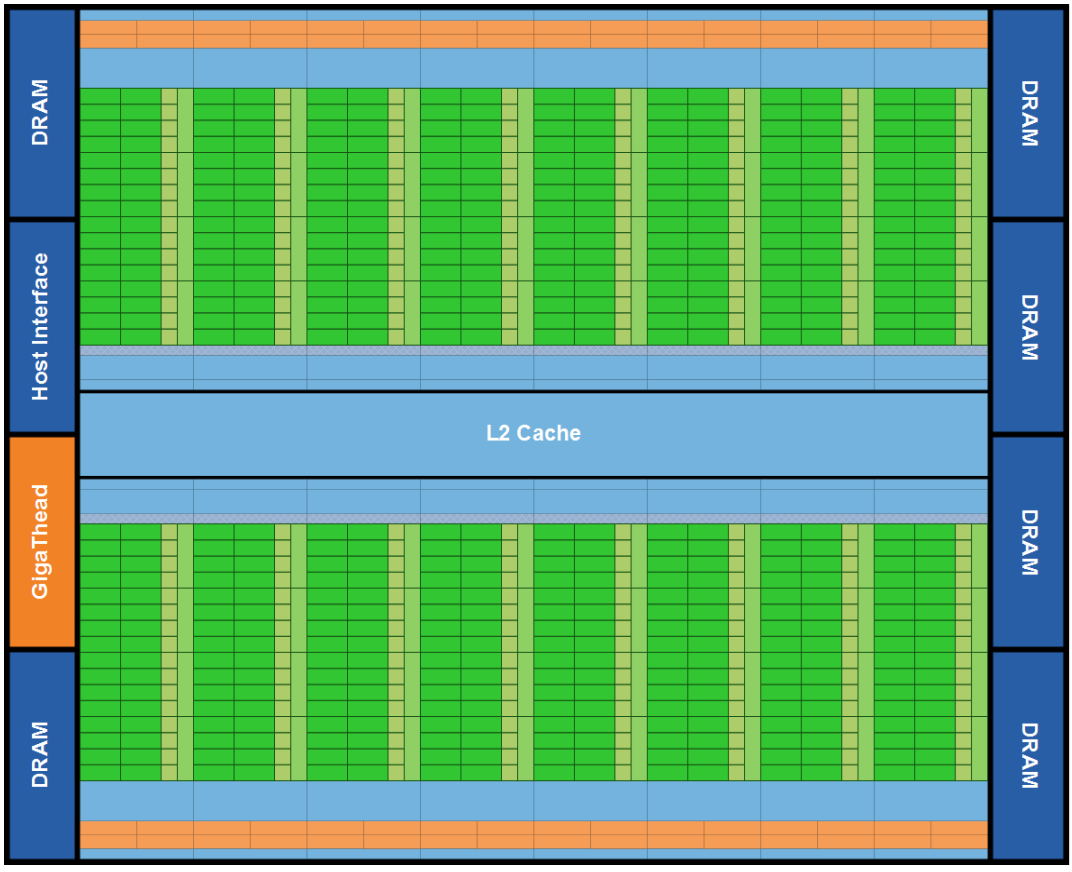
de núcleos que a executará em ordem. Este modelo de execução de instruções é conhecido como *SIMT* (*Single Instruction Multiple Threads*), derivado do modelo *SIMD* (*Single Instruction Multiple Data*).

Outra característica das *GPUs* é possuírem uma memória compartilhada entre todos os núcleos. As aceleradoras mais novas, porém, já possuem caches L1 e L2; além disso, algumas possuem memórias diferenciadas para melhoria de desempenho (NVIDIA, 2012).

2.4.1.1 Arquitetura Fermi

Esta arquitetura da *NVIDIA* prioriza uma maior quantidade de transístores às unidades de execução, ao invés de às unidades de controle e cache. A Figura 1 apresenta uma visão geral da arquitetura Fermi (NVIDIA, 2012).

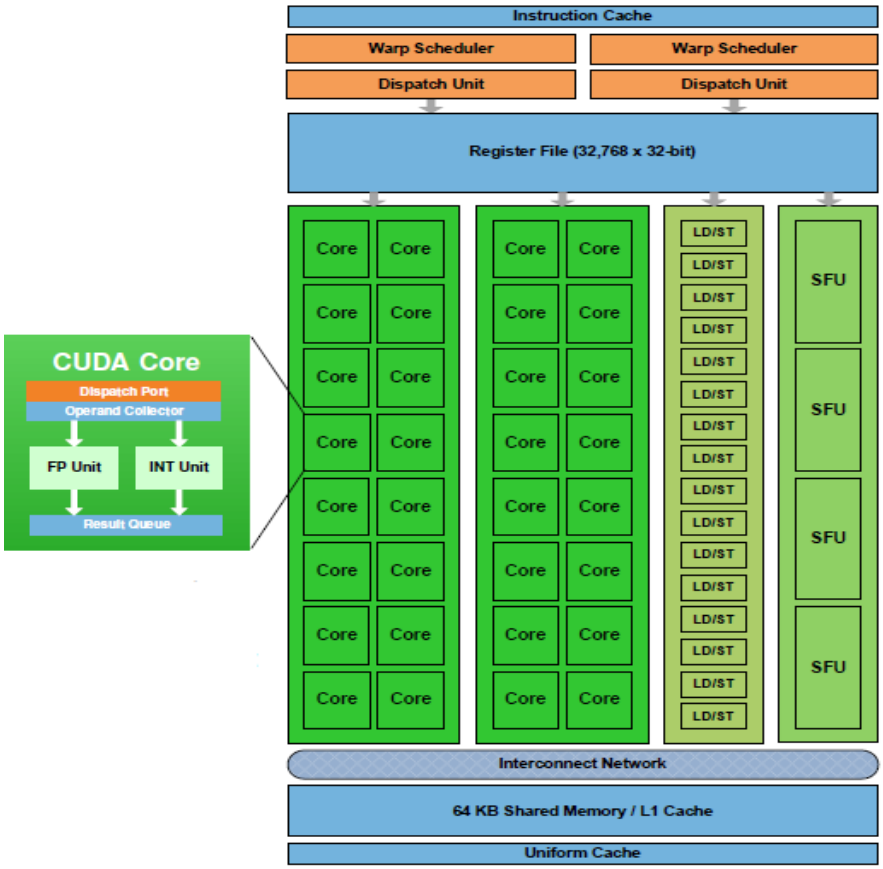
Figura 1 – Visão geral da arquitetura Fermi.
Fonte: Nvidia, 2012.



A arquitetura de Fermi conta com: 16 *Streaming Multiprocessors* (*SM*), cada um composto de 32 núcleos de processamento (totalizando 512 núcleos) uma memória *cache* de segundo nível (*L2*) que é compartilhada por todos os *SM* – cada *SM* possui sua própria cache de primeiro nível (*L1*) e seis partições de memória *DRAM GDDR5* de 64 *bits*. Sua interface de comunicação com a *CPU* é via *PCI-Express*; um escalonador global de *threads* (*GigaThread*) é utilizado para a distribuição dos blocos de *threads* para os escalonadores das *SMs* (NVIDIA, 2012).

A Figura 2 apresenta os componentes de uma *SM*, seus registradores, núcleos de processamento, as unidades de escalonamento e despacho de instruções, a cache de nível 1, entre outros componentes.

Figura 2 – Arquitetura de uma *SM*.
Fonte: Nvidia, 2012.

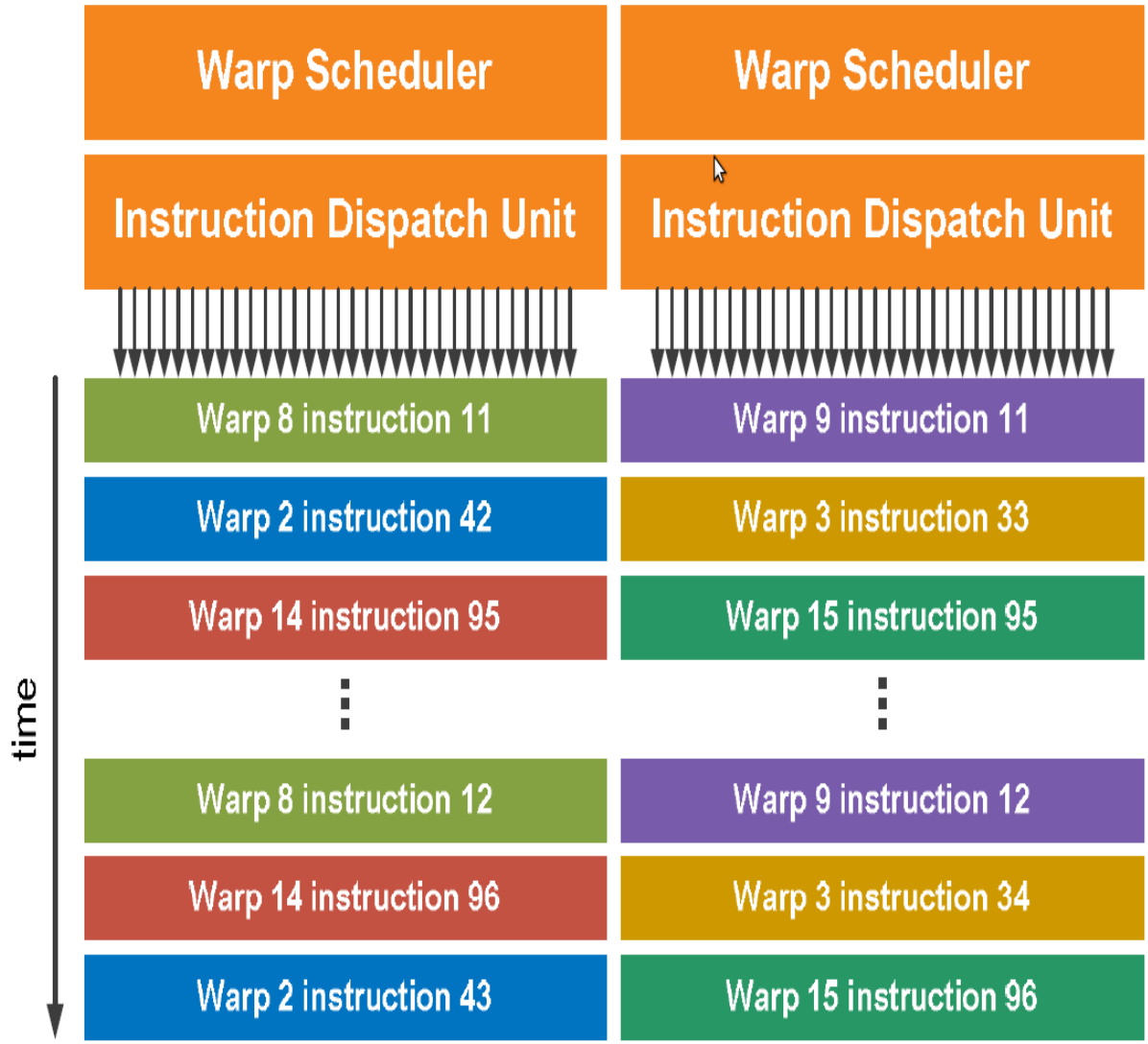


Uma *GPU* resume-se basicamente a um conjunto de multiprocessadores (*SMs*) conectados a uma *cache* de nível 2 e a uma memória global. O escalonamento das *threads* é feito em grupos de 32 que executam paralelamente (*warp*). Na arquitetura Fermi cada multiprocessador tem dois escalonadores de *warps* e duas

unidades de despacho de instruções, possibilitando a distribuição e execução de dois *warps* concorrentemente (dual *warps*) (NVIDIA, 2012).

A Figura 3 apresenta o esquema de escalonamento da arquitetura Fermi, onde o escalonador seleciona dois *warps* independentes possibilitando o despacho de uma instrução de cada warp para um grupo de 16 *cores* melhorando o desempenho.

Figura 3 – Fermi Dual Warp.
Fonte: Nvidia, 2012.



2.4.1.2 Arquitetura Kepler

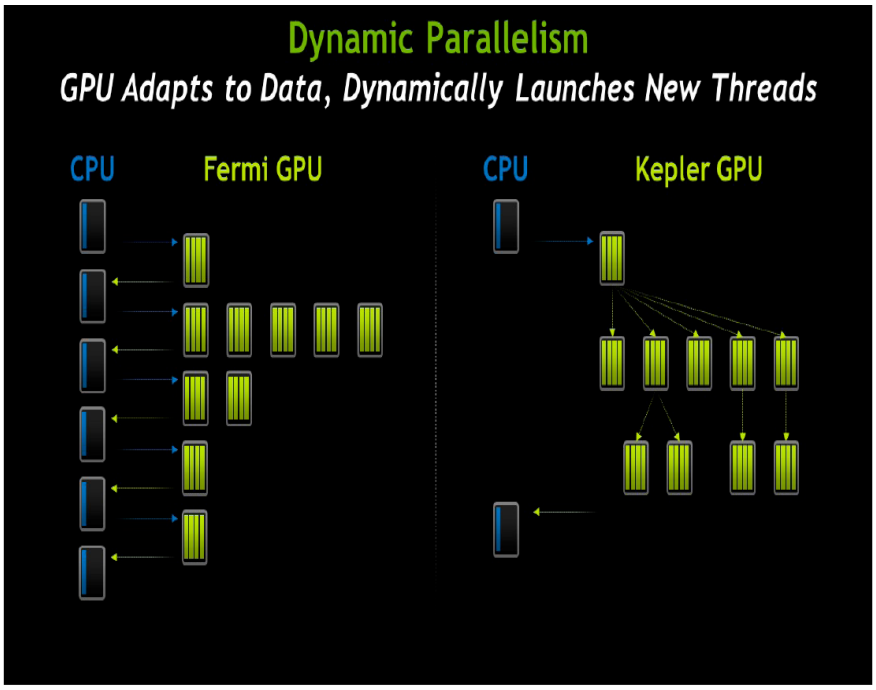
Esta arquitetura da *NVIDIA* é implementada nas novas *GPUs* da *NVIDIA*. Na Kepler GK110 as características mais importantes estão relacionadas com melhorias nos multiprocessadores, paralelismo dinâmico e a tecnologia *Hyper Q*. As *GPUs* apresentam um conjunto de até 15 multiprocessadores (*SMX*) com 192 núcleos cada como mostrado na Figura 4 (NVIDIA, 2012).

Figura 4 – Multiprocessador (SMX) Kepler.
Fonte: Nvidia, 2012.

Essa organização confere mais autonomia à *GPU*. Com o conceito de Paralelismo Dinâmico introduzido por esta arquitetura, a *GPU* dispara dinamicamente a execução de novas *threads*, sincroniza os resultados e controla o escalonamento, adaptando-se ao fluxo de execução sem interferência do processo sendo executado

na *CPU*. Esta independência possibilita que vários processos sejam executados diretamente na *GPU*, e ainda que *kernels* possam fazer chamadas a outros *kernels* criando os recursos necessários para a execução deles, independente do código executado no *host*. Este paralelismo dinâmico pode ser visto na comparação entre a execução de *threads* das arquiteturas Fermi e Kepler da Figura 5 (NVIDIA, 2012).

Figura 5 – Paralelismo Dinâmico.
Fonte: Nvidia, 2012.

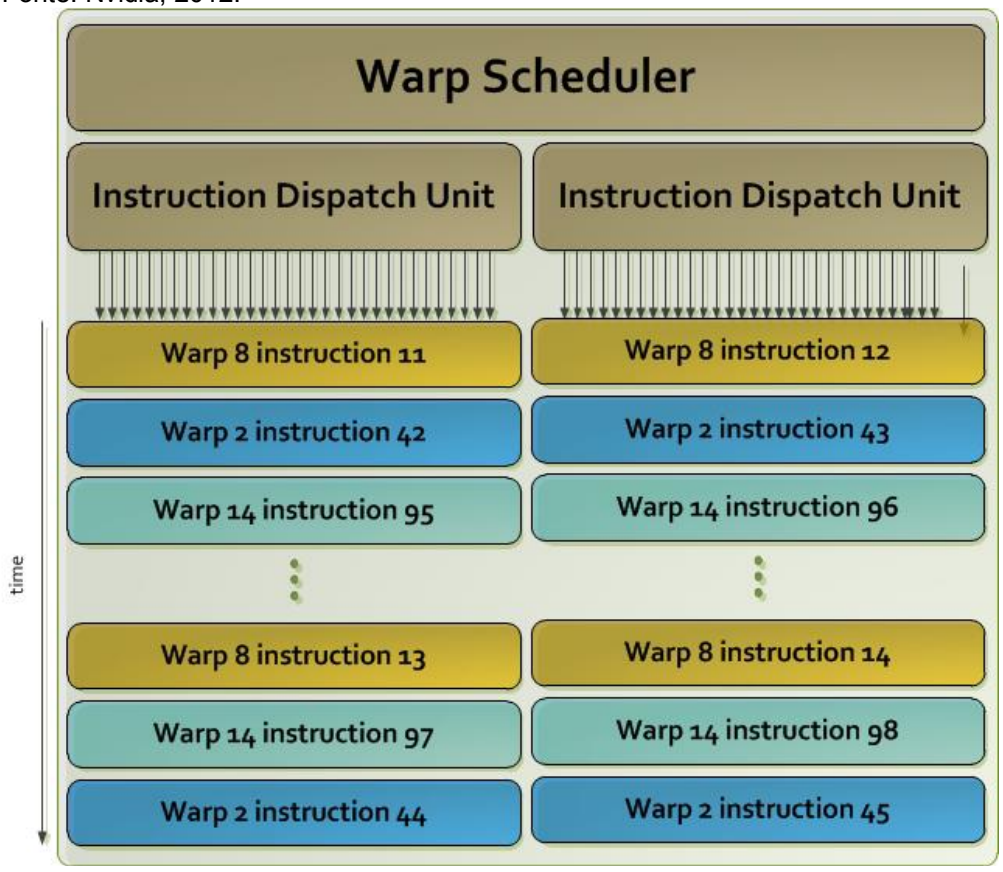


No comparativo mostrado na Figura **5** percebe-se a autonomia dada às *GPUs* da arquitetura Kepler. Enquanto que nas arquiteturas anteriores as instruções despachadas à *GPU* dependiam do código principal executado no *host* para sincronizar dados e disparar novos *kernels*, na nova arquitetura Kepler uma função *kernel* pode gerar quantas chamadas a outros *kernels* forem necessárias dentro da própria *GPU* adaptando-se ao fluxo de execução (NVIDIA, 2012).

Na tecnologia *Hyper-Q*, diferentes *threads* do *host* disparam a execução de *kernels* simultaneamente, sendo possíveis 32 conexões ao mesmo tempo com os cores da *CPU*, contra 1 conexão das arquiteturas anteriores. Com essa nova proposta a *GPU* pode ser utilizada de modo que os processos podem disparar a execução simultânea de *kernels* na *GPU*, melhorando o desempenho dessas aplicações (NVIDIA, 2012).

O escalonamento na arquitetura Kepler, em que cada *SMX* possui quatro escalonadores de *warps* e oito unidades de despacho de instruções, possibilita que quatro *warps* com 32 *threads* paralelas possam ser escalonados e executados concorrentemente (*quad warp scheduler*). A organização dos escalonadores é mostrada na Figura 6, cada um deles com duas unidades de despacho de instruções (NVIDIA, 2012).

Figura 6 – Kepler Quad-Warp
Fonte: Nvidia, 2012.



2.5 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Neste capítulo foram expostos os principais conceitos responsáveis por definir um ambiente computacional heterogêneo; passando por um apanhado geral do que é computação heterogênea e algumas de suas características (processamento de alto

desempenho, suas vantagens, utilização de paralelismo – paralelismo implícito e explícito – distribuição, *grid* e *cluster*).

Após a construção do embasamento foram expostos conceitos da *GPU* (componente a ser utilizado na solução proposta – computação heterogênea fazendo uso da *GPU* como recurso acelerador); foram abordados tópicos sobre algumas arquiteturas deste componente (arquitetura Fermi e Kepler), elucidando através de imagens seu funcionamento.

Além dos conceitos de *hardware*, se explanaram pontos sobre a estrutura do *software* paralelo tais como: modelos de controle, *threads*, granularidade do paralelismo, paradigmas computacionais (SPMD, SIMT, MSIMD), métodos de comunicação, primitivas de sincronização e bibliotecas auxiliares (*CUDA* e *OpenCL*).

3 MÉTODO

De acordo com Silva e Menezes (2005), a metodologia tem como objetivo auxiliar o(s) pesquisador(es) durante a pesquisa, demonstrando a este(s) o caminho que deve ser trilhado. Os itens deste capítulo representam o método utilizado para o desenvolvimento deste projeto de monografia; encontram-se subdivididos em: caracterização do tipo de pesquisa, etapas da pesquisa, proposta ou arquitetura da solução e delimitações, o cronograma se encontra no APÊNDICE A – Cronograma.

3.1 CARACTERIZAÇÃO DO TIPO DE PESQUISA

Este trabalho tem como objetivo apresentar os conceitos base acerca de computação heterogênea, com finalidade de criar uma estrutura teórica que sustente (justifique) a utilização de heterogeneidade a nível de *GPU* para melhoria no tempo de execução de uma aplicação.

Com base nos conceitos apresentados, foi modelada (no padrão *UML*) e desenvolvida (em JAVA, fazendo uso de bibliotecas externas – *JCUDA*, uma interposição entre o *CUDA* e o *JAVA*) um protótipo, a fim de demonstrar um comparativo entre o desempenho em sistemas computacionais que utilizam e não utilizam de heterogeneidade.

Do ponto de vista da natureza, esta pesquisa enquadra-se no molde de pesquisa aplicada, pois possui objetivo de gerar conhecimentos para a aplicação prática, dirigindo-os à solução de problemas (SILVA; MENEZES, 2005). A forma de abordagem do problema é baseada no propósito de pesquisa quantitativa. Os dados coletados serão referentes ao tempo de execução da aplicação, diretamente ligados ao processamento. Estes dados serão apresentados em forma de uma tabela, para facilitar a análise de desempenho.

A pesquisa é classificada como quantitativa, pois considera que tudo pode ser mensurável, ou seja, pode se traduzir por via de números, informações para

classificação e análise. Requer o uso de recursos e de técnicas estatísticas (porcentagem) (LAKATOS; MARCONI, 1985).

Para Bonat (2009), a pesquisa quantitativa infere acerca do que pode ser mensurado, contado, medido; nota-se portanto, um alto teor descritivo. A análise das questões é afastada de opiniões pessoais, tornando-a privilegiada pelo positivismo. Exemplificando: o tempo de processamento necessário para que a aplicação execute o cálculo da matriz identidade.

Quanto aos objetivos, a pesquisa é de ordem exploratória, pelo fato desta, de acordo com Gil (1999), visar proporcionar maior familiaridade com o problema a fim de torná-lo explícito ou a construir hipóteses. Envolve levantamento bibliográfico; testes realizados pelos autores, para comprovar os estudos acerca da solução proposta; análise de um protótipo, que estimule a compreensão e a dimensão da solução proposta. Assume, em geral, as formas de pesquisas bibliográficas e estudo de caso.

3.2 ETAPAS

As etapas de desenvolvimento da proposta (utilização de computação heterogênea para melhoria de processamento em uma aplicação modelo fazendo uso de *GPU*) subdividem-se em:

a) revisão bibliográfica: consiste no levantamento do material responsável por determinar o embasamento teórico disposto no capítulo 2; apresenta os principais conceitos de computação heterogênea (direcionados a hardware e software), juntamente com a apresentação de bibliotecas que irão auxiliar no desenvolvimento da aplicação modelo;
b) modelagem e desenvolvimento da aplicação modelo: representa o processo de modelagem da aplicação (fazendo uso de *UML*) e o desenvolvimento da mesma (em JAVA), incluindo a utilização da biblioteca externa (*CUDA/JCUDA*);
c) processamento em ambiente computacional comum x processamento em ambiente computacional heterogêneo: com base na estrutura apresentada no

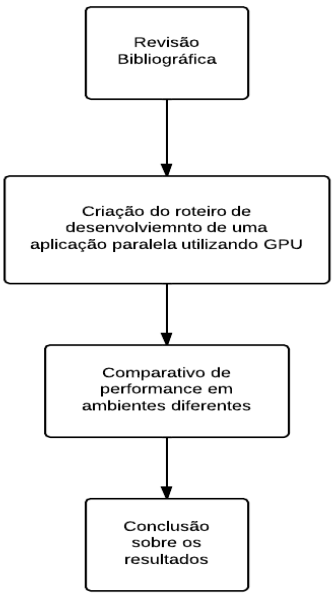
capítulo 2, apresenta um comparativo entre os sistemas computacionais que fazem uso dos que não fazem uso de heterogeneidade;

d) propor a utilização da *GPU* como alternativa de processamento de aplicações (trechos de código) quando este recurso acelerador encontrar-se ocioso, ou quando a *CPU* encontrar-se sobrecarregada;

e) conclusões sobre a utilização da *GPU* (computação heterogênea) para melhoria de processamento em um ambiente computacional: apresentar os resultados obtidos por via da aplicação modelo e propor a utilização deste tipo de solução quando viável.

A Figura 7 mostra o fluxograma da sequência de realização das etapas deste trabalho.

Figura 7 – Fluxograma de etapas.
Fonte: Os autores, 2013.



3.3 PROPOSTA OU ARQUITETURA DA SOLUÇÃO

Para dar início a parte prática do trabalho, foi criado e configurado o ambiente de desenvolvimento (instalação e configuração das aplicações necessárias

para criar a aplicação modelo). Feito isto, foi construída uma aplicação modelo que pode ser executada tanto em um ambiente que se utiliza de heterogeneidade, quanto em um ambiente que não faz uso da mesma. A aplicação modelo executa o cálculo da matriz identidade de uma matriz quadrada com tamanhos pré-definidos (512 x 512, 1024 x 1024, 2048 x 2048 e 4096 x 4096), através do método Gaussiano (Gauss-Jordan).

Em álgebra linear a Eliminação Gaussiana (conhecida também como redução) é um algoritmo utilizado para resolver sistemas de equações lineares; como variação deste método tem-se o Gauss-Jordan, que consiste na aplicação de operações elementares às linhas da matriz aumentada do sistema até que se obtenha uma matriz numa forma em que o sistema associado a esta matriz seja de fácil resolução (SANTOS, 2013).

O método de Gauss-Jordan foi selecionado para efetuar as operações de matriz pois utiliza operações elementares em uma matriz unitária; é adequado para rotinas computacionais que envolvem sistemas lineares, uma vez que envolve um algoritmo mais rápido de ser implementado e calculado por processadores (CANTÃO; STARK, 2013).

Dentro do algoritmo de Gauss-Jordan podemos extrair um trecho de código que executa a chamada eliminação com a subtração das linhas por um múltiplo da linha em que se encontra o algoritmo a fim de eliminar todos os elementos na coluna do elemento em questão:

```
for ( j = 1:(n - 1) ) //para cada coluna
    …
    for ( i = (j + 1):n ) //para cada linha
            …
            for ( k = (j + 1):(n + 1) ) // para cada elemento
                    …
            endfor
    endfor
endfor
```

Podemos então medir a eficiência computacional do algoritmo descobrindo o número de operações aritméticas necessárias para realizar a redução de linha.

Verificamos que temos operações para todos os elementos “k = (j + 1):(n + 1)” em cada iteração pela matriz “for (i = (j + 1):n)” e “for (j = 1:(n - 1))“. Temos então (n*n)*n*n que nos leva a uma complexidade aritmética de O(n4).

O usuário seleciona se está operando em um ambiente computacional heterogêneo (que conta com a presença do recurso acelerador proposto como solução – A *GPU*) ou não. Para mensurar o desempenho são medidos e analisados os tempos de execução de cada uma das matrizes quadradas executando o método Gaussiano (Gauss-Jordan), com e sem a utilização da *GPU*.

3.4 DELIMITAÇÕES

Nesta seção estão destacadas as delimitações do trabalho, com intuito de focar na criação do roteiro de um ambiente de desenvolvimento de uma aplicação paralela, que faz uso de *GPU* (computação heterogênea) para obtenção de melhoria de processamento em um ambiente computacional, conforme os itens colocados a seguir:

a) são decorridas as informações mais relevantes para o entendimento do funcionamento da computação heterogênea, entretanto, não são aprofundados, nem expostos conceitos que tem relação com paralelismo mas não possuem relação com computação heterogênea a nível de *GPU*;
b) apesar de apresentado, o *OpenCL* não é objeto de estudo nesta monografia; o protótipo faz uso exclusivo da biblioteca *CUDA*, utilizando da biblioteca auxiliar *JCUDA* para seu funcionamento na linguagem de programação *JAVA*;
c) a aplicação modelo, apesar de produzida na linguagem de programação *JAVA* não foi testada em qualquer outro sistema operacional compatível com esta linguagem, que não o *Windows* (Versão 8 *Professional)*;
d) os resultados comparativos são apresentados por via de uma tabela, onde cada uma das linhas representa uma matriz quadrada e as colunas representam a utilização e a não utilização do *CUDA*;
e) as etapas metodológicas seguidas neste trabalho estão apresentadas no capítulo 3, item 3.2 (Etapas);

f) a confecção da monografia seguirá estritamente as Etapas propostas (item 3.2) e no APÊNDICE A – CRONOGRAMA.

4 DESENVOLVIMENTO E RESULTADOS

O desenvolvimento desta monografia consiste na elaboração de um roteiro de desenvolvimento para um software paralelo utilizando a GPU como recurso acelerador, exemplificando o funcionamento de aplicativos heterogêneos por via de uma aplicação exemplo – protótipo; este protótipo tem como finalidade a demonstração do ganho de desempenho fazendo uso do recurso acelerador escolhido pelos autores – a *GPU*. A aplicação é constituída de uma matriz do tipo ponto flutuante (*float*), gerada automaticamente pelo protótipo.

Através do método Gaussiano (Gauss-Jordan) é calculada a matriz identidade. Com o auxílio das bibliotecas auxiliares (*CUDA/JCUDA*), a carga de processamento é direcionada para a *GPU*. Posteriormente, um relatório contendo os tempos de processamento é gerado, para que seja comprovado o ganho de desempenho fazendo uso de um recurso computacional que encontra-se, normalmente, ocioso – e raramente é utilizado neste tipo de situação.

Os subitens apresentados a seguir têm como objetivo demonstrar a modelagem do sistema; estes requisitos de engenharia de software são considerados indispensáveis pelos autores deste trabalho – quando há envolvimento do desenvolvimento de software – tanto no que se refere a organização do projeto, quanto a elucidação de cada uma das etapas quando apresentadas para terceiros.

4.1 REQUISITOS DE SOFTWARE

Segundo Sommerville (2011), os requisitos de software para este projeto podem ser classificados da seguinte maneira:

4.1.1 Requisitos Funcionais

Para melhor entendimento acerca do processo de construção da aplicação exemplo, serão colocados alguns requisitos funcionais do sistema, ou seja, definições sobre os componentes ou funções do software. A seguir estão listados os requisitos funcionais levantados para o exemplo:

4.1.1.1 Requisitos de Usuário

Os autores consideram para o protótipo os seguintes requisitos de usuário – O papel do usuário na utilização do sistema, ou seja, o que o usuário será capaz de fazer (interação entre usuário/sistema):

- Geração das matrizes base;
- Inicialização da operação de cálculo da matriz identidade – através do método Gaussiano (Gauss-Jordan).

4.1.1.2 Requisitos de Sistema

Para o desenvolvimento do protótipo foram levantados os seguintes requisitos de sistema, que são informações de grau técnico envolvidos na modelagem e desenvolvimento do protótipo:

- a criação das matrizes base deve ocorrer de maneira aleatória;
- as matrizes base e a matriz resultante deverão ser armazenadas em arquivos texto separados para uso posterior;
- os resultados mostrados no relatório devem ser armazenados em um arquivo texto;
- o Software deve emitir o seguinte relatório:

- Relatório de uso de recurso de processamento da *CPU* e da *GPU* durante a operação de multiplicação das matrizes, especificando se a carga computacional está ou não sendo distribuída entre ambas.

4.1.2 Requisitos Não Funcionais

Do mesmo modo, seguem os requisitos não funcionais necessários para aplicação modelo. Estes requisitos encontram-se agrupados em requisitos de produto e requisitos externos.

4.1.2.1 Requisitos de produto

Destacam-se dentro dos requisitos de produto os seguintes itens:

a) **Usabilidade**:

- A aplicação deve ser desenvolvida na linguagem *JAVA* versão 1.7 ou superior;
- a aplicação deve ser desenvolvida para *desktop*;
- a aplicação deve ser desenvolvida utilizando os conceitos de orientação a objetos;
- a aplicação deve utilizar interfaces padrão para realizar a interação com o usuário;
- a aplicação deverá utilizar arquivo texto para realizar a persistência dos dados.

b) **Desempenho**:

- A aplicação deve consumir menos de 20*MB* de memória durante a sua execução;
- uma operação efetuada através do sistema deverá levar poucos milissegundos para ser executada.

4.1.2.2 Requisitos externos

Sobre os requisitos externos, os autores avaliaram, inicialmente, que o seguinte requisito externo deverá ser cumprido:

- Os dados devem ser visíveis através de programas externos, como Bloco de Notas do *Windows* e *Notepad*++.

4.2 MODELAGEM DA IMPLEMENTAÇÃO

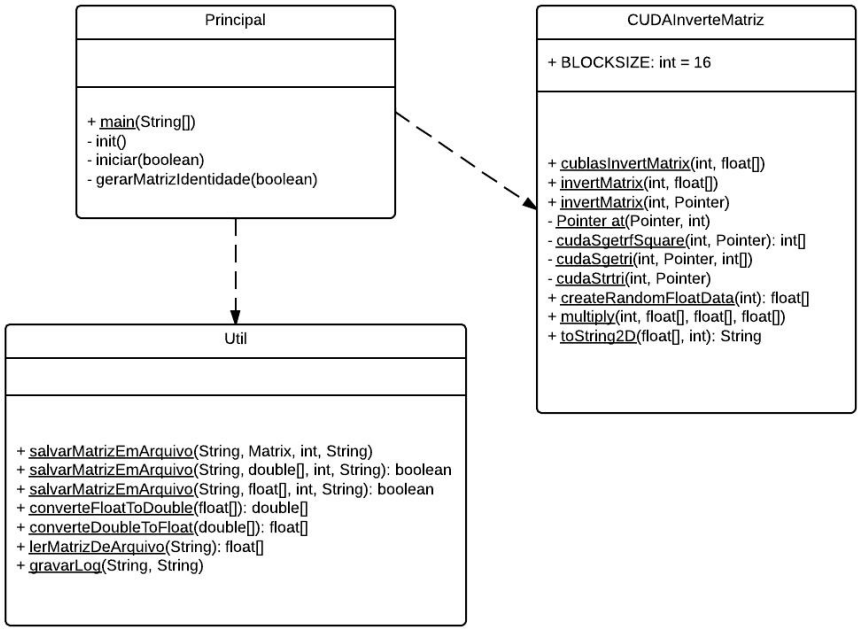
Para embasar a proposta, construiu-se um diagrama de classe no padrão *UML*, com a finalidade de apresentar a ideia do protótipo, executado diretamente no método *main* (Principal) da aplicação.

4.2.1 DIAGRAMA DE CLASSE

A

Figura **8** apresenta o diagrama de classe da aplicação modelo, utilizada com o intuito de elucidar o funcionamento de uma aplicação paralela utilizando a *GPU* como recurso acelerador:

Figura 8 – Diagrama de Classes do Protótipo.
Fonte: Os autores, 2012.



O programa consiste na classe Principal, responsável pelo gerenciamento das operações para a obtenção da matriz identidade. Ela utiliza em relacionamento de dependência a classe Util para as operações de leitura e gravação dos arquivos de armazenamento das matrizes base dos cálculos e das matrizes resultantes dos mesmos e ainda faz a gravação dos logs das operações, utilizados na descrição dos resultados e confecção dos gráficos. A classe CUDAInverteMatriz, também utilizada pela classe principal através de dependência, realiza as operações aritméticas com o uso da *GPU* e do *JCUDA*; Nas operações aritméticas sem o uso da *GPU* a classe Principal se encarrega de utilizar a biblioteca *la4j* nos cálculos – classe esta que calcula a identidade da matriz selecionada através do método Gaussiano (Gauss-Jordan).

4.3 TECNOLOGIAS UTILIZADAS

As configurações de *hardware* dos computadores utilizados no desenvolvimento/execução do protótipo foram as mesmas (nos computadores dos

dois autores); que são: Processador *Intel Core i7 2670QM 2.20GHZ Sandy Bridge*, Hynix/Hyundai 8*GB* de memória *RAM DDR3 (1600MHZ)* – dispostos em dois *slots* de *4GB*, Placa de Vídeo *nVIDIA* 650M 2*GB* (dedicada) e disco rígido *Vertex SSD 120GB*.

O ambiente de desenvolvimento foi construído sobre a plataforma *Windows*, versão 8 *Professional*. A linguagem de programação escolhida foi *JAVA*, por tratar-se de uma linguagem de programação *open source* (de código aberto); a versão do *JAVA* instalada no ambiente de desenvolvimento foi a que acompanha o *kit* do *JDK 1.7.013*.

No desenvolvimento do protótipo foi utilizada a *IDE* Eclipse, versão *Juno*, que também é uma ferramenta *open source* para desenvolvimento de *software*. Para utilização do hardware acelerador (placa gráfica – *Nvidia Geforce GT 650M*) foi instalada a camada intermediária de software responsável por criar o ambiente heterogêneo, o *CUDA*, de versão 5.0.

O *CUDA* foi selecionado para ser utilizado como solução para criação de aplicações paralelas por ser uma biblioteca proprietária da NVIDIA, fabricante das *GPU*'s disponíveis para os desenvolvedores desta monografia, ou seja, a documentação desta biblioteca é mais direcionada e específica para o hardware em questão.

Para que o *CUDA* funcionasse na linguagem de programação *JAVA* foi necessária a utilização de uma biblioteca auxiliar, com o intuito de abstrair um pouco mais a camada de codificação, para que não fosse necessário trabalhar com outra linguagem de baixo nível. Este intermédio entre *JAVA* e *CUDA* é exercido pela biblioteca *JCUDA*.

O *JCUDA* foi escolhido como biblioteca intermediária por possuir uma quantidade maior de material de apoio, além de contar com uma quantidade de usuários utilizadores significativa, resultando em um background mais robusto para a solução de problemas ocorridos.

4.4 ROTEIRO DE CONFIGURAÇÃO DO AMBIENTE DE DESENVOLVIMENTO

Para criação da aplicação exemplo (e outras aplicações paralelas utilizando as tecnologias citadas) foi confeccionado um roteiro, com finalidade de apresentar os passos seguidos pelos autores para alcançar o objetivo proposto:

1. Instalação do *CUDA Toolkit 5.0 64bit*. Deve se ignorar o alerta sobre o *.Net2.0*, pois o *Windows 8* já possui o *.Net4* instalado por padrão. O *CUDA* instala automaticamente o *Visual Studio C++ 8*;

2. Instalação do JAVA *JDK 1.7* (versão mais atual – *1.7.013* utilizada);

3. Instalação do *Eclipse Juno 4.2* (*Download* do *ZIP* e descompactação na pasta de destino);

4. Instalação do *Microsoft Visual Studio 2010*;

5. Configuração da Variável de Ambiente *PATH* da seguinte maneira: ***%PATH%;DRIVER_LOCAL:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin;DRIVER_LOCAL:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE***;

6. *Download* dos arquivos *.dll* e *.jar* do *JCUDA* em: http://www.jcuda.org/downloads/downloads.html, onde o arquivo baixado deve ser: *JCuda-All-0.5.0-bin-windows-x86_64*;

7. *Download* do arquivo de *utils* do *JCUDA* em: http://www.jcuda.org/utilities/utilities.html, onde o arquivo baixado deve ser: *jcudaUtils-0.0.4.jar*;

8. *Download* de uma aplicação exemplo utilizando *JCUDA*: http://www.jcuda.org/samples/samples.html, onde o arquivo baixado deve ser: *Matrix Inversion*;

9. Criação de um novo "*Java Project*" no *Eclipse*;

10. Importar aplicação (arquivos *src* de exemplo baixados);

11. Importar todos os *.jars* necessários para o funcionamento do exemplo, que são

- *jcusparse-0.5.0.jar;*
- *jcurand-0.5.0.jar;*
- *jcufft-0.5.0.jar;*
- *jcudaUtils-0.0.4.jar;*
- *jcuda-0.5.0.jar;*
- *jcublas-0.5.0.jar*.

12. Copiar arquivos de *.dll* para a raiz do projeto, que são:

- *JCublas2-windows-x86_64.dll;*
- *JCublas-windows-x86_64.dll;*
- *JCudaDriver-windows-x86_64.dll;*
- *JCudaRuntime-windows-x86_64.dll;*
- *JCufft-windows-x86_64.dll;*
- *JCurand-windows-x86_64.dll;*
- *JCusparse2-windows-x86_64.dll;*
- *JCusparse-windows-x86_64.dll.*

13. Copiar a pasta com os *kernels* para a raiz do projeto, a pasta contém os seguintes itens:

- *kernel/GPUadjustRow_kernel.cu;*
- *kernel/GPUeliminateBlock_kernel.cu;*
- *kernel/GPUeliminateCol_kernel.cu;*
- *kernel/GPUeliminateRest_kernel.cu;*
- *kernel/GPUnormalizeDiag_kernel.cu;*
- *kernel/GPUsetIdentity_kernel.cu;*

14. Executar o projeto *MatrixInvertSample.main()*.

Seguindo estes passos, o ambiente de desenvolvimento está preparado para desenvolver aplicações paralelas utilizando *CUDA* e *JCUDA*.

4.5 APRESENTAÇÃO DO SISTEMA

A aplicação exemplo não possui uma complexidade elevada; seu objetivo é apresentar a diferença de processamento entre um ambiente que conta com o recurso acelerador *versus* um ambiente que não possui este tipo de hardware. A interatividade do usuário com o sistema se dá via console; o usuário deve escolher se quer gerar uma matriz aleatoriamente ou se deseja ler uma matriz gerada anteriormente, armazenada em um arquivo de texto.

Caso o usuário opte por gerar uma matriz nova, ele deve selecionar o seu tamanho, que é pré-definido pelo sistema (512 x 512, 1024 x 1024, 2048 x 2048 ou 4096 x 4096); caso o usuário opte por efetuar a leitura de uma matriz a partir de um arquivo de texto (matriz gerada anteriormente), ele deve selecionar o tamanho desta matriz, para que o sistema saiba qual será a matriz quadrada a ser utilizada na próxima etapa de execução do protótipo.

Independente da seleção do usuário, na etapa subsequente deve ser efetuada a seleção do tipo de paradigma computacional responsável por efetuar a o cálculo da matriz identidade, através do método Gaussiano (Gauss-Jordan); nesta etapa o usuário poderá escolher entre utilizar ou não o *CUDA*.

Para efetuar um controle do conjunto de dados utilizado nos cálculos do protótipo, estes são salvos em um arquivo de texto, juntamente com o *log* de cada opção selecionada, para posteriores verificações e comparações, por parte dos autores e examinadores.

4.6 DEMONSTRAÇÃO DE RESULTADOS

Para demonstrar os resultados foi realizada a medição dos tempos de execução, sendo que os tempos resultantes (em milissegundos) correspondem ao desempenho da aplicação utilizando computação heterogênea (*CUDA*) e não utilizando computação heterogênea. Para garantir que o procedimento obtivesse um

resultado legítimo, a amostragem utilizada nas duas situações estava composta do mesmo conjunto de dados. A

Tabela **1** mostra os resultados obtidos, em milissegundos:

Tabela 1 – Tempos de execução do protótipo (em milissegundos)

		Desempenho (milissegundos - *ms*)
Linhas x Colunas	**Sem *CUDA***	**Com *CUDA***
512	47.929	2.718
1024	787.098	3.200
2048	7.191.531	7.600
4096	106.176.500	36.988

Fonte: Os Autores, 2013.

A Tabela 2 mostra os mesmos tempos de execução convertidos em minutos:

Tabela 2 – Tempos de execução do protótipo (em minutos)

		Desempenho (minutos - *min*)
Linhas x Colunas	**Sem *CUDA***	**Com *CUDA***
512	0.7988	0.0453
1024	13.118	0.0533
2048	119.858	0,1266
4096	1.769.608	0.6164

Fonte: Os Autores, 2013.

A Tabela 3 mostra os mesmos tempos de execução convertidos em horas:

Tabela 3 – Tempos de execução do protótipo (em horas)

		Desempenho (horas - *hrs*)
Linhas x Colunas	**Sem *CUDA***	**Com *CUDA***
512	0.0133	0.000755
1024	0.2186	0.000888
2048	1.9976	0.002111
4096	29.4934	0.010274

Fonte: Os Autores, 2013.

O algoritmo utilizado para o cálculo da matriz identidade é o algoritmo de Gauss (Gauss-Jordan), nas duas situações (com e sem *CUDA*). Foi observado que já nas matrizes de tamanho “menor” o tempo de processamento da aplicação utilizando computação heterogênea foi significativamente inferior; para uma matriz de 512 linhas por 512 colunas o protótipo sem a solução proposta teve um tempo de 47929*ms versus* 2718*ms* no protótipo utilizando o hardware acelerador – uma diferença de aproximadamente 95% no tempo de execução.

Dobrando-se a quantidade de linhas e de colunas (1024 linhas por 1024 colunas), o tempo de execução da aplicação modelo com *CUDA* aumentou em torno de 16% (subiu para 3200*ms*), em relação ao resultado obtido anteriormente, enquanto que o protótipo executado sem hardware acelerador levou aproximadamente 94% de tempo a mais para o dobro de linhas (sendo executado em um tempo de 787098*ms*). A diferença do tempo de execução entre protótipo com e sem a solução proposta, gira nesta situação, em tordo de 99%.

Para o cálculo da identidade das matrizes de 2048 linhas por 2048 colunas o protótipo com aceleração da *GPU* teve um tempo de execução de 7600*ms* (um tempo de execução 58% maior que para a matriz de 1024 linhas por 1024 colunas). Quando o protótipo foi executado sem a *GPU*, o tempo de execução foi de 7191531*ms* – um tempo de execução 90% maior que para a matriz de 1024 linhas por 1024 colunas. A diferença do tempo de execução entre o protótipo com e sem a solução proposta, gira, nesta situação em torno de 99%.

Por fim, nas matrizes de 4096 linhas por 4096 colunas, o protótipo com aceleração teve um tempo de execução de 36988*ms* (um tempo de execução 85% maior que para a matriz de 2048 linhas por 2048 colunas). Quando o protótipo foi executado sem a *GPU*, o tempo de execução foi de 106176500*ms* – um tempo de execução 94% maior que para a matriz de 4096 linhas por 4096 colunas. A diferença do tempo de execução entre o protótipo com e sem a solução proposta, gira, nesta situação em torno de 99%.

4.7 DIFICULDADES NO PROJETO

Como em todo projeto, houveram algumas dificuldades iniciais; devido a inexperiência dos autores em relação a utilização da tecnologia em questão (*CUDA* e *JCUDA*) foi necessário realizar estudos em páginas da internet (fóruns e *blogs*) com tutoriais para configuração do ambiente e execução de aplicações de exemplo fornecidas nestes mesmos sites.

O maior impedimento no desenvolvimento do protótipo foi a configuração da camada *JCUDA*. Após o entendimento do funcionamento desta biblioteca foi realizada a implementação da proposta dos autores, conforme as delimitações propostas no item 3.4 (Capítulo 3 – Método, subitem Delimitações) e no capítulo 4 (Modelagem da Solução) desta monografia.

4.8 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Neste capítulo foram expostos alguns pontos referentes a modelagem do protótipo (solução proposta), baseados em conceitos de engenharia de software, considerados pelos autores como peças primordiais quando é necessário o desenvolvimento de qualquer sistema, por mais simples que este possa ser. Os requisitos de software (e suas subdivisões) além de auxiliar no desenvolvimento do projeto, são capazes de fornecer a terceiros uma ideia geral de como foi construído o protótipo; para complementar a visão geral da aplicação, o diagrama de classes apresenta a estrutura básica do sistema, seus principais objetos e operações.

Foram expostas também as tecnologias utilizadas para construção do ambiente de desenvolvimento necessário para a programação do protótipo, o *hardware* deste ambiente, bem como as tecnologias utilizadas diretamente na aplicação modelo. O sistema construído foi brevemente apresentado, para criar uma ideia de seu funcionamento. Juntamente com estas informações, os autores expuseram as dificuldades de entendimento relacionadas à configuração das bibliotecas do *CUDA* e *JCUDA*.

A demonstração dos resultados foi disposta na forma de 3 tabelas, nas quais cada linha corresponde ao tamanho de uma matriz quadrada, pré-definida na aplicação; nas colunas da tabela estão colocadas as informações de utilização do protótipo, que podem ser com *CUDA* ou sem *CUDA*. Para cada uma das matrizes foi executado o cálculo da sua identidade, utilizando (ou não) o *CUDA* para que fossem gerados os tempos de execução.

5 CONSIDERAÇÕES FINAIS

A *GPU* se apresenta aos usuários de computadores na forma de um grande potencial de processamento disponível para as aplicações. Quando jogos e aplicativos gráficos que exigem grandes quantidades de cálculos não a utilizam, este recurso é, consequentemente, desperdiçado. Isso é acarretado, muitas vezes, pelo desconhecimento desse recurso pelos desenvolvedores de softwares. Dessa forma, com a finalidade de demonstrar como este potencial pode ser melhor aproveitado, este trabalho apresenta sua utilização por meio de um protótipo. Este protótipo calcula a matriz identidade (através do método de Gauss-Jordan) de uma matriz de ponto flutuante gerada aleatoriamente.

Para embasar a construção da aplicação, foram expostos ao decorrer da revisão bibliográfica os conceitos de computação heterogênea. Cada uma das seções apresentados neste capítulo foram pontos importantes para embasar o desenvolvimento do protótipo e para que se entendam as tecnologias envolvidas.

A implementação do protótipo não foi de alta complexidade. Para seu entendimento e construção foi efetuada uma breve modelagem, destacada no capítulo 4. Esta foi realizada baseada nas bibliotecas auxiliares e utilizadas para execução dos cálculos da matriz identidade com e sem heterogeneidade computacional.

A utilização da *GPU*, como recurso acelerador aplicado na implementação do protótipo, proporcionou uma diminuição significativa no tempo de execução necessário para obtenção dos resultados. A diminuição dos tempos de execução pôde ser observada com clareza na

Tabela **1**, na qual constata-se que, desde as matrizes quadradas de tamanho menor (512 x 512), já existe um ganho de desempenho (relativo ao tempo de execução) em torno de 95%.

A diferença de tempo de execução seguiu por cada uma das matrizes pré-definidas (1024 x 1024, 2048 x 2048 e 4096 x 4096), sempre apresentando um ganho de tempo de processamento maior que 90% para os cálculos da matriz identidade utilizando *CUDA*. Ressaltando que, conforme apresentado anteriormente, o computador no qual foi executado o protótipo sem CUDA é uma máquina que conta com uma configuração considerável, com 7 núcleos de processamento.

Alguns pontos dificultosos foram percebidos na parte da utilização da biblioteca do *JCUDA*, camada intermediária entre o *CUDA* e o *JAVA*. Mais especificamente podem-se citar as seguintes dificuldades encontradas: (i) dificuldade na construção do ambiente de desenvolvimento com problemas seguidos de correções que levavam a novos problemas seguidos de novas correções; (ii) falta de material com de apoio e *API* da biblioteca *JCUDA*; (iii) baixa aplicabilidade dos exemplos existentes apresentados no site da biblioteca, entre outros.

O aprendizado com a parte de configuração do ambiente de desenvolvimento (descrito no capítulo 5 desta monografia, subitem 5.1) impulsionou a continuidade do desenvolvimento do projeto utilizando a estrutura selecionada. Transposto este obstáculo inicial, coube aos autores o entendimento do método utilizado para cálculo da matriz identidade (oferecido como método auxiliar da biblioteca do *JCUDA*) e aplicação deste mesmo método (Gauss-Jordan) “fora” do ambiente heterogêneo (por via de métodos da biblioteca *la4j*).

Recomenda-se para terceiros que desejem construir uma aplicação heterogênea com *JCUDA*/*CUDA*, que sigam os passos apresentados no capítulo 5 desta monografia, subitem 5.1, atentando para atualizações das partes de software envolvidas (*JAVA*, *CUDA*, *JCUDA*, *Visual Studio*, e demais softwares/bibliotecas utilizadas), com intuito de que estejam sempre trabalhando com as últimas versões existentes, contado com possíveis agregações de recursos. É importante ressaltar que qualquer combinação diferente da apresentada é passível de problemas de compatibilidade.

5.1 SUGESTÕES PARA TRABALHOS FUTUROS

Como sugestão para trabalhos futuros a serem efetuados, tomando-se como ponto de partida o que foi realizado nessa monografia, tem-se: (i) a criação de uma biblioteca facilitadora, para acesso à *GPU*, com funções para a execução de cálculos matemáticos gerais para fórmulas matemáticas complexas, com a utilização de *CUDA*; e (ii) montagem de um material de apoio, ou manual, contendo as principais diretrizes de funcionamento da biblioteca *JCUDA*.

REFERÊNCIAS

BONAT, Debora. **Metodologia da Pesquisa.** 3ª Edição. Curitiba: Iesde Brasil S.a., 2009. 132 p.

BRIDGES, T.. **The gpa machine:** a generally partitionable msimd architecture. In Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the, pages 196–203, 1990.

CANTÃO, Luiza Amalio Pinto; STARK, Felipe Sanches. **Apostila de Programação Linear:** PL. Departamento de Ciência da Computação da UNESP (Universidade Estadual Paulista Júlio de Mesquita Filho). Disponível em: <http://www2.sorocaba.unesp.br/professor/luiza/PL/apostila.pdf>. Acesso em: 23 jun. 2013.

DAREMA-ROGERS, D. A. G. V. A. N. and Pfister, G. F.. A vm parallel environment. Technical Report RC-11225 (#9161), IBM Thomas J. Watson Research Center, 1985.

DU, Peng et al. **From CUDA to OpenCL:** Towards a performance-portable solution for multi-platform GPU programming. Departamento de Ciência da Computação e Engenharia – Universidade Estadual de Ohio. Disponível em: < http://www.cse.ohio-state.edu/~agrawal/788-sp13/Papers/5.opencl.pdf>. Acesso em: 03 out. 2012.

FOSTER, Ian. **What is the Grid? A Three Point Checklist.** Disponível em: <http://dlib.cs.odu.edu/WhatIsTheGrid.pdf>. Acesso em: 04 out. 2012.

GIL, A.C. **Métodos e técnicas de pesquisa social.** São Paulo: Atlas, 1999.

GRUNE, Dick et al. **Projeto Moderno de Compiladores:** Implementação e Aplicações. Rio de Janeiro: Campus, 2001.

KHRONOS GROUP. **OpenCL - The open standard for parallel programming of heterogeneous systems.** Disponível em: <http://www.khronos.org/opencl/>. Acesso em: 27 ago. 2012.

KIRK, David B.; HWU, Wen-mei W.. **Programando para processadores paralelos:** Uma abordagem prática à programação de GPU. Rio de Janeiro: Elsevier, 2011.

LAKATOS, E. M.; MARCONI, M. A. **Fundamentos de Metodologia Científica.** São Paulo: Atlas, 1985.

MACHADO, Francis B.; MAIA, Luiz Paulo. **Arquitetura de Sistemas Operacionais.** 2. ed. Rio de Janeiro: Ltc, 2002. 232 p.

MANOEL, Edson e FERREIRA, Luis. **Linux Clustering with CSM and GPFS.** Disponível em: <http://my.safaribooksonline.com/book/operating-systems-and-server-administration/linux/0738428086> Acesso em: 27 ago. 2012.

NVIDIA (Org.) 2012a. **O que é computação com a GPU:** GPGPU, CUDA e Kepler. Disponível em: <http://www.nvidia.com.br/object/what-is-gpu-computing-br.html>. Acesso em: 22 ago. 2012.

NVIDIA (Org.) 2012b. **NVIDIA's Next Generation CUDATM Compute Architecture:** Fermi. Disponível em: <http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>. Acesso em: 04 out. 2012.

NVIDIA (Org.) 2012c. **NVIDIA's Next Generation CUDATM Compute Architecture:** Kepler TM GK110 The Fastest, Most Efficient HPC Architecture Ever Built. Disponível em: <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>. Acesso em: 04 out. 2012.

ROCHA, Ricardo. **Computação Paralela LCC/LERSI:** Fundamentos de Computação Paralela. Departamento de Ciência da Computação da Universidade do Porto. Disponível em: <http://www.dcc.fc.up.pt/~ricroc/aulas/0607/cp/apontamentos/partel.pdf>. Acesso em: 03 out. 2012.

SHATZ, Sol M.; WANG Jia-Ping. Tutorial: Distributed-Software Engineering. New York: IEEE Computer Society Press, 1989.
279p.

SILBERSCHATZ, Abraham. **Fundamentos de Sistemas Operacionais.** 6. ed. Rio de Janeiro: Livros Técnicos e Científicos, 2004.

SILVA, Edna Lúcia da; MENEZES, Estera Muszkat. **Metodologia da Pesquisa e elaboração de Dissertação.** Florianópolis: UFSC. 2005. p. 9-36.

SANTOS, Reginaldo J. **Introdução à Álgebra Linear.** Belo Horizonte: Imprensa Universitária da UFMG, 2013.

STERLING, Thomas. **Beowulf:** A Parallel Workstation for Scientif Computation. Center of Excellence in Space Data and Information Sciences - NASA Goddard Space Flight Center Disponível em: < http://www.phy.duke.edu/~rgb/brahma/Resources/beowulf/papers/ICPP95/icpp95.html >. Acesso em: 20 ago. 2012.

STRINGHINI, Denise; GONÇALVES, Rogério A.; GOLDMAN, Alfredo. **Introdução à Computação Heterogênea**. XXXI Jornada de Atualização em Informática (JAI). Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/jai/2012/007.pdf>. Acesso em: 20 ago. 2012.

SOMMERVILLE, Ian. **Engenharia de Software.** 9. ed. São Paulo: Pearson Education, 2011.

TANEMBAUM, Andrew S. **Sistemas Operacionais Modernos.** 2. ed. São Paulo: Pearson Education do Brasil, 2003.

APÊNDICE A – Cronograma

Etapas	Março				Abril				Maio				Junho			
Configuração do Ambiente de Desenvolvimento (Reunião de bibliotecas e aplicativos necessários)			X	X												
Adaptação da Modelagem Proposta (Relacionamento com as bibliotecas externas)					X	X	X	X								
Desenvolvimento da aplicação modelo (com e sem uso de computação heterogênea)					X	X	X	X								
Adaptação do Projeto escrito de monografia	X	X	X	X	X	X	X	X	X	X	X	X				
Testes no Protótipo					X	X	X	X	X	X	X	X				
Estudo dos Resultados					X	X	X	X	X	X	X	X				
Adaptação do Projeto Escrito de monografia – Inclusão dos resultados no projeto escrito					X	X	X	X	X	X	X	X				
Defesa do Projeto de Monografia															X	

Quadro 1 – Cronograma do Projeto de Monografia.
Fonte: Os autores, 2013.