

UNIVERSIDADE ANHEMBI MORUMBI  
**ALESSANDRO VIEIRA GRAMMELSBACHER**  
**JOÃO CARLOS CAMPOI MEDRADO**

COMPARAÇÃO DE DESEMPENHO ENTRE GPGPU E SISTEMAS PARALELOS

São Paulo  
2009

ALESSANDRO VIEIRA GRAMMELSBACHER  
JOÃO CARLOS CAMPOI MEDRADO

**COMPARAÇÃO DE DESEMPENHO ENTRE GPGPU E SISTEMAS PARALELOS**

Trabalho de Conclusão de Curso apresentado como exigência parcial para obtenção do título de Bacharel em Ciência da Computação da Universidade Anhembi Morumbi.

Orientador: Prof. Marcelo Alexandre Couto de Jesus.

São Paulo  
2009

ALESSANDRO VIEIRA GRAMMELSBACHER  
JOÃO CARLOS CAMPOI MEDRADO

## **COMPARAÇÃO DE DESEMPENHO ENTRE GPGPU E SISTEMAS PARALELOS**

Trabalho de Conclusão de Curso apresentado como exigência parcial para obtenção do título de Bacharel em Ciência da Computação da Universidade Anhembi Morumbi.

Aprovado em

---

Prof. Augusto Mendes Gomes Júnior  
Universidade Anhembi Morumbi

---

Prof. Marcelo Alexandre Couto de Jesus  
Universidade Anhembi Morumbi

---

Profa. Simone de Abreu  
Universidade Anhembi Morumbi

Esta monografia é dedicada a todos que precisam trabalhar para custear a faculdade e se esforçam fazendo malabarismos com os horários para conseguir o melhor desempenho em ambas as atividades. Não é tarefa simples, mas é recompensadora.

## **AGRADECIMENTOS**

Conquistar o título de Bacharel em Ciência da Computação tornou-se um sonho que há quatro anos atrás parecia distante e hoje torná-lo realidade está diante de nossas mãos. As oportunidades de crescimento pessoal e profissional com tal título são gigantescas e farão uma grande diferença em nossas vidas. Sabemos que em nosso país poucas pessoas possuem a oportunidade de subir os degraus que subimos. Contudo não chegaríamos aonde estamos sem a ajuda de grandes pessoas e portanto queremos agradecê-las.

Gostaríamos de agradecer ao nosso orientador Marcelo Alexandre Couto de Jesus que desde o início do trabalho até o final forneceu subsídio para o desenvolvimento desta monografia contribuindo com idéias e sugestões de grande valia, não se importando de muitas vezes ficar até mais tarde após a aula discutindo questões do trabalho. Também agradecemos ao professor Augusto Gomes pelo apoio pois apesar de não ser nosso orientador também esteve sempre disposto a tirar nossas dúvidas sendo atencioso e paciente contribuindo com sugestões valiosas.

Aos professores das disciplinas de trabalho de conclusão de curso Nelson Shimada, Roberta Aragon e Judith Pavón pelas revisões e tempo expendido analisando nosso trabalho.

A esta Universidade, seu corpo de direção e administrativo, que oportunizaram a conclusão desse curso superior. A nossas famílias e namoradas, que nos momentos de ausência dedicados a conclusão desta pesquisa, sempre fizeram entender que o futuro, é feito das escolhas e constante dedicação no presente.

Agradecemos, sinceramente, a todos que nos ajudaram na conclusão desta pesquisa!

## **RESUMO**

Este trabalho visa fornecer um estudo comparando a execução de algoritmos paralelos na CPU (Central Process Unit) e na GPU (Graphics Process Unit), elucidando se é mais vantajoso executar determinados algoritmos na CPU ou na GPU. Para realizar as comparações são feitos benchmarks da execução de algoritmos com a finalidade de medir a capacidade de processamento em FLOPS (Floating Point Operation Per Second), a cópia de dados na memória entre outras características do sistema. Para que os algoritmos fossem implementados foi necessário um estudo sobre o funcionamento de GPGPU e de sistemas paralelos. Utilizou-se as bibliotecas CUDA (Compute Unified Device Architecture) e OpenMP para implementação da execução de código paralelo na GPU e na CPU respectivamente.

**Palavras-chave:** CUDA, GPGPU, sistemas paralelos, OpenMP, GPU

## **ABSTRACT**

This paper wants to provide a study comparing the performance of parallel algorithms on the CPU and the GPU, explaining if it is more advantageous to execute certain algorithms on the CPU or the GPU. To perform the comparisons benchmarks of the performance of algorithms were made in order to measure the processing power in FLOPS (Floating Point Operation Per Second), copying data in memory and other system features. In order to implement the algorithms it was necessary a study on the functioning of GPGPU and parallel systems. It was used the CUDA libraries and OpenMP to implement the execution of parallel code on the GPU and CPU respectively.

**Keywords:** CUDA, GPGPU, parallel systems, OpenMP, GPU

## LISTA DE TABELAS

Tabela 1- Arquitetura de sistema com memória compartilhada .....	32
Tabela 2 - Arquitetura de sistema com memória distribuída. ....	32
Tabela 3 - Tabela de Desempenho C Serial .....	48
Tabela 4 - Tabela de Desempenho OpenMP.....	49
Tabela 5 - Tabela de Desempenho CUDA.....	49
Tabela 6 - Tabela de Desempenho Cublas .....	49

## LISTA DE ILUSTRAÇÕES

Figura 1 - Exemplo de uma CPU e GPU.....	15
Figura 2 - Super Computador .....	16
Figura 3 - Funcionamento CUDA .....	17
Figura 4 - Funcionamento OpenCL.....	19
Figura 5 - Linha do Desenvolvimento do OpenCL.....	19
Figura 6 - Gráfico comparativo do desempenho entre GPU e CPU.....	20
Figura 7 - Arquitetura de uma CPU.....	21
Figura 8 - A figura ilustra o processamento de dados em uma GPU recente .....	22
Figura 9 - Exemplo de AMD FireStream. ....	23
Figura 10 - Exemplo de Arquitetura TESLA.....	24
Figura 11- Exemplo de gráfico que pode ser utilizado em um benchmark.....	28
Figura 12 - Memória compartilhada em Pthreads .....	34
Figura 13 - Código exemplo PThreads.....	34
Figura 14 - Execução de um pthread_join() .....	35
Figura 15 - Código simples em PThreads .....	35
Figura 16- Modelo Fork-Join no OpenMP .....	36
Figura 17 - Exemplo código OpenMP.....	36
Figura 18 - Exemplo de Diretiva. ....	37
Figura 19 - Exemplo de uma região paralela.....	37
Figura 20 - Exemplo de Código OpenMP.....	38
Figura 21 - Organização da Threads.....	40
Figura 22 - Execução de Programa com CUDA. ....	42
Figura 23 - Alocação automática de blocos nos multiprocessadores. ....	43
Figura 24 - Cuda acesso a memórias. ....	44
Figura 25 - Acesso aos tipos de memórias do Cuda. ....	45
Figura 26 - Gráfico de desempenho C Serial .....	50
Figura 27 - Gráfico de desempenho CUBLAS .....	50
Figura 28 - Gráfico de desempenho CUDA .....	51
Figura 29 - Gráfico de desempenho OpenMP.....	51
Figura 30 - Comparação entre as tecnologias.....	52

## SUMÁRIO

<b>1 INTRODUÇÃO</b> .....	12
1.1 OBJETIVO.....	12
1.2 JUSTIFICATIVA.....	12
1.3 ABRANGÊNCIA.....	13
1.4 ESTRUTURA DO TRABALHO .....	13
<b>2 O QUE É GPGPU</b> .....	15
2.1 CUDA (COMPUTE UNIFIED DEVICE ARCHITETURE) .....	16
2.2 OPENCL.....	18
<b>3 ARQUITETURAS DA CPU E GPU</b> .....	20
3.1 UNIDADE CENTRAL DE PROCESSAMENTO .....	20
3.2 UNIDADE DE PROCESSAMENTO GRÁFICO.....	22
3.2.1 AMD FireStream .....	22
3.2.2 Nvidia Tesla .....	23
<b>4 METODOLOGIA DE BENCHMARK</b> .....	27
<b>5 SISTEMAS PARALELOS</b> .....	30
5.1 PROGRAMAÇÃO PARALELA.....	30
5.2 PTHREADS .....	33
5.3 OPENMP .....	36
5.4 CUDA.....	39
<b>5.4.1 Threads, blocks e grids</b> .....	39
<b>5.4.2 Gerenciamento de memória</b> .....	43
<b>6 BENCHMARKS ENTRE CPU, SISTEMAS PARALELOS E GPGPU</b> .....	46
6.1 CONFIGURAÇÕES DO AMBIENTE .....	46
6.2 CARACTERÍSTICAS DAS MATRIZES UTILIZADAS .....	47
6.3 CODIGOS DAS MULTIPLICAÇÕES .....	47
6.4 RESULTADOS .....	48
6.5 ANÁLISE DO RESULTADO .....	52
<b>6.5.1 CUDA</b> .....	53
<b>6.5.2 Cublas</b> .....	53
<b>6.5.3 OpenMP</b> .....	54
<b>7 CONCLUSÃO</b> .....	55
7.1 TRABALHOS FUTUROS .....	56

<b>REFERÊNCIA BIBLIOGRÁFICAS</b> .....	57
APENDICE A – CÓDIGO MULTIPLICAÇÃO DE MATRIZES – C SERIAL.....	60
APENDICE B - CÓDIGO MULTIPLICAÇÃO DE MATRIZES – OPENMP .....	61
APENDICE C - CÓDIGO MULTIPLICAÇÃO DE MATRIZES – CUDA .....	63

## 1 INTRODUÇÃO

O crescimento de poder computacional atualmente não tem se dado pelo aumento de velocidade dos processadores, mas sim pelo número de núcleos do processador e pela capacidade dos softwares executarem em paralelo aproveitando o processamento de todos os núcleos existentes na CPU (Central Processing Unit).

Ao mesmo tempo as placas de vídeo vêm evoluindo e fornecem em seus processadores, alguns com diversos núcleos, uma capacidade de processamento ociosa grande, com funcionalidades de processamento paralelo e cálculo vetorial entre outras. A GPU (Graphical Processing Unit) possui uma tecnologia que pode ser aproveitada para executar códigos de aplicativos.

Este trabalho mostra o estudo das técnicas de GPGPU (General-purpose Computing on Graphics Processing Units), comparando com benchmarks sistemas paralelos para chegar à conclusão de qual técnica é melhor para cada caso.

### 1.1 OBJETIVO

O objetivo é elaborar um estudo comparativo das técnicas de Sistemas Paralelos e GPGPU definindo qual tem o melhor desempenho e mostrando os pontos fortes e fracos da GPGPU.

### 1.2 JUSTIFICATIVA

O uso de GPGPU vem se popularizando, os sistemas operacionais Windows 7 e Mac OSX Snow Leopard já pretendem integrar essas técnicas no próprio sistema operacional (CBS CORPORATION, 2009). O popular software Nero para gravação de Cds e DVDs também já se aproveita da GPU para acelerar a aplicação, principalmente para efetuar conversão de vídeo. (NERO INC., 2009).

A AMD vem trabalhando há alguns anos em um processador chamado Fusion (ADVANCED MICRO DEVICES, 2008) que pretende unir a CPU e a GPU sem necessidade de haver uma placa de video separada com uma GPU dedicada a essa tarefa. A Intel também trabalha em um processador para unir CPU e GPU chamado Larrabee (INTEL CORPORATION, 2008).

Portanto esse assunto deve se desenvolver bastante nos próximos anos e no entanto

existem poucos materiais em português sobre o assunto. Também não foram encontrados benchmarks acadêmicos comparando o desempenho da GPU com sistemas paralelos.

Adicionalmente processadores gráficos estão ficando mais poderosos e baratos facilitando sua aquisição para uso em GPGPU. Com o estudo deste tema pretende-se fornecer benchmarks que compare o desempenho das duas arquiteturas. Tal estudo acadêmico poderia ajudar outras pesquisas científicas e até mesmo empresas que necessitem de alto desempenho computacional.

### 1.3 ABRANGÊNCIA

O universo do trabalho trata de uma pesquisa acadêmica, referente a implementação e realização de testes de algoritmos que exijam alto poder computacional. Esta implementação se dará em sistema paralelo e em GPGPU, efetuando-se benchmarks para analisar os tempos de execução de ambas as implementações e identificando qual implementação fornece melhor desempenho para determinado algoritmo.

Não se entrará em detalhes específicos da arquitetura da placa de vídeo para verificar o motivo de uma implementação em sistema paralelo fornecer desempenho melhor ou pior que outra em GPGPU. Neste trabalho basta identificar qual possui melhor desempenho.

### 1.4 ESTRUTURA DO TRABALHO

Este estudo se encontra dividido em sete capítulos.

No segundo capítulo define-se o que é GPGPU, onde é utilizada e como é aplicada nos sistemas, como a tecnologia é utilizada por empresas e exemplifica-se com 2 bibliotecas existentes: CUDA e OpenCL.

O terceiro capítulo explica a utilização de uma CPU (Central processing unit) e uma GPU (Graphics processing unit), a arquitetura de cada hardware, e uma visão geral para o entendimento de GPGPU.

O capítulo quatro define o que é benchmark, algumas das metodologias de benchmarks existentes e como elas serão utilizadas neste trabalho para comparação de desempenho entre CPU e GPGPU.

O quinto capítulo explica como funciona um sistema paralelo, e efetua a comparação entre as técnicas mais populares de programação paralela, fornecendo assim subsídio para a escolha da técnica que será utilizada ao se comparar sistemas paralelos com GPGPU.

No sexto capítulo é efetuado o benchmark mostrando a configuração do ambiente, os resultados obtidos e análise dos resultados.

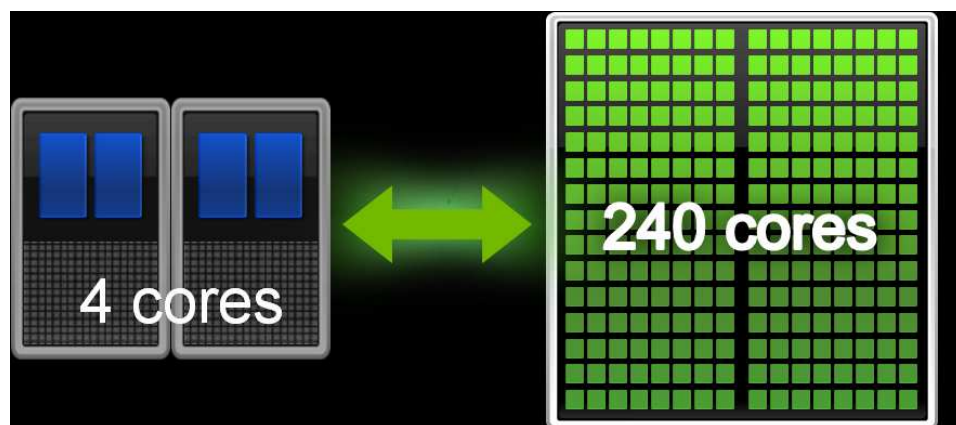
O sétimo capítulo contém a conclusão do trabalho e os problemas encontrados.

## 2 O QUE É GPGPU

GPGPU significa General-Purpose computation on Graphics Processing Units (Computação de Propósito Geral na Unidade de Processamento Gráfico) e também é conhecida como GPU Computing (computação GPU), descoberto por Mark Harris em 2002 quando reconheceu o uso de GPUs em aplicações não gráficas (GPGPU TEAM, 2009).

A GPU (Graphics Processing Units) é a unidade de processamento gráfico, um sistema especializado em processar imagens. A GPU é projetada com a responsabilidade de lidar com a parte visual do computador, era difícil de programá-la para outra finalidade, como resolver uma equação de propósito geral, há poucos anos diriam que programar em uma placa de vídeo, era incompreensível, devido falta de um framework que facilitasse o processo. Hoje as GPUs evoluíram para um processador de diversos núcleos tornando-a interessante para o desenvolvimento de aplicações em sistemas paralelos, a figura 1 ilustra a diferença de núcleos que pode existir. As GPUs possuem um modelo de programação paralela explícito e possuem uma performance muito maior para alguns tipos de processamentos de dados quando comparados com uma CPU. Isso explica o aumento do uso de GPUs como um co-processador para executar aplicações paralelas de alta performance (GRAÇA; DALI,2006). Para facilitar a execução desses algoritmos na GPU criaram as bibliotecas como CUDA e OpenCL que possuem recursos de interfaces para programação em linguagens conhecidas como C.

Utilizar GPGPU é aproveitar o processador da placa de vídeo (GPU) para realizar tarefas (de propósito geral) que tradicionalmente a CPU faria, trabalhando como um co-processador (NVIDIA CORPORATION, 2009 b).



**Figura 1 - Exemplo de uma CPU e GPU**

Fonte: NVIDIA CORPORATION (2009 b)

Os desenvolvedores que portam suas aplicações paralelas para executarem com GPGPU

frequentemente alcançam ganhos de velocidade em comparação com as versões das aplicações otimizadas para a CPU. Essas aplicações devem atender a alguns quesitos, pois sua arquitetura não atende aplicações pequenas ou com pouco processamento em paralelo. Utilizando as bibliotecas de GPGPU pode-se reduzir não só tempo, mas também o espaço físico que a máquina ocupa, a manutenção e a troca de máquinas. Por exemplo o TACC Range (VIZWORLD.COM, 2009), um super computador com a seguinte configuração:

- a) Nós: 3.936
- b) Núcleos: 62.976
- c) Pico de Performance: 579.4 TFlops
- d) Número de Racks: 82



**Figura 2 - Super Computador**

Fonte: VIZWORLD.COM (2009)

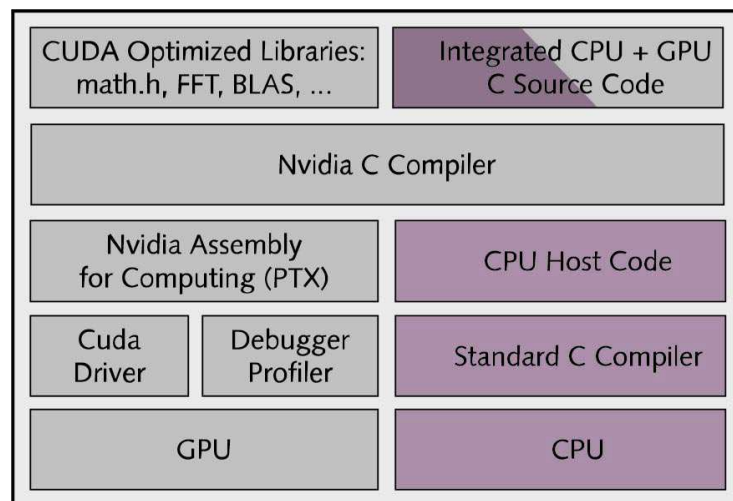
São 82 Racks repletos de equipamentos para receber 580 Teraflops e ocupar uma sala inteira. Para ter um computador com a mesma performance, utilizando a arquitetura de Nvidia Tesla, seriam necessárias máquinas com as seguintes configurações:

- a) 145 Tesla's = 580 TeraFLOPS (1 Tesla S1070 = 4 TeraFLOPS )
- b) Número de Racks: 3.5 (42 Tesla S1070's por Rack)

Existem diversas ferramentas e arquiteturas para desenvolvimento de software com tecnologia GPGPU, a maioria se baseia na linguagem C de programação, por ser de fácil aprendizado e muito utilizada, mesmo assim cada uma delas tem suas próprias características deixando o desenvolvedor livre para escolher a aplicação que mais se adéqua ao seu programa ou a sua necessidade. Algumas das principais bibliotecas de desenvolvimento serão citadas a seguir (VIZWORLD.COM, 2009).

## 2.1 CUDA (COMPUTE UNIFIED DEVICE ARCHITETURE)

CUDA é uma arquitetura de computação paralela desenvolvida pela NVIDIA que tira proveito do mecanismo de computação paralela das GPUs (Graphics Processing Unit) para resolver problemas computacionais complexos que teoricamente uma CPU levaria mais tempo para executar. A GPU se utiliza de algumas vantagens como memória compartilhada, processamento dedicado, e por isso pode levar vantagem em relação a uma CPU (NVIDIA, 2009 c). Hoje para programar pela arquitetura CUDA o desenvolvedor utiliza C que é uma linguagem de programação de baixo nível bastante utilizada, no futuro outras linguagens serão admitidas. Além da linguagem também é necessário uma GPU compatível com sistema tornando assim mais fácil a programação em GPGPU do que era antigamente.



**Figura 3 - Funcionamento CUDA**

Fonte: HALFHILL (2008)

A figura 3 mostra o funcionamento da plataforma CUDA para processamento paralelo em GPUs NVIDIA. O CUDA inclui ferramentas de desenvolvimento C/C++, bibliotecas para funcionamento e o compilador que esconde do programador o funcionamento do processador gráfico. Na ferramenta de desenvolvimento CUDA o programador pode desenvolver em CUDA, com funções específicas, mas na compilação do programa o compilador divide os algoritmos que serão executados no processador gráfico e os que serão executados da CPU. As funções matemáticas padrões do header math.h podem ser substituídas automaticamente pelo Nvidia C Compiler para executarem na GPU (Halfhill, 2008).

Inúmeros desenvolvedores já estão utilizando a ferramenta gratuita de desenvolvimento de software da NVIDIA CUDA para resolver problemas em aplicativos comerciais e domésticos. O Nero, Photoshop, Windows 7, MACOSX Snow Leopard, processamento de áudio, vídeo, simulação de exemplos físicos e até exploração de petróleo são exemplos de

aplicações comerciais que já utilizam a arquitetura para resolver problemas ou aumentar desempenho de seus aplicativos (NVIDIA CORPORATION, 2009 a).

A NVIDIA tem mais de 100 milhões de GPUs compatíveis com a arquitetura partindo das versões G8X. No início desse ano a fabricante de GPUs investiu bilhões no aplicativo que implementa a arquitetura GPGPU apostando que nos próximos 2 anos a área irá se expandir aumentando o comércio de GPUs.

Ainda existem algumas limitações para tecnologia como:

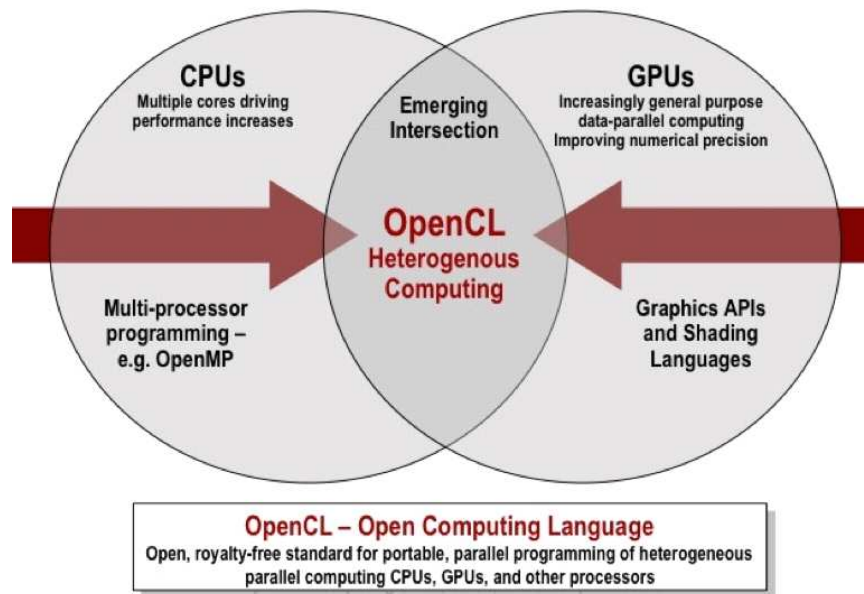
- a) Recursividade que não é permitida mas pode ser contornada alterando para loops;
- b) Utiliza ponteiros e recursões simples resultantes da linguagem C, ou seja, um único processo que roda em espaços separados na memória não otimizado o uso, ao contrario de outros ambientes que utilizam C linguagem de programação;
- c) Somente alguns GPUs da NVIDIA são compatíveis, mas a empresa já está trabalhando para alterar esse cenário;
- d) São aceitos no mínimo 32 threads para um desempenho considerável, abaixo disso não vale a pena utilizar a placa de vídeo;

Segundo NVIDIA os principais recursos da tecnologia são:

- a) Linguagem C padrão para desenvolvimento de aplicativos em paralelo na GPU;
- b) Bibliotecas numéricas padrão para FFT (Fast Fourier Transform) e BLAS (Basic Linear Algebra Subroutines) através do CUFFT e CUBLAS;
- c) Driver CUDA dedicado para computação com caminho de transferência rápida de dados entre a GPU e a CPU;
- d) Interoperação do driver CUDA com os drivers gráficos OpenGL e DirectX (BERILLO, 2008).

## 2.2 OPENCL

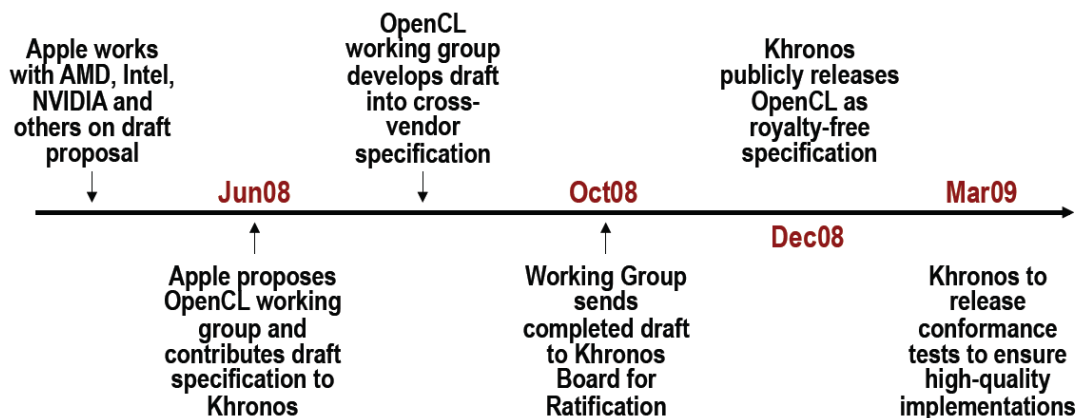
Outro framework criado para escrita de programas em plataformas heterogêneas como CPUs e GPUs é o OpenCL (Open Computing Language). O OpenCL permite programação paralela uniforme para desenvolvimento de software portáteis com alta performance para servidores, desktops e portáteis. A diferença para o CUDA é que essa biblioteca permite a programação em qualquer sistema paralelo, ou seja, tanto multi-core CPUs quanto GPUs e qualquer outro tipo de processadores paralelos (Khronos, 2009).



**Figura 4 - Funcionamento OpenCL**

Fonte: KHRONOS GROUP (2009)

Criado inicialmente pela Apple com parceria da NVIDIA, OpenCL foi apresentado ao grupo Khronos no meio do ano de 2008, com o objetivo de criar uma plataforma versátil para desenvolvimento de aplicações em GPU. A NVIDIA promoveu a linguagem com ajuda direta de seu departamento de desenvolvimento, deixando a plataforma portátil tanto para placas NVIDIA quanto para aplicativos paralelos. A linguagem também baseada em C é a maior concorrente com CUDA (NVIDIA, 2009 d).



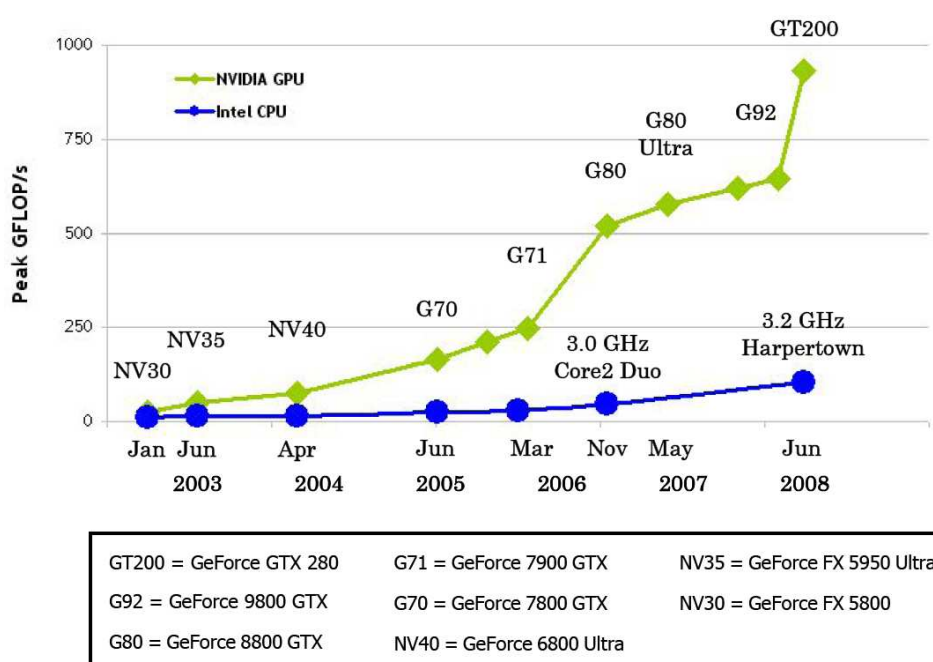
**Figura 5 - Linha do Desenvolvimento do OpenCL**

Fonte: KHRONOS GROUP (2009)

As limitações do framework são quase as mesmas do sistema CUDA, com a vantagem de poder executar em qualquer sistema paralelo. Isso mostra que as limitações estão mais ligadas com o hardware de GPUs do que com a linguagem de desenvolvimento. A figura 5 ilustra a linha de desenvolvimento do OpenCL enquanto a figura 4 ilustra seu funcionamento.

### 3 ARQUITETURAS DA CPU E GPU

Nos últimos anos as unidades de processamento gráfico tiveram um aumento impressionante em seu processamento. Com múltiplos núcleos e impulsionados pelo elevado desempenho da memória de alta velocidade, os recursos oferecidos pela GPU em processos gráficos e não gráficos são incríveis, como é apresentado na figura 6 um comparativo entre CPU e GPU do aumento de GFLOPS – medida de desempenho comparando operações de ponto flutuante que um processador pode gerar por segundo - por ano (NVIDIA, 2009).



**Figura 6 - Gráfico comparativo do desempenho entre GPU e CPU.**

Fonte: NVIDIA (2008)

Este capítulo aborda a arquitetura de cada sistema, suas principais características e exemplos.

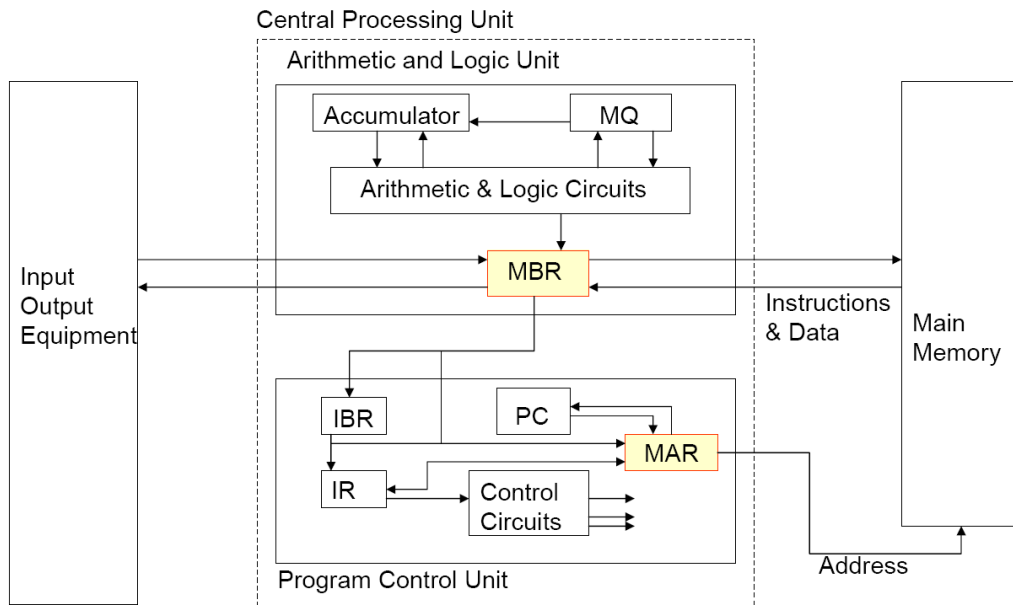
#### 3.1 UNIDADE CENTRAL DE PROCESSAMENTO

A CPU (Central Processing Unit) pode ter um único chip ou uma série, os multi-cores, que realizam cálculos lógicos e aritméticos, e controlam operações dos demais componentes do sistema. A maioria dos chips da CPU é composta por quatro seções funcionais: uma unidade lógica/aritmética, áreas de registro, barramento interno e uma seção de controle.

A seção de controle é onde se regula e temporiza as operações da totalidade do

sistema, lê as configurações dos dados e converte em uma atividade; e o ultimo segmento de um chip da CPU é o seu barramento interno.

Na figura 7 vemos a arquitetura desenvolvida de uma CPU segundo Von Neumann



**Figura 7 - Arquitetura de uma CPU**

Fonte: CHIAN (2009)

De acordo com a figura 7 temos em uma CPU:

a) Arithmetic & Logic Circuits: A unidade lógica/aritmética é unidade central do processador que realmente efetua os cálculos lógicos e aritméticos, proporciona a capacidade de calculo, funciona como uma “grande calculadora”;

b) MBR (Memory Buffer Register): Área de registros usadas para armazenamento de dados e resultados, dados serão escritos ou copiados desse registro;

c) MAR (Memory Address Register): Guarda o endereço do qual a próxima instrução será executada;

d) IR (Instruction Register): Parte do processador cujo controla unidades para guardar os dados do processamento que será executado.

e) IBR (Instruction Buffer Register): é um registro na CPU estritamente dedicado para armazenar instruções e prover outras temporariamente com o processador.

f) PC (Program Counter): processador de registros cuja função é indicar onde está na seqüência de instruções, contém o endereço da instrução que será executada.

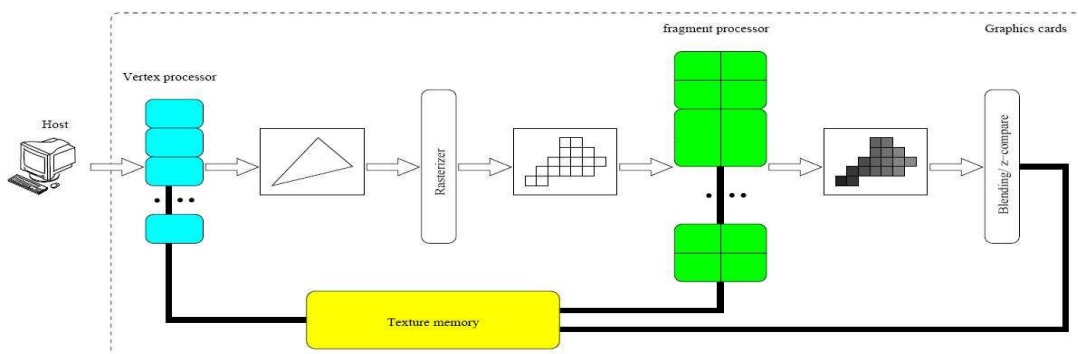
g) Accumulator: Armazena resultados ou partes de resultados de operações lógicas e aritméticas.

h) MQ (Multiplier Quotient): é um registro igual a duas palavras em que o quociente é desenvolvido para divisão e o multiplicador é utilizado para multiplicação (CHIAN, 2009).

### 3.2 UNIDADE DE PROCESSAMENTO GRÁFICO

A maior parte dos dados processados pela unidade de processador gráfico (GPU - Graphic Processor Unit, em inglês) são pixels, objetos geométricos e elementos que retornam no estado final uma figura. Esses objetos precisam de um processamento intenso antes de exibir a imagem final. Esse processamento é feito com “Graphics Hardware Pipeline”. O Pipeline contém varias etapas onde as aplicações em 3D enviam seqüências de vértices para GPU que transforma o dado em figuras geométricas primitivas (polígonos, linhas e ponteiros). Esses vértices são enviados e resolvidos pelo processador programável de vértices (Programmable Vertex Processor), que tem a capacidade de efetuar operações matemáticas rapidamente. O resultado dessa operação ainda é um objeto primitivo, esse é enviado para o processador de fragmentos (Programmable Fragment processor), onde faz as mesmas operações matemáticas com a adição de operações de textura.

Os principais processadores da GPU são os processadores de vértices e fragmentos, são eles quem fazem o trabalho pesado exaustivamente no processador gráfico, e são exatamente esses dois tipos de processadores que existem em grande quantidade em uma GPU, que ao longo dos anos aumentou absurdamente o número de processadores dentro dela (GRAÇA; DALI,2006).



**Figura 8 - A figura ilustra o processamento de dados em uma GPU recente**

Fonte: GRAÇA; DALI (2006)

#### 3.2.1 AMD FireStream

A AMD desenvolveu junto com ATI um processador paralelo para computadores de alto desempenho, para uso de processos paralelos com ajuda da GPU, antigamente chamado de ATI FireStream o processador possibilita o cálculo de dados pesados em ponto flutuante e pode ser usado para ajudar no desempenho de sua máquina, não como um segundo processador e sim como um co-processador, ajudando o principal.



**Figura 9 - Exemplo de AMD FireStream.**

Fonte: AMD (2009)

De acordo com a Advanced Micro Devices, Inc. (AMD) – algumas limitações ainda são impostas ao sistema.

- a) Recursividade assim como no CUDA ainda não pode ser implementada.
- b) Comparado com um processador de 64 bits o FireStream apresenta somente arquiteturas de 32 bits, o que pode ser um gargalo.
- c) Funções não pode conter entradas de tamanho variáveis (vetor, matriz)
- d) O uso de threads na GPU ainda não está totalmente otimizado, para ter um desempenho considerável é necessário usar no mínimo 64 threads.

### **3.2.2 Nvidia Tesla**


A tecnologia Nvidia Tesla permite montar sistemas multiprocessados com GPU, assim como existem sistemas computacionais de alto desempenho com diversas CPUs (por exemplo: super computadores), Tesla fornece uma tecnologia da Nvidia que permite montar sistemas com diversos processadores de GPU (multi-GPU) com alto desempenho computacional no qual se utiliza técnicas de GPGPU para programar estes diversos processadores fornecidos(NVIDIA, 2009 a).

Também é possível mesclar o uso do Tesla com sistemas multi-CPU (sistemas com múltiplas CPUs), aproveitando os benefícios de cada tecnologia para obter um desempenho maior dependendo da aplicação utilizada.

O processador de computação NVIDIA® Tesla™ C870 é o primeiro a disponibilizar uma arquitetura massivamente multiencadeada para os aplicativos usados na computação de alto desempenho (HPC) por cientistas, engenheiros e outros profissionais técnicos.

Processadores de computação da GPU Tesla C1060 transformam um sistema padrão em um supercomputador pessoal com pico de mais de 500 gigaflops de desempenho de ponto flutuante.

Com um núcleo de computação de 240 núcleos (modelo Tesla C1060), um ambiente de desenvolvimento para a GPU baseado na linguagem C, um conjunto de ferramentas de desenvolvimento e a maior comunidade do mundo em desenvolvimento ISV para computação GPU, o processador de computação GPU Tesla C870 permite aos profissionais desenvolver aplicativos mais rapidamente e implementá-los em diversas gerações de processadores. O processador de computação GPU Tesla C870 pode ser usado em conjunto com sistemas de CPUs multinúcleo a fim de criar uma solução flexível para a supercomputação pessoal (NVIDIA, 2008 d).



The image shows a Tesla C1060 Computing Processor, a black PCIe card with a fan and various connectors. To the right of the card is a table of specifications.

<b>Tesla C1060 Computing Processor</b>	
<b>Processor</b>	1 x Tesla T10
<b>Number of cores</b>	240
<b>Core Clock</b>	1.33 GHz
<b>On-board memory</b>	4.0 GB
<b>Memory bandwidth</b>	102 GB/sec peak
<b>Memory I/O</b>	512-bit, 800MHz GDDR3
<b>Form factor</b>	Full ATX: 4.736" (H) x 10.5" (L) Dual slot wide
<b>System I/O</b>	PCIe x16 Gen2
<b>Typical power</b>	160 W

**Figura 10 - Exemplo de Arquitetura TESLA.**

Fonte: NVIDIA, (2008) d

### Arquitetura NVIDIA Tesla:

- a) Arquitetura de computação massivamente paralela com processadores multiencadeados por GPU
- b) Processador de encadeamento escalar com operações de inteiros e de ponto flutuante
- c) O Thread Execution Manager (Gerenciador de Execução de Encadeamentos) possibilita milhares de encadeamentos por GPU
- d) O Parallel Data Cache (Cache de Dados Paralelo) possibilita que processadores trabalhem em conjunto com informações compartilhadas no desempenho do cache local
- e) Acesso de memória ultra-rápido com pico de largura de banda de 76,8 GB/seg. por GPU
- f) Ponto flutuante de precisão IEEE 754

### Soluções escaláveis

- a) Escalável de uma a milhares de GPUs
- b) Disponível em processadores de computação GPU, supercomputador deskside e servidor de computação GPU com montagem de rack 1U

### Ferramentas de desenvolvimento de software

- a) Compilador da linguagem C, analisador e modo de emulação para depuração
- b) Bibliotecas numéricas padrão para FFT (Fast Fourier Transform) e BLAS (Basic Linear Algebra Subroutines)

Entre os recursos interessantes existentes na tecnologia Tesla, destaca-se:

É possível programar em Testa compilando um código desenvolvido na linguagem C, de alto nível e extremamente popular entre profissionais de computação, para linguagem do chip gráfico da placa. Como é uma linguagem já conhecida, não é necessário o aprendizado de uma nova linguagem para poder lidar com esta tecnologia facilitando assim a migração de aplicações existentes e a criação de novas aplicações que utilizem do alto desempenho computacional fornecido por esta solução.

Os picos de mais de 500 gigaflops ilustram como esta tecnologia fornece desempenho de alta computação.

Outro fator interessante da tecnologia é a possibilidade de controlar múltiplas GPUs com uma única GPU através do driver da GPU, proporcionando um volume de transferência incrível em aplicativos de computação. Sendo assim, dividindo o problema entre diversas GPUs a capacidade de resolver problemas em grande escala pode ser multiplicada.

O cache de dados paralelos multiplica a largura de banda e reduz a latência do cache, diversos processadores trabalham em conjunto nas informações compartilhadas pelo cache local. Isso faz com que os dados sejam copiados menos vezes e sejam disponibilizados

imediatamente para os processadores que compartilham o mesmo cache de dados paralelo.

A transferência de dados entre a GPU e a placa mãe ocorre através de barramentos PCI-Express que possibilitam baixa latência e alta largura de banda fornecendo assim a mais alta taxa de transferência de dados.

A tecnologia Tesla é compatível com as arquiteturas padrões (e mais utilizadas) no mercado. A compatibilidade é existente para microprocessadores x86 de 32bits e 64bits da Intel e da AMD. Também é compatível com sistemas operacionais Linux e Windows. No caso do Linux são suportadas (homologadas pela NVidia) as distribuições Red Hat Enterprise 3, Red Hat Enterprise 4, Red Hat Enterprise 5, Suse Linux 10.1, Suse Linux 10.2, Suse Linux.

As características citadas fornecem uma noção da monstruosidade de processamento que esta tecnologia fornece.

## 4 METODOLOGIA DE BENCHMARK

Benchmarks são utilizados para medir a performance dos componentes de sistemas de computação. Costumam ser utilizados para comparação destes componentes ou um conjunto deles, para se verificar qual possui melhor desempenho, qual tem a melhor relação de custo benefício. Também serve como uma excelente ferramenta para detecção de gargalos no sistema (IDA, 2000).

Entre os componentes de um sistema de computação que um benchmark pode medir o desempenho estão a CPU, GPU, memória RAM, disco rígido e placa de rede entre outros.

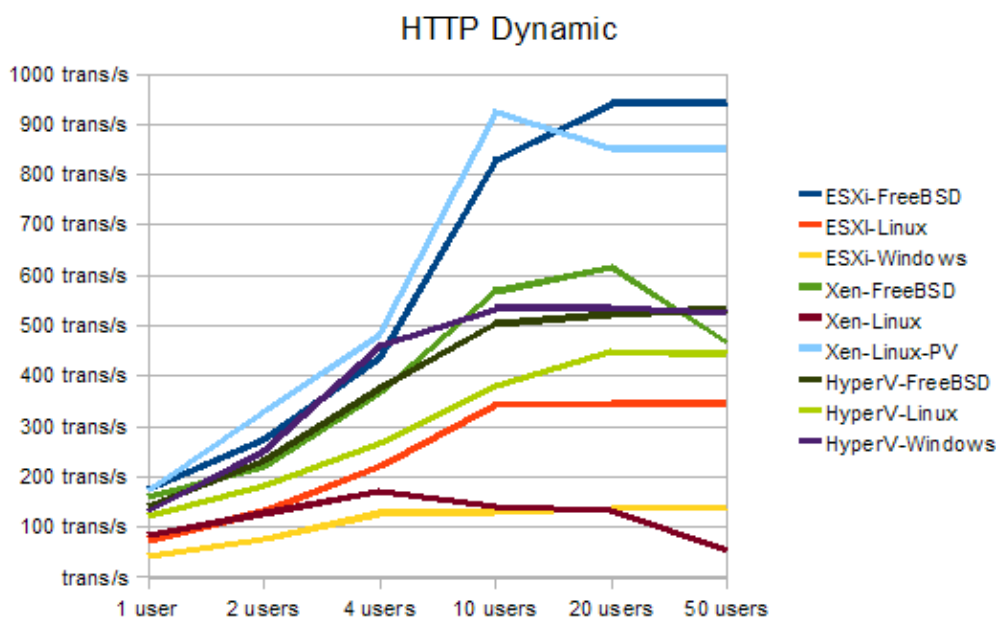
Neste trabalho será necessário analisar o desempenho de códigos executados tanto na GPU quanto na CPU, incluindo CPUs multi-núcleos que necessitam de códigos e medições que executem em sistemas paralelos, para tanto serão utilizadas metodologias de benchmark.

Para que a performance possa ser adequadamente comparada, torna-se necessário a utilização de unidades de medidas que reflitam adequadamente a capacidade do que está sendo comparado. Dentre as unidades de medidas existentes, as mais conhecidas são MIPS (Millions of Instructions per Second – Milhões de Instruções por Segundo) e o MFLOPS (Millions of Floating-Point Operations per Second – Milhões de Operações de Ponto-Flutuante por segundo). Também pode-se utilizar métricas para comparar o tempo de acesso para leitura de um determinado dado na memória, bem como o tempo de acesso para gravação de dados na memória.

Ao efetuar um benchmark geralmente se escolhe uma unidade de medida que represente a característica dos componentes que se deseja comparar, por ex.: ao se medir a performance de geração de gráficos de uma placa de vídeo pode-se utilizar o FPS (Frames per second) como unidade de medida, ao se medir a performance de um servidor web pode-se utilizar uma unidade como requisições por milissegundo.

Para realizar um benchmark pode-se desenvolver um programa ou algoritmo próprio, utilizar um programa que já tenha implementado diversos algoritmos que analisem um componente específico do computador, como por exemplo, criação de arquivos para testar o desempenho do disco rígido, comparar as próprias especificações técnicas na caixa do produto sem realizar testes, ou escolher outra forma de análise.

Para facilitar a visualização dos dados costuma-se utilizar gráficos como o da figura 11.



**Figura 11- Exemplo de gráfico que pode ser utilizado em um benchmark**

Fonte: VORAS; BASCH; ŽAGAR, (2008)

Para medir a performance da CPU e da GPU, além de medir os MIPS e MFLOPS, também pode-se medir o tempo que a CPU leva para leitura e gravação de determinado dado na memória RAM, o tempo de acesso que a GPU leva para ler e gravar dados na memória RAM contida na própria placa de vídeo, e o tempo que a CPU demora para transferir dados da memória RAM instalada na placa mãe para a memória RAM contida na placa de vídeo.

Existem diversos tipos de benchmarks:

a) Sintético: São aqueles cujo código não faz nenhuma computação útil, não representa nenhuma aplicação real; somente exercita alguns componentes básicos do computador (como por exemplo, aritmética de inteiros, fazer a chamada a um procedimento, fazer desvios condicionais). Geralmente, tentam determinar uma frequência média de instruções típicas, comumente utilizadas, e recriá-las em um programa.

b) Kernel: são baseados no fato de que a maior parte da computação de um programa é centrada em uma pequena parte de seu código. Esta pequena parte chamada de núcleo (kernel), é extraída do programa e usada como benchmark. Deve ser ressaltado que eles não servem para avaliar completamente a performance de uma máquina. São bastante interessantes por sua simplicidade e pequeno tamanho.

c) Algoritmo: são algoritmos bem definidos, geralmente implementações de métodos conhecidos em computação numérica, como por exemplo métodos de equação lineares.

d) Aplicação: são programas completos, que resolvem problemas científicos bem

definidos.

Existem suites de benchmarks prontas para serem executadas nos sistemas e medirem o desempenho. Tais como Whetstone, Dhrystone, Livermore Loops, Linpack, NAS Parallel Benchmarks, SPEC, entre outros.

Não se utilizará estas suites por diversos motivos. As suites Whetstone, Dhrystone, Livermore Loops se utilizam de códigos para rodar em uma única CPU porém queremos códigos que executem em mais de uma CPU para conseguir comparar sistemas paralelos com a GPU, eles também não possuem implementações que executam na GPU. O SPEC e o NAS Parallel Benchmarks apesar de possuírem benchmarks paralelos para CPU, não possuem benchmarks implementados para GPGPU, o mais próximo que possuem são benchmarks que avaliam a GPU sob o ponto de vista gráfico medindo frames por segundo.

O fato das suites existentes não possuírem uma versão específica para rodar em uma GPU inviabiliza seu uso neste trabalho, alguns deles possuem versão com código aberto, contudo reescrever o código destes programas utilizando GPGPU foge a abrangência deste trabalho.

Para tanto decidiu-se criar uma metodologia própria utilizando benchmarks do tipo algoritmo que implemente rotinas iguais na CPU e na GPU onde é possível medir e comparar os desempenhos.

## 5 SISTEMAS PARALELOS

Tradicionalmente os programas são feitos para executarem sequencialmente. Eles executam em uma única CPU na qual o problema é dividido em uma série de problemas menores e as instruções são executadas uma após a outra, isto é, sequencialmente. Somente uma instrução é executada por vez.

Já um sistema paralelo utiliza de diversos recursos disponíveis no computador para resolver um determinado problema. A resolução ocorre dividindo o problema em partes menores que podem ser resolvidas simultaneamente sem dependerem uma das outras.

Atualmente o poder computacional dos processadores x86 voltados a desktops tem crescido principalmente com base no número de núcleos do processador, as futuras gerações de aplicativos aproveitarão cada vez a capacidade multi-núcleos dos processadores para executarem tarefas em paralelo. Contudo os códigos existentes necessitam de ajustes para executarem em paralelo, como a utilização de threads, processos ou mesmo algoritmos específicos que trabalhem em paralelo. Pretende-se neste capítulo contextualizar formas de programação paralela existentes indicando as características de cada uma (APSTC, 2008), (BARNEY, 2009).

### 5.1 PROGRAMAÇÃO PARALELA

O objetivo de utilizar programação paralela é aumentar a performance da aplicação, executando-a através dos diversos núcleos e/ou processadores existentes na mesma máquina. Para isso existem duas abordagens:

- a) Auto-paralelização
- b) Programação Paralela

Quando já se tem uma aplicação pronta, desenvolvida sequencialmente que não foi otimizada para programação paralela, a auto-paralelização tenta automaticamente paralelizar a aplicação sequencial para que esta possa se aproveitar do hardware que está apto a executar código paralelo. Exemplos em que isso corre é utilizando compiladores com possibilidade de otimização paralela, isto é, compiladores paralelos como Sun Studio 12 (SUN, 2007) e Intel C++ Compiler (INTEL CORPORATION, 2008). As vantagens dessa abordagem é o fato de que aplicações já existentes não precisam ser reescritas para funcionar com paralelismo, pois basta recompila-las nestes compiladores para funcionarem sem necessidade do programador

ter de aprender novos conceitos de programação paralela. Contudo a desvantagem é que o compilador não consegue chegar em um mesmo nível de otimização que um programador conseguiria utilizando paralelismo em seus algoritmos.

Já com programação paralela a aplicação é desenvolvida para aproveitar o paralelismo desde seu próprio algoritmo. A vantagem dessa abordagem é que ela fornece maior ganho de performance que a auto-paralelização, e a desvantagem é que exige um maior esforço no desenvolvimento da aplicação.

Avaliou-se as plataformas existentes de programação paralela com base nos sete critérios qualitativos propostos pela APSTC (2008), que são:

a) Arquitetura de sistema: pode ser de memória compartilhada que se refere a sistemas na qual o processador utiliza uma área de memória compartilhada (compartilham dos mesmos endereços de acesso a memória), ou de memória distribuída que se refere a aos casos em que cada nó de processamento tem seu próprio espaço de memória não compartilhado por outros.

b) Metodologia de programação: de que forma os programadores conseguem utilizar os recursos de paralelismo. Por ex.: API, nova linguagem, diretivas especiais.

c) Gerenciamento de trabalho: o paralelismo pode ocorrer através de processos ou de threads. Caso programador precise cuidar da criação e destruição de threads dizemos que o gerenciamento de trabalho é explícito senão ele é implícito e basta especificar a seção de código que vai rodar em paralelo.

d) Esquema de particionamento da carga de trabalho: A carga de trabalho que será executadas é dividida em pequenas porções chamadas tarefas, no critério implícito o programadores precisam apenas especificar qual carga de trabalho vai ser processada em paralelo sem se preocupar em gerenciar isso. Enquanto no critério explícito os programadores precisam decidir manualmente como essa carga de trabalho será dividida.

e) Mapeamento entre tarefa e a thread ou processo: no critério implícito o programador não precisa especificar qual thread/processo é responsável pela tarefa. Já no explícito gerenciar isso é responsabilidade do programador.

f) Sincronização: define em que sequência as threads/processos acessam os dados que compartilham. Na sincronização implícita não há esforço necessário do programador, ou este é mínimo e não é necessário, ou basta especificar que naquele trecho de código ocorrerá uma sincronização, enquanto na sincronização explícita os programadores precisam gerenciar como se dará o acesso dos processos e threads nesta área compartilhada.

g) Modelo de comunicação: Este modelo foca no paradigma de comunicação utilizado por um modelo.

Analisando seis modelos de programação paralela se obtém as tabelas abaixo, na Tabela 1 se vê a arquitetura de um sistema com memória compartilhada, e na Tabela 2 a arquitetura de um sistema que tem sua memória distribuída.

**Tabela 1- Arquitetura de sistema com memória compartilhada**

<b>Critério</b>	<b>Pthreads</b>	<b>OpenMP</b>	<b>CUDA</b>
Execução	Thread	Thread	Thread
Metodologia de programação	API, C, Fortran	API, C, Fortran	API, Extension to C
Gerenciamento de trabalho	Explicito	Implicito	Implicito
particionamento da carga de trabalho	Explicito	Implicito	Explicito
Mapeamento entre tarefa e a thread	Explicito	Implicito	Explicito
Sincronização	Explicito	Implicito/Explicito	Implicito
Modelo de comunicação	Espaço de memória compartilhado	Espaço de memória compartilhado	Espaço de memória compartilhado

Fonte: APSTC (2008)

**Tabela 2 - Arquitetura de sistema com memória distribuída.**

<b>Critério</b>	<b>MPI</b>	<b>UPC</b>	<b>Fortress</b>
Execução	Processo	Thread	Thread
Metodologia de programação	API, C, Fortran	API, C	Nova linguagem
Gerenciamento de trabalho	Implicito	Implicito/Explicito	Implicito/Explicito
Particionamento da carga de trabalho	Explicito	Implicito/Explicito	Implicito/Explicito
Mapeamento	Explicito	Implicito/Explicito	Implicito/Explicito

entre tarefa e a thread			
Sincronização	Implícito	Implícito/Explícito	Implícito/Explícito
Modelo de comunicação	Troca de mensagens	Partição do espaço de endereço de memória global	Espaço de memória global

Fonte: APSTC (2008)

Para manter o escopo deste trabalho serão analisadas apenas arquiteturas de sistema de memória compartilhada, pois possuem características semelhantes a placa de vídeo, ou seja, a GPU compartilha a memória da placa de vídeo.

## 5.2 PTHREADS

Pthreads é um padrão UNIX especificado pelo IEEE POSIX 1003.1c standard, as implementações que aderem a este padrão são conhecidas como Portable Operating System Interface, também conhecido como POSIX Threads ou Pthreads para o uso de threads. Pthreads é um conjunto de tipos e chamadas de funções da linguagem C implementadas no arquivo de header/include pthread.h em conjunto com uma biblioteca de thread (JOHNSON, 2006).

A principal motivação para se utilizar Pthreads é a possibilidade de desempenho que se ganha utilizando essa API e o controle total que se tem sob as threads criadas, apesar desse controle ser um ponto contra pois seu aprendizado e desenvolvimento pode ser mais trabalhoso, visto que o controle sob as threads é feito pelo desenvolvedor.

Uma curiosidade da biblioteca é que todos os identificadores começam por pthreads\_ e para explicar o controle da API do Pthreads sobre as threads a API pode ser agrupada em três classes principais:

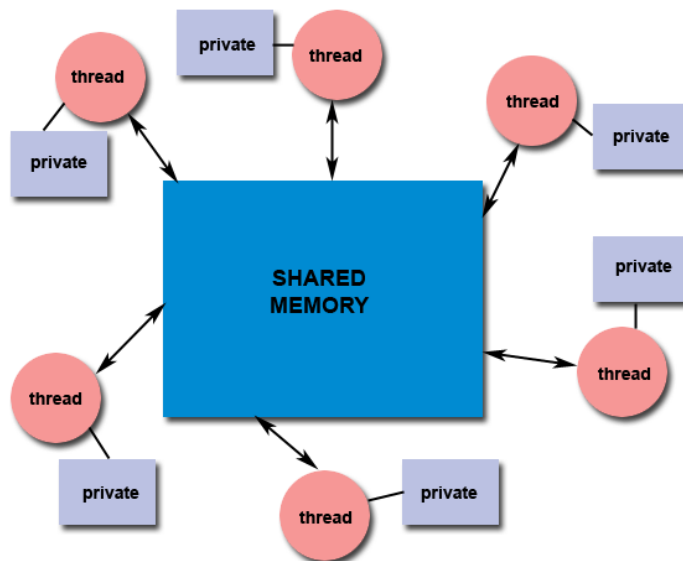
a) A primeira classe de funções trabalha diretamente com as threads, possibilitando o desenvolvedor criar, dividir, juntar, threads. Essa classe pode ser considerada uma classe de Gerenciamento de Threads, que inclui funções de consulta ou criação dos atributos e das threads.

b) A segunda classe de funções é ideal para sincronização da thread, as funções de Mutex, que provêm a criação, exclusão dos mutexes para sincronização e manipulação de threads, a classe Mutex consegue alterar atributos associados ou incluídos com as threads.

c) A terceira classe de funções são funções para comunicação entre as threads. Essa

classe também possui funções para criar, destruir, pausar e marcar as variáveis (THE OPEN GROUP, 2004).

Em Pthreads cada thread tem sua memória privada, e todas tem acesso a uma memória global, como ilustra a figura 12. Os programadores são responsáveis pela sincronização da memória compartilhada, pois em nenhum momento a API sincroniza os dados para que se tenha dados consistentes, com segurança.



**Figura 12 - Memória compartilhada em Pthreads**

Fonte: BARNEY, (2009) B

Para criar uma thread utiliza-se a função `pthread_create()`, que pode ser vista na figura 13, (IBM, 2000) mostra, caso a função seja executada com sucesso retorna 1, caso contrario a função retorna 0.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void*), void *arg);
```

**Figura 13 - Código exemplo PThreads**

Fonte: BARNEY, (2009) B

Na função são utilizados os parâmetros (BARNEY, 2009):

- a) Thread: do tipo `pthread_t`, caso a função ocorra sem problemas retorna o ID da thread no campo;
- b) Attr: Uma estrutura de opções, do tipo `pthread_attr_t`, quando pode ser atribuída como null, pois a função `pthread_create` já tem atributos que são padrão no uso da função;
- c) Start\_routine: que é o nome de uma função do tipo `void` que representa o trecho que será executado dentro da thread;

d) Por ultimo o ponteiro pra arg, que é o único argumento de PrintHello;

A primitiva `pthread_join()` suspende o processamento da thread que a chamou até que a thread identificada na função termine normalmente através da função `pthread_exit()` ou seja cancelada.

A figura 14 mostra o fluxo de execução de um `pthread_join()`.

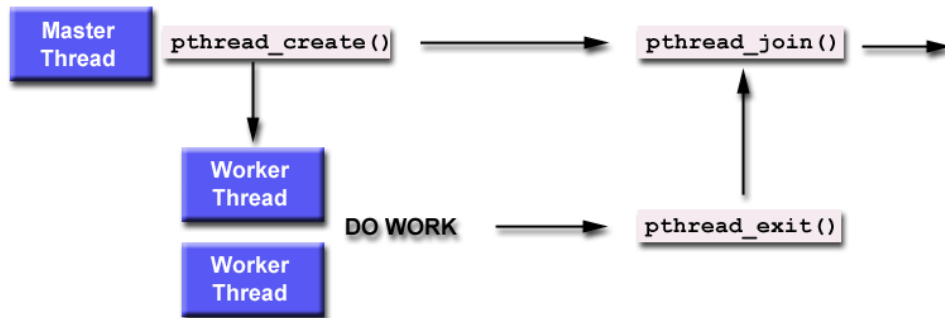


Figura 14 - Execução de um `pthread_join()`

Fonte: BARNEY, (2009) B

Para finalizar uma chamada de thread utilizamos a função `pthread_exit()`, onde `value_ptr` é o valor disponível para retorno. A função não pode encerrar sem este comando (ANTUNES, 2009).

Com as primitivas já é possível criar um código simples, como mostrado na figura 15.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS      5

void *PrintHello(void *threadid){
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread %ld!\n", tid);
    pthread_exit(NULL);}

int main (int argc, char *argv[]){
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);}
```

Figura 15 - Código simples em PThreads

BARNEY, (2009) B

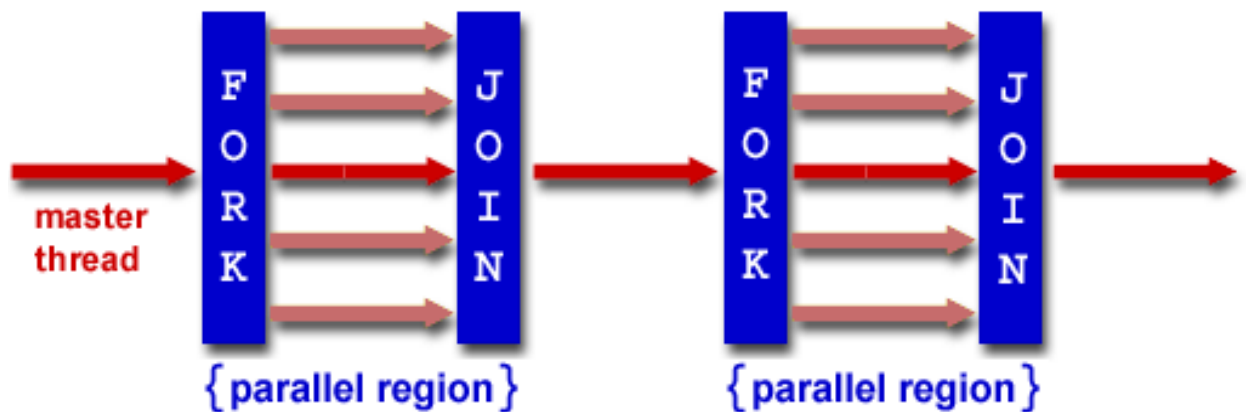
Em Pthreads o controle da thread é feito pelo desenvolvedor, ao contrario de outras APIs de fácil desenvolvimento, como OPENMP que faz todo controle das threads.

### 5.3 OPENMP

OpenMP é uma API para C/C++ e Fortran utilizada no desenvolvimento de programas para sistemas paralelos. A organização sem fins lucrativos ARB é responsável pela especificação da biblioteca. (OPENMP ARCHITECTURE REVIEW BOARD, 2009)

Particularidades do desenvolvimento com OpenMP:

- a) Utiliza diversas threads e memória compartilhada para executar código paralelo
- b) Modelo fork -join: no momento do paralelismo ocorre um fork da thread principal em um grupo de threads escravas. Após a execução do código paralelo as threads se sincronizam e terminam continuando em execução apenas a thread principal, isto é ocorre um join. A figura 16 ilustra o modelo fork-join



**Figura 16- Modelo Fork-Join no OpenMP**

Fonte: BARNEY, (2009)

c) A especificação do OpenMP não cita nada em relação a I/O. Se múltiplas threads precisarem ler/escrever em um mesmo arquivo, é responsabilidade do programador cuidar para que não ocorram conflitos/sobrescritas na leitura do arquivo.

A programação paralela ocorre com o uso de diretivas de compilação. No caso da linguagem C elas ocorrem com o símbolo #. O formato das diretivas é mostrado nas figuras 17 e 18.

```
#pragma omp nome-da-diretiva [clausulas]
```

**Figura 17 - Exemplo código OpenMP**

Fonte: OPENMP ARCHITECTURE REVIEW BOARD, (2009)

Elas são case-sensitive e deve existir uma quebra de linha ao seu final.

Como neste trabalho é utilizada a linguagem C, não foram abordadas as diretivas para Fortran.

```
#pragma omp parallel default(shared) private(beta, pri)
```

**Figura 18 - Exemplo de Diretiva.**

Fonte: BARNEY, (2009)

Uma diretiva para definir uma região de código paralela que seria executada por múltiplas threads seria igual ao trecho de código mostrado na figura 19, encontrado no artigo de Barney (2009), onde mostra um código em C como Exemplo de uma região paralela.

```
#include <omp.h>
main () {
    int nthreads, tid;
    /* Faz o fork das threads e coloca o id de cada uma delas na
    variavel tid */
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }
}
```

**Figura 19 - Exemplo de uma região paralela.**

Fonte: BARNEY, (2009)

No exemplo de código é importante notar a diretiva `#pragma omp parallel` e o bloco de código que está contido entre chaves logo após ela pois este é o trecho de código do bloco que será executado em paralelo. Este bloco também contém o trecho `private(tid)` que indica que a variável `tid` é `private`, e isso implica que cada thread vai cuidar de variáveis `private` de forma independente, então supondo que a thread 1 altere o valor de `tid`, a thread 2 continuará com o valor antigo não inicializado a menos que a thread 2 altere a variável. Após a execução o valor continua como antes do bloco. Ao invés de `private(tid)` a diretiva poderia conter `shared(tid)` e neste caso a variável `tid` seria compartilhada por todas as threads implicando que se a thread 1 alterar o valor dela, as demais threads acessariam este valor alterado.

No código acima a função `omp_get_thread_num()` responsável por fornecer o número da thread que está executando dentro do bloco de código e a função `omp_get_num_threads()` é responsável por fornecer o número total de threads alocadas para execução. O número de threads em execução pode ser alterado com o uso da função `omp_set_num_threads()` ou

fazendo uma atribuição na variável de ambiente OMP\_NUM\_THREADS, por padrão o número de threads costuma ser equivalente ao número de núcleos do processador (OPENMP ARCHITECTURE REVIEW BOARD, 2009), (BARNEY, 2009 A).

A figura 20 mostra um outro exemplo de código.

```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
main ()

int i, chunk;
float a[N], b[N], c[N];
/* alguma inicializacao */
for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i)
{
    #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
        c[i] = a[i] + b[i];
} /* fim da região paralela */
}
```

**Figura 20 - Exemplo de Código OpenMP**

Fonte: BARNEY, (2009)

Neste código a novidade está na diretiva `#pragma omp for schedule(dynamic,chunk) nowait`, `#pragma omp for` serve para fazer o código do laço `for` executar em todas as threads disponíveis. Por exemplo, se existir um laço `for` com 16 iterações e 4 threads no OpenMP, 4 iterações executaram simultaneamente até que todas as 16 iterações do laço sejam concluídas. A diretiva citada também contém `schedule(dynamic, chunk)`, este `schedule` serve para definir como a tarefa será dividida entre as threads que poderia ser por exemplo `schedule(dynamic, chunk)` ou `schedule(static, chunk)`. Nos dois casos o vetor é dividido em pedaços com o tamanho especificado em `chunk` para ser calculado pelas threads, a diferença de como esses pedaços são calculados é especificado através do `dynamic` ou `static`, o `static` quer dizer que cada thread já tem pré-determinado os pedaços que serão calculados enquanto no `dynamic` isso não é pré-determinado, conforme a thread termina o seu trabalho ela já pega o próximo `chunk` para continuar o cálculo. No final da diretiva citada também existe a palavra `nowait`, se ela estiver presente quando o loop finalizar as threads não irão sincronizar, isto é, esperar pelo término da execução das demais threads para continuar sua execução (BARNEY, 2009 A).

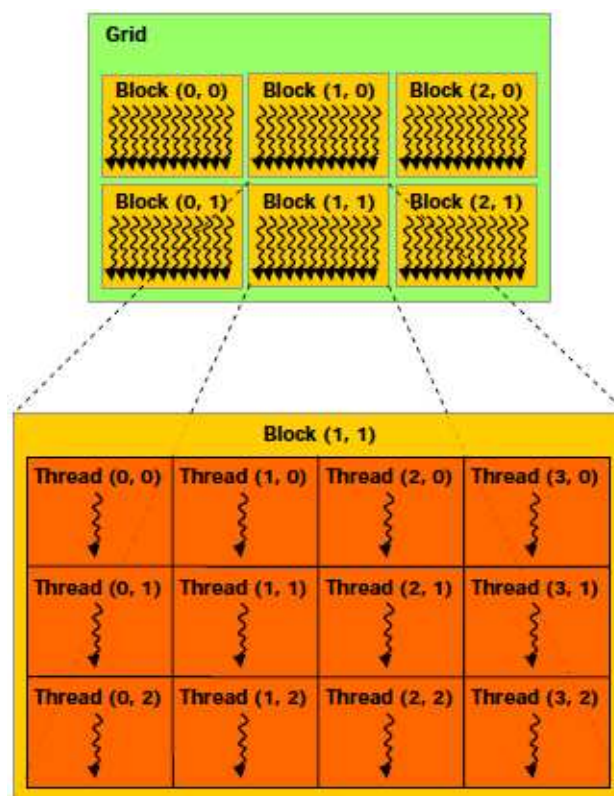
## 5.4 CUDA

Já foi citado o que é CUDA e seu modelo de funcionamento no capítulo sobre GPGPU. C for CUDA é uma extensão do C que permite programar em paralelo aproveitando os diversos multiprocessadores da placa de vídeo. A NVIDIA utiliza a nomenclatura device e host para identificar o que executa na placa de vídeo e o que executa na CPU respectivamente. Neste trabalho esta nomenclatura será seguida para facilitar a identificação de onde determinado código é executado.

### 5.4.1 Threads, blocks e grids

Antes de detalhar o funcionamento de threads, blocos e grids em CUDA, cabe uma breve explicação de como os códigos CUDA são executados. Nesta forma de programação as funções que rodam no device são criadas em código C e chamadas de kernels, quando ocorre uma chamada de função a um kernel o fluxo de execução sai do host, ou seja da CPU, e vai para o device que é a GPU. A placa de vídeo então executa o código do kernel em paralelo e após finalizar a execução, a CPU continua através da thread principal do host o fluxo de execução.

CUDA agrupa as threads em grupos que são chamados de blocks, os blocks por sua vez também são agrupados e chamados de grids. Threads, blocks e grids possuem acessos a tipos de memória diferentes e isto será explicado mais adiante.



**Figura 21 - Organização da Threads**

Fonte: NVIDIA, (2009) d

A figura 21 ilustra como ocorre o agrupamento de threads, blocks e grids. As threads e os blocos podem ser alocadas em até 3 dimensões, os grids por sua vez podem ser alocados em duas dimensões. O hardware no qual o trabalho foi desenvolvido e é descrito no ítem deste trabalho possui limitações, cada block pode conter no máximo 512, 512 e 64 threads nos eixos x, y e z respectivamente. Já cada grid pode conter até 65535 blocos em cada um dos eixos x e y. Também existem 16 multiprocessadores com 8 núcelos cada (NVIDIA CUDA, 2009).

A NVIDIA disponibiliza no Appendix A do Programming Guide as limitações e características existentes para cada uma de suas placas de vídeo suportadas pelo CUDA.

O CUDA precisa lidar com centenas de threads como se pode notar pelos números de citados, contudo nem todas estas threads executam simultaneamente com paralelismo real. Para gerenciar a execução destas threads existe um escalonador implementado via hardware em cada multiprocessador da GPU, cada um desses multiprocessadores é responsável por cuidar de um agrupamento de 32 threads conhecido como warp, dentre as threads do warp apenas 8 por multiprocessador são executadas simultaneamente em cada ciclo da GP, isto é, com paralelismo real, em cada um dos 8 núcelos existentes em cada multiprocessador.

(NVIDIA, 2009 F). O escalonamento destas threads ocorre segundo a NVIDIA com zero overhead. (NVIDIA, 2009 G).

Se durante a execução ocorrer um branch, ou seja uma ramificação, em uma thread devido a alguma operação de controle de fluxo como um “if”, a thread também se divide fazendo que um ramo da execução execute em um ciclo e o outro ramo no outro ciclo levando a uma queda de performance. (NVIDIA, 2009 G).

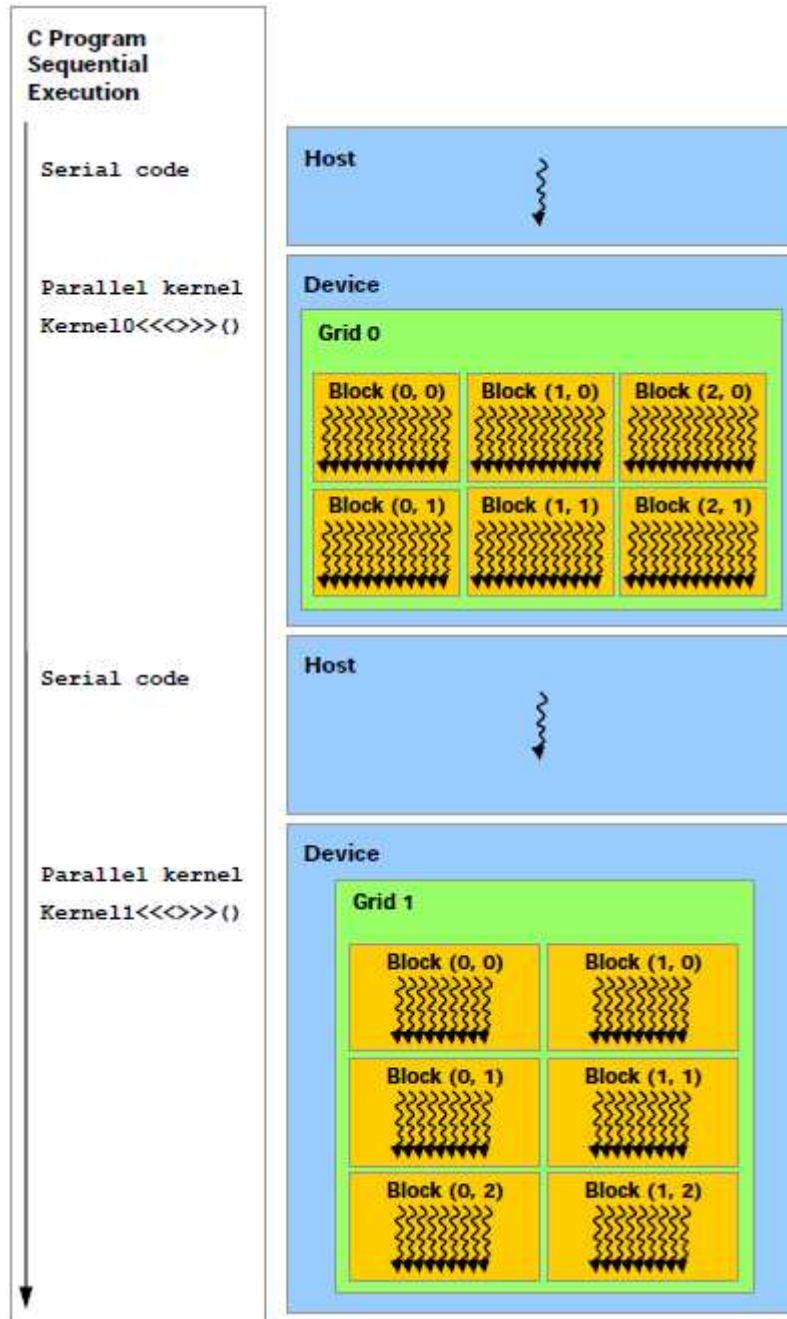
Para que o máximo de threads execute com paralelismo real e se tire o maior proveito possível da placa de vídeo a Nvidia disponibiliza em seu SDK uma tabela de Excel chamada `CUDA_Occupancy_calculator.xls`

Para poder acessar as threads, CUDA fornece as estruturas `threadIdx.x`, `threadIdx.y` e `threadIdx.z` que retornam o número da thread em cada um dos eixos. Também existem as funções `blockDim.x`, `blockDim.y` que fornecem o tamanho dos blocks e as funções `blockIdx.x`, `blockIdx.y`, `blockIdx.z` para fornecer um indicador do blocks.

Uma construção comum para identificar uma thread é  $id = blockIdx.x * blockDim.x + threadIdx.x$ , pois os números de threadix se repetem para cada um dos blocks.

A função `__syncthreads()` fornece sincronismo entre as threads.

A figura 22 ilustra a execução de um programa em C que utiliza CUDA.

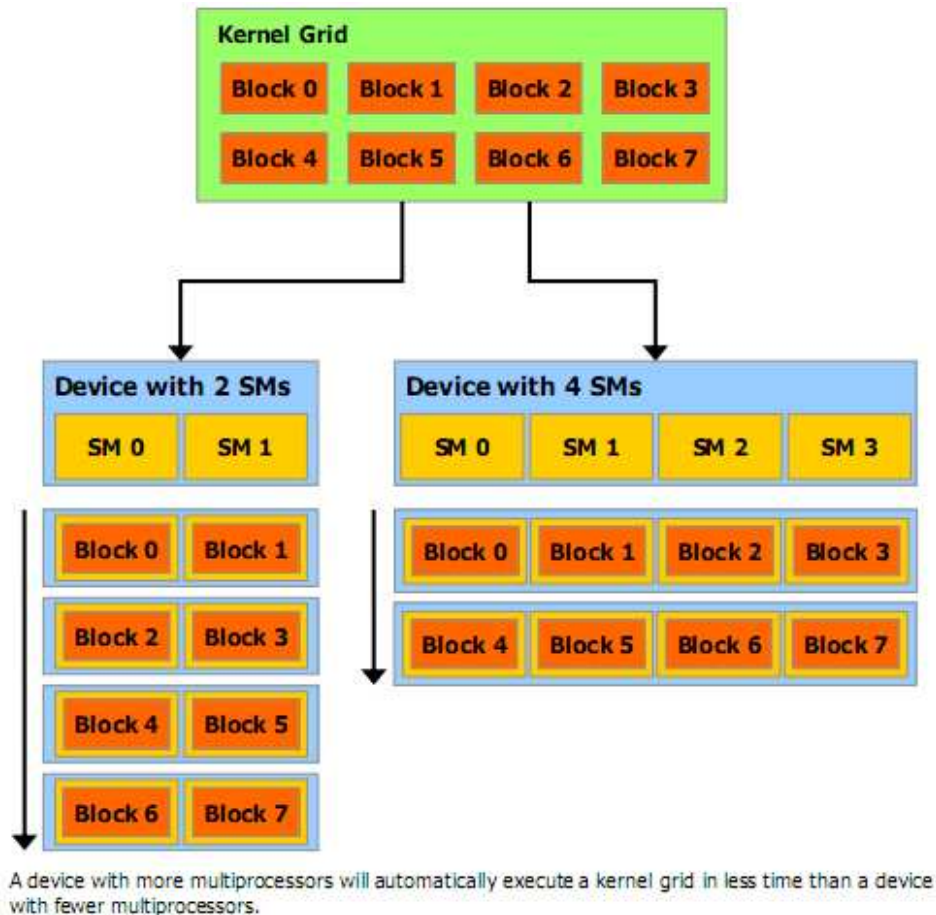


Serial code executes on the host while parallel code executes on the device.

**Figura 22 - Execução de Programa com CUDA.**

Fonte: NVIDIA , (2009) A

Outro ponto interessante é que para o programador é transparente a quantidade de multiprocessadores existente na placa de vídeo, o Cuda automaticamente agrupa os blocos do kernel nos multiprocessadores conforme a disponibilidade da placa de vídeo e faz o escalonamento necessário. A figura 23 ilustra este ponto.



**Figura 23 - Alocação automática de blocos nos multiprocessadores.**

Fonte: NVIDIA CUDA, (2009) A

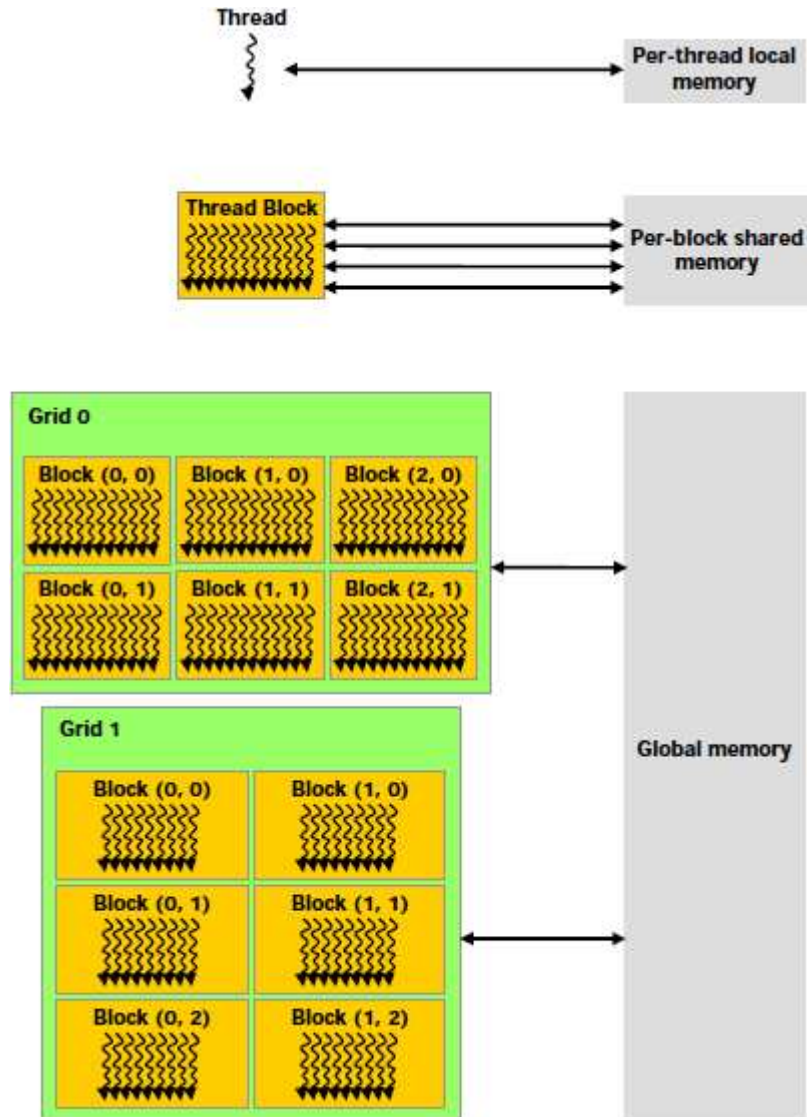
Esta alocação automática faz com que placas de vídeo com mais multiprocessadores executem código mais rápido do que as que possuem menos multiprocessadores e esta escalabilidade ocorre de automaticamente e transparente para o programador.

#### 5.4.2 Gerenciamento de memória

Cada thread possui acesso a uma memória local dela que outras threads não acessam, denominada local memory. Também existe uma memória que é compartilhada dentro do block conhecida como shared memory.

Já no caso dos grids a memória que todos os grids conseguem acessar é conhecida como global memory.

A figura 24 ilustra o acesso a estas memórias.



**Figura 24 - Cuda acesso a memórias.**

Fonte: NVIDIA, (2009 H)

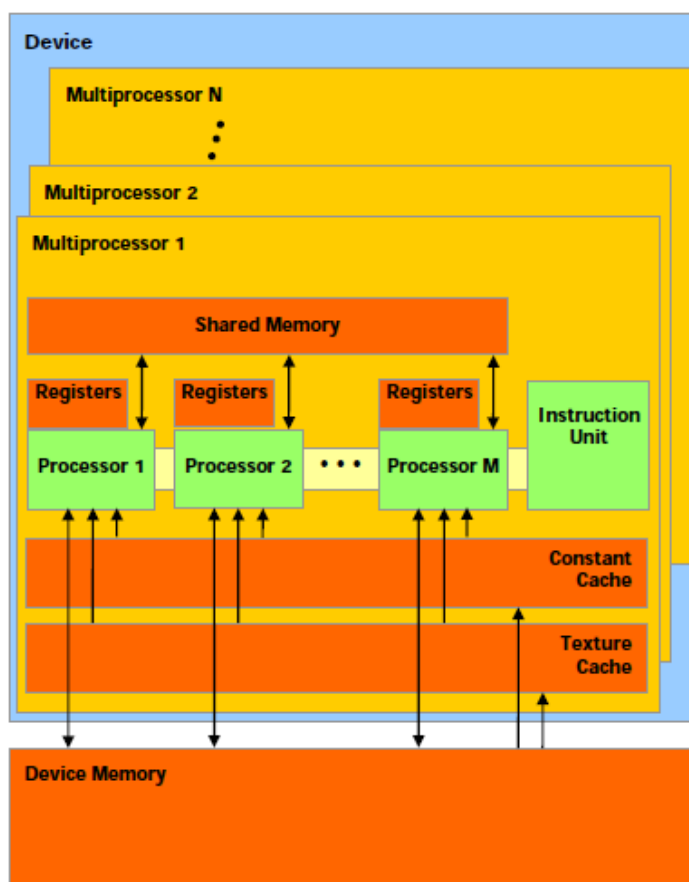
Além das memórias citadas também existe a register memory, texture memory, constant memory.

Existe uma diferença de desempenho no acesso aos dados destas memórias, a memória local e a global possuem o mesmo desempenho. As memórias constant e texture possuem melhor desempenho do que local e global porque utilizam cache no acesso aos dados. Já as memórias shared e register tem o melhor desempenho entre todos os tipos por estarem no mesmo chip do microprocessador (NVIDIA, 2009 H).

As threads possuem acesso somente a leitura nas memórias constant e texture, e consegue ler e escrever nas memórias shared, global, local e register.

O sentido das setas da figura 25 indicam se o acesso ao tipo de memória é somente

leitura ou também inclui escrita.



A set of SIMT multiprocessors with on-chip shared memory.

**Figura 25 - Acesso aos tipos de memórias do Cuda.**

Fonte: NVIDIA, (2009 H)

É importante notar que quando um kernel é executado no device ele consegue acesso apenas a memória RAM do device, portanto não consegue acessar diretamente a memória do host. Normalmente ao se programar os dados estão na memória do host e devido a isso torna-se necessário copiar dados da memória do host para o device para depois fazer a chamada de função ao kernel que acessará estes dados diretamente no device. Um fluxo comum em aplicativos CUDA é alocar a memória no device, copiar os dados do host para o device, fazer a chamada ao kernel, copiar os dados do device para o host, liberar a memória do device e continuar a execução do programa no host.

As funções `cudaMalloc()` e `cudaFree()` são equivalentes as funções `malloc()` e `free()` da linguagem C e permitem alocar e liberar memória global. Além destas duas funções também existe a função `cudaMemcpy()` que tem como finalidade copiar dados do host para o device e vice-versa.

A alocação de memória shared deve ocorrer dentro do kernel com o uso da palavra reservada `__shared__`, por exemplo: `__shared__ float variavel;`

## 6 BENCHMARKS ENTRE CPU, SISTEMAS PARALELOS E GPGPU

Para efetuar o benchmark foram realizadas multiplicações de matrizes preenchidas com números de ponto flutuante, implicando assim em um benchmark com FLOPs.

Foram utilizados dois algoritmos de multiplicação para CPU, um deles na versão não paralela, isto é executado sequencialmente que foi nomeado neste trabalho como C serial. O outro foi construído com paralelismo na CPU utilizando-se OpenMP.

Na GPU implementou-se mais dois algoritmos de multiplicação, sendo um deles utilizando CUDA e o outro com a biblioteca Cublas (NVIDIA CUBLAS, 2008) que é voltada para efetuar operações otimizadas de álgebra linear nas GPUs com suporte a CUDA.

Efetuuou-se cada um dos testes por 3 vezes e foram extraídas a média aritmética dos tempos destas 3 execuções para geração das tabelas e gráficos presentes neste capítulo.

Foi decidido executar cada teste por 3 vezes para se obter um tempo de execução mais homogêneo no qual escalonamentos, execução de outros programas em background no sistema operacional, e demais variáveis envolvidas não interferissem nos valores obtidos. A execução de 3 vezes foi arbitrária e naturalmente um número maior tornaria o tempo de execução mais homogêneo.

Os testes no OpenMP foram realizados com 4 threads, porque o processador utilizado possui 4 núcleos. Já em CUDA o número de threads varia conforme o tamanho das matrizes.

### 6.1 CONFIGURAÇÕES DO AMBIENTE

As multiplicações foram executadas sob o seguinte hardware:

- a) Placa-mãe Asus M3A78
- b) CPU AMD Phenom II X4 940 3.00 Ghz
  - Quantidade de Núcleos: 4
  - Memória interna: L2 - 4 x 512 KB - L3 Cache - 6 MB
  - Velocidade do clock: 3Ghz
- c) 4,00 GB de memória RAM DDR2
- d) Placa de vídeo NVIDIA GeForce GTS 250
  - Quantidade multiprocessadores: 16
  - Quantidade de núcleos: 128
  - Memória RAM: 512MB DDR3
  - Largura da Memória: 256-bit

Largura da Banda da memória(GB/sec): 70.4

Velocidade de clock: 1100Mz

Já em relação ao software utilizou-se:

- a) Windows 7 Ultimate 64bits versão RC 6.1.7100
- b) Microsoft Visual Studio 2008
- c) NVIDIA CUDA Toolkit 2.3 64bits
- d) Driver da placa de video da NVIDIA versão 190.38

Todos os programas foram compilados no Visual Studio com a geração de binário para plataforma de 64bits.

## 6.2 CARACTERÍSTICAS DAS MATRIZES UTILIZADAS

A dimensão das matrizes variou de 500 a 4000, havendo um incremento de 500 a cada multiplicação. Portanto, multiplicou-se matrizes quadradas de 500x500, 1000x1000, 1500x1500, 2000x2000, 2500x2500, 3000x3000, 3500x3500 e 4000x4000. As multiplicações ocorreram dessa forma para analisar o que ocorre em relação ao desempenho conforme a dimensão da matriz é alterada.

Entre as matrizes envolvidas no processo de multiplicação uma delas foi preenchida com valor float 0.5, outra com valor 2.1 e a de resultados com 0.

## 6.3 CODIGOS DAS MULTIPLICAÇÕES

No Código fonte C serial, o valor que aparece em #define TAM foi naturalmente modificado a cada multiplicação para refletir o tamanho da matriz, como pode ser observado no trecho a seguir:

```
#include <stdio.h>
#include <windows.h>
#define TAM 4000
```

O código pode ser visto completo no Apêndice A.

O Código fonte OpenMP foi modificado a partir do código criado por Blaise Barney (2005) e está disponível no apêndice B.

O Código fonte do CUDA foi adaptado do exemplo MatrixMul disponível no Cuda Toolkit. Naturalmente os valores de #define de NRA, NCA e NCB foram adaptados para os tamanhos de matrizes utilizados. Como se pode ver no trecho a seguir extraído do arquivo matrixMul.h:

```
#define NRA 2500          /* numero de linhas da matrix A */
#define NCA 2500          /* numero de colunas da matrix A */
#define NCB 2500          /* numero de colunas da matrix B */
```

O código pode ser visto na íntegra no Apêndice C.

## 6.4 RESULTADOS

As tabelas a seguir apresentam os resultados da execução de cada um dos códigos que podem ser vistos nos apêndices. A coluna “Tamanho” indica o tamanho da matriz quadrada utilizada, todos os tempos são representados em milissegundos sendo que a coluna "Total" indica o tempo total de execução do programa. "Alocações GPU" representa o tempo gasto para copiar os dados da CPU para placa de vídeo, "Multiplicação" é o tempo utilizado pelo trecho de código que faz exclusivamente a multiplicação das matrizes e "Outros" representa o tempo gasto pelas demais operações do programa. “Outros” é equivalente a subtrair do tempo total o tempo gasto com multiplicação e alocações GPU.

A Tabela 3 mostra o desempenho do código em C serial na multiplicação de matrizes com dimensão de 500 a 4000. É importante notar que quando aparece o número zero na coluna "Outros" das tabelas 3 e 4, e zero na coluna "Multiplicação" da tabela 6, esse valor de fato não foi zero, mas sim um valor no qual não houve precisão suficiente na medição de tempo para detectar quanto tempo foi gasto de fato.

**Tabela 3 - Tabela de Desempenho C Serial**

Tamanho	Multiplicação (ms)	Alocações GPU (ms)	Total (ms)	Outros (ms)
500	234	0	234	0
1000	10967	0	10967	0
1500	46738	0	46754	16
2000	118717	0	118748	31
2500	230366	0	230413	47
3000	408769	0	408832	63
3500	657716	0	657794	78
4000	1217307	0	1217401	94

Fonte: Autores (2009)

A Tabela 4 mostra o desempenho do OpenMP multiplicando os mesmos valores das matrizes da Tabela 3, porém utilizando OpenMP.

**Tabela 4 - Tabela de Desempenho OpenMP**

Tamanho	Multiplicação (ms)	Alocações GPU (ms)	Total (ms)	Outros (ms)
500	109	0	109	0
1000	3338	0	3354	16
1500	12651	0	12683	32
2000	29765	0	29812	47
2500	64210	0	64273	63
3000	108935	0	109029	94
3500	167920	0	168060	140
4000	298727	0	298851	124

Fonte: Autores (2009)

A tabela 5 similar as precedentes, mostra o desempenho na multiplicação de matrizes de utilizando CUDA.

**Tabela 5 - Tabela de Desempenho CUDA**

Tamanho	Multiplicação (ms)	Alocações GPU (ms)	Total(ms)	Outros(ms)
500	7,77	3,76	386,8	375,27
1000	39,08	8,76	311,41	263,57
1500	181,49	17,16	543,35	344,7
2000	147,65	24,77	459,33	286,91
2500	851,74	39,27	1181,69	290,68
3000	1060,02	53,75	1463,03	349,26
3500	1222,3	96,6	1717,45	398,55
4000	1223,99	100,42	1790,7	466,29

Fonte: Autores (2009)

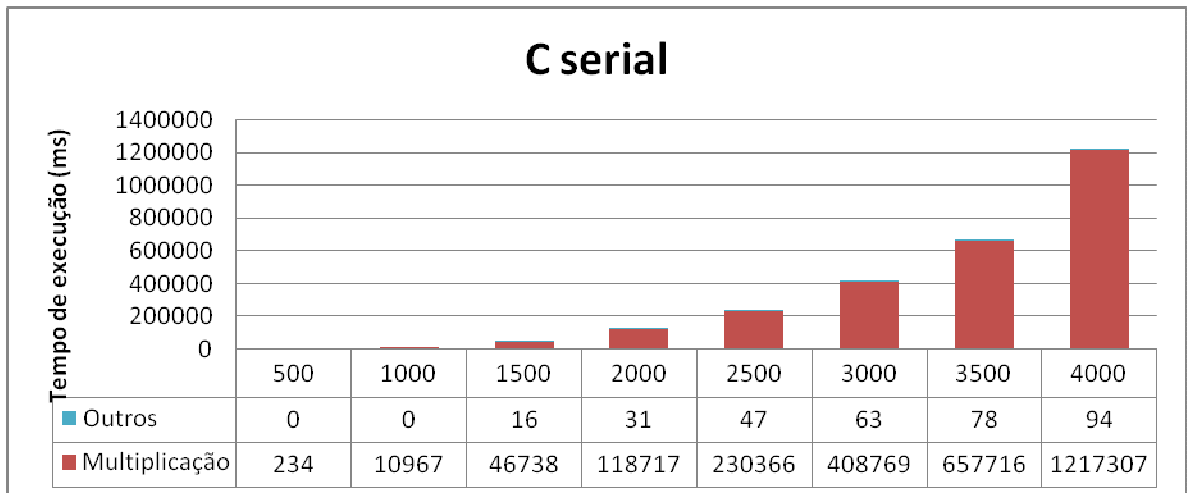
A última arquitetura testada foi a CUBLAS, cujo desempenho é mostrado na Tabela 6

**Tabela 6 - Tabela de Desempenho Cublas**

Tamanho	Multiplicação (ms)	Alocações GPU (ms)	Total (ms)	Outros (ms)
500	0	16	827	811
1000	16	16	905	873
1500	63	63	952	826
2000	125	31	1061	905
2500	250	47	1310	1013
3000	437	62	1560	1061
3500	718	62	1857	1077
4000	796	93	2106	1217

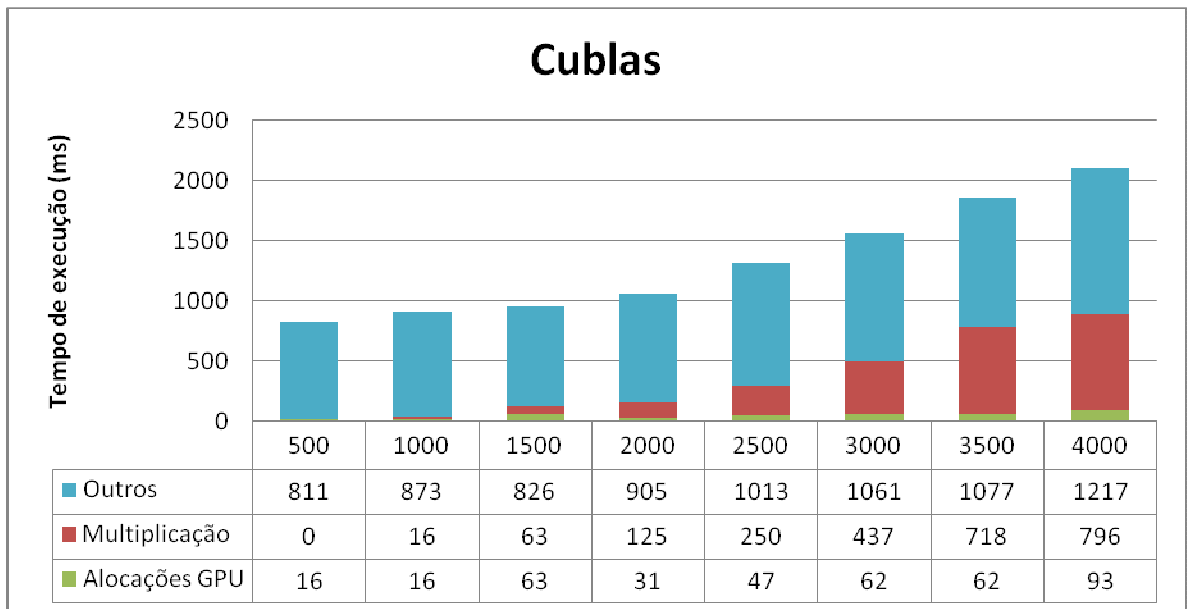
Fonte: Autores (2009)

Os gráficos a seguir facilitam a visualização dos dados contidos nas tabelas. Mostrando no eixo 'x' o tamanho das matrizes e no 'y' o tempo de execução. As legendas indicam o tempo de execução dos trechos dos algoritmos.



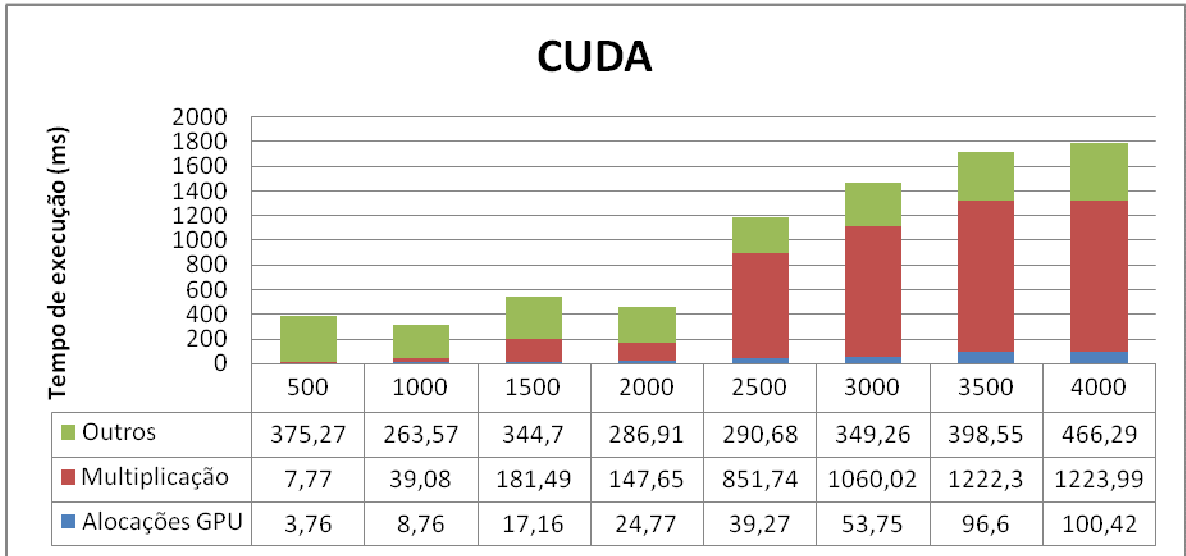
**Figura 26 - Gráfico de desempenho C Serial**

Fonte: Autores (2009)



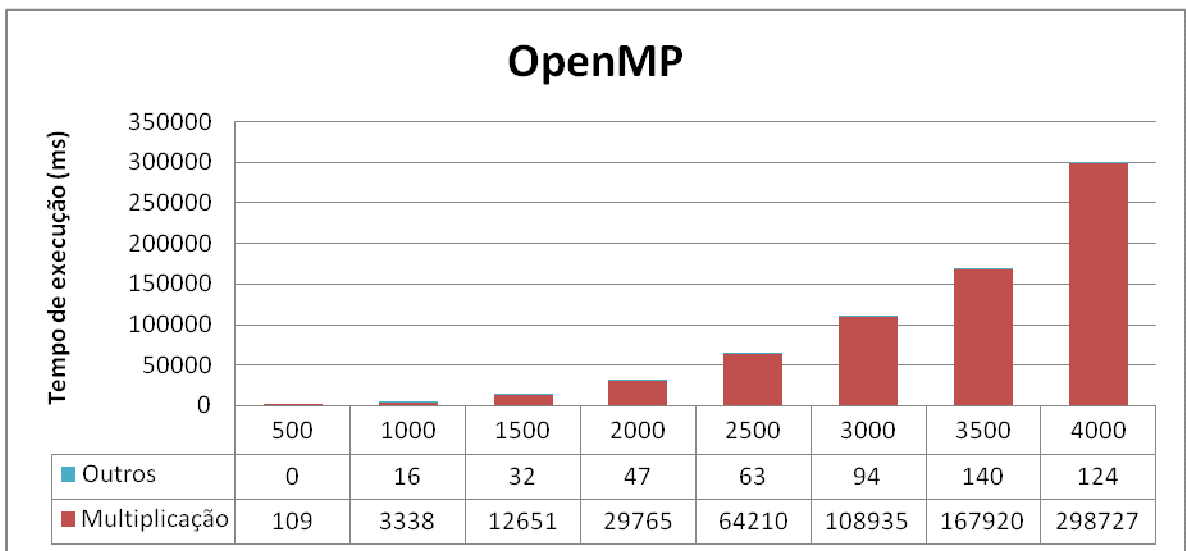
**Figura 27 - Gráfico de desempenho Cublas**

Fonte: Autores (2009)



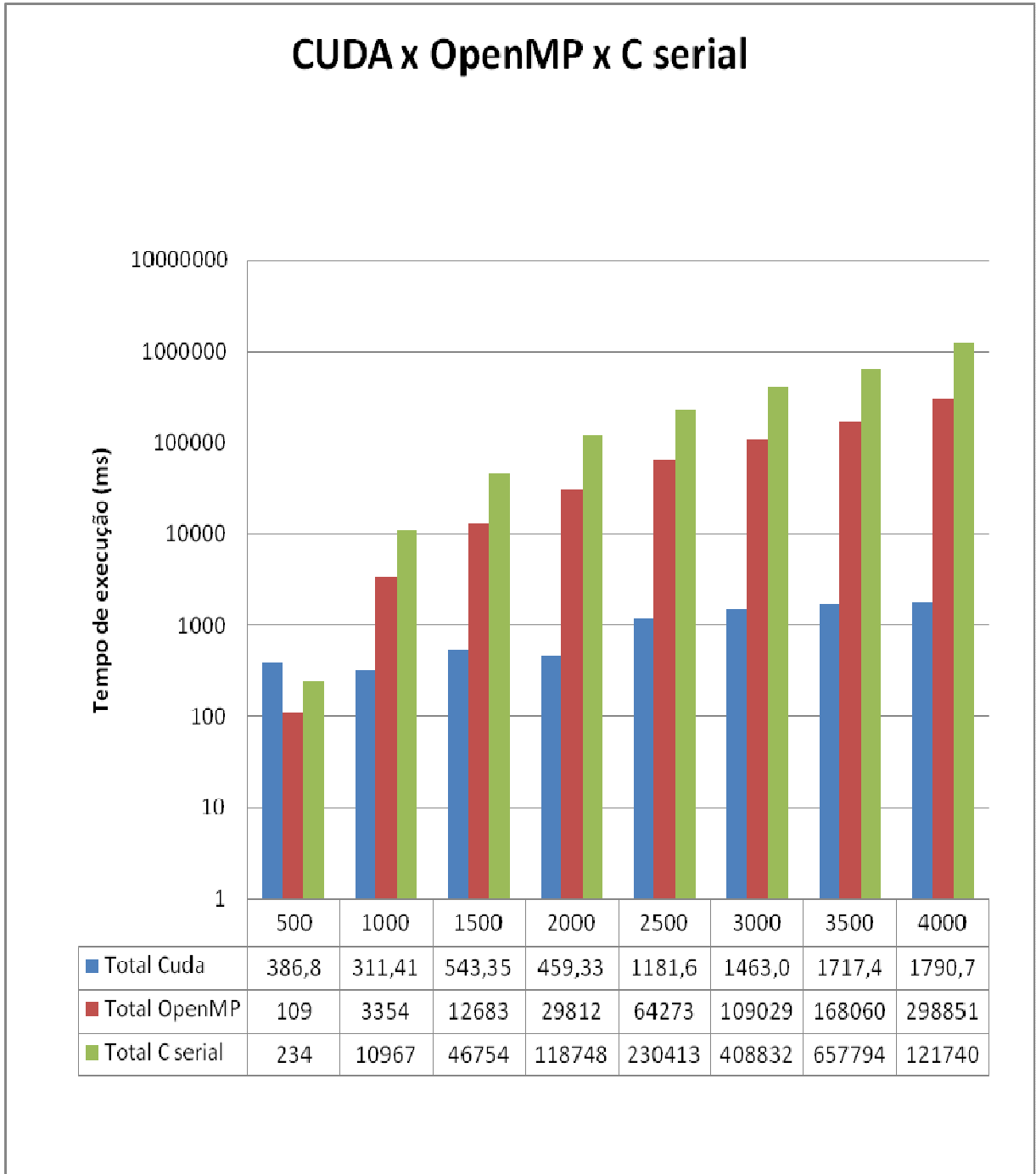
**Figura 28 - Gráfico de desempenho CUDA**

Fonte: Autores (2009)



**Figura 29 - Gráfico de desempenho OpenMP**

Fonte: Autores (2009)



**Figura 30 - Comparação entre as tecnologias**

Fonte: Autores (2009)

## 6.5 ANÁLISE DO RESULTADO

Observando os dados de tempo total obviamente nota-se que o melhor desempenho foi do CUDA, seguido de CUBLAS, OpenMP e C serial. A seguir são descritas observações específicas de cada tecnologia.

### 6.5.1 CUDA

Observando isoladamente os dados do CUDA chega-se a conclusões interessantes, observou-se que para tamanhos menores da matriz como 500, 1000, 1500 e 2000 é gasto mais tempo na categoria "outros" que engloba os contadores de tempo, inicialização serial da matriz e outras funções do Cuda do que na multiplicação em si. Para um valor muito pequeno, a cópia de dados da CPU para GPU leva praticamente metade do tempo da multiplicação, enquanto que para valores grandes a alocação consome menos de 1/10 do tempo da multiplicação.

Outro ponto curioso no gráfico do CUDA, é que o tempo de multiplicação de matrizes com 3500 elementos e das com 4000 elementos varia apenas em 1 ms, enquanto que a multiplicação das matrizes de 1500 elementos levou cerca de 80ms a mais que a multiplicação da matriz de 2000 elementos.

Neste trabalho o item 5.4.1 abordou conhecimentos relacionados ao escalonamento das threads, warp de threads, escalabilidade, e que existe um paralelismo real de apenas 8 threads por multiprocessador. Estes fatores influenciam na divisão do trabalho das threads e por consequência no desempenho apresentado para os diversos tamanhos de matrizes fazendo com que os tempos apresentem a variação encontrada.

Comparando a execução do CUDA com as demais tecnologias, observa-se que ela teve o melhor tempo total, chegando a executar 166 vezes mais rápido que a implementação em OpenMP e 678 vezes mais rápido que a versão C serial. Levando em conta apenas o código que faz a multiplicação, o ganho de desempenho chega a ser 994 vezes maior que o código C serial.

O motivo do Cuda oferecer um desempenho tão grande, é que através do hardware utilizado e da plataforma, ocorre a execução de 128 threads ao mesmo tempo com paralelismo real em núcleos com clock de 1.1Ghz. Conforme o tamanho da matriz aumenta, o Cuda também aumenta o número de threads e faz o escalonamento entre elas automaticamente, com zero overhead. Por outro lado, o OpenMP consegue executar somente 4 threads simultaneas apesar de executar em uma CPU com maior clock.

### 6.5.2 Cublas

Cublas é uma biblioteca da NVIDIA que utiliza Cuda para realizar diversas operações de álgebra linear. A NVIDIA alega que a biblioteca Cublas fornece alta performance para

operações de álgebra linear como a multiplicação de matrizes que foi realizada neste trabalho. Porém, ao efetuar a multiplicação observa-se que no tempo total ela perde por pouco para o CUDA e ganha de todas as demais implementações. Em contra partida, o tempo de multiplicação foi imbatível em comparação com todas as implementações incluindo CUDA, o tempo de execução da multiplicação isoladamente chega a ser 1529 vezes mais rápido que a versão C serial, e chega a ser 375 vezes mais rápida que a versão OpenMP. O tempo de execução da coluna "Outros" revela que este foi o grande motivo para ela perder em desempenho para o Cuda, pois seu tempo para inicializar a biblioteca e as demais operações foi mais custoso. Contudo o tempo da coluna "Outros" cresce menos que o tempo de multiplicação quando a dimensão da matriz aumenta, e devido a isso a tendência é que para matrizes maiores a CUBLAS apresentasse melhor desempenho em relação as outras tecnologias.

### **6.5.3 OpenMP**

A implementação de OpenMP revelou que sistemas paralelos perde em desempenho para o CUDA, com exceção do tempo total para a matriz de dimensão 500, pois neste caso o "Outros" do CUDA torna-se custoso. Porém, levando em conta apenas o tempo de multiplicação, GPGPU leva grande vantagem sobre sistemas paralelos. OpenMP ainda ofereceu melhor desempenho que a versão C serial, chegando a executar 4x mais rápido que ela, sendo um número interessante visto que o OpenMP foi executado com 4 threads em paralelo obtendo 4 vezes mais desempenho.

## 7 CONCLUSÃO

Segundo a análise dos resultados do capítulo 6, CUDA levou vantagem no processamento de FLOPs, mostrando que possui grande capacidade computacional sobre a CPU.

Contudo, alguns problemas foram encontrados durante o desenvolvimento deste estudo.

A documentação oficial da NVIDIA apesar de conter muita informação, possui erros como, por exemplo, os exemplos que copiados do Programming Guide para o compilador não funcionam devido a código incompleto ou erros de codificação.

Outras inconsistências são, por exemplo, o fato de o SDK fornecer exemplos de código, como o de multiplicação de matrizes que foi adaptado neste trabalho, com informações inconsistentes. O código do exemplo de multiplicação citava que era referente ao capítulo 7 do Programming Guide, mas este manual é baixado no site da NVIDIA e não possui capítulo 7. Para completar, ao abrir o NVIDIA Browser SDK que é instalado no computador junto com o Toolkit, o mesmo código que possui comentários sobre o capítulo 7 também é referenciado como pertencente ao capítulo 6 do Programming Guide sendo que também não há capítulo 6 no mesmo.

O software não apresenta completa estabilidade, pois o driver de vídeo reiniciava quando alocava-se matrizes de tamanho grande.

O tratamento de erro nem sempre fornece mensagens claras que ajudam a identificar o problema, mensagens como "unknown error" podem aparecer alternadas com "out of memory".

Observa-se que o tempo despendido para desenvolver um simples algoritmo em CUDA é maior do que utilizar o OpenMP, isso é explicado devido a mudança de paradigma utilizado para se programar em CUDA.

Programar em CUDA requer uma série de conceitos novos que o programador acostumado com a CPU não teria, por exemplo, CUDA não possui apenas um tipo de memória, pode trabalhar com a memória da thread, memória do bloco, memória global e de textura, e é responsabilidade do programador identificar qual memória é mais adequada a cada caso e gerenciá-la. Outro conceito é que as threads são distribuídas ao longo dos eixos x, y e z, aumentando a complexidade para iterar dentro de uma matriz utilizando conceitos do CUDA.

A maior dificuldade neste trabalho foi verificar se as matrizes alocadas excediam o

espaço livre de memória da GPU, pois não existe função para saber quanto espaço está sendo utilizado e quanto está disponível para alocação.

Em OpenMP as dificuldades encontradas foram facilmente sanadas na leitura do manual, livros, documentos e exemplos.

O fato de programar na GPU ser algo recente e pouco utilizado faz com que exista pouca documentação sobre o assunto, dificultando a solução de problemas encontrados. Porém, GPGPU apresenta grande potencial para ganho de desempenho nos algoritmos e a tendência é surgirem mais ferramentas e documentação para facilitar seu uso.

## 7.1 TRABALHOS FUTUROS

Dentre os trabalhos futuros identificados estão à análise aprofundada de como GPGPU tem sido utilizada em softwares como o Nero e Photoshop entre outros para ganho de desempenho. Quando este trabalho foi iniciado, OpenCL ainda era uma tecnologia em desenvolvimento mas que hoje em dia já é utilizada no Mac OS X Snow Leopard, e que agora possui um SDK para a placa de vídeo utilizada nos benchmarks. OpenCL se mostra promissor por ser um padrão aberto ao contrário de CUDA e portanto seria interessante também adicionar benchmarks de OpenCL neste trabalho.

Também seria interessante fazer um trabalho de benchmarks comparando o desempenho entre GPGPU e sistemas distribuídos.

## REFERÊNCIA BIBLIOGRÁFICAS

ADVANCED MICRO DEVICES (AMD) (Estados Unidos). **The Industry-Changing Impact of Accelerated Computing.** Disponível em: <[http://sites.amd.com/us/Documents/AMD\\_fusion\\_Whitepaper.pdf](http://sites.amd.com/us/Documents/AMD_fusion_Whitepaper.pdf)>. Acesso em: 27 maio 2009.

ANTUNES, Rodrigo O. R.. **Servidor TCP em paralelo usando POSIX threads.** Belo Horizonte: I2, 2009. Disponível em: <<http://www.i2.com.br/~rora/aulas/topicos/tcpclient3/>>. Acesso em: 11 nov. 2009.

ASIA-PACIFIC SCIENCE AND TECHNOLOGY CENTER (APSTC) SUN MICROSYSTEMS (China). **Survey on Parallel Programming Model.** Disponível em: <<http://apstc.sun.com.sg/activities/events/past/download/SurveyOnPPM.pdf>>. Acesso em: 14 maio 2009.

BARNEY, Blaise. **Introduction to Parallel Computing.** Disponível em: <[http://computing.llnl.gov/tutorials/parallel\\_comp/](http://computing.llnl.gov/tutorials/parallel_comp/)>. Acesso em: 26 jan. 2009.

BARNEY (A), Blaise. **OpenMP.** California: Lawrence Livermore National Laboratory, 2009. Disponível em: <<https://computing.llnl.gov/tutorials/openMP/>>. Acesso em: 12 nov. 2009.

BARNEY (B), Blaise. **Pthreads.** California: Lawrence Livermore National Laboratory, 2009. Disponível em: <<https://computing.llnl.gov/tutorials/pthreads/>>. Acesso em: 11 nov. 2009.

CHIAN, Felicia. **CPU Register.** Disponível em: <<http://thetechnology4u.blogspot.com/2008/08/forum-1-cnb1003.html>>. Acesso em: 28 maio 2009.

GPGPU TEAM (Estados Unidos). **About GPGPU.org.** Disponível em: <<http://gpgpu.org/about/>>. Acesso em: 20 maio 2009.

GRAÇA, Guillaume Da; DALI, David Defour. **Implementation of float-float operators on graphics hardware.** Disponível em: <<http://hal.archives-ouvertes.fr/docs/00/06/33/56/PDF/float-float.pdf>>. Acesso em: 20 maio 2009.

HALFHILL, Tom R.. **Parallel Processing With CUDA.** Disponível em: <[http://www.nvidia.com/docs/IO/55972/220401\\_Reprint.pdf](http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf)>. Acesso em: 27 maio 2009.

IDA, Cesar Ossamu. **Benchmarks.** Disponível em: <<http://www.inf.ufrgs.br/procpar/disc/cmp134/trabs/T1/001/benchmarks/Benchmarks-cap.htm>>. Acesso em: 14 maio 2009.

INTEL CORPORATION (Estados Unidos). **Product Brief Intel C++ Compiler 11.0 Professional Edition.** Disponível em: <<http://software.intel.com/file/15384>>. Acesso em: 21 maio 2009.

INTERNATIONAL BUSINESS MACHINES (IBM). **POSIX threads explained.** Disponível em: <<http://www.ibm.com/developerworks/linux/library/l-posix1.html>>. Acesso em: 21 maio 2009.

JOHNSON, Ross. **POSIX Threads (pthreads) for Win32.** Disponível em: <<http://sourceware.org/pthreads-win32/>>. Acesso em: 23 maio 2009.

LAWRENCE LIVERMORE NATIONAL LABORATORY. **Introduction to Parallel Computing.** Elaborado por Blaise Barney. Disponível em: <[https://computing.llnl.gov/tutorials/parallel\\_comp](https://computing.llnl.gov/tutorials/parallel_comp)>. Acesso em: 26 jan. 2009.

NVIDIA CORPORATION (A) (Santa Clara). **Get Ready for Windows 7 With NVIDIA Graphics Processors.** Disponível em: <[http://www.nvidia.com/object/windows\\_7.html](http://www.nvidia.com/object/windows_7.html)>. Acesso em: 27 maio 2009.

NVIDIA CORPORATION (B) (Santa Clara). **OpenCL GPU Computing Support on NVIDIA.** Disponível em: <[http://www.nvidia.com/object/cuda\\_opencl.html](http://www.nvidia.com/object/cuda_opencl.html)>. Acesso em: 27 maio 2009.

NVIDIA CORPORATION (C) (Santa Clara). **The Industry-Changing Impact of Accelerated Computing.** Disponível em: <[http://www.nvidia.com/object/product\\_tesla\\_c1060\\_us.html](http://www.nvidia.com/object/product_tesla_c1060_us.html)>. Acesso em: 27 maio 2009.

NVIDIA CORPORATION (D) (Santa Clara). **What is GPU Computing?** Disponível em: <[http://www.nvidia.com/object/GPU\\_Computing.html](http://www.nvidia.com/object/GPU_Computing.html)>. Acesso em: 19 maio 2009.

NVIDIA CORPORATION (E) (Santa Clara). Chapter 2. Programming Model. In: NVIDIA CUDA. **Programming Guide.** 2.3 Santa Clara: Nvidia, 2009. Cap. 2, p. 07-15. Disponível em: <[http://developer.download.nvidia.com/compute/cuda/2\\_2/toolkit/docs/NVIDIA\\_CUDA\\_Programming\\_Guide\\_2.3.pdf](http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf)>. Acesso em: 22 maio 2009.

NVIDIA CORPORATION (G). Chapter 5. Performance Guidelines. In: NVIDIA CUDA. **Programming Guide.** 2.3 Santa Clara: Nvidia, 2009. Cap. 5, p. 77-100. Disponível em: <[http://developer.download.nvidia.com/compute/cuda/2\\_2/toolkit/docs/NVIDIA\\_CUDA\\_Programming\\_Guide\\_2.2.pdf](http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.pdf)>. Acesso em: 25 set. 2009.

NVIDIA CORPORATION (F) (Santa Clara). Chapter 4. Hardware Implementation. In: NVIDIA CUDA. **Chapter 4. Hardware Implementation.** 2.3 Santa Clara: Nvidia, 2009.

Cap. 4, p. 71-75. Disponível em:  
<[http://www.nvidia.de/docs/IO/55972/220401\\_Reprint.pdf](http://www.nvidia.de/docs/IO/55972/220401_Reprint.pdf)>. Acesso em: 25 set. 2009.

NVIDIA CORPORATION (G) (Santa Clara). Programming Interface. In: NVIDIA CUDA. **Programming Guide**. 2.3 Santa Clara: Nvidia, 2009. Cap. 5, p. 72 - 105. Disponível em:  
<[http://developer.download.nvidia.com/compute/cuda/2\\_3/toolkit/docs/NVIDIA\\_CUDA\\_Programming\\_Guide\\_2.3.pdf](http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf)>. Acesso em: 09 nov. 2009.

NVIDIA CUBLAS. **CUBLAS Library**. 2.0 Santa Clara: Nvidia, 2008. 115 p. Disponível em:  
<[http://developer.download.nvidia.com/compute/cuda/2\\_0/docs/CUBLAS\\_Library\\_2.0.pdf](http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf)>. Acesso em: 27 set. 2009.

OPENMP ARCHITECTURE REVIEW BOARD. **About OpenMP and OpenMP.org**. Disponível em: <<http://openmp.org/wp/about-openmp>>. Acesso em: 21 maio 2009.

SUN MICROSYSTEMS. **Sun Studio Features**. Disponível em:  
<<http://developers.sun.com/sunstudio/features/>>. Acesso em: 21 maio 2009.

THE OPEN GROUP. **IEEE Std 1003.1,2004 Edition**. Disponível em: <[http://www.unix-systems.org/version3/ieee\\_std.html](http://www.unix-systems.org/version3/ieee_std.html)>. Acesso em: 21 maio 2009.

ZALUTSKI, Peter. **CUDA advantages and limitations**. Disponível em:  
<<http://ixbtlabs.com/articles3/video/cuda-1-p3.html>>. Acesso em: 19 maio 2009

## APENDICE A – CÓDIGO MULTIPLICAÇÃO DE MATRIZES – C SERIAL

```

#include <stdio.h>
#include <windows.h>
#define TAM 4000

float mtz1[TAM][TAM];
float mtz2[TAM][TAM];
float mtzr[TAM][TAM];

int main(int argc, char** argv) {
    unsigned int i, j, k;
    long t_total1 = GetTickCount();
    for (i = 0; i < TAM; i++) {
        for (j = 0; j < TAM; j++) {
            mtz1[i][j] = 0.5;
            mtz2[i][j] = 2.1;
            mtzr[i][j] = 0;
        }
    }
    /* multiplicando a matriz */
    long t_procl = GetTickCount();
    for (i = 0; i < TAM; i++) {
        for (j = 0; j < TAM; j++) {
            for (k = 0; k < TAM; k++) {
                mtzr[i][j] += (mtz1[i][k] * mtz2[k][j]);
            }
        }
    }
    long t_proc2 = GetTickCount();
    long t_total2 = GetTickCount();
    printf("\nTempo de multiplicacao %f (ms)", (float)(t_proc2-t_procl));
    printf("\nTempo total de execucao %f (ms)", (float)(t_total2-
t_total1));
    printf("\nResultado: %.2f", mtzr[TAM-1][TAM-1]);
    scanf("%d");
    return (EXIT_SUCCESS);
}

```

## APENDICE B - CÓDIGO MULTIPLICAÇÃO DE MATRIZES – OPENMP

```

/*****
****
* FILE: omp_mm.c
* DESCRIPTION:
*   OpenMp Example - Matrix Multiply - C Version
*   Demonstrates a matrix multiply using OpenMP. Threads share row
iterations
*   Disponível em:
https://computing.llnl.gov/tutorials/openMP/samples/C/omp\_mm.c
*   according to a predefined chunk size.
* AUTHOR: Blaise Barney
* LAST REVISED: 06/28/05
* Modificado por Alessandro Vieira e João Medrado
*****/
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define NRA 2500          /* numero de linhas da matrix A */
#define NCA 2500          /* numero de colunas da matrix A */
#define NCB 2500          /* numero de colunas da matrix B */

float a[NRA][NCA],        /* matrix A */
      b[NCA][NCB],        /* matrix B */
      c[NRA][NCB];        /* matrix C que contem resultados */

int main (int argc, char *argv[])
{
  int  tid, nthreads, i, j, k, chunk;

  chunk = 100;            /* set loop iteration chunk size */
  long t_totall, t_total2; /* variavel para armazenar o tempo total de
execução */
  long t_procl, t_proc2; /* variavel para armazenar o tempo de processamento
*/
  t_totall = GetTickCount();
  printf("%d",NRA*NRA*3*sizeof(float));
  omp_set_num_threads(4); // numero de threads
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
  {
    tid = omp_get_thread_num();
    if (tid == 0)
    {
      {
        nthreads = omp_get_num_threads();
        printf("Multiplicando as matrizes com %d threads\n",nthreads);
        printf("Inicializando as matrizes em paralelo...\n");
      }
      /* Inicializando as matrizes */
#pragma omp for schedule (static, chunk)
      for (i=0; i<NRA; i++)
        for (j=0; j<NCA; j++)
          a[i][j]= 0.5;
#pragma omp for schedule (static, chunk)
      for (i=0; i<NCA; i++)

```

```

    for (j=0; j<NCB; j++)
        b[i][j]= 2.1;
#pragma omp for schedule (static, chunk)
for (i=0; i<NRA; i++)
    for (j=0; j<NCB; j++)
        c[i][j]= 0;

/* Multiplicando as matrizes compartilhando iterações do loop externo */
printf("Thread %d iniciando a multiplicacao..\n",tid); // Mostrando qual
thread faz qual interação
t_procl = GetTickCount();
#pragma omp for schedule (static, chunk)
for (i=0; i<NRA; i++)
{
    //printf("Thread=%d did row=%d\n",tid,i);
    for(j=0; j<NCB; j++)
        for (k=0; k<NCA; k++)
            c[i][j] += a[i][k] * b[k][j];
}
} /* fim da região paralela */
t_proc2 = GetTickCount();
t_total2 = GetTickCount();

printf("Tempo de processamento: %f (ms)\n", (float)(t_proc2-t_procl));
printf("Tempo total: %f (ms)\n ",(float)(t_total2-t_total1));

printf("\n Resultado: %6.2f   ", c[NRA-1][NCB-1]);
scanf ("%d");
}

```

## APENDICE C - CÓDIGO MULTIPLICAÇÃO DE MATRIZES – CUDA

matrixMul.h:

```
#define NRA 2500          /* numero de linhas da matrix A */
#define NCA 2500          /* numero de colunas da matrix A */
#define NCB 2500
```

Naturalmente os valores de #define de NRA, NCA e NCB foram adaptados para os tamanhos de matrizes utilizados.

matrixMul.cu:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// para utilizar o timer
#include <cutil_inline.h>

// arquivo com o kernel
#include <matrixMul_kernel.cu>

// prototipos de funcoes utilizadas
void runTest(int argc, char** argv); // executa o teste de multiplicacao
void inicializaMatriz(float*, int, float); // inicializa a matriz

int main(int argc, char** argv){
    unsigned int t_total = 0;
    cutCreateTimer(&t_total);
    cutStartTimer(t_total);
    runTest(argc, argv);
    cutStopTimer(t_total);
    printf("Tempo total de execucao: %f (ms) \n",
    cutGetTimerValue(t_total));
    cutDeleteTimer(t_total);
    cutilExit(argc, argv);
    return 0;
}

void runTest(int argc, char** argv){
    if( cutCheckCmdLineFlag(argc, (const char**)argv, "device") )
        cutilDeviceInit(argc, argv);
    else
        cudaSetDevice( cutGetMaxGflopsDeviceId() );

    unsigned int size_A = WA * HA;
    unsigned int size_C = WC * HC;
    unsigned int size_B = WB * HB;

    unsigned int mem_size_A = sizeof(float) * size_A;
    unsigned int mem_size_B = sizeof(float) * size_B;
    unsigned int mem_size_C = sizeof(float) * size_C;
    float* h_A = (float*) malloc(mem_size_A);
```

```

float* h_B = (float*) malloc(mem_size_B);
float* h_C = (float*) malloc(mem_size_C);

float* d_A;
float* d_B;
float* d_C;

// inicializando a matriz no host
inicializaMatriz(h_A, size_A, 0.5);
inicializaMatriz(h_B, size_B, 2.1);

// alocando memoria na placa de video
cudaMalloc((void**) &d_A, mem_size_A);
cudaMalloc((void**) &d_B, mem_size_B);
cudaMalloc((void**) &d_C, mem_size_C);

// tempo de cópia
unsigned int t_copia = 0;
cutCreateTimer(&t_copia);
cutStartTimer(t_copia);

// copiando dados do host para device
cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice);
cutStopTimer(t_copia);
printf("Tempo de copia das matrizes A e B: %f (ms) \n",
cutGetTimerValue(t_copia));
cutDeleteTimer(t_copia);

// parametros de execução
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(WC / threads.x, HC / threads.y);

// tempo de processamento
unsigned int t_proc = 0;
cutCreateTimer(&t_proc);
cutStartTimer(t_proc);
// chamando a multiplicação
matrixMul<<< grid, threads >>>(d_C, d_A, d_B, WA, WB);
/* após a execução do kernel, o programa continua...
para ele não cair no timer de baixo e atrapalhar o tempo de execução
da alocação
colocamos um cudaThreadSynchronize() aqui, para garantir que a
multiplicação terminou
antes dele continuar executando o programa */
cudaThreadSynchronize();
// checa erros na execucao do kernel, como alocao de memoria
cutStopTimer(t_proc);
printf("Tempo de execucao do kernel: %f (ms) \n",
cutGetTimerValue(t_proc));
cutDeleteTimer(t_proc);

// copiando resultado do device para host
t_copia = 0;
cutCreateTimer(&t_copia);
cutStartTimer(t_copia);
cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);
cutStopTimer(t_copia);
printf("Tempo de copia da matrizes C: %f (ms) \n",
cutGetTimerValue(t_copia));
cutDeleteTimer(t_copia);

```

```
    printf("%.2f ",h_C[(HC-1) * BLOCK_SIZE + (WC-1)]); // imprimindo
ultima linha e ultima coluna
```

```
    // liberando memoria
    free(h_A);
    free(h_B);
    free(h_C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}

// inicializa a matriz data, de tamanho size, com valor
void inicializaMatriz(float* data, int size, float valor)
{
    for (int i = 0; i < size; ++i)
        data[i] = valor;
}

```

matrixMul\_kernel.cu:

```
/* Multiplicação de matrizes: C = A * B.
 * Código que roda na GPU
 */

#ifdef _MATRIXMUL_KERNEL_H_
#define _MATRIXMUL_KERNEL_H_

#include <stdio.h>
#include "matrixMul.h"

#define CHECK_BANK_CONFLICTS 0
#ifdef CHECK_BANK_CONFLICTS
#define AS(i, j) cutilBankChecker(((float*)&As[0][0]), (BLOCK_SIZE * i +
j))
#define BS(i, j) cutilBankChecker(((float*)&Bs[0][0]), (BLOCK_SIZE * i +
j))
#else
#define AS(i, j) As[i][j]
#define BS(i, j) Bs[i][j]
#endif

//////////
/////
///! Multiplicação da Matriz na GPU: C = A * B
///! wA é o largura de A e wB é a largura de B
//////////
/////
__global__ void matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    // Index do Bloco
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Index do Bloco
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index da primeira sub matriz de A processada pelo bloco

```

```

int aBegin = wA * BLOCK_SIZE * by;

// Index da ultima sub-matriz de A processada pelo bloco
int aEnd   = aBegin + wA - 1;

// Passo usado para iterar pelas sub-matrizes de A
int aStep  = BLOCK_SIZE;

// Index da primeira sub-matriz de B processada pelo bloco
int bBegin = BLOCK_SIZE * bx;

// Passo usado para iterar pelas sub-matrizes de B
int bStep  = BLOCK_SIZE * wB;

// Csub é usado para guardar os elementos da submatriz no bloco
// ele é calculado pela thread
float Csub = 0;

// Loop iterando por todas as sub-matrizes de A e B
// requer calculo do bloco de sub-matriz
for (int a = aBegin, b = bBegin;
     a <= aEnd;
     a += aStep, b += bStep) {

    // Declaração do memória compartilhada usada para
    // guardar as sub-matrizes de A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

    // Declaração do memória compartilhada usada para
    // guardar as sub-matrizes de B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Obter os elementos das matrizes da memória do device
    // para a memória compartilhada; cada carregamento de thread
    // implica em um elemento para cada matrix
    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];

    // Sincroniza para ter certeza de que as matrizes foram obtidas.
    __syncthreads();

    // Multiplicar as duas matrizes juntas
    // cada thread calcula um elemento
    // do bloco da sub-matrix
    for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += AS(ty, k) * BS(k, tx);

    // Sincronizar para ter certeza de que os calculos
    // foram feitos antes de criar novas
    // sub-matrizes de A e B na nova iteração
    // sub-matrices of A and B in the next iteration
    __syncthreads();
}

// Escreve um bloco de sub-matrizes na memória do device;
// cada thread escreve um elemento
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
}

#endif // #ifndef _MATRIXMUL_KERNEL_H_

```

Código fonte Cublas, feito apartir da documentação CUBLAS\_Library\_2.0.pdf

```

#include <stdio.h>
#include <windows.h>
#include "cublas.h"
#define N (4000)

int main(int argc, char** argv)
{
    cublasStatus status;
    float* h_A;
    float* h_B;
    float* h_C;
    float* d_A = 0;
    float* d_B = 0;
    float* d_C = 0;
    float alpha = 1.0f;
    float beta = 0.0f;
    int n2 = N * N;
    int i;
    long t_procl, t_proc2, t_alloc1, t_alloc2, t_total1, t_total2;
    t_total1 = GetTickCount();
    cublasInit();
    h_A = (float*)malloc(n2 * sizeof(h_A[0]));
    h_B = (float*)malloc(n2 * sizeof(h_B[0]));
    h_C = (float*)malloc(n2 * sizeof(h_C[0]));
    for (i = 0; i < n2; i++) {
        h_A[i] = 0.5f;
        h_B[i] = 2.1f;
        h_C[i] = 0;
    }
    t_alloc1 = GetTickCount();
    cublasAlloc(n2, sizeof(d_A[0]), (void**)&d_A);
    cublasAlloc(n2, sizeof(d_B[0]), (void**)&d_B);
    cublasAlloc(n2, sizeof(d_C[0]), (void**)&d_C);
    cublasSetVector(n2, sizeof(h_A[0]), h_A, 1, d_A, 1);
    cublasSetVector(n2, sizeof(h_B[0]), h_B, 1, d_B, 1);
    cublasSetVector(n2, sizeof(h_C[0]), h_C, 1, d_C, 1);
    t_alloc2 = GetTickCount();
    t_procl = GetTickCount();
    cublasSgemm('n', 'n', N, N, N, alpha, d_A, N, d_B, N, beta, d_C, N);
    cublasGetVector(n2, sizeof(h_C[0]), d_C, 1, h_C, 1);
    t_proc2 = GetTickCount();
    free(h_A);
    free(h_B);
    cublasFree(d_A);
    cublasFree(d_B);
    cublasFree(d_C);
    cublasShutdown();
    t_total2 = GetTickCount();
    printf("\nResultado: %f",h_C[3999]);
    printf("\nTempo de alocao e copia %f (ms)",(float)(t_alloc2-
t_alloc1));
    printf("\nTempo de multiplicacao %f (ms)",(float)(t_proc2-t_procl));
    printf("\nTempo total %f (ms)",(float)(t_total2-t_total1));
    free(h_C);
    scanf("%d");
    return EXIT_SUCCESS;
}

```