UNIVERSIDADE ESTADUAL DO NORTE DO PARANÁ

***CAMPUS* LUIZ MENEGHEL**

GIULIANA REIS CARDOSO

COMPARATIVO DO DESEMPENHO DE PROCESSAMENTO EM MÁQUINAS COM ARQUITETURAS CPU E GPU

Bandeirantes

2013

GIULIANA REIS CARDOSO

COMPARATIVO DO DESEMPENHO DE PROCESSAMENTO EM MÁQUINAS COM ARQUITETURAS CPU E GPU

Trabalho de conclusão de curso apresentado como requisito parcial para obtenção do título de Bacharelado e Licenciatura em Sistemas de Informação pela Universidade Estadual do Norte do Paraná – *Campus* Luiz Meneghel, sob orientação do professor Estevan Braz Brandt Costa.

Bandeirantes
2013

GIULIANA REIS CARDOSO

COMPARATIVO DO DESEMPENHO DE PROCESSAMENTO EM MÁQUINAS COM ARQUITETURAS CPU E GPU

Trabalho de conclusão de curso apresentado como requisito parcial para obtenção do título de Bacharelado e Licenciatura em Sistemas de Informação pela Universidade Estadual do Norte do Paraná – *Campus* Luiz Meneghel, conferido pela Banca Avaliadora formada pelos professores:

Prof. Estevan Braz Brandt Costa
Universidade Estadual do Norte do Paraná – UENP

Prof. Me. Christian James de Castro Bussmann
Universidade Estadual do Norte do Paraná – UENP

Prof. Me. Luiz Fernando Legore do Nascimento
Universidade Estadual do Norte do Paraná – UENP

Bandeirantes, 24 de junho de 2013.

Dedico este trabalho primeiramente a Deus. Aos meus pais, irmã, professores e amigas que estiveram sempre presentes ao longo desta jornada. Dedico também ao Fabricio Carneiro Mendes, que de forma especial e carinhosa me apoiou na realização deste trabalho.

RESUMO

Com o aumento da quantidade de dados coletados, o desempenho computacional se torna um fator crucial para indicar o sucesso ou o fracasso da execução de uma aplicação científica. Existem opções de arquiteturas que demonstraram grande poder de processamento, no entanto são subtilizadas. Uma delas é a Unidade de Processamento Gráfico, GPU (*Graphic Processing Unit*). O presente trabalho analisa o desempenho computacional deste ambiente, através da implementação do algoritmo de multiplicação matricial. Este foi escolhido devido a sua grande variedade de implementações, desenvolvido na linguagem OpenCL. A problemática é identificar a arquitetura mais vantajosa, através de comparações entre máquinas com o processamento convencional, CPU (*Central Processing Unit*), com as que possuam GPU acoplada à placa de vídeo. Foi criado e aperfeiçoado o código em C++ para que atuasse de forma a acelerar o processamento, e posteriormente transcrito em OpenCL. Em seguida, foram executados os testes em máquinas com CPU e em outro dispositivo que também possua GPU. Após a obtenção dos resultados, os mesmos foram analisados e comparados, e demonstraram que a GPU se sobressai quando há uma grande quantidade de dados.

Palavras – chave: GPU, CPU, OpenCL, Desempenho Computacional.

ABSTRACT

With the increasing amount of data collected, the computational performance becomes a crucial factor to indicate the success or failure of the execution of a scientific application. There are options architectures that showed great processing power, yet they are underused. One is the Graphic Processing Unit, GPU. This paper analyzes the performance of this computing environment, through the implementation of the matrix multiplication algorithm. This was chosen because of its variety of implementations, developed in the language OpenCL. The problem is to identify the most advantageous architecture, by comparing the conventional processing machines, CPU (Central Processing Unit), with machines possessing GPU coupled to the video card. The code was created and optimized in C++ so that acted in a way to accelerate processing, and later transcribed into OpenCL. Then, tests were run on machines with CPU and another device that also has the GPU. After obtaining the results, they were compared and analyzed, and showed that the GPU excels when there is a large amount of data.

Keywords: GPU, CPU, OpenCL, Computational Performance.

LISTA DE FIGURAS

LISTA DE SIGLAS

ALU	*Arithmetic Logic Unit*
	(Unidade Lógica e Aritmética)
API	*Applications Programming Interface*
	(Interface de Programação de Aplicativos)
CISC	*Complex Instructions Set Computer*
	(Computador com um Conjunto Complexo de Instruções)
CPU	*Central Processing Unit*
	(Unidade Central de Processamento)
CRISC	*Complex And Reduced Instructions Set Computer*
	(Computador com um Conjunto Complexo e Reduzido de Instruções)
GPGPU	*General-Purpose computation on GPU*
	(Computação de uso geral em GPU)
GPU	*Graphic Processing Unit*
	(Unidade Gráfica de Processamento)
MIMD	*Multiple Instruction, Multiple Data*
	(Múltiplas Instruções, Múltiplos Dados)
MISD	*Multiple Instruction, Single Data*
	(Múltiplas Instruções, Dados Simples)
RISC	*Reduce Instruction Set Computer*
	(Computador com um Conjunto Reduzido de Instruções)
SIMD	*Single Instruction, Multiple Data*
	(Instruções Simples, Múltiplos Dados)
SISD	*Single Instruction, Single Data*
	(Instruções Simples, Dados Simples)

SUMÁRIO

1 INTRODUÇÃO

Originalmente os computadores foram criados para nos auxiliar na realização de operações matemáticas de forma precisa em uma velocidade muito maior. O tempo que o computador demora realizando estas operações pode determinar seu desempenho. Logo a busca por um melhor desempenho se fez necessária desde o inicio, objetivando computadores executarem mais cálculos em menos tempo, resolvendo problemas de magnitudes cada vez maiores.

Primeiramente a opção de melhorar o desempenho estava diretamente vinculada a aumentar o poder computacional, esse representado pelo processador. Diversas técnicas foram utilizadas, como a divisão do processador em vários estágios distintos e a miniaturização dos componentes. No entanto, essas técnicas não estão mais surtindo efeito por atingirem as barreiras do limite físico. Por conseguinte surgiram outras soluções permitindo que o desempenho melhorasse continuamente, tal como trabalhar com mais de um processador ou com processadores com múltiplos núcleos (*multicore*).

Uma solução recente que teve destaque foi aliar a Unidade Central de Processamento, CPU (*Central Processing Unit*), aos aceleradores gráficos compostos por um processador e memória RAM, cujo processamento é realizado paralelamente por milhares de processos (*threads*), executando com rapidez e agilidade grande parte dos cálculos, sendo anteriormente executado somente pela CPU.

Esses aceleradores gráficos referem-se as placas de vídeo, mais conhecidas como Unidade de Processamento Gráfico, GPU (*Graphic Processing Unit*). Originalmente foram desenvolvidas para melhorar em tempo real os efeitos em jogos de computador, agora são onipresentes e fornecem poder computacional às aplicações científicas sem precedentes (SCHMEISSER, 2009). Contudo é indispensável um conhecimento aprofundado sobre as técnicas e arquitetura dessas placas por parte do pesquisador para que ocorram ganhos a contento de desempenho.

Para facilitar a execução das GPU's despontou a necessidade de criar padrões, o que impulsionou a criação do consórcio tecnológico Khronos *Group*, gerenciador da

plataforma OpenCL. Essa plataforma permite escrever programas que executam em arquiteturas heterogêneas como CPUs, GPUs e outros processadores *multicore*. Ademais, nela está inserida uma linguagem baseada em C e em Interfaces de Programação de Aplicações, API (*Applications Programming Interface*), que são usadas para definir e controlar as arquiteturas heterogêneas.

O trabalho apresentado nesta dissertação consiste em aperfeiçoar esse ambiente. Analisando e equiparando o desempenho de execução de algoritmos matemáticos em computadores pessoais que possuam apenas CPU, com arquiteturas que tenham GPU acopladas. Evidenciando a melhoria de desempenho em operações com pontos flutuantes, sendo esse de grande valia para alavancar o conhecimento sobre o poder computacional das GPU's.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Comparar o desempenho de processamento em máquinas que contenham a arquitetura CPU com as que também possuam a arquitetura GPU.

1.1.2 Objetivos Específicos

- Elencar as principais técnicas para melhoria de desempenho em operações com pontos flutuantes;
- Criar um algoritmo de multiplicação de matriz na plataforma OpenCL; e
- Comparar o impacto da utilização de máquinas com as arquiteturas CPU e GPU para resolução do algoritmo.

1.2 JUSTIFICATIVA

No Brasil e em diversos países o foco da utilização das GPUs continua sendo os aceleradores gráficos para jogos 3D, os usuários não utilizam sua capacidade para execuções de algoritmos com propósitos gerais. Por essa razão se dá a justificativa deste trabalho, a urgência em fazer os algoritmos utilizarem da melhor forma possível o ambiente que lhes é ofertado.

Para alcançar essa melhora, diversas técnicas e métodos surgiram para o uso da CPU. Porém, a expansão do conhecimento da arquitetura GPU entre a população trouxe uma nova forma de otimização dos algoritmos.

Logo, surgiram plataformas que auxiliam o programador a desenvolver algoritmos para GPU facilmente, porém a sua utilização está longe de ser trivial, já que há a necessidade de ter um conhecimento aprofundado sobre a estrutura e a utilização destas plataformas a fim de obter resultados satisfatórios ao desempenho computacional.

1.3 METODOLOGIA

Essa pesquisa é classificada como descritiva por não ter como objetivo a proposição de soluções, mas sim a descrição de fenômenos. O objetivo é analisado de forma a aprofundar no ambiente estudado, descrevendo todos os seus lados e verificando o problema e não a solução, uma vez que esse não é seu objeto. A descrição da necessidade dos dados que foram levantados, é caracterizada por quantitativa, pois aferi o que pode ser mensurado, medido e contado (BONAT, 2009).

Baseada nessa classificação, foram levantados os avanços para se alcançar novos patamares de desempenho com operações de ponto flutuante. Após esta análise, foram encontrados métodos para verificar as melhorias de desempenho alcançadas. Para tal, foi utilizado o algoritmo de multiplicação matricial devido a grande variedade de implementações encontradas na literatura. Uma vez finalizada essa etapa,

o algoritmo de multiplicação matricial foi refeito em OpenCL para uma comparação entre estas implementações e as soluções feitas para CPU.

1.4 HIPÓTESE

O algoritmo de multiplicação matricial será calculado mais rápido ao ser utilizado em uma máquina que possua GPU.

1.5 ORGANIZAÇÃO DO TRABALHO

A disposição deste trabalho está segmentada da seguinte forma, no primeiro Capítulo são apresentadas as áreas abordadas na pesquisa, tal como as funções delas. No segundo Capítulo, estão descritos os assuntos essenciais abordados no trabalho e que foram utilizados para o desenvolvimento. No terceiro Capítulo contém a proposta de desenvolvimento, assim como o método que foi utilizado para alcançar os objetivos. Encontra-se no quarto Capítulo os resultados obtidos após a execução dos testes e suas explicações. No quinto Capítulo estão as conclusões e conhecimentos aprendidos ao longo da realização deste trabalho. Por fim, as referências utilizadas como base para o referencial teórico e prático do estudo.

2 FUNDAMENTAÇÃO TEÓRICA

O presente Capítulo levantará os principais conceitos relacionados ao trabalho.

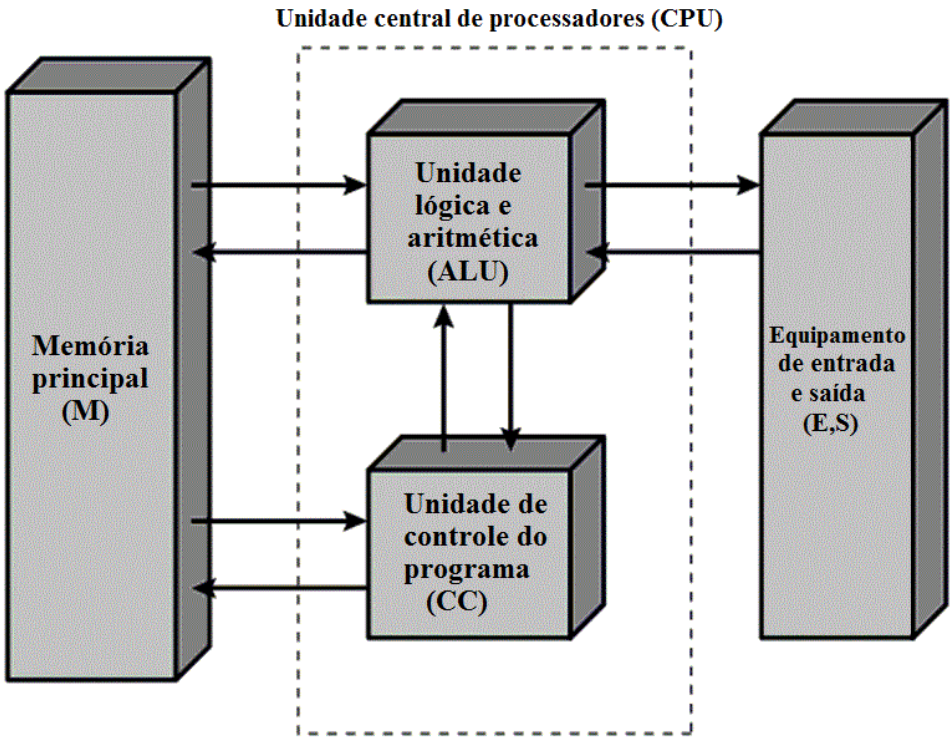
2.1 ARQUITETURA DE COMPUTADORES

O termo arquitetura de um computador segundo Stallings (2010, p. 6) “refere-se aos atributos de um sistema visíveis a um programador ou, em outras palavras, aqueles atributos que possuem um impacto direto sobre a execução lógica de um programa”. Alguns exemplos destes atributos são: conjunto de instruções, número de bits que representam dados, mecanismos de entradas e saídas e técnicas para endereçamento de memória (STALLINGS, 2010).

Para melhorar esse impacto e alcançar um desempenho superior às arquiteturas de computadores, o matemático John von Neumann projetou um protótipo de computadores para uso geral, que recebeu o nome de Máquina de von Neumann.

Como observado na Figura 1, essa estrutura é composta de uma memória principal, que armazenam dados e instruções; uma unidade lógica e aritmética, capaz de operar sobre dados binários; uma unidade de controle, que interpreta as instruções na memória e faz com que sejam executadas; e equipamento de entrada e saída operado pela unidade de controle (STALLINGS, 2010).

Figura 1 - Estrutura da Máquina de von Neumann.



Fonte: Adaptado de Stallings (2010).

Além disso, Stallings (2010) relata que von Neumann baseou-se dos seguintes fundamentos para idealizar seu feito:

- O computador tem que realizar operações aritméticas constantemente, portanto o mesmo tem a necessidade de uma unidade que cumpra apenas essas funções, referente à parte da unidade lógica e aritmética, ALU (*Arithmetic Logic Unit*);
- Uma memória considerável (M) é de grande relevância para execução de cálculos longos e complicados;
- O dispositivo tem que manter contato de entrada e saída (E/S) com algum meio; e
- A execução das operações tem que estar sequenciadas, função que é realizada com eficiência por um órgão de controle central (CC).

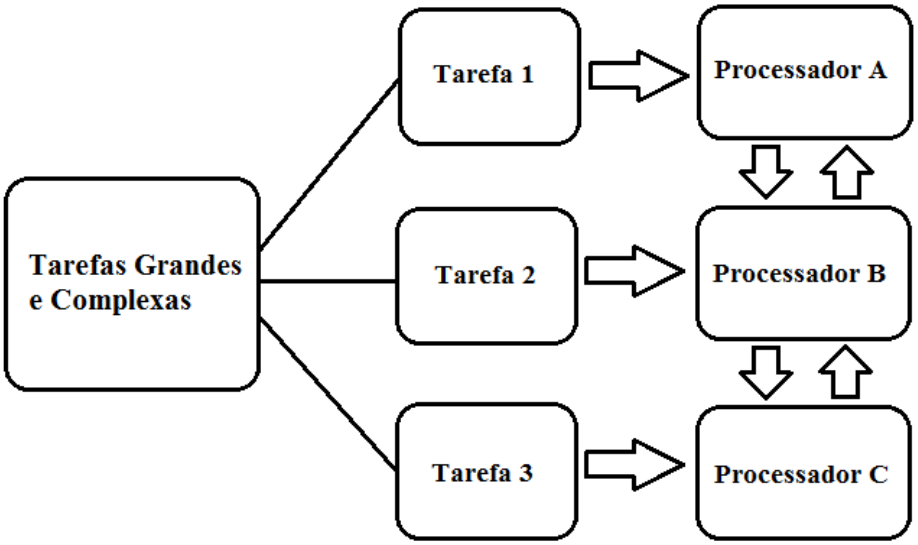
Na unidade central citada, é executado o processamento das tarefas sequenciais, cujo principal componente é o processador. Este *hardware* por sua vez pode ser considerado o cérebro do computador, pois é incumbido de processar a maioria das informações. Ele recebeu melhorias expressivas quanto ao desempenho, sendo que uma delas foi o acréscimo de dois ou mais núcleos em um único processador, permitindo a divisão das tarefas entre si, o ganho de velocidade nas resoluções de algoritmos matemáticos, o aumento no desempenho de modo geral e trabalhando em um ambiente multitarefa.

Com essas melhorias se deu o surgimento do multiprocessamento, que permite a execução de vários programas simultaneamente por meio de processadores que utilizam o mesmo sistema operacional. Desse modo, explorando o paralelismo das tarefas e tornando possível ampliar a capacidade de computação de um sistema (ENGHOLM, 2013).

Para o aproveitamento total desse ambiente se faz necessário uso de *threads*. Estas são uma parte de um programa que compartilha recursos do processador com outras *threads*. Ela pode ser considerada como uma tarefa simples, essas tarefas são obtidas através da divisão de um problema complexo em partes menores. Dessa forma, o processador que trabalha corretamente com a *thread* é capaz de executar de maneira simultânea vários programas (GEBALI, 2011).

Ademais, Pitanga (2008) discorre que o paralelismo de núcleos do multiprocessamento pode ser definido como uma técnica usada em processos grandes e complexos para atingir resultados com rapidez, dividindo as tarefas em porções menores, *threads*, e distribuindo-as em vários processadores para executarem de forma simultânea, demonstrado na Figura 2.

Figura 2 - Paralelização das Tarefas.



Fonte: Adaptado de Pitanga (2008).

Além de tudo, o multiprocessamento necessita trabalhar junto a um *hardware* que faça o trabalho lógico, o mais popular é o multiprocessador. O desenvolvimento de *software* convencional não usufrui das qualidades que esse meio oferece sem passar por alguns problemas. Estes são enfrentados usando refinamento sucessivo, começando com um modelo idealizado em que as preocupações matemáticas são fundamentais, e gradualmente passando para modelos mais usuais (HERLIHY e SHAVIT, 2008).

Para auxiliar os desenvolvedores na utilização desses multiprocessadores, foi criada uma taxonomia usual e eficaz, elaborada por Michael Flynn, com a classificação adequada de modelos de arquiteturas de computadores, baseada no número de fluxos de dados e de instruções contidas em cada instante (PITANGA, 2008). A classificação de Flynn pode ser verificada na Tabela 1.

Tabela 1 - Taxonomia da Arquitetura de Computadores de Flynn.

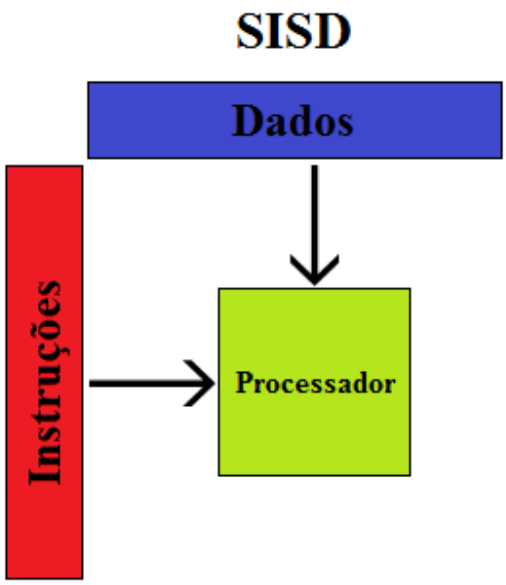
	Instruções Simples	Várias Instruções
Dados Simples	SISD	MISD
Vários Dados	SIMD	MIMD

Fonte: Adaptado de Siewert (2012).

Para mais, Pitanga (2008) apresenta as arquiteturas que são usadas para descrever o tipo de paralelismo da classificação de Flynn:

- **SISD (*Single Instruction, Single Data*):** só é executada uma instrução por vez para cada dado enviado, por isso o equipamento é considerado sequencial. Trata-se de máquinas de von Neuman tradicionais, sem paralelismo algum;

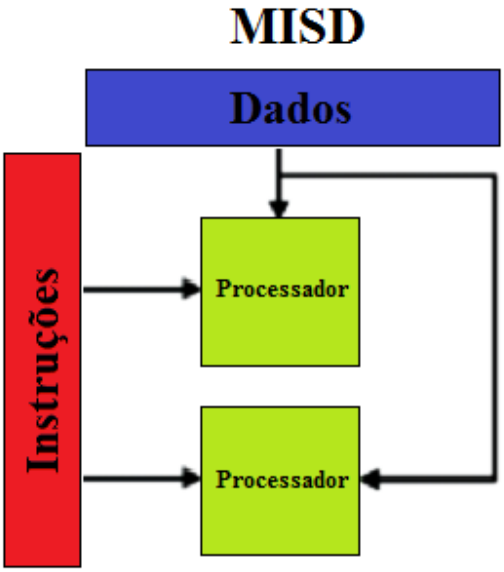
Figura 3 - SISD (Single Instruction, Single Data.



Fonte: Adaptado de Schmeisser (2009).

- **MISD (*Multiple Instruction, Single Data*):** são máquinas que executam várias instruções ao mesmo tempo sobre um único dado;

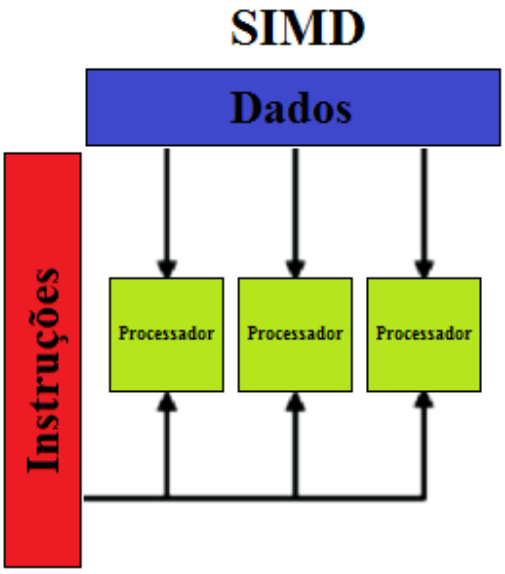
Figura 4 - MISD (Multiple Instruction, Single Data).



Fonte: Adaptado de Schmeisser (2009).

- **SIMD (*Single Instruction, Multiple Data*):** é o equivalente ao paralelismo de dados, uma instrução simples é executada paralelamente utilizando vários dados; e

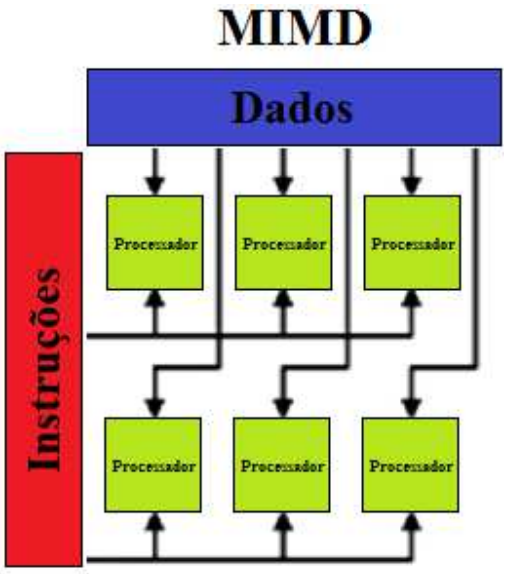
Figura 5 - SIMD (Single Instruction, Multiple Data).



Fonte: Adaptado de Schmeisser (2009).

- **MIMD (*Multiple Instruction, Multiple Data*):** cada processador age independentemente, havendo múltiplos fluxos de instruções e múltiplos dados.

Figura 6 - MIMD (Multiple Instruction, Multiple Data).



Fonte: Adaptado de Schmeisser (2009).

Apesar das categorias estarem bem distintas, a utilização delas precisa que o *hardware* e o *software* trabalhem focados na categoria escolhida. Algumas arquiteturas utilizadas para compor os *hardwares* se tornaram conhecidas por sua eficiência e sua portabilidade entre as categorias de multiprocessamento. Dentre estas, estão a CPU e a GPU.

2.1.1 CPU

Conforme Ruas (2008) a CPU, também chamada de microprocessador, é considerada o principal elemento do computador, com a função de processar toda informação. É o componente de *hardware* que realiza atividades de cálculos, executa instruções e controla o fluxo de informações. Nela as instruções e comandos de

programas são interpretados, controlados e executados, tal qual gerencia todo o equipamento.

As principais funções dessa arquitetura é expladana por Oliveira (2008):

- Executar as instruções dos programas;
- Controlar os programas que estão sendo executados;
- Realizar os cálculos lógicos e aritméticos; e
- Controlar os dispositivos de entrada e saída.

Segundo Rainer e Cegielski (2012) o microprocessador possui pares distintos que realizam tarefas diferentes. A unidade de controle acessa de maneira sequencial as instruções do programa, decodifica e controla o fluxo de dados nos sentidos de ida e volta para a ALU, registradores, *caches,* armazenamento primário, armazenamento secundário e vários dispositivos de saída.

Pertinente a ALU, esta assume os cálculos matemáticos e faz comparações lógicas. Já as informações necessárias para esses cálculos se encontram nos registradores. Como estes são muito pequenos, eles não têm capacidade de guardar uma grande quantidade de dados, delegando essa função aos outros componentes.

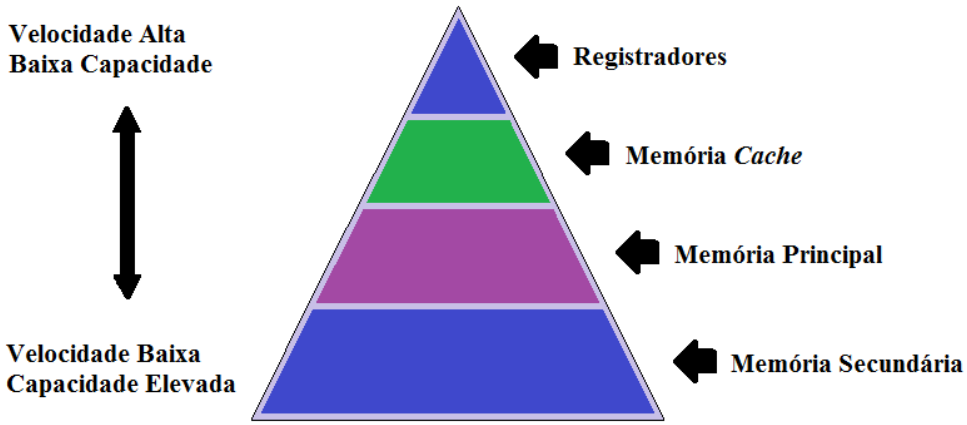
Quanto à unidade central, ela realiza ligação entre o local onde os dados estão armazenados e o local que serão processados. Os componentes para armazenagem são categorizados por Velloso (2011):

- Registradores - servem como endereçamento para os operadores presentes em cada operação, além de outros propósitos especiais. São células de memória, em número limitado com capacidade de 8, 16, 32 e 64 bits cada;
- *Cache* ou memória *cache* - é um bloco de memória para o armazenamento temporário de dados que possuem grande probabilidade de serem utilizados novamente, ou melhor, uma área de armazenamento temporária onde os dados frequentemente utilizados são armazenados para acesso rápido;
- Memória principal - é a memória que o processador pode endereçar diretamente, ela fornece uma ponte para as memórias secundárias e

contém informação necessária para o processamento, como enfileirar as instruções correntes; e

- Memórias secundárias - não podem ser endereçadas diretamente pelo processador e não são especificamente necessárias para a operação do computador, nela é possível armazenar os dados permanentemente. Este nível é composto por discos rígidos, CDs, DVDs e disquetes.

Figura 7 - Hierarquia de Memórias.



Fonte: Filho (2013).

Essas categorias formam uma hierarquia das memórias (Figura 7), em cada nível é possível resolver um problema específico, por exemplo, os registradores e memória *cache* trabalham em uma velocidade alta, já a memória principal e memória secundária possuem como característica uma capacidade maior. Com a soma do trabalho de todos os níveis, pode ser sanada a necessidade de resolução dos problemas como um todo.

Além da arquitetura da CPU possuir uma hierarquia de memória, Ruas (2008) relata que ela pode ser classificada em CISC (*Complex Instructions Set Computer*) ou RISC (*Reduce Instruction Set Computer*) a partir do número de instruções de processamento que a identifica.

Acrescentando, Ruas (2008) explica que CISC são processadores com diversas instruções, resultando em um processamento prolongado, pois para as instruções serem executadas demandam de vários ciclos. Já as instruções RISC são como processadores que possuem um conjunto de instruções reduzidas, tornando a busca, a

decodificação e a execução mais ágeis, e suas instruções necessitam de um ou dois ciclos para executarem, tal que seu desempenho é maior que o da arquitetura CISC.

Existe também a CRISC (*Complex And Reduced Instructions Set Computer*) que usa a técnica de execução dinâmica para converter as instruções CISC em RISC, acelerando o processo (RUAS, 2008).

Outra arquitetura que atua em conjunto com a CPU, é a GPU, um processador inicialmente dedicado apenas para a renderização de gráficos em tempo real.

2.1.2 GPU

Historicamente, em concordância com Ikeda (2011), a GPU teve quatro gerações e no ano corrente deste trabalho está na quinta geração. Ao longo destas houve três grandes empresas no mercado: ATI, NVIDIA e 3Dfx. A primeira geração se trata de placas de vídeos sem qualquer processamento vinculado.

Na segunda geração, em 1999, com a introdução do conceito de divisão do processador em vários estágios gráficos, pela empresa NVIDIA, se deu o surgimento a primeira GPU, o que acarretou na queda da 3Dfx e a NVIDIA passa a ter a liderança no mercado. Ainda na segunda geração a NVIDIA adquiri a 3Dfx e passa a ter apenas a ATI como concorrente, ambas lançam novas versões cujo a GPU assume funcionalidades que anteriormente eram exclusivas da CPU. Contudo as placas ainda não eram programáveis, apenas mais configuráveis (IKEDA, 2011).

Ikeda (2011) também relata que na terceira geração, em 2001, a NVIDIA lançou a primeira versão programável do mercado. Após esse avanço foram sendo lançadas, pelas duas empresas concorrentes, novas versões com melhorias. Nessa geração a GPU é capaz de executar aplicações, onde as funções limitavam-se por 128 instruções e até 96 parâmetros. Para a implementação, era necessária a linguagem de montagem da GPU, por não haver uma linguagem compatível.

Na quarta geração os lançamentos já suportavam milhares de instruções e pela GPU se portar como uma ferramenta auxiliar ao processamento, culminou em abrir caminho para a implementação de propósito geral. Este recebe o nome de GPU

Computing ou GPGPU (*General-Purpose computation on GPU*), ele visa explorar as vantagens das placas gráficas com finalidade de executar intenso fluxo de cálculos altamente paralelizáveis abrangendo de modo geral os processos dos computadores (IKEDA, 2011).

A quinta e atual geração, iniciou no fim de 2006, com o lançamento da nova linha da NVIDIA, que suporta o OpenCL, uma plataforma aberta, compatível com as novas tecnologias das empresas líder do mercado e trabalha com programação paralela.

2.1.2.1 Arquitetura GPU

As GPUs são classificadas como dispositivos aceleradores que operam em parceria com as CPUs. Elas são compostas de núcleos simples que processam o mesmo código através de milhares de *threads* paralelamente (STRINGHINI ET AL., 2012). A classificação de Flynn que melhor se adapta a ela é a SIMD, por trabalhar com muitos dados e instruções simples.

Stringhini et al. (2012) também menciona que a hierarquia de memória das GPUs possui memória global que pode ser acessada por todas as *threads*, contudo, as mais modernas já contam com dois níveis de memória *cache*, *caches* de nível 1 e de nível 2.

Já em relação aos fabricantes que se destacam no mercado atualmente, se encontram a NVIDIA e a AMD que podem operar juntamente com as fabricantes de CPUs Intel e AMD (STRINGHINI ET AL., 2012).

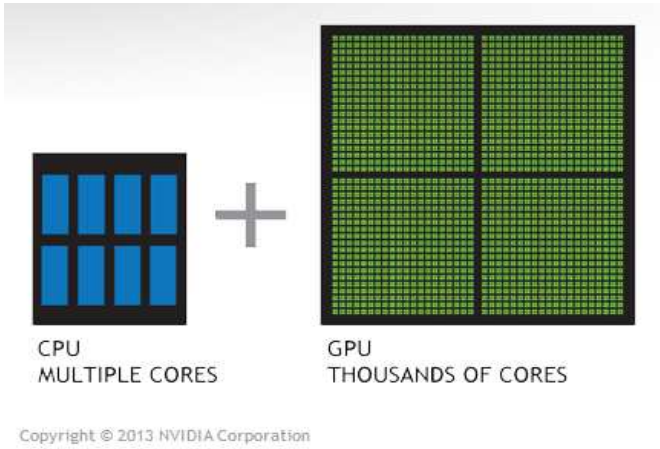
2.1.3 GPU x CPU

A GPU evoluiu e agora possui poder computacional e grande largura de barramento de memória. Quanto a CPU, mesmo com seu poder de cálculo dos processadores gráficos apresentando uma melhora substancial, ela está limitada por problemas físicos e de consumo de energia.

Em adição, Ikeda (2011) salienta a diferença da arquitetura dos processadores da CPU e da GPU:

> “Enquanto as CPUs modernas possuem no máximo de 4 ou 8 núcleos, uma GPU é formada por centenas; desde o princípio são especializadas em tarefas de computação intensiva e altamente paralelas.”

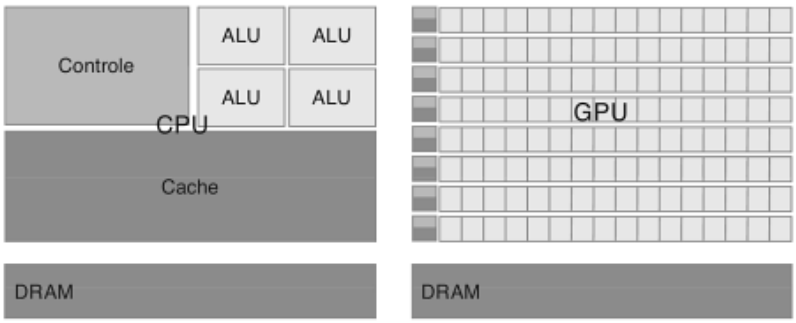
Figura 8 - Núcleos CPU e GPU.



Fonte: NVIDIA (2013).

Na Figura 9, é possível notar a diferença crucial que existe entre o projeto de uma CPU e uma GPU (IKEDA, 2011). Na arquitetura da CPU encontram-se poucas unidades para o processamento dos cálculos, ALU. À medida que a GPU possui centenas, permitindo a esta uma capacidade maior para execução de cálculos paralelamente.

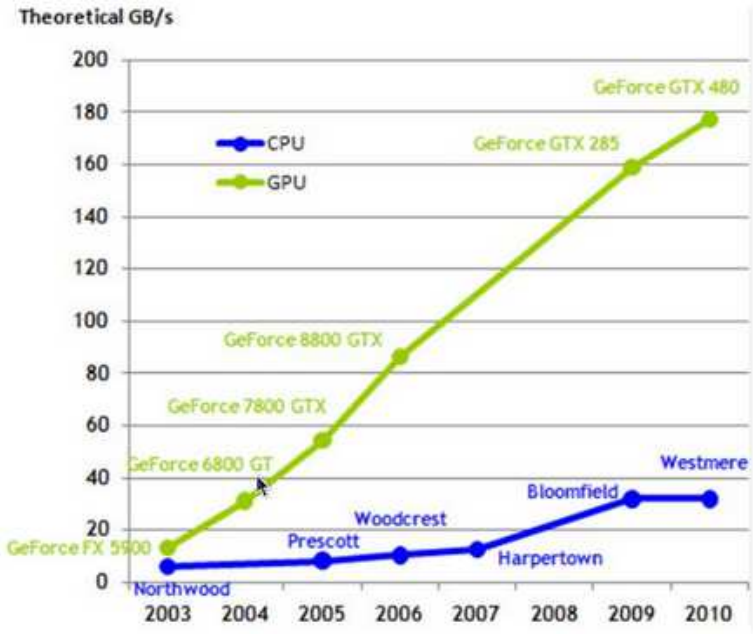
Figura 9 - CPUs e GPUs têm filosofias de projeto diferentes.



Fonte: Kirk e Hwu (2011).

De mais a mais, Tsuda (2012) evidencia que houve uma evolução da CPU para uma arquitetura com múltiplos núcleos, e no mesmo período houve também a evolução do poder de processamento e largura de banda de memória das GPUs (Figuras 10), sustentada pela alta demanda do mercado por gráficos tridimensionais interativos em alta resolução e processamento em tempo real.

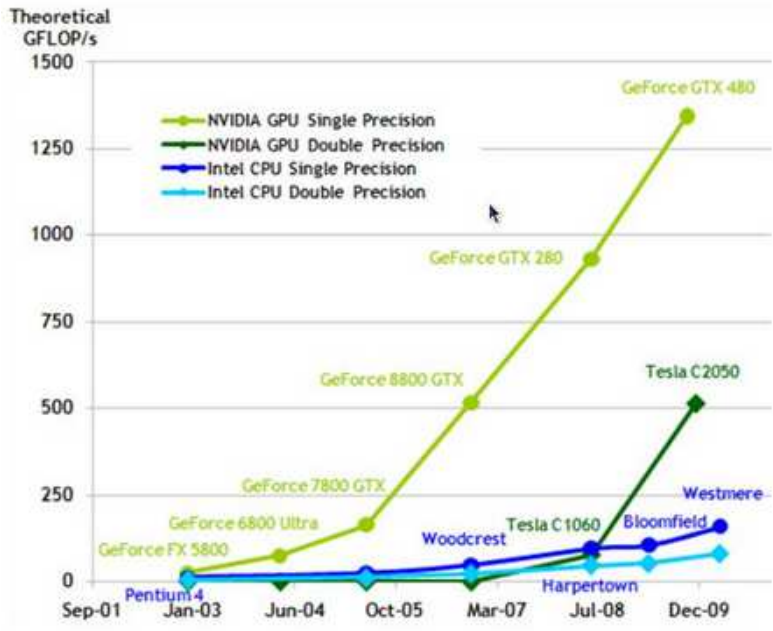
Figura 10 - Evolução da largura de banda entre o processador e a memória das GPUs e CPUs.



Fonte: NVIDIA (2013).

Somando-se a isto, verifica-se na Figura 11 o ganho na quantidade de cálculos feitos for segundo na CPU e GPU por meio de precisão numérica sem o uso de numerais depois da vírgula (*Single Precision*) e de precisão numérica com o uso de numerais depois da vírgula (*Double Precision*).

Figura 11- Evolução da capacidade de processamento em ponto flutuante entre o processador e a memória das GPUs e CPUs.



Fonte: NVIDIA (2013).

A criação dessa arquitetura paralela permitiu o desenvolvimento de técnicas que possibilitam o uso das GPUs como co-processador, auxiliando a CPU na execução de programas, resultando em um ganho de desempenho em grande parte dos casos (TSUDA, 2012).

Além disso, Ikeda (2011) compara diferentes abordagens e desafios entre CPU e GPU:

- Objetivo: para a CPU, seu núcleo foi desenvolvido para executar com maior velocidade uma única *thread* composta de instruções sequenciais. Já para a GPU, milhares de *threads* são executadas paralelamente, com mais rapidez;
- Transistores: para CPU é utilizado os transistores com intuito de aprimorar o desempenho de execução em uma sequência de tarefas. Para GPU, a especialidade é executar milhares de instruções paralelas e em sua maior

parte o trabalho se faz para o processamento de dado em si, ao invés de ocupar com o controle de fluxo;

- *Cache*: a CPU têm grandes *caches*, no qual o acesso é realizado de maneira aleatória. Essa estrutura é responsável por acelerar a execução de alguns comandos e reduzir a latência de memória, mesmo consumindo muito energia. Na GPU encontra os controladores de memória para vários canais e um *cache* pequeno e rápido, facilitando uma largura de banda ser maior e o processamento de milhares de *threads* concomitantemente; e
- Multiprocessamento: a CPU é apta para executar 1 ou 2 *threads* por núcleo, e a alternância de uma *thread* para a outra leva à centenas de ciclos, à medida que a GPU pode manter até 1024 por multiprocessador existente, bem como fazer trocas de *thread* a cada ciclo.

Logo, como exposto por Stringhini et al. (2012), com o constante crescimento de ambas as arquiteturas fez surgir arquiteturas heterogêneas. Essas utilizam núcleos mais simples, que executam no modelo SIMD, em companhia com certa quantia de núcleos mais sofisticados, e são caracterizadas como computação heterogênea.

2.2 Computação Heterogênea

Consoante com os estudos de Stringhini et al. (2012), o termo “computação heterogênea” tem sido usual para designar o uso de diferentes arquiteturas em um ambiente computacional, com a finalidade de conseguir melhorias no desempenho da aplicação que será processada nesse meio.

Além desse fato, Dongarra (2009, *apud* Stringhini et al., 2012) recomenda utilizar para contextualizar as arquiteturas heterogêneas, uma taxonomia para plataformas heterogêneas. Na taxonomia sugerida inclui máquinas paralelas e sistemas distribuídos e compreende as decorrentes categorias em ordem crescente de heterogeneidade e complexidade:

- Sistemas heterogêneos projetados por fabricantes específicos;
- Computadores heterogêneos que utilizam sistemas distribuídos;
- Redes locais de computadores;
- Redes globais de computadores em um mesmo nível organizacional; e
- Redes globais de computadores com propósitos gerais.

Na computação heterogênea também se destaca o OpenCL, sua linguagem e a maneira que organiza a execução das tarefas podem ser utilizados como uma camada de abstração ao *hardware* heterogêneo. Desse modo, um programa OpenCL tem o objetivo de usufruir de todos os dispositivos da máquina, assim como processadores *multicore,* GPUs, entre outros (STRINGHINI ET AL., 2012).

2.2.1 OpenCL

Ansoni (2010) afirma que o desenvolvimento inicial do OpenCL foi realizado pela Apple com a participação de fabricantes de processadores. E no ano de 2008 foi criado o Khronos *Compute Working Group* com integrantes de empresas fabricantes de CPU, GPU e *software*. Munshi et al. (2011) completa dizendo que o OpenCL é um padrão da indústria, pois houve uma colaboração entre fornecedores de *software*, designers de sistemas de computador e fabricantes de microprocessadores.

Esta plataforma propicia a escrita de programas que atuem tanto em CPUs, GPUs, quanto em processadores *multicore*, pois foi desenvolvido para satisfazer plataformas heterogêneas (ANSONI, 2010). Acrescentando, Tsuda (2012) relata que ela visa padronizar a programação paralela em diversos processadores modernos. Ela também evidencia abstrações que definem funcionalidades que são executadas por GPUs, CPUs, processadores digitais de sinais, entre outros (TSUDA, 2012). Essas funcionalidades são desenvolvidas utilizando a linguagem C e recebem o nome de núcleos ou *kernel*. O OpenCL cria e gerencia suas *threads* com base nesses núcleos (ANSONI, 2010).

3 DESENVOLVIMENTO

O desenvolvimento deu-se em duas etapas. A primeira refere-se à codificação da multiplicação matricial e a segunda sobre a preparação para a coleta dos valores de tempo de execução.

3.1 CÓDIGO DE MULTIPLICAÇÃO MATRICIAL

Foi escolhido utilizar a multiplicação matricial para submeter o trabalho das arquiteturas citadas no Capítulo 2.1., a partir dela puderam ser analisadas as execuções com ordens de tamanhos diferentes, permitindo uma explanação completa dos resultados gerados.

O algoritmo dessa multiplicação está representado na Figura 12 em pseudocódigo. O código da linha número 1 percorre as variáveis de cada linha da matriz A e na linha 2 é percorrido as variáveis de cada coluna da matriz B. Na terceira linha percorre-se cada posição das matrizes efetuando a multiplicação entre elas e armazenando o resultado na matriz C.

Figura 12 - Algoritmo de Multiplicação Matricial em Pseudocódigo.

```
1 Para (linha = 0; linha < dimensao da matriz; linha recebe +1) {
2     Para (coluna = 0; coluna < dimensao; coluna recebe +1) {
3         Para (posição = 0; posição < dimensao; posição recebe +1) {
4             matriz C[linha][coluna] += (matriz A[linha][posição]
5                 * matriz B[posição][coluna]);
6         }
7     }
8 }
```

Fonte: Produção do próprio autor.

Conseguinte o código foi transcrito para C++, passando a ter como diferencial a substituição do nome das variáveis *linha, coluna* e *posição* da Figura 12 para *i*, *j* e *x* da Figura 13 respectivamente.

Figura 13 - Algoritmo de Multiplicação Matricial em C++ utilizando Matrizes.

```
for (int i = 0; i < dimensao; i++) {
    for (int j = 0; j < dimensao; j++) {
        for (int x = 0; x < dimensao; x++) {
            matriz_C[i][j] = matriz_C[i][j] +
                (matriz_A[i][x] * matriz_B[x][j]);
        }
    }
}
```

Fonte: Produção do próprio autor.

Posteriormente foi realizada uma alteração no código, utilizando vetores ao invés de matrizes. Essa mudança foi efetuada devido aos vetores serem considerados estruturas de dados mais simples em relação à escrita e leitura do código, suas variáveis podem armazenar vários valores na memória, o acesso a essas variáveis pode ser feito diretamente ou por meio de índices e seu processamento é executado serialmente, tornando ágil o resultado final, intuito à obtenção de respostas rápidas (SCHILDT, 1996).

O novo algoritmo de cálculo, como observado na Figura 14, continua utilizando três estruturas de repetição que processam a multiplicação de matrizes, entretanto foram adicionadas variáveis temporárias para auxiliar na execução do processo por meio de vetores.

Figura 14 - Algoritmo de Multiplicação Matricial em C++ utilizando Vetores.

```
1  int contC = 0;
2  for (int i = 0; i < total; i+= dimensao) {
3      int tempI = i;
4      for (int j = 0; j < dimensao; j++) {
5          int temp = 0;
6          int tempJ = j;
7          tempI = i;
8          for (int x = 0; x < dimensao; x++) {
9              temp += matriz_A [tempI] * matriz_B[tempJ];
10             tempI++;
11             tempJ += dimensao;
12         }
13         matriz_C[contC] = temp;
14         contC++;
15     }
16 }
```

Fonte: Produção do próprio autor.

Logo após, o código foi reproduzido para a versão final em OpenCL, plataforma citada no Capítulo 2.2.1., cujo núcleo de cálculo é chamado de *kernel*. Este foi implementado e recebeu o nome de “multiplicacaomatricial”, como visto na linha 1 da Figura 15. Nas linhas 2, 3, e 4 estão as variáveis que recebem o tamanho da dimensão das matrizes; nas linhas 5, 6, e 7 estão os ponteiros que indicam onde os dados estão localizados; nas linhas 10 e 11 é selecionado qual o valor será buscado na memória principal através dos ponteiros passados por parâmetro, visto que o conjunto de funções não vem junto com os dados, são tratados separadamente; na linha 13 é verificado se esses valores não são inválidos; da linha 16 à 18 é executado a multiplicação das matrizes em uma única estrutura de repetição.

Figura 15 - Algoritmo de Multiplicação Matricial em OpenCL.

```
1  __kernel void multiplicacaomatricial(
2       const int Mdim,
3       const int Ndim,
4       const int Pdim,
5       __global float* A,
6       __global float* B,
7       __global float* C)
8  {
9       int k;
10      int i = get_global_id(0);
11      int j = get_global_id(1);
12      float tmp;
13      if( (i < Ndim) && (j <Mdim))
14      {
15         tmp = 0.0;
16         for(k=0;k<Pdim;k++)
17            tmp += A[i*Ndim+k] * B[k*Pdim+j];
18         C[i*Ndim+j] = tmp;
19      }
20 }
```

Fonte: Produção do próprio autor.

Na Figura 15 também é possível observar a ação do modelo de arquitetura SIMD, pois da linha 5 à 7 foram armazenados múltiplos dados e da linha 16 à 18 está a estrutura de repetição com uma instrução simples.

Após a elaboração do código de multiplicação matricial foi desenvolvido o método de execução dos testes para aplicação nas arquiteturas CPU e GPU.

3.2 MÉTODO DE EXECUÇÃO

Foi definido que a execução do programa seria realizada 30 vezes para cada ordem de matriz subsequente: 16, 32, 64, 128, 256, 512, 1024 e 2048. Para tornar o processo mais dinâmico ficou definido que o tamanho da matriz seria indicado na hora em que o programa fosse chamado, e não dentro do algoritmo. O Quadro 1 mostra a linha de comando necessária para executar a multiplicação matricial 16X16.

Quadro 1 - Linha de Comando de Execução da Multiplicação Matricial.

```
C:\>multiplicacaomatricial 16
```

Fonte: Produção do próprio autor.

Com base nesta linha foi produzido um script contendo o seguinte comando copiado 240 vezes: multiplicacaomatricial N >> resultadoN.txt. A letra N é substituída pela ordem da matriz, como fora testadas 8 ordens de matrizes diferentes ficaram 30 chamadas para cada.

Quando executado o comando citado acima, este fica com a responsabilidade de gravar no arquivo os resultados obtidos. Ao fim da execução do script foram gerados 8 arquivos, um para cada ordem de matriz, possuindo 30 resultados de tempo de execução.

Foi fundamental cronometrar esse tempo, pois a partir dele foram feitas as comparações e apontamento do melhor desempenho. Para calcular o tempo foi utilizada uma biblioteca chamada *timer*, observada com maior facilidade no algoritmo em C++ na Figura 17.

Figura 16 - Biblioteca *Timer* para Cronometrar o Tempo.

```
1  Timer tempo;
2  int contC = 0;
3  tempo.start();
4  for (int i = 0; i < total; i+= dimensao) {
5      int tempI = i;
6      for (int j = 0; j < dimensao; j++) {
7          int temp = 0;
8          int tempJ = j;
9          tempI = i;
10         for (int x = 0; x < dimensao; x++) {
11             temp += matriz_A [tempI] * matriz_B[tempJ];
12             tempI++;
13             tempJ += dimensao;
14         }
15         matriz_C[contC] = temp;
16         contC++;
17     }
18 }
19 tempo.stop();
20 printf ("%f ", tempo.getElapsedTimeInMicroSec ());
```

Fonte: Produção do próprio autor.

Essa biblioteca foi inicializada com o nome “tempo” na linha 1. O início da cronometragem é feito na linha 3, seguido da multiplicação matricial, e seu término ocorre na linha 19 ao final do núcleo de cálculo. Na linha 20 é impresso na tela o tempo de execução.

As atribuições de valores das matrizes não foram englobadas na contagem do tempo. Essas foram realizadas separadamente, de forma que a matriz C é inicializada com valor zero e as matrizes A e B recebem valores aleatórios através do método randômico.

Quanto aos valores adquiridos pelas matrizes A e B, foi estipulado que cada posição iria receber a importância variando de 1 à 5. O método utilizado foi que cada posição passa a ter um valor aleatório entre 0 e 4, e em seguida esse valor é somado à 1, como pode ser observado na Figura 18.

Figura 17 - Algoritmo de Atribuição de Valores Aleatórios.

```
1  for (int i = 0; i < total; i++) {
2          matriz_A[i] = (rand ()%5)+1;
3          matriz_B[i] = (rand ()%5)+1;
4          matriz_C[i] = 0;
5      }
```

Fonte: Produção do próprio autor.

Ao término da criação do método de execução foram aplicados os testes para obtenção dos dados resultantes. Com os arquivos gerados, foram enviados os resultados a um programa que calcula a média simples, método escolhido por preferência do autor. Os dados resultantes de cada ordem de matriz foram somados e divididos por 30, obtendo assim os resultados finais.

4 RESULTADOS OBTIDOS

Foi utilizado o código em C++ na CPU e o código em OpenCL tanto na CPU quanto na arquitetura GPU. Logo após, foram obtidos os resultados de tempo de execução de cada ordem de matriz para análise do desempenho das arquiteturas. Na Tabela 2 é possível observar as médias simples dos tempos de execução, em segundos:

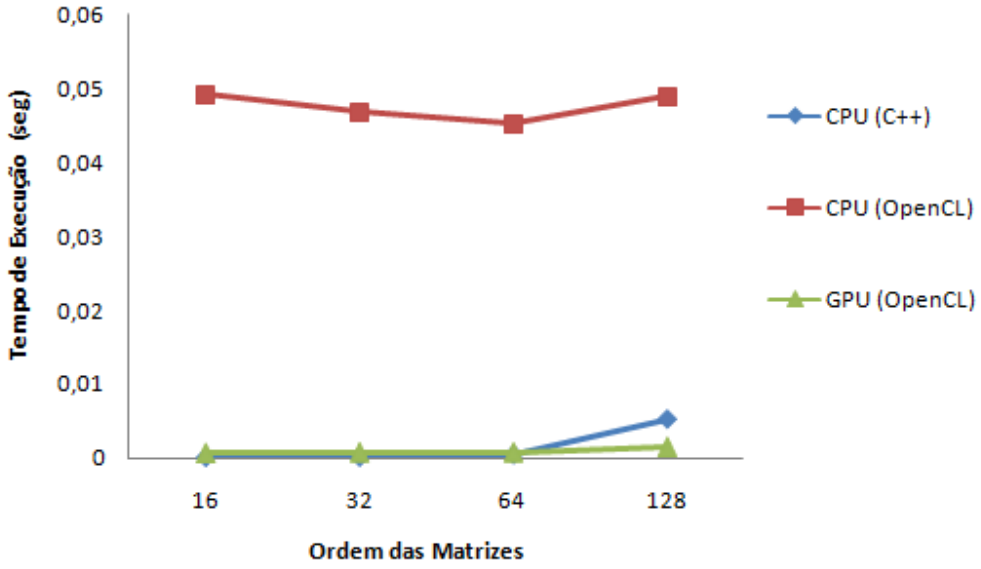
Tabela 2 - Resultados obtidos para cada arquitetura testada.

Ordem das Matrizes Quadradas	C++	OpenCL	
	CPU	CPU	GPU
16	0,000008	0,0492	0,0007
32	0,000061	0,0468	0,0008
64	0,0004	0,0453	0,0008
128	0,0052	0,049	0,0015
256	0,0454	0,0754	0,0081
512	0,4204	0,3333	0,0724
1024	16,6567	9,3116	0,679521
2048	141,3349	154,862	7,238521

Fonte: Produção do próprio autor.

Para uma análise eficaz dos dados colhidos nos testes, foi criado um gráfico das quatro primeiras ordens de matriz *versus* o tempo de execução que as arquiteturas CPU e GPU tiveram ao utilizar as linguagens C++ e OpenCL.

Gráfico 1 - Tempos de Execuções pelas Ordens de Matrizes 16, 32, 64 e 128.



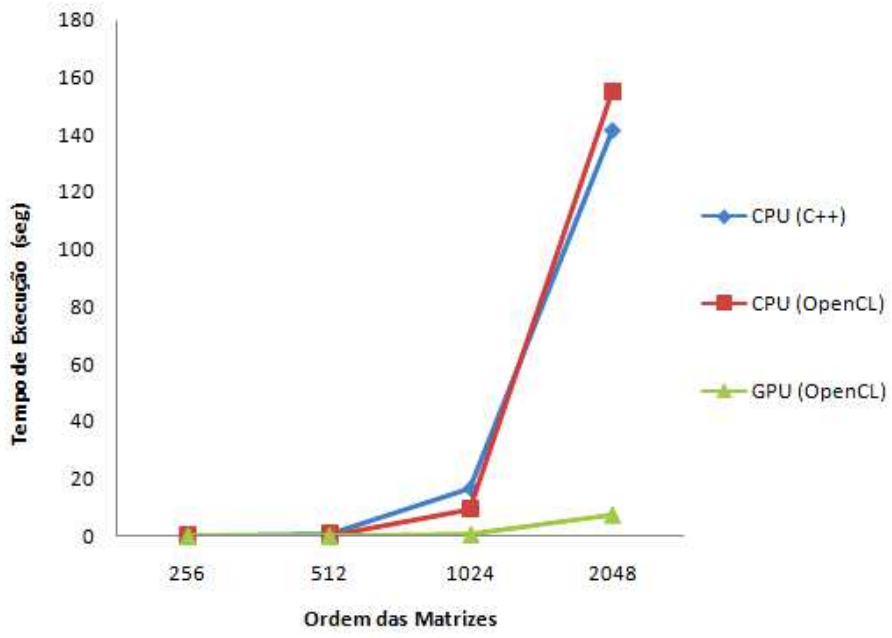
Fonte: Produção do próprio autor.

No Gráfico 1 pode-se notar que a execução da CPU utilizando a linguagem em OpenCL teve os resultados mais demorados, consumindo um tempo maior. Isto se dá devido a plataforma OpenCL forçar a CPU a trabalhar com o modelo SIMD, valendo-se que ela é mais compatível com o modelo MIMD, explicado no Capítulo 2.1, não utilizando o modelo SIMD de forma completa.

Já a execução da CPU usufruindo da linguagem em C++ teve resultados aproximados aos da GPU, contudo, esta se sobressaiu na ordem de matriz 128, possuindo um tempo mais curto em sua execução.

Em seguida foi gerado um gráfico das outras quatro ordens de matrizes pelo tempo de execução. No Gráfico 2, foi possível verificar que na ordem de matriz 1024 a GPU processou mais rápido, seguida da CPU com o código em OpenCL e posteriormente da CPU com C++.

Gráfico 2 - Tempos de Execuções pelas Ordens de Matrizes 256, 512, 1024 e 2048.



Fonte: Produção do próprio autor.

Já para a ordem de matriz 2048, a maior e mais exaustiva execução dos testes realizados, a GPU teve um tempo de execução expressivamente menor quando comparada aos resultados da CPU, tanto executado em C++ quanto em OpenCL. O desempenho mais considerável se deu porque essa arquitetura é composta de muitos núcleos trabalhando para a execução dos cálculos, ao passo que a CPU não possui essa vantagem.

Visto que alguns valores são muito baixos para serem visualizados e comparados através dos gráficos, foi produzida uma tabela com os resultados da divisão dos tempos de execução, cujo intuito é comparar e reconhecer facilmente a estrutura que alcançou o desempenho mais significativo.

A Tabela 3 é analisada da seguinte forma, se o valor obtido na divisão for abaixo de zero entende-se que a primeira arquitetura da divisão teve o desempenho mais relevante, caso contrário, se o valor for maior que zero, a segunda arquitetura que possuirá o desempenho satisfatório.

Tabela 3 - Resultado da divisão entre os tempos de execução das arquiteturas.

Ordem das Matrizes Quadradas	CPU (C++) / CPU (OpenCL)	CPU (C++) / GPU	CPU (OpenCL) / GPU
16	0,0002	0,0114	70,2857
32	0,0013	0,0763	58,5000
64	0,0088	0,5000	56,6250
128	0,1061	3,4667	32,6667
256	0,6021	5,6049	9,3086
512	1,2613	5,8066	4,6036
1024	1,7888	24,5124	13,7032
2048	0,9127	19,5254	21,3942

Fonte: Produção do próprio autor.

A primeira coluna da tabela corresponde à divisão dos tempos de execução na CPU que utiliza C++ pela que usa a plataforma OpenCL. Nessa a CPU (OpenCL) obteve um desempenho expressivo apenas na execução das ordens de matrizes 512 e 1024. Ressaltando que a CPU trabalha melhor no ambiente MIMD, cuja linguagem em C++ também é mais hábil.

Na coluna seguinte a GPU não ultrapassou a velocidade de execução da CPU (C++) nas ordens de matriz compostas de valores pequenos, contudo, seu desempenho aumentou consideravelmente quanto às ordens com um número maior de dados. Este ocorreu devido a CPU trabalhar de forma eficiente quando não são ultrapassados seus limites físicos, à medida que a GPU tem um melhor potencial para trabalhar com eficiência com uma grande quantidade de cálculos.

E na ultima coluna todos os valores foram acima de zero, interpretando assim que a GPU se sobressaiu.

5 CONCLUSÃO

O principal foco deste trabalho foi realizar um levantamento de dados estatístico, certificando seu impacto e comprovando a autenticidade dos testes. O comparativo realizado entre o desempenho de processamento em máquinas com arquiteturas CPU e GPU obteve valores relevantes, entendendo-se que as aplicações podem usufruir de cada unidade de processamento de forma distinta.

Um ponto pertinente à arquitetura CPU, foi que ao utilizar o algoritmo em OpenCL, seu tempo de execução atuou de modo lento, mesmo desfrutando de todas as *threads*. Esse evento se deu pelo fato desta plataforma enviar primeiro a instrução e posteriormente os dados, se tornando pouco viável ao processamento da arquitetura em questão, a deixando compassada.

Referente à plataforma OpenCL, é notório que a mesma usufruiu de forma completa do poder computacional da GPU. Ela também faz com que a programação na GPU progrida pela facilidade que é atribuída, pois se trata de uma extensão da linguagem C, habitual aos programadores.

Através dos resultados obtidos também pode ser visto que no teste realizado com a maior ordem de matriz, 2048, a GPU alcançou um desempenho elevado, com um ganho de 95% no tempo de execução quando comparado ao processamento da CPU. Isto ocorre porque ela trabalha com rapidez e de forma efetiva com grandes quantidades de cálculos.

Ademais, o uso da GPU é de grande importância às aplicações científicas que visam usufruir do seu poder de paralelização, acelerando os cálculos aritméticos substancialmente. Por outro lado, a CPU demonstrou ser oportuna ao emprego de tarefas que se adéquem aos seus limites.

Para perspectivas de trabalhos futuros é de grande relevância realizar testes exclusivamente com os processadores de ambas as arquiteturas, valendo-se que os testes realizados neste se deu em máquinas que além de executarem os comandos solicitados também realizavam tarefas concorrentes.

Outra perspectiva é quanto à união dos *hardwares* da CPU e GPU em um

mesmo encapsulamento, por consequência do aumento da participação de processadores, visando aproveitar a capacidade de processamento da GPU quanto à execução de algoritmos mais complexos, aumentando seu ganho quantitativo e seu desempenho.

6 REFERÊNCIAS

AGULLO, E.; AUGONNET, C.; DONGARRA, J.; FAVERGE, M.; LANGOU, J.; LTAIEF, H.; TOMOV, S. LU Factorization for Accelerator-based Systems, 2011.

ANSONI, J. L. Resolução de um Problema Térmico Inverso Utilizando Processamento paralelo em Arquiteturas de Memória Compartilhada. Escola de Engenharia de São Carlos da Universidade de São Paulo, 2010.

BABOULIN, M.; DONFACK, S.; DONGARRA, J.; GRIGORI, L.; RÉMY, A.; TOMOV, S. A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines. International Conference on Computational Science, 2012.

BONAT, Debora. Metodologia da Pesquisa. 3ª edição – Curitiba: IESDE Brasil S.S., 2009.

D'AZEVEDO, E.; HILL, J. C. Parallel LU Factorization on GPU cluster. International Conference on Computational Science, 2012.

ENGHOLM, Hélio Jr. Apostila Sistema Operacional. Cyber Tech CSE Informática. Disponível em:<http://www.cybertechcse.com.br/sistOperacionais/Frame.htm> Acesso em: 01 de julho de 2013.

FILHO, Raimundo de G. N. Hierarquia de Memórias. Caderno de Textos da disciplina Introdução ao Computação. Departamento de Informática - UFPB, 2012. Disponível em:<http://www.di.ufpb.br/raimundo/Hierarquia/Hierarquia.html> Acesso em: 11 de junho de 2013.

GASTER, B. R.; HOWES, L.; KAELI, D.; MISTRY, P.; SCHAA, D. Heterogeneous Computing with OpenCL, 2012.

GEBALI, Fayez. Algorithms and parallel computing. United States of America: Wiley series on parallel and distributed computing, 2011.

HERLIHY, M.; SHAVIT, N. The Art of Multiprocessor Programming. United States of America: Elsevier, 2008.

HORTON, M.; TOMOV, S.; DONGARRA J. A Class of Hybrid LAPACK Algorithms for Multicore and GPU Architectures, 2011.

IKEDA, P. A. Um Estudo do Uso Eficiente de Programas em Placas Gráficas. Instituto de Matemática e Estatística da Universidade de São Paulo, 2011.

INO, F.; MATSUI, M.; GODA, K.; HAGIHARA, K. Performance Study of LU Decomposition on the Programmable GPU. Graduate School of Information Science and Technology, Osaka University. Japão, 2012.

JIA, Y.; LUSZCZEK, P.; DONGARRA, J. Multi-GPU Implementation of LU Factorization. International Conference on Computational Science, 2012.

KIRK, David B., HWU, Wen-Mei W. Programando para processadores paralelos: uma abordagem prática à programação de GPU. Tradução de Daniel Vieira. Rio de Janeiro: Elsevier, 2011.

LEE, V. W.; KIM, C.; CHHUGANI, J.; DEISHER, M.; KIM, D.; NGUYEN, A. D.; SATISH, N.; SMELYANSKIY, M.; CHENNUPATY, S.; HAMMARLUND, P.; SINGHAL, R.; DUBEY, P. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU, 2010.

MENEZES, Paulo. Matemática discreta para computação e informática. 3ª edição – Dados eletrônicos – Porto Alegre: Bookman, 2010.

MUNSHI, A.; GASTER, B. R.; MATTSON, T. G.; FUNG, J.; GINSBURG, D. OpenCL Programming Guide, 2012.

NVIDIA, Corporation. Computação de Alta Performance – O que é computação com GPU?. San Tomas Expressway Santa Clara. USA, 2013.

OLIVEIRA, Rogério Amigo de. Informática. 2ª edição – Rio de Janeiro: Elsevier, 2008.

PITANGA, Marcos. Construindo Supercomputadores com Linux. 3ª edição – Rio de Janeiro: Brasport, 2008.

RAINER, R. Kelly; CEGIELSKI, Casey G. Introdução a sistemas de informação. 3ª edição – Rio de Janeiro: Elsevier, 2012.

RUAS, Jorge. Informática para concursos: teoria e mais de 450 questões. 6ª edição – Rio de Janeiro: Elsevier, 2008.

SCHILDT, H. C Completo e Total. 3ª edição – São Paulo: Editora Makron Books, São Paulo, 1996.

SCHMEISSER M, HEISEN BC, LUETTICH M, BUSCHE B, HAUER F, KOSKE T, KNAUBER KH, STARK H. Parallel, distributed and GPU computing technologies in single-particle electron microscopy - Acta Crystallogr. D Biol. Crystallogr, 2009.

SIEWERT, Sam. Educação baseada em nuvem, Parte 2: Computação de alto desempenho para a educação. IBM, 2012. Disponível em:<

http://www.ibm.com/developerworks/br/industry/library/ind-cloud-e-learning2/> Acesso em: 19 de abril de 2013.

STALLINGS, William. Arquitetura e Organização de Computadores. 8ª edição – São Paulo: Prentice-Hall Brasil, 2010.

STRINGHINI, D., GONÇALVEZ, R. A., GOLDMAN, A. Introdução à computação Heterogênea, 2012.

STRZODKA, R.; DOGGETT, M.; KOLB. A. Scientific computation for simulations on programmable graphics hardware, 2005.

TOMOV, S.; NATH, R.; LTAIEF, H.; DONGARRA J. Dense Linear Algebra Solvers for Multicore with GPU Accelerators. Department of Electrical Engineering and Computer Science, University of Tennessee, 2010.

TSUDA, F. Utilização de Técnicas de GPGPU em Sistemas de Vídeo-Avatar. Escola Politécnica da Universidade de São Paulo, 2012.

VELLOSO, Fernando de Castro. Informática – conceitos básicos. 8ª edição – Rio de Janeiro: Elsevier, 2011.

YANG, Y.; XIANG, P.; MANTOR, M.; ZHOU, H. CPU – Assisted GPGPU on Fused CPU – GPU Architectures. Internacinal Symposium on High Performance Computer Architecture. HPCA, 2012.