Multiplicação de Matrizes Utilizando CUDA – Análise de Performance

Jairo Lucas , Pedro Hoppe

Universidade Federal do Espirito Santo - Laboratório de Computação de Alto Desempenho - LCAD

***Resumo* - Com a evolução das placas GPU, que passaram de simples dispositivos que ofereciam funções gráficas para complexas arquiteturas com centenas de cores e funções de uso geral, a programação paralela ganhou enorme destaque nos últimos anos, permitindo que um computador pessoal possa ter um desempenho comparável a um supercomputador da década passada. Este enorme poder computacional oferecido pelo paralelismo torna cada vez maior a demanda por soluções que assimilam o uso de paralelismo e efetivamente consigam aproveitar este poder.**

Neste cenário, o objetivo deste trabalho foi analisar o potencial de um algoritmo paralelo utilizando uma arquitetura de GPU em relação a um algoritmo serial. O problema escolhido foi a multiplicação de matrizes, e para isso, foi implementado uma versão paralelizada do algoritmo em C-CUDA e OPEN-MP , e uma versão serial em C que utiliza a somente a CPU.

Nos testes foi usada uma GPU Nvidia Tesla C2050 e uma CPU Intel Xeon, ambas detalhadas na seção 3.

Os resultados obtidos demonstram o enorme poder de processamento oferecido pela programação paralela em GPU's, chegando a oferecer um *speedup* de mais de 2000x em comparação a CPU em operações com matrizes de números float.

Termos – GPU, programação paralela, Speedup, CUDA, NVIDIA, multiplicação de matrizes, Threads, paralelismo.

1.INTRODUÇÃO

Historicamente, o aumento de poder de processamento de um processador foi focado no aumento de seu clock (número de operação que o processador consegue efetuar em um ciclo), em 2004, quando a Intel cancelou o lançamento de dois novos processadores (processadores *Tejas* e *Jayhawk*) devido o alto consumo de energia e do calor gerado [3], esse paradigma começou a mudar. Desde então os grandes fabricantes passaram a aumentar a performance do processador aumentando a quantidade de núcleos (cores) do mesmo, criando processadores com vários cores, sendo que o clock dos mesmos pouco variou deste então.

Este novo paradigma de processadores paralelos incentivou gigantes mundiais da tecnologia a investir pesado na criação e aperfeiçoamento de placas com processadores altamente paralelizáveis, que pudessem operar em conjunto com um CPU comum e oferecer centenas de processadores adicionais.

Desta forma, as GPU's (unidades de processamento gráfico), que até então eram utilizadas por suas funções gráficas, foram incrementadas com esta tecnologia com o objetivo de oferecer uma solução para um uso mais amplo.

1.1 – GPU (Unidade de Processamento Gráfico)

Segundo [2], uma GPU (*Graphics Processing Units*) é a "*uma unidade de processamento gráfica especializada em processar imagens, através de uma combinação de um processador e uma minúscula memória de alta velocidade*".

Elas existem desde a década de 90, porém evoluíram de dispositivos que ofereciam funções gráficas, para complexas arquiteturas com centenas de cores, que suportam programação em linguagem de alto nível e funções antes exclusiva da CPU.

Atualmente, estas placas são conhecidas também como GPGPU (unidades de processamento gráfico de propósito geral), visto que o uso das mesmas não é mais restrito a aplicações gráficas. Vale lembrar, que o poder de processamento de uma GPU extrapolou em muito o de uma CPU. Na figura 1 é possível visualizar a evolução do poder de processamento das GPU's Nvidia em comparação com uma CPU 's Intel ao longo dos anos.

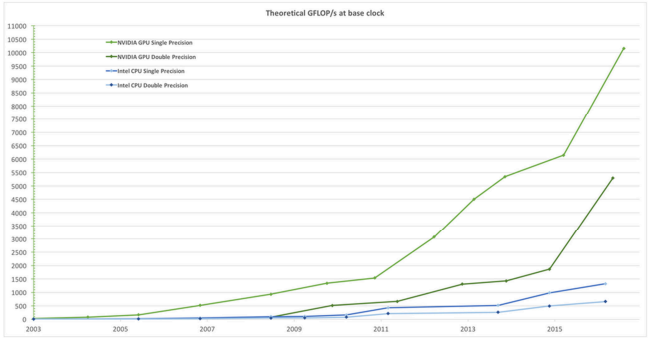

Fig. 1 – Comparativo entre GPU Nvidia e CPU Intel.[3]

Além do poder de processamento, a largura de banda da memória das GPU's também evoluiu muito nos últimos anos enquanto que nas CPU's este valor ficou praticamente estagnado. Na figura 2 é possível visualizar esta evolução[4].

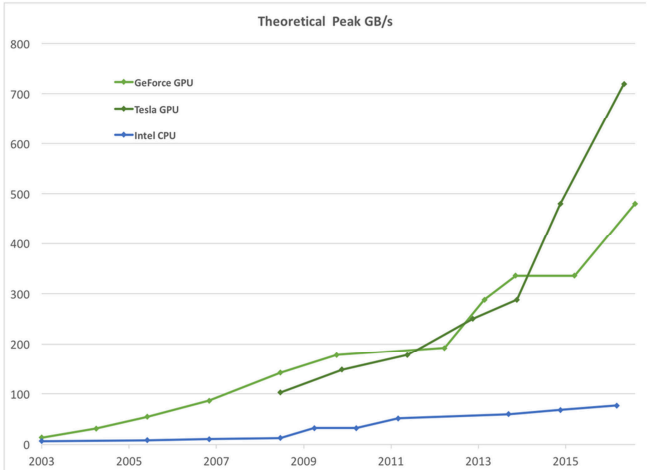

Figura 2- Largura de banda nas CPU's E GPU's [3]

1.1.2 – Arquitetura de uma GPU Nvidia

Como visto no tópico anterior, a capacidade de processamento das GPU ultrapassa bastante o poder das CPU's. O principal motivo desta eficiência no desempenho é a arquitetura das GPU's , que são dedicadas exclusivamente ao processamento de dados, não se preocupando com o controle de caches e fluxos de dados, como uma CPU. Desta forma, toda a área do processador que seria destinada a memória cache e ao controle de fluxo é ocupada por processadores. A figura 3 exibe a disposição de espaço nas duas arquiteturas.

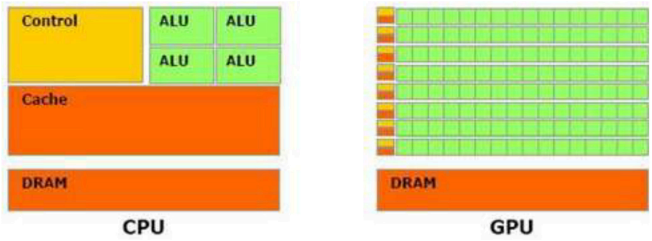

Figura 3 – Disposição das arquiteturas CPU e GPU [3].

Através dessa arquitetura, a NVIDIA criou um novo conceito na computação paralela chamado de SIMT (single-instruction multiple-thread). Esta arquitetura cria, gerencia, agenda e executa grupos de threads (32 no caso das placas Tesla) chamadas de *warp*. [4].

Para programação das placas Nvidia é utilizado a API CUDA, que veremos mais detalhadamente no próximo tópico.

1.2 – CUDA - *Computing Unified Device Architecture*

A arquitetura CUDA é um modelo de programação paralelo disponibilizado pela Nvidia para ser utilizado em suas GPU's. A mesma procura oferecer ao programador familiarizado com linguagens como o C uma baixa curva de aprendizado.

O CUDA possui três tipos de abstrações no cores: Hierarquia de grupos de Threads, memória compartilhada e barreiras de sincronização de threads. Estas abstrações permitem ao programador efetuar uma programação paralela refinada, de forma que um problema pode ser quebrado em subproblemas menores que são resolvidos em paralelo por blocos de threads, e cada subproblema pode ser quebrado em pedaços menores e resolvido de forma cooperada em paralelo por todas as threads dentro do bloco.

1.2.1 – Conceitos básicos de CUDA.

O escalonamento das threads da plataforma CUDA utiliza dois conceitos básicos : blocos e grids.

Blocos : Um bloco é a unidade básica de organização das threads e de mapeamento para o hardware. É formado por um conjunto de threads que serão executadas de forma sincronizada e terão memória compartilhada entre elas. A comunicação entre blocos é feita através da memória global, que é mais lenta que a memória compartilhada entre as threads de um bloco. Blocos podem ter uma, duas ou três dimensões.

Grids : O grid é onde estão distribuídos os blocos. O grid é a estrutura completa de distribuição das threads que executam uma função. É nele que está definido o número total de blocos e de threads que serão criados e gerenciados pela GPU para uma determinada função. Grid's podem ter uma ou duas dimensões.

A figura 4 mostra uma estrutura de blocos e grids.

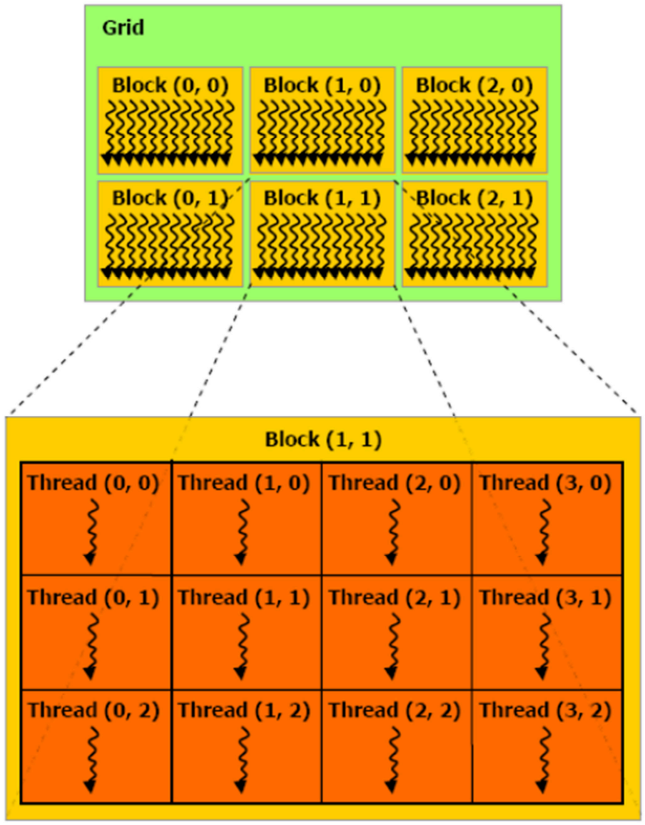

Figura 4 – Estrutura de blocos e grids [3]

As GPU's possuem vários tipos de memorias, cada uma com suas particularidades e tempos de acesso. Entender e

principalmente utilizar corretamente os diversos tipos de memórias é essencial para um bom desempenho de um algoritmo em CUDA. A hierarquia de memória das GPU's Nvidia seguem o padrão abaixo, sendo listadas da mais rápida para a mais lenta.

a) **Registradores:** É o tipo de memória mais rápido da GPU, porém é um recursos escasso, por exemplo, a GPU utilizada neste trabalho oferece apenas 32768 registradores por bloco. Os registradores ficam localizado dentro do chip e cada thread possui um conjunto privado de registradores de forma que quanto mais registradores uma thread usar, menos threads poderão existir em um bloco.

b) **Memoria compartilhada:** É compartilhada com todas as threads no mesmo bloco.

c) **Memoria local :** Cada thread possui um espaço de memória local (além dos registradores). Apesar de ser chamada de local, esta memória fica fora do microchip e tem a mesma velocidade de acesso da memória global.

d) **Global:** Memória disponível para todas as threads de cada bloco e para todos os grids. A memória global é a única maneira de threads em um grid conversarem com threads em outra. O acesso é bastante lento

Existe ainda a memória DRAM, que fica fora GPU e é responsável pela comunicação entre a CPU e a GPU. Esta memoria não faz parte da GPU e o tempo de acesso a mesma é extremamente lento.

A figura 5 mostra a organização de memória dentro da GPU.

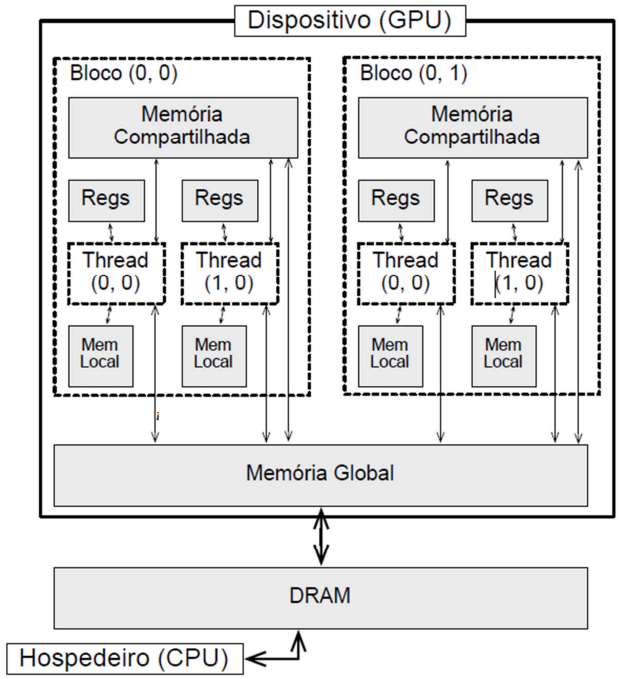

Figura 5 – organização da memoria da GPU.

A programação paralela exige algumas considerações sobre a decisão de utilizar ou não a GPU, sendo que alguns algoritmos podem resultar em um desempenho abaixo do obtido com o uso exclusivo da CPU. Um ponto que deve ser observado com cuidado é a transferência de dados entre a CPU e a GPU e o tipo de operação que será aplicada sobre estes dados. A transferência de dados sempre será um enorme gargalo de tempo , e dependendo do tipo de aplicação, este gargalo será maior que o tempo ganho com o processamento dos dados na GPU. Tipicamente, o ganho de performance de rodar um programa na GPU é maior quando existem processamento complexo e repetitivo sobre o mesmo grupo de dados, como por exemplo, na multiplicação de matrizes. Neste tipo de tarefa, os dados seriam transferidos apenas uma vez, e haveria várias operações de soma e adição sobre os mesmos.

3.METODOLOGIA

Neste trabalho foram utilizadas as seguintes metodologias e ferramentas.

3.1 – Software

- Sistema operacional Linux Ubuntu
- Framework C-CUDA
- Biblioteca Open MP
- Linguagem de programação C

3.2– Hardware

GPU

Nvidia – Modelo "Tesla C2050"

Versão do Driver	: 7.5 / 7.5
Compatibilidade	: 2.0
Total memoria Global	: 2687 MBytes
14 SM com 32 Cores cada	: 448 CUDA Cores
Clock dos processadores	: 1147 MHz (1.15 GHz)
Memory Bus Width	: 384-bit
Total de memória	: 65536 bytes
Registradores por bloco	: 32768
Tamanho do Warp	: 32
Máximo de threads/bloco	: 1024
Máximo de threads/SP	: 1536

CPU

Intel Xeon

Número de cores	: 4 cores
Clock dos processadores	: 2.1 Ghz
Memória Ram	: 12 gb

Para apurar o desempenho da GPU em relação a CPU, foram criados 3 versões do algoritmo para a tarefa de multiplicação de matrizes. Uma versão utilizando programação serial, uma versão utilizando a biblioteca OPEN MP e outra versão utilizando somente CUDA.

Cada versão do algoritmo multiplica matrizes quadradas de tamanhos 128x 128, 256x256, 512x512 e 1024x1024. Os algoritmos estão programados para repetir a multiplicação 1x, 100x e 1000 vezes. Em cada uma das rodadas, o tempo de cada um é apurado utilizando funções do C e do Cuda.

Foi avaliado o comportamento de cada algoritmo em relação ao tamanho das matrizes, e o desempenho em relação

aos outros algoritmos. A métrica de comparação utilizada para avaliar o desempenho entre os algoritmos é o *speedup,* calculado conforme a formulara da figura 6.

$$S = \frac{T_{wo}}{T_w}$$

Fig. 6 Formula do calculo do Speedup.

Onde, *wo* é o tempo de execução apurado no algoritmo base, e *w* é o tempo de execução apurado no algoritmo testado.

4. Experimentos e Resultados

Inicialmente, foi apurado de forma teórica e posteriormente através de testes empíricos vários parâmetros da GPU para que a mesma pudesse oferecer a melhor performance possível.

Uma vez apurados os melhores resultados para a execução na GPU os mesmos foram fixados e utilizados para todos os experimentos.

4.1 – Experimento I

Este experimento mostra o comportamento do algoritmo sequencial utilizando float e double na multiplicação de matrizes de tamanhos diferentes. Foram executadas 100 iterações do algoritmo.

Conforme esperado, o gráfico mostra um aumento exponencial no tempo à medida que aumenta o tamanho da matriz. Enquanto uma matriz de 128x128 tem um tempo de execução de em torno de 1 segundo, a matriz de 1024 x 1024 necessita de aproximadamente 2.600 segundos para concluir o processo. Por outro lado, é possível constatar que o uso de float ou double nas matrizes, pouco afeta a performance o algoritmo quando utilizado na CPU.

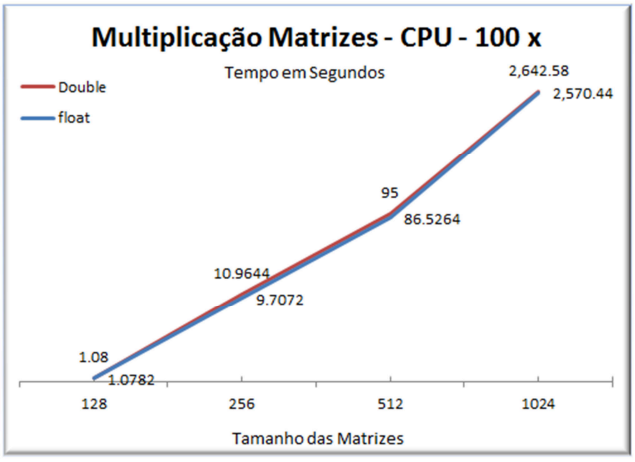

Fig. 7 - Resultados do experimento I

4.2 – Experimento II

O experimento I foi replicado utilizado agora o algoritmo de programação paralela, usando a GPU.

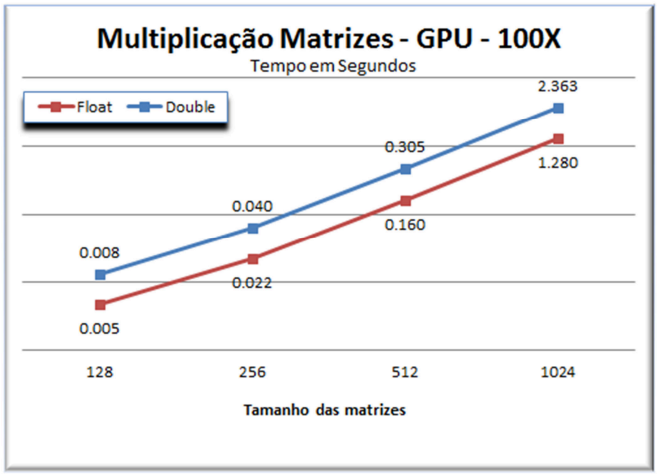

Fig. 8 - Resultado do experimento II

Pelo gráfico, podemos constatar que na execução paralela, utilizando GPU, o algoritmo executa até 2000 vezes mais rápido que o algoritmo serial. Para uma matriz de 1024x1024 o algoritmo necessita somente de 2.3 segundos para matrizes float e 1.28 segundos para matrizes do tipo float.

É possível constatar também que, diferente do comportamento do algoritmo sequencial, o uso do double tem uma grande influência no tempo de execução do algoritmo paralelo usando a GPU. A multiplicação de matrizes com valores double tem um custo aproximadamente 85% maior que matrizes com float. Este degradação no desempenho é explicada pelo fato que cada processador da GPU tem um número limitado de threads e registradores por blocos. Como o double consume o dobro de espaço de um float, as threads terão que usar mais registradores para cada elemento da matriz, e consequentemente, terão que processar menos elementos a cada bloco.

4.3 – Experimento III

O experimento anterior foi novamente replicado, desta vez aplicado ao algoritmo paralelo usando a biblioteca OpenMp para paralelizar o código na CPU. A figura 9 mostra o resultado do experimento.

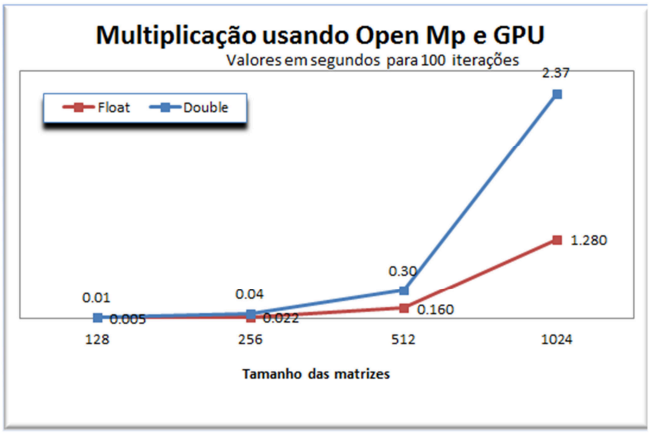

Figura 9 – Resultado experimento III

O resultado deste experimento mostra que a utilização do Open Mp para paralelizar o código entre os vários cores da CPU não traz nenhum ganho de performance em relação ao código que utiliza somente um core para interagir com a GPU.

Os tempos de execução tiveram uma diferença desprezível, a nível de milésimos de segundos. Este resultado já era esperado, visto que a paralelização entre os vários cores da CPU farão com que os mesmos concorram entre si para utilizar os mesmos recursos da GPU.

4.4 – Experimento III

Neste experimento, avaliamos o ganho de desempenho do algoritmo paralelo utilizando GPU em relação ao algoritmo serial que utiliza a CPU. O ganho de desempenho está expresso em termos de s*peedup*, como definido na seção anterior.

Este resultado vale também para o algoritmo paralelo que utiliza OpenMP, pois conforme visto no experimento III, seu desempenho é igual ao código paralelo que não utiliza esta biblioteca.

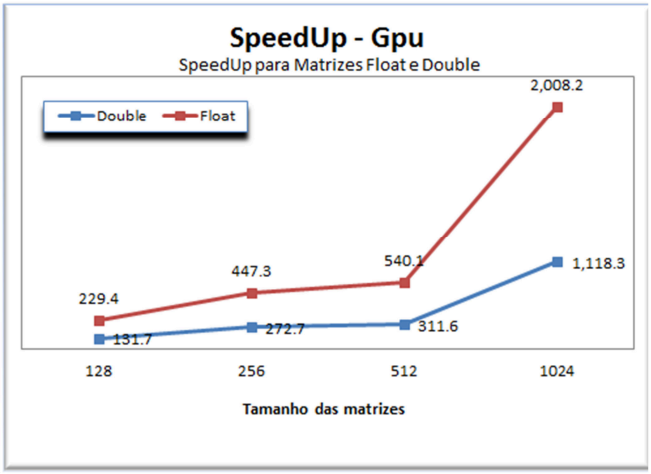

Fig.9. Resultado do experimento III

Pelo gráfico, podemos constatar que o ganho de performance oferecido pela GPU é bem maior nas matrizes maiores, chegando a um *speedup* de mais de 2000 nas matrizes com tamanho de 1024.

Também é possível observar que o *speedup* nas matrizes que utilizam double é bem menor. Este fato era esperado, conforme explicado no experimento II.

5. Conclusões

Este trabalho avaliou o desempenho de uma solução paralelizada que utiliza uma GPU para o problema de multiplicação de matrizes de tamanhos diversos.

Os resultados obtidos, mostrando um *speedup* acima de 2000 para alguns tamanhos de matrizes, tornam indiscutível o ganho de desempenho oferecido por algoritmos paralelos utilizando uma GPU.

Porém, estes algoritmos devem levar em conta uma série de nuances que podem afetar drasticamente este ganho de desempenho. Uma correta definição do tamanho dos blocos, tipo memoria a ser utilizado, tipo de sincronizaçao, entre outras customizações, podem fazer com que o ganho de desempenho esteja na ordem das dezenas ou na ordem dos milhares. Por exemplo, o uso da memoria global ao invés da memoria compartilhada, que é muito mais rápida, ira fazer desempenho do algoritmo de multiplicação de matrizes cair cerca de 58%, o uso de copia assíncrona para troca de dados entre CPU e GPU melhora em aproximadamente 25% o desempenho, a definição de blocos com tamanho que não sejam múltiplos do tamanho dos warps (32 no caso da placa utilizada nos testes) também fara o desempenho cair consideravelmente.

Para o tipo de problema analisado, podemos concluir, que o uso de algoritmos paralelos com placas GPU sempre irá trazer ótimos ganhos de desempenho, porém, esses ganhos serão proporcional ao conhecimento que o programador possui da linguagem, da arquitetura oferecida pela placa e dos conceitos da programação paralela.

6. Bibliografia

[1] David B. Kirk; NVIDIA - **CUDA Software and GPU Parallel Computing Architecture** *;* NVIDIA Research, 2012

[2] JOHN NICKOLLS, IAN BUCK, AND MICHAEL GARLAND,: ***Scalable Parallel PROGRAMMING With CUDA,*** ACM QUEUE, March 2008

[3] NVIDIA CUDA tool Kid DocumentationDisponível em:http://docs.nvidia.com/cuda/cuda-quick-start guide/index.html#axzz4OcOKsIU; acessado em out/ 2016.

[4] FLN, L. J. ***Intel Halts Development of 2 New Microprocessors.***: The New York Times, 2004. Disponível em: <http://www.nytimes.com/2004/05/08/business/08chip.html>. Acesso em: out. 2016.

[5] A. W. Lim and M. S. Lam. ***Maximizing parallelism and minimizing synchronization with affine transforms.*** In POPL, pages 201–214, 1997.

[6] Volkov V. and Demmel J. ***Benchmarking GPUs to Tune Dense Linear Algebra***; SC2008 November 2008, Austin, Texas,