Luis Gustavo Abe Yano

Avaliação e comparação de desempenho utilizando tecnologia CUDA

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

São José do Rio Preto - SP, Brasil

Junho – 2010

Luis Gustavo Abe Yano

Avaliação e comparação de desempenho utilizando tecnologia CUDA

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

Orientadora:

Prof. Dra. Renata Spolon Lobato

São José do Rio Preto - SP, Brasil

Junho – 2010

Luis Gustavo Abe Yano

Avaliação e comparação de desempenho utilizando tecnologia CUDA

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

_____	_____
Luis Gustavo Abe Yano Aluno	Prof. Dra. Renata Spolon Lobato Orientadora

Banca Examinadora:

Prof. Dr. Aleardo Manacero Junior
UNESP – São José do Rio Preto

Prof. Dr. José Márcio Machado
UNESP – São José do Rio Preto

São José do Rio Preto - SP, Brasil

Junho – 2010

RESUMO

O presente trabalho consiste na exploração do potencial computacional das GPU's da NVIDIA, através do uso da tecnologia CUDA, com a finalidade de otimizar o tempo de execução dos algoritmos em paralelo. Serão implementados os algoritmos de multiplicação de matrizes, de produto escalar, do método iterativo de Jacobi-Richardson e do método de ordenação bitônica. Para tanto, se faz necessário um estudo sobre a arquitetura das GPU's NVIDIA e a tecnologia CUDA. Duas implementações de cada algoritmo são propostas: uma paralelizada utilizando a tecnologia CUDA e outra na maneira sequencial. Por fim, apresenta-se os resultados obtidos para todas as implementações.

Palavras-Chave: ordenação bitônica, Jacobi-Richardson, matriz, produto escalar, CUDA, GPU.

Evaluation and comparison of performance using technology CUDA

ABSTRACT

The present work consists in exploring the computational potential of NVIDIA GPU, using CUDA technology, aiming to optimize the runtime of parallel algorithms. Algorithms will be done for matrix multiplication, scalar product, the iterative Jacobi-Richardson method and the bitonic sorting method. To do that, a short study is needed about the architecture of the GPU and NVIDIA CUDA technology. Two implementations of each algorithm are proposed: a parallel algorithm using CUDA technology and another in sequential method. Finally, the results obtained with these implementations are presented.

Keywords: bitonic sorting, Jacobi-Richardson, matrix, scalar product, CUDA, GPU.

Sumário

Lista de figuras

Lista de códigos

1. INTRODUÇÃO

A Ciência da Computação é, em sua essência, uma área que estuda e desenvolve solução em processamento de dados. O objetivo principal desta área é produzir resultados computacionais através da entrada de variáveis ou dados em um sistema. Problemas de um sistema são descritos através de passos, instruções para a máquina. Estes passos direcionam o que a máquina deve fazer. O nome deste esquema de passos é algoritmo, e através de algoritmos consegue-se uma abstração de um problema real. Implementar um algoritmo, seria transcrever para a máquina um dado sistema descrito no algoritmo. O ato de implementar algoritmos seria fazer computação. Sendo assim, com o advento dos computadores, a computação passou a ser realizada por estas máquinas, que são programáveis. Os computadores, ao executarem programas, interpretam o código que foi passado através dos algoritmos e seguem passo a passo a tarefa pela qual foram incubidos.

O principal ganho que se obteve com o advento dos computadores foi a rapidez na execução das tarefas. O que se gastava minutos, até horas, os computadores conseguiam fazer em apenas alguns segundos. Então, o maior objetivo dessa evolução, obtida pelo avanço da tecnologia, é resolver problemas em menos tempo de computação.
Historicamente, essa redução de tempo foi obtida através do aumento do poder de processamento das CPU ao longo do tempo. Entretanto, em 8 de maio de 2004 [FLYNN, 2004], a Intel anunciou o cancelamento de dois projetos de processadores: Tejas e Jayhawk. O motivo desse cancelamento foi o alto de consumo de energia e o aquecimento do hardware, o que tornou inviável a continuidade do incremento do *clock* dos processadores. A partir de então, a Intel concentra-se em projetos de processadores de dois ou mais núcleos.

O tempo gasto pela computação é o tempo necessário para que todas as instruções sejam executadas pelo processador. Como elas são executadas em sequencia, o tempo total da computação é a soma dos tempos de cada instrução. Logo, com a evolução do hardware, o mesmo programa era executado em um tempo cada vez menor, bastando a aquisição de um processador mais atual. Com o fim do incremento do *clock* das CPU's, essa facilidade acabou.

Assim, uma forma viável de reduzir o tempo da computação é através da paralelização da execução das instruções, ou seja, buscar executar mais de uma instrução ao mesmo tempo. Para que isso seja possível, é preciso executar o código em

um ou mais computadores paralelos. Um computador paralelo pode ser um conjunto de processadores capazes de trabalhar cooperativamente para resolver um dado problema [FOSTER, 1995].

Considerando que duas instruções são executadas em paralelo, apenas metade do tempo utilizado pelo algoritmo sequencial é necessário para realizar a mesma computação, com o acréscimo de uma instrução para dividir a instrução e outra que junta os resultados parciais.

Impulsionado pela crescente complexidade do processamento gráfico, especialmente em aplicações de jogos, as placas de vídeo passaram por uma grande revitalização tecnológica. Atualmente, as placas de vídeo são extremamente paralelas. A figura 1.1 exibe uma comparação entre a capacidade de processamento das GPU's da NVIDIA e dos processadores da Intel, no que diz respeito às operações de ponto flutuante. Pode-se observar que as GPU's da NVIDIA já possuem um poder computacional maior em relação às CPU's da Intel.

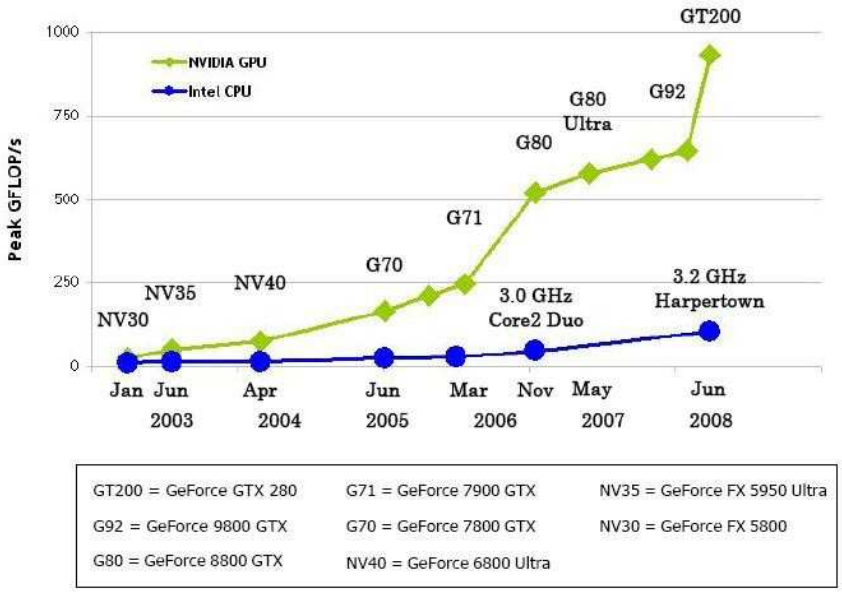


Fig. 1.1: Evolução dos GFLOPS das GPU's e CPU [CUDA PROGRAMMING, 2009].

Outro aspecto importante na análise do poder de processamento de um hardware é a largura de banda da memória (*memory bandwidth*). As GPU's da NVIDIA também ultrapassaram as CPU's da Intel, conforme pode ser visto na figura 1.2.

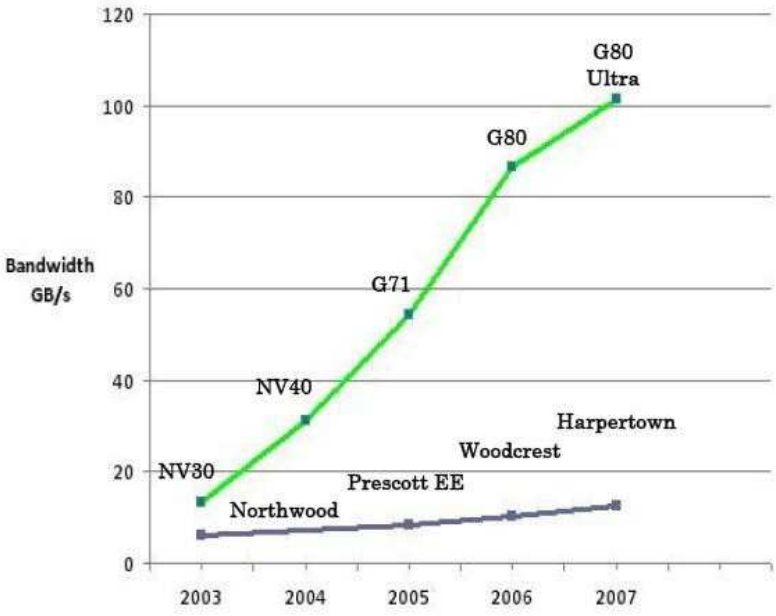


Fig. 1.2: Evolução da *bandwidth* das GPU's e CPU [CUDA PROGRAMMING, 2009].

Além disso, há a questão financeira que chama a atenção. O valor financeiro das GPGPU é menor do que o processador convencional. Para ilustrar isso, a figura 1.3 mostra uma comparação das placas mais atuais do mercado. Os preços foram cotados pela internet em junho de 2010.

Fig. 1.3: Comparação entre preços e *bandwidth* de GPU's e CPU.

Tendo em vista esse enorme potencial computacional oferecido pelas GPU's da NVIDIA, essa empresa, em 15 de fevereiro de 2007, tornou público uma nova plataforma de software: o CUDA. Com o CUDA é possível explorar o potencial computacional das GPU's, antes restrita à computação gráfica, para computação em geral. Gerando o conceito das GPGPU, ou seja, placa de vídeo que servem para propósitos gerais.

Neste cenário, este trabalho procura analisar o potencial da GPGPU e compará-la com o potencial da CPU. Serão implementados algoritmos paralelos utilizando do processamento da GPGPU e algoritmos sequenciais utilizando a maneira convencional de processamento da CPU.

Sendo assim, o capítulo 2 apresenta a teoria envolvida por parte dos algoritmos a serem implementados, uma contextualização da programação paralela, um estudo geral sobre GPGPU e CUDA, através de uma breve análise da arquitetura das GPU's da NVIDIA e da API CUDA.

No capítulo 3, apresenta-se os principais trechos dos códigos implementados com comentários sobre o seu desenvolvimento. Os código-fonte serão disponibilizados nos apêndices.

O capítulo 4 apresenta todos os testes de desempenho realizados nos algoritmos programados. Demonstrando os ganhos obtidos com a exploração dessa nova tecnologia, seguidos de uma breve análise desses resultados. O capítulo 5 apresenta as considerações finais deste trabalho e propostas de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA DO PROJETO

Operações matriciais da álgebra linear são a base da computação científica[CUMMINS, 2008].

Neste capítulo, serão apresentados conceitos da álgebra linear, entre eles a multiplicação de matrizes e do produto escalar, características do método iterativo de Jacobi-Richardson, também conhecido como Método dos Deslocamentos Simultâneos, de resolução de sistemas lineares e características sobre o método de ordenação bitônica. Também será apresentado o contexto histórico de programação paralela e a definição de GPGPU.

Finalmente, enuncia-se o conceito de CUDA e suas características, tais como a arquitetura, a estrutura de sua API, como é feito o gerenciamento de memória e alguns aspectos sobre seu desempenho.

2.1 Álgebra linear

Álgebra linear é uma parte da álgebra que, por sua vez, é um ramo da matemática na qual são estudados matrizes, espaços vetoriais e transformações lineares. Todos esses itens servem para um estudo detalhado de sistemas lineares de equações. É um fato histórico que a invenção da álgebra linear tenha origem nos estudos de sistemas lineares de equações. Assim, o fato da álgebra linear ser um campo abstrato da matemática, ela tem um grande número de aplicações dentro e fora da matemática [LEON, 1998].

A álgebra linear é o estudo dos espaços vetoriais e das transformações lineares entre eles [LIMA, 1995] e [LIMA, 2001]. Quando os espaços têm dimensões finitas, as transformações lineares possuem matrizes. São numerosas e bastante variadas as situações em matemática e em suas aplicações onde ocorre a utilização destes objetos. A álgebra linear aplica-se a várias áreas, por exemplo, em criptografia, computação gráfica, genética e programação linear.

O produto escalar de dois vetores, tal como o comprimento de um vetor, pode ser calculado por uma fórmula simples a partir das coordenadas dos vetores num referencial ortonormado dado. Na base do estabelecimento dessa fórmula, estão as propriedades algébricas do produto escalar, nomeadamente a sua bilinearidade (propriedades distributiva e associativa mista) e comutatividade [BOLDRINI, 1980].

Neste trabalho, serão implementados os algoritmos de multiplicação de matrizes, de produto escalar entre vetores e do método iterativo de Jacobi-Richardson para resolução de sistemas lineares, o qual será explicado a seguir.

2.2 Método iterativo de Jacobi-Richardson

Métodos numéricos para solução de sistemas de equações lineares são divididos principalmente em dois grupos:

• Métodos Exatos: são aqueles que forneceriam a solução exata, se não fossem os erros de arredondamento, com um número finito de operações.

• Métodos Iterativos: são aqueles que permitem obter a solução de um sistema com uma dada precisão através de um processo infinito convergente e que possui um erro de truncamento.

Em sistemas de grande porte os erros de arredondamento de um método exato podem tornar a solução sem significado, enquanto que nos métodos iterativos os erros de arredondamento não se acumulam.

Os métodos iterativos, em certos casos, são melhores do que os métodos exatos, por exemplo, quando a matriz dos coeficientes é uma matriz esparsa (muitos elementos iguais a zero). Os métodos iterativos utilizam menos memória do computador, e além disso, possuem a vantagem de se auto corrigir se um erro é cometido. Ainda podem ser usados para reduzir os erros de arredondamento na solução obtida por métodos exatos, e também, sob certas condições, serem aplicados para resolver um conjunto de equações não lineares[FRANCO, 2006].

O método iterativo de Jacobi-Richardson se baseia inicialmente numa verificação de convergência, ou seja, verifica-se se a matriz é estritamente diagonalmente dominante. Se ela atender o critério de convergência, define-se a margem de erro máximo (truncamento) que o resultado terá. Assim, o método inicia-se através da solução inicial (no caso de um sistema linear, zero é uma das soluções possíveis), em seguida, calcula-se o resultado e ocorre a substituição no vetor resultado na próxima iteração, e assim em diante até que se tenha a solução que atenda a margem de erro. O diagrama da figura 2.1, a seguir, resume a execução do método.

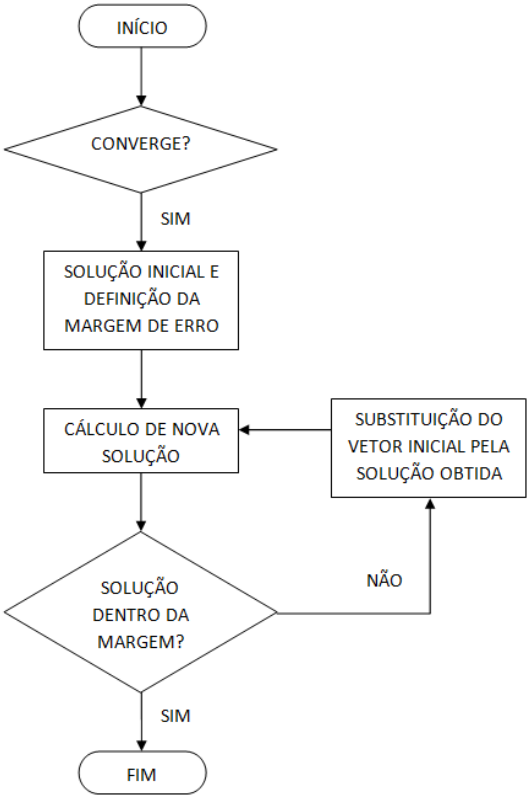


Fig. 2.1: Execução do método de Jacobi-Richardson

Apesar da existência estudos que visam melhorar o desempenho deste método [KHATCHATOURIAN, 2006], utilizaremos o método trivial por ser amplamente mais utilizado.

2.3 Ordenação bitônica

Este algoritmo de ordenação utiliza a idéia de unir pares de subsequências alternadamente com ordenações locais. O algoritmo é baseado na idéia do algoritmo apresentado em [CÁCERES, 2001] e demais demonstrações estão em [GONDA, 2004].

Uma sequência de números é denominada bitônica se pudermos deslocá-la ciclicamente, tal que a sequência resultante S seja bitônica, ou seja, existe um número inteiro $1 \leq k \leq n$, tal que $S1 \leq S2 \leq ... \leq Sk \geq Sk+1 \geq ... \geq Sn$.

Assim, dada uma sequência bitônica, obtemos uma sequência ordenada, através da aplicação de operações de divisões bitônicas sucessivas. Para comprovar, considere uma sequência bitônica H = (h(0), h(1),..,h(n-1)), e as sequências, a seguir:

Smin = (min{h(0), h(n/2)}, min{h(1), h((n/2)+1)},..., min{h((n/2)-1), h(n-1)})

Smax = (max{h(0), h(n/2)}, max{h(1), h((n/2)+1)},..., max{h((n/2)-1), h(n-1)})

Smin e Smax são também sequências bitônicas e todos os elementos de Smin são menores que todos do Smax.

Por exemplo, a sequência (2 5 7 9 8 4 3 0), produz o seguinte resultado: Smin = (2,4,3,0) e Smax= (8,5,7,9). O exemplo a seguir, mostra o início da ordenação, aplicando as divisões bitônicas sucessivas:

Passo	*Elementos*
0	2 5 7 9 8 4 3 0 ↑ ↑
1	2 4 3 0 8 5 7 9 ↑ ↑ ↑ ↑
2	2 0 3 4 7 5 8 9 ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
3	0 2 3 4 5 7 8 9

Fig. 2.2: Execução do método de ordenação bitônica [CÁCERES, 2001].

Se uma sequência S não for bitônica, devemos transformá-la para que seja bitônica S(0), através de deslocações cíclicas. Assim, em seguida, podemos ordenar os elementos de S(0) através de divisões bitônicas sucessivas, como descrito acima.

Para tranformar uma sequência em bitônica, inicialmente, observa-se que qualquer sequência contendo dois elementos é bitônica e também já está ordenada de forma crescente ou decrescente. Dadas duas sequências s1 = (a0, a1) e s2 = (a2, a3), transformamos s1 em crescente e s2 em decrescente, comparando os elementos de cada sequência. Dessa forma, obtemos uma sequência bitônica S = s1 U s2 com quatro elementos. Para um sequência com n elementos, podemos dividir a sequência em subsequências bitônicas de n/2 elementos.

Passo	*Elementos*
0	8 3 4 7 9 2 0 5
1	3 8 7 4 2 9 5 0
2	3 4 7 8 5 9 2 0
3	3 4 7 8 9 5 2 0

Fig. 2.3: Execução da tranformação para uma sequência bitônica[CÁCERES, 2001].

Podemos dividir S em duas subsequências S1 = (a0, a1, a2, a3) e S2 = (a4, a5, a6, a7). Como mostrado anteriormente, S1 e S2 podem ser transformadas em sequências bitônicas. Para que S seja bitônica, devemos ordenar uma das sequências de forma

crescente e outra de forma decrescente. A subsequência S1 pode ser ordenada de forma crescente através de divisões bitônicas sucessivas, como descrito anteriormente.

Para ordenar S2 de forma decrescente, podemos aplicar, recursivamente, divisões bitônicas modificadas, colocando os máximos na primeira metade e os mínimos na segunda metade da sequência. Após estas operações, temos que S0 = S1 U S2 é uma sequência bitônica, onde os elementos de a00 a a03 estão ordenados em ordem crescente e os elementos de a04 a a07 estão ordenados em ordem decrescente.

2.4 Programação paralela

Programação paralela é aquela onde diversos processos cooperam entre si para executar simultaneamente, de forma coordenada com o objetivo de resolver um problema específico em comum [OLIVEIRA, 2001]. Para que um programa possa ser executado concorrentemente é preciso que não haja dependências entre os dados a serem processados. Uma instrução y depende de x quando ela necessita do resultado da computação de x para a sua execução.

Enquanto a programação concorrente foca mais na interação entre as tarefas, o objetivo de utilizar paralelização é aumentar a desempenho da aplicação, executando-a através dos diversos núcleos e/ou processadores existentes na mesma máquina.

Um sistema paralelo pode ser classificado por diversas formas, uma delas é pela taxionomia de Flynn. Ele classificou pelo número de fluxos de instruções e pelo número de fluxos de dados. Uma maior explicação sobre esta taxionomia pode ser encontrada em [PACHECO, 1997]. Focaremos apenas no modelo SIMD, pois CUDA utiliza um modelo bastante parecido (SIMT). NVIDIA não detalha com maior precisão sobre este novo modelo de arquitetura, porém há fontes que dizem que ela se utiliza do modelo SIMD [FATAHALIAN, 2009] e [PERILS, 2008]. No modelo SIMD, existe apenas um unidade de controle, que manda a instrução que será executada por todos os processadores. Assim, em cada instante, todos os processadores executam a mesma instrução sobre dados possivelmente diferentes.

A programação paralela, por definição, divide a execução de um programa em partes, e cada parte é processada através de uma *thread. Thread* é um fluxo de execução independente que possui uma memória compartilhada com o processo pai e que pode ser escalonada pelo sistema operacional [BARNEY, 2009]. Dessa forma, cada *thread* pode processar um subconjunto dos dados paralelamente.

As grandes motivações para se utilizar programação paralela podem ser resolver problemas computacionais grandes ou simplesmente não ter que se sujeitar a limitações física (limites de miniaturização e de velocidade de transmissão de informação entre os componentes), e econômica (é mais caro fazer um processador ficar mais rápido do que juntar vários processadores menos velozes), da computação sequencial [BARNEY, 2010].

Para realizar a paralelização existem duas abordagens, a auto-paralelização e a programação paralela.

Quando já se tem uma aplicação pronta, desenvolvida sequencialmente que não foi modificada para ser processada paralelamente, a auto-paralelização tenta automaticamente paralelizar a aplicação sequencial para que esta possa se aproveitar do hardware 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 e Intel C++ Compiler [INTEL, 2010]. As vantagens dessa abordagem é o fato de que aplicações já existentes não precisam ser refatoradas para funcionar com paralelismo, pois basta utilizar estes compiladores para funcionarem sem necessidade do programador aplicar novos conceitos de programação paralela. Contudo, a desvantagem é que o compilador não consegue otimizar no mesmo nível que um programador conseguiria utilizando paralelismo em seus algoritmos.

Com programação paralela a aplicação é desenvolvida para aproveitar o paralelismo desde o algoritmo. A vantagem dessa abordagem é que ela fornece maior ganho de desempenho, mas a desvantagem é que exige um maior esforço no desenvolvimento da aplicação.

Algumas plataformas existentes de programação paralela foram avaliadas com base nos sete critérios qualitativos propostos em [KASIM, 2008], que são:

• 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 aos casos em que cada nó de processamento tem seu próprio espaço de memória não compartilhado por outros.

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

• 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, ou então ele é implícito, e basta especificar a seção de código que vai rodar em paralelo.

• Esquema de particionamento da carga de trabalho: A carga de trabalho que será executada é divida em pequenas porções chamadas tarefas, no critério implícito os 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 divida.

• 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.

• 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 simplesmente 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.

• 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 memória distribuída.

Critério	Pthreads	OpenMP	CUDA
Execução	Thread	Thread	Thread
Metodologia de programação	API, C, Fortran	API, C, Fortran	API,Extensão de C
Gerenciamento de trabalho	Explícito	Implícito	Implícito
Particionamento da carga de trabalho	Explícito	Implícito	Explícito
Mapeamento entre tarefa e a thread	Explícito	Implícito	Explícito
Sincronização	Explícito	Implícito/Explícito	Implícito
Modelo de comunicação	Espaço de memória compartilhado	Espaço de memória compartilhado	Espaço de memória compartilhado

Tabela 1 - Arquitetura de sistema com memória compartilhada [KASIM, 2008].

Critério	MPI	UPC	Fortress
Execução	Processo	Thread	Thread
Metodologia de programação	API, C, Fortran	API, C	Nova linguagem
Gerenciamento de trabalho	Implícito	Implícito	Implícito/Explícito
Particionamento da carga de trabalho	Explícito	Implícito/Explícito	Implícito/Explícito
Mapeamento entre tarefa e a thread	Explícito	Implícito/Explícito	Implícito/Explícito
Sincronização	Implícito	Implícito/Explícito	Implícito/Explícito
Modelo de comunicação	Troca de mensagens	Partição do espaço de endereço de memória global	Espaço de memória global

Tabela 2 - Arquitetura de sistema com memória distribuída [KASIM, 2008].

2.4.1 Conceito de *speedup*

Seja P um problema computacional qualquer e seja n o tamanho da entrada. Denotemos a complexidade sequencial de P por T*(n). Existe um algoritmo sequencial que resolve P com esse tempo mínimo, e ainda mais, podemos provar que nenhum algoritmo sequencial pode resolver P de forma mais rápida. Seja A um algoritmo paralelo que resolve P num tempo Tp(n) em um computador paralelo com p processadores.

Define-se o *speedup* alcançado por A como:

$$Sp(n) = T^*(n) / Tp(n)$$

O valor Sp(n) mede o *speedup* obtido por um algoritmo A quando p processadores estão disponíveis para utilização. O objetivo é projetar algoritmos paralelos que alcancem Sp(n) próximos de p.

Na realidade, existem vários fatores que introduzem ineficiência nos algoritmos paralelos. Entre eles, destaca-se os atrasos que são introduzidos pela comunicação, a sobrecarga ocorrida na sincronização das atividades dos vários processadores e no controle do sistema.

Note que T1(n), é o tempo do algoritmo paralelo A quando o número p de processadores é igual a 1, não é necessariamente o mesmo que T*(n), portanto, o *speedup* é medido relativamente de acordo com o melhor algoritmo sequencial possível. É comum considerar o T*(n) como o tempo do melhor algoritmo sequencial conhecido, sempre que a complexidade não é conhecida.

Utilizando CUDA, segundo [CUDA PROGRAMMING, 2009], cada bloco de *threads*, é distribuído para cada núcleo automaticamente. Sendo assim, se a GPU tivesse 4 núcleos de processamento, e existisse 8 blocos de *threads* para serem processados, o primeiro e o quinto blocos ficariam no primeiro núcleo, o segundo e o sexto bloco ficariam no segundo núcleo, por assim em diante. Como neste trabalho, o número de blocos será variável, o número de núcleos processadores empregados será variável também.

2.5 GPGPU

GPGPU é um acrônimo de 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), desenvolvido por Mark Harris em 2002, quando aproveitou a capacidade das GPU's em aplicações não gráficas [GPGPU, 2010].

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.

A GPU (Graphics Processing Units) é a unidade de processamento gráfico, um sistema especializado em processar imagens, através de uma combinação de um processador e um minúscula memória de alta velocidade (GRAM) [BOGGAN et al., 2007]. 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, dizia-se que desenvolver programas para serem executadas em uma placa de vídeo, era incompreensível, devido à falta de um framework que facilitasse o processo. Hoje, as GPU's evoluíram para um processador de diversos núcleos tornando-as interessantes para o desenvolvimento de aplicações em sistemas paralelos, a figura 2.4 ilustra a diferença de núcleos que pode existir. As GPU'ss 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 [BERRILLO, 2008]. Portanto, se explica o aumento do uso de GPU como um co-processador para executar aplicações paralelas de alta performance [GRAÇA, 2006]. Para facilitar a execução desses algoritmos na GPU criaram as bibliotecas como CUDA, que possui recursos de interfaces para programação em linguagens conhecidas como C.

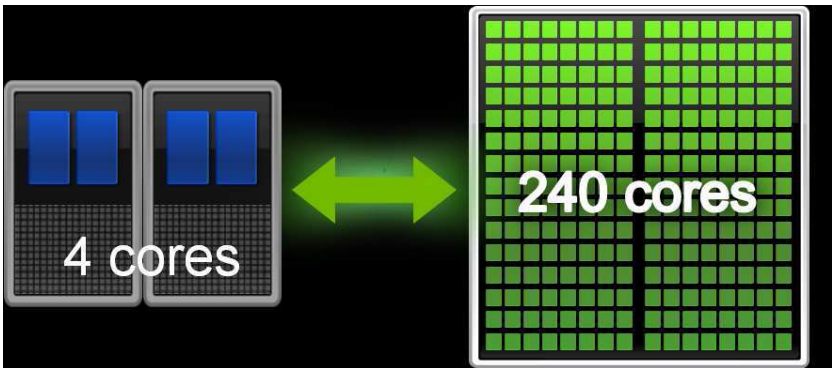


Fig. 2.4: Exemplo de comparação entre CPU e GPU[CUDA PROGRAMMING, 2009].

Os desenvolvedores que portam suas aplicações paralelas para executarem com GPGPU frequentemente alcançam ganhos de velocidade em comparação com as versões das aplicações personalizadas 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 (Texas Advanced Computing Center) Ranger [RANGER, 2010], um super computador com a seguinte configuração:

- Nós: 3.936
- Núcleos: 62.976
- Pico de Desempenho: 579.4 TFlops
- Número de Racks: 82

Fig. 2.5: Uma das seis fileiras do super computador TACC Ranger[RANGER, 2010].

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

- 145 Tesla's = 580 TeraFLOPS (1 Tesla S1070 = 4 TeraFLOPS)
- 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 tem suas bases na linguagem C de programação, por ser bastante didática 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. CUDA, que é uma das principais bibliotecas de desenvolvimento atualmente e de melhor desempenho [GAO, 2008], será detalhada a seguir.

2.6 CUDA

CUDA é uma plataforma que utiliza software e hardware para computação paralela de alto desempenho de propósito geral que utiliza o poder de processamento dos núcleos das GPU's's da NVIDIA. A arquitetura da GeForce 8 possui 128 processadores de *threads*, suportando um total de 12.288 *threads* concorrentes [HALFHILL, 2008]. A plataforma foi formalmente apresentada em fevereiro de 2007 [NVIDIA CUDA, 2007]. Atualmente, encontra-se em sua versão 3.0. Ela possui muitos usuários e demanda nos campos da matemática [DU, 2007], [AMORIM, 2009], [ZOU, 2009] e [DZIEKONSKI, 2008], científico [TAHER, 2009] e [CUMMINS, 2008], biomédico [LIGOWSKI, 2009], [LUEBKE, 2008] e [LING, 2009], da computação [MANAVSKI, 2007] e da engenharia [SPAMPINATO, 2009], entre outros. Devido às características das aplicações nesses campos, que são altamente paralelizáveis, CUDA é bastante útil para elas. Essa tecnologia, apesar de algumas desvantagens em relação aos atuais super computadores [SUDA, 2009], tem chamado bastante atenção de toda a comunidade acadêmica devido ao seu grande poder computacional.

2.6.1 Arquitetura das GPU's NVIDIA

Conforme foi visto na introdução, a capacidade de processamento das GPU's's da NVIDIA ultrapassou a das CPU da Intel. O principal motivo dessa eficiência de desempenho é que as GPU's's são dedicadas exclusivamente ao processamento de dados, não tendo a tarefa de guardar informações em memórias cache nem de tratar um controle de fluxo. Isso se deve ao fato das aplicações gráficas serem extremamente paralelas, realizando a mesma operação com grande volume de dados. Desta forma, as áreas que seriam destinadas à memória cache e controle de fluxo nas CPU, são

utilizadas para processamento de dados nas GPU's. A figura 2.6 exibe uma comparação entre a destinação dos transistores em GPU e CPU.

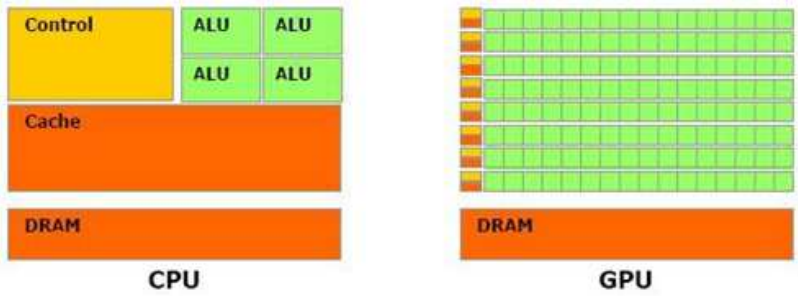


Fig. 2.6: Destinação dos transistores em GPU e CPU [CUDA PROGRAMMING, 2009].

Em novembro de 2006, a NVIDIA apresentou para o mercado, sua primeira GPU destinada à computação de propósitos gerais (GPGPU): a arquitetura Tesla [CUDA PROGRAMMING, 2009]. Essa arquitetura é totalmente dedicada ao processamento de dados. Através dessa nova arquitetura, a NVIDIA criou um novo conceito na computação paralela: *single-instruction multiple-thread*, ou SIMT (como foi citado, anteriormente na seção 2.4). Essa arquitetura cria, gerencia, agenda e executa *threads* automaticamente em grupos de 32 *threads* paralelas, o que a NVIDIA chama de *warp*, sem *overhead* de agendamento [CUDA PROGRAMMING, 2009]. Caso um multiprocessador receba um bloco com mais de 32 *threads* para processar, ele quebra esse bloco em *warps*, agrupando as *threads* de acordo com o seu *thread ID*.

Uma GPU Tesla consiste em um *array* de multiprocessadores de *threads*, conforme pode ser observado na figura 2.7.

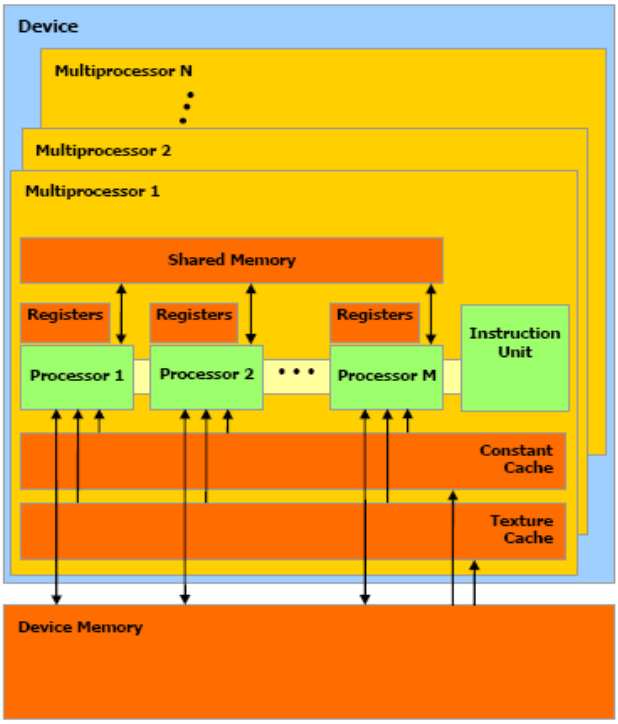


Figura 2.7: A arquitetura da GPU Tesla [CUDA PROGRAMMING, 2009].

2.6.2 A API CUDA

Programar em CUDA exige componentes de software e hardware. Em relação ao hardware, uma lista de GPU que são compatíveis para executar código CUDA pode ser encontrada em [CUDA PROGRAMMING, 2009]. Sendo satisfeita essa condição, é necessária a utilização de alguns softwares: um *driver* específico e um *toolkit* contendo um compilador e algumas ferramentas adicionais. Todos são obtidos no *site* da plataforma [DONWLOAD CUDA, 2010]. A figura 2.8 ilustra a organização da plataforma CUDA, composta pelo *driver* de acesso ao hardware, um componente de *runtime* e de duas bibliotecas matemáticas: CUBLAS (*Basic Linear Algebra Subroutines*) e CUFFT (*Fast Fourier Transform*). No topo da cadeia encontra-se a API da plataforma CUDA.

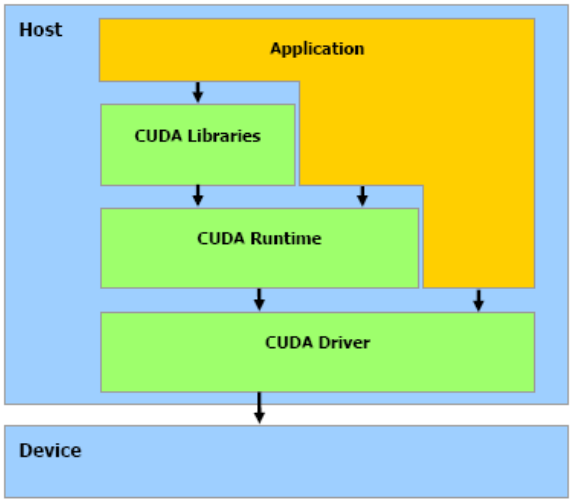


Figura 2.8: A pilha de *software* da plataforma CUDA [CUDA PROGRAMMING, 2009].

O escalonamento das *threads* da plataforma CUDA utiliza dois conceitos: bloco e *grid*. Através deles se organiza a repartição dos dados entre as *threads* e a organização e distribuição dos dados ao hardware.

Um bloco é a unidade básica de organização das *threads* e de mapeamento para o hardware. Um bloco de *threads* é alocado a um multiprocessador da GPU. Dessa forma, o tamanho mínimo recomendado a um bloco é de 8 *threads*, caso contrário deve haver processadores desocupados. Os blocos podem ter até três dimensões.

O *grid* é a unidade básica onde estão distribuídos os blocos. O *grid* é a estrutura completa de distribuição das *threads* que executam uma função. E nele que está definido o número total de blocos e de *threads* que serão criados e gerenciados pela GPU para uma determinada função. Um *grid* pode ter até duas dimensões.

A figura 2.9 ilustra um *grid* de dimensões 2x3 com blocos de tamanho 3x4.

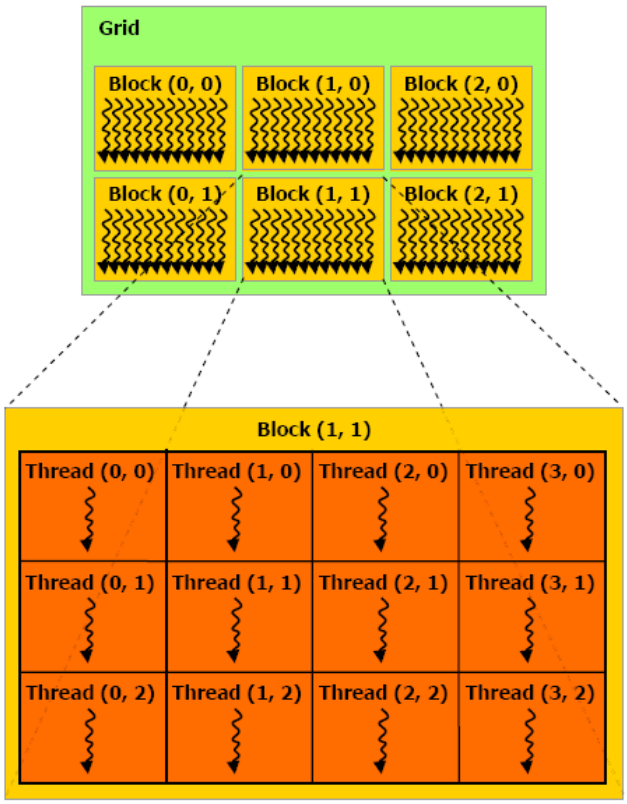


Figura 2.9: Exemplo de organização dos blocos pertencentes a um *grid* [CUDA PROGRAMMING, 2009].

A API da plataforma CUDA introduz quatro extensões a linguagem C, sendo elas:

• Qualificadores de tipo de função, para definir a unidade lógica de execução do código: CPU ou GPU;

• Qualificadores de tipo de variável, para definir onde elas serão armazenadas: na CPU ou na GPU;

• Nova sintaxe de chamada de função para configurar os blocos e o *grid*;

• Variáveis internas para acessar os índices e dimensões dos blocos, do *grid* e das *threads*.

Existem três qualificadores de tipo de função: __device__, __global__ e __host__. O qualificador __device__ define uma função que será executada na GPU e somente poderá ser iniciada a partir da GPU.

O qualificador __global__ define o que a plataforma CUDA chama de *kernel*, ou seja, uma função que é executada na GPU e é iniciada a partir da CPU.

Por fim, o qualificador __host__ define uma função que será executada na CPU e que somente poderá ser iniciada a partir da CPU.

São três os qualificadores de tipo de variáveis: __device__, __constant__ e __shared__.

O qualificador __device__ define uma variável que reside na memória global da GPU. Elas são acessíveis por todas as *threads* de um *grid* e também a partir da CPU através do uso da biblioteca de *runtime* do CUDA e possui em tempo de vida da aplicação.

O qualificador __constant__ é similar ao __device__, porém se difere no fato de que a variável é alocada no espaço de memória constante da GPU.

Por outro lado, as variáveis __shared__ residem na memória compartilhada da GPU e são acessíveis apenas pelas *threads* de um mesmo bloco e possui em tempo de vida do bloco.

Para realizar uma chamada de função na GPU e preciso informar as dimensões do *grid* e do bloco. Isso é feito através de uma sintaxe na chamada da função. Utiliza-se, entre o nome da função e os argumentos passados a ela, um *array* bidimensional onde constam as dimensões do *grid* e do bloco, respectivamente. Esse *array* e delimitado pelos caracteres <<< e >>>.

Para acessar o valor dos índices das *threads* utiliza-se a variável threadIdx, a qual é um vetor de até três dimensões e o acesso a cada dimensão é feito através das componentes x, y e z. Cada bloco dentro do *grid* fornece seu índice identificador, como em x e y. Além disso, os índices das dimensões dos blocos de *threads* são acessíveis através da variável blockDim, através das componentes x, y e z. Por fim, é possível acessar os valores das dimensões do *grid* através da variável gridDim, como em gridDim.x e gridDim.y.

Tomando-se como exemplo a *thread* (3,2) do bloco (1,1) da figura 2.9, tem-se:

threadIdx.x = 3;

threadIdx.y = 2;

blockIdx.x = 1;

blockIdx.y = 1;

blockDim.x = 4;

blockDim.y = 3;

gridDim.x = 3;

gridDim.y = 2.

As funções da plataforma CUDA possuem algumas restrições, como por exemplo:

• As funções __device__ e __global__ não suportam recursão, nem declarar variáveis estáticas e sem possibilidade de variar o número de argumentos.

• As funções __device__ não fornecem seu endereço, porém, ponteiros para funções __global__ podem existir.

• As funções __global__ retornam *void*, obrigatoriamente.

• Qualquer chamada a uma função __global__ deve especificar a dimensão do *grid* e dos blocos.

• Chamadas as funções __global__ são assíncronas, ou seja, a execução continua na CPU mesmo que não tenha terminado na GPU.

• Os parâmetros de uma função __global__ são passados através da memória compartilhada e estão limitados a 256 bytes.

A figura 2.10 mostra o fluxo normal de execução de um programa feito em CUDA. Inicialmente, o programa copia os dados da memória RAM que serão processados para a memória da GPU. Logo após, configura-se os blocos que receberão os dados para de fato ocorrer o processamento. Assim, os resultados são devolvidos para a memória RAM.

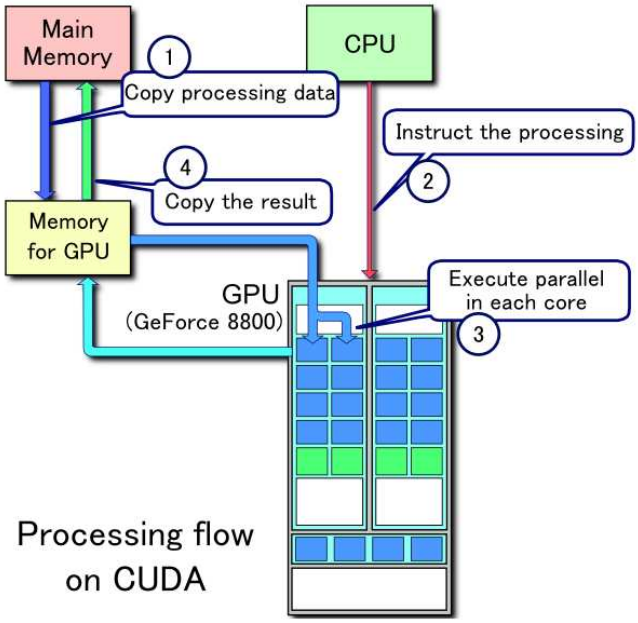


Figura 2.10: Fluxo de execução de um programa CUDA [CUDA PROGRAMMING, 2009].

Analisando a figura 2.11, percebe-se que o ciclo de execução de uma aplicação CUDA é alternada entre execuções ora na CPU, ora na GPU.

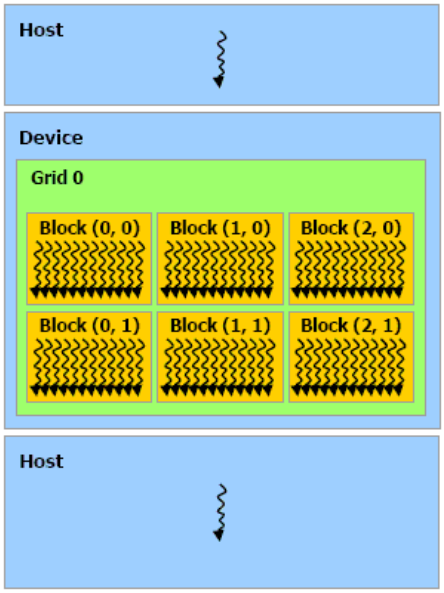


Figura 2.11: Caminho de execução de uma aplicação CUDA [CUDA PROGRAMMING, 2009].

No geral, o programa em CUDA é organizado no programa principal, consistindo em um ou mais *threads* sequenciais executando na CPU, e um ou mais *kernels* que são executados na GPU. Um *kernel* executa um programa sequencial num conjunto de *threads* paralelas. O programador divide os blocos de *threads* em *grids.* As *threads* de um mesmo bloco são capazes de sincronizar entre elas através via barreiras e terem acesso a uma comunicação muito rápida. As *threads* de blocos diferentes no mesmo *grid* podem ser coordenadas apenas via operações na memória global compartilhada visível a todas as *threads*. CUDA requer que os blocos sejam independentes, para que o *kernel* seja executado corretamente não importando a ordem que as *threads* são executadas [GARLAND, 2008].

2.7 Gerenciamento de memória

A tecnologia CUDA não permite a utilização de ponteiros para ponteiros por ser uma operação ilegal referenciar um ponteiro do código *device* (executado na GPU) no código *host* (executado na CPU), o que resulta em falha de segmentação. Portanto, existem duas formas de alocar memória dinamicamente: como memória linear ou CUDA *arrays*. Para a alocação de memória linear utiliza-se uma das seguintes funções:

cudaMalloc, *cudaMallocPitch*, *cudaMalloc2D* ou *cudaMalloc3D*. Já para a alocação de um CUDA *array* utiliza-se a função *cudaMallocArray*.

Com exceção da função *cudaMalloc*, no que tange a alocação de memória linear, as demais funções efetuam um ajuste dos dados na memória para que esta fique alocada de forma a aperfeiçoar o acesso aos dados. Este ajuste significa aumentar o tamanho das linhas da estrutura, preenchendo com zeros os elementos excedentes. Além disso, essas funções retornam um valor chamado *pitch*, o qual deve ser utilizado para acessar o *array* alocado. Cabe ressaltar aqui que o *array* alocado é linearizado na memória, independentemente do número de dimensões que este possua. O código 2.1 abaixo ilustra a utilização desse modelo de alocação de memória.

```
float* devPtr;
int pitch;
cudaMallocPitch((void**)&devPtr, &pitch, width * sizeof(float),
height);
myKernel<<<100, 512>>>(devPtr, pitch);
__global__ void myKernel(float* devPtr, int pitch) {
for (int r = 0; r < height; ++r) {
float* row = (float*)((char*)devPtr + r * pitch);
for (int c = 0; c < width; ++c) {
float element = row[c];
}}}
```

Código 2.1: Alocando um array bidimensional [CUDA PROGRAMMING, 2009].

Conforme pode ser observado, a função *cudaMallocPitch* aloca espaço suficiente para um *array* bidimensional de tamanho *width * height* do tipo *float*.

Entretanto, o acesso aos dados deste *array* é linearizado.

Por outro lado, um CUDA *array* pode ter uma, duas ou três dimensões e seus elementos podem ser inteiros de 8, 16 ou 32 bits ou *floats* de 32 bits. São estruturas otimizadas para leitura a partir da memória de textura. Para utilizá-los e preciso gravar os dados na memória de textura e acessá-las através de uma função cujo retorno e o elemento associado. O código 2.2 ilustra um exemplo de utilização dessa estrutura.

```
texture<float, 2, cudaReadModeElementType> texRef;
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(32, 0, 0,
0, cudaChannelFormatKindFloat);
cudaArray* cuArray;
cudaMallocArray(&cuArray, &channelDesc, width, height);
cudaMemcpyToArray(cuArray, 0, 0, C_data, size,
cudaMemcpyHostToDevice);
cudaBindTextureToArray(texRef, cuArray, channelDesc);
...
float elemento = tex2D(texRef, x, y);
```

Código 2.2: Utilizando um CUDA array [CUDA PROGRAMMING, 2009].

Conforme pode ser observado na listagem acima, para utilizar um CUDA *array* é preciso definir uma textura, definir um CUDA *array*, alocar o CUDA *array*, copiar os dados para o CUDA *array*, e por fim, mapear o CUDA *array* para a textura. Por fim, o acesso aos dados em uma função *kernel* é realizado através da função *tex2D*, para um *array* bidimensional. Cabe ressaltar o fato de que a memória de textura é apenas-leitura.

Alternativamente, é possível alocar memória estática. Dessa forma, é possível acessar um *array* de varias dimensões utilizando a sintaxe usual de colchetes. O código 2.3 ilustra um exemplo.

```
typedef struct {
float array[largura][altura];
} tipobloco;
…
tipobloco *bloco;
cudaMalloc((void**)&bloco, sizeof(tipobloco));
…
float elemento = bloco->array[x][y];
```

Código 2.3: Alocação estática de memória.

De acordo com a listagem acima, as linhas 1 a 3 definem a estrutura de dados para conter o *array* bidimensional, as linhas 5 e 6 tratam de definir e alocar espaço para esta estrutura e a linha 8 ilustra como acessar dados dessa estrutura dentro de uma função *kernel*.

2.8 Aspectos de desempenho

De acordo com [CUDA PROGRAMMING, 2009], para obter o melhor desempenho da GPU é preciso que o *grid* aloque ao menos um bloco para cada multiprocessador.

Adicionalmente, cada bloco deve conter pelo menos 64 *threads*. Entretanto, o acesso aos registradores, que normalmente não demanda nenhum ciclo extra, pode implicar em atrasos devido a dependências de *read-after-write* e conflitos de endereçamento. Sendo assim, estes atrasos podem ser ignorados se houverem pelo menos 192 *threads* ativas por multiprocessador. Dessa forma, quando ocorre algum atraso na execução de *warp*, imediatamente outro *warp* e colocado em execução, mantendo a GPU ocupada. É possível manter até 32 *warps* ativos por multiprocessador.

Outro aspecto importante é que o acesso a memória global pode ser otimizado de acordo com determinadas condições. O que ocorre é que o acesso a memória por um *half-warp* (um *half-warp* pode ser as 16 *threads* superiores ou inferiores de um *warp*) pode ser realizado com até uma única instrução de leitura. Para palavras de 32 bits, um *float* por exemplo, o segmento de memória é de 128 bytes, ou 32 *floats*. Sendo assim, quanto mais *threads* de um *half-warp* ativo no multiprocessador acessarem endereços de um mesmo segmento, mais rápida será a sua execução. A escolha do segmento a ser lido segue o seguinte algoritmo:

• Buscar o segmento de memória que contem o endereço requisitado pela *thread* ativa de menor *thread ID*;

• Encontrar todas as outras *threads* ativas que requisitaram dados do mesmo segmento;

• Reduzir o tamanho do segmento, se possível, de 128 bytes para 64 bytes caso todas *threads* acessem apenas a metade superior ou inferior do segmento;

• Reduzir o tamanho do segmento, se possível, de 64 bytes para 32 bytes caso todas *threads* acessem apenas a metade superior ou inferior do segmento;

• Executar a instrução de leitura do segmento e marcar as *threads* servidas como inativas;

• Executar até que todas as *threads* do *half-warp* sejam servidas.

Como a alocação de memória CUDA é linearizada, quando todas as *threads* de um *half-warp* requisitarem elementos de uma mesma coluna em sequência, este acesso

será otimizado para uma única instrução de leitura da memória, devido ao fato de que os dados requisitados pertencem ao mesmo segmento de memória. Esta é a técnica adequada para se obter altos índices de transferência de dados da memória.

O compilador realiza um esforço a fim de tornar o acesso à memória o mais otimizado possível, entretanto, para se obter um resultado melhor é preciso que o programador organize o código e os dados de forma a facilitar este processo. Para que seja possível organizar o código de forma a obter estes ganhos de desempenho e preciso entender a regra de formação dos *warps*. A regra de formação de um *warp* e a seguinte:

Dado um bloco com mais de 32 *threads*, estas serão agrupadas em grupos de 32 de acordo com o seu *thread ID*, iniciando com a *thread* de menor *thread ID*, prosseguindo de forma incremental ate a *thread* com o maior *thread ID* no bloco. O cálculo do *thread ID* é direto: para blocos unidimensionais, o *thread ID* e o próprio indice da *thread*. Para um bloco bidimensional de tamanho (*Dx*, *Dy*), o *thread ID* da *thread* de índice (*x*, *y*) e (*x + y * Dx)*.

Sendo assim, para realizar a soma de dois vetores compostos por elementos de 32 bits de tamanho 512 da forma mais eficiente possível, basta alocar um bloco unidimensional de tamanho 512 onde a função kernel que realiza a soma e como segue: *c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x]*. O bloco de 512 *threads* será particionado em 16 *warps* de modo que o primeiro *warp* conterá as *threads* de índice [0,1, 2, …, 31], o segundo *warp* conterá as *threads* de índice [32, 33, 34, …, 63], e assim por diante. No momento de execução, cada *thread* irá substituir no código o valor da variável *threadIdx.x* pelo seu respectivo índice. Dessa forma, no momento da execução do primeiro *warp* cada *thread* irá executar o seguinte código:

```
c[0] = a[0] + b[0];
c[1] = a[1] + b[1];
c[2] = a[2] + b[2];
...
```

Quando a *thread* de índice 0 solicita o dado *a[0]*, o algoritmo de acesso a memória é executado. Como as *threads* requisitam os dados *a[0]*, *a[1]*,*a[2]*, ..., *a[31*] e estes encontram-se em um mesmo segmento de memória de 128 bytes, apenas uma instrução de leitura da memória é realizada.

Vale a pena frisar que as placas da NVIDIA possuem divergências em relação à norma IEEE-754, a qual padroniza a notação de ponto flutuante. É interessante citar que operações de multiplicação seguidas de adição, utilizando CUDA, geralmente são

executadas em uma única instrução no hardware chamada FMAD. Instruções FMAD são capazes de realizar duas operações de ponto flutuante em precisão simples por ciclo de *clock*. Esta instrução faz o truncamento dos valores intermediários das operações, fazendo com que haja perda de precisão [CUDA PROGRAMMING, 2009]. Embora CUDA forneça, via software, uma solução para resolver esta disparidade, a solução apresentada promove perda de desempenho para realizar operações seguindo o padrão IEEE-754.

Este capítulo apresentou a importância da multiplicação de matrizes e o produto escalar, a arquitetura das GPU's da NVIDIA e a tecnologia CUDA. No próximo capítulo, serão apresentados os algoritmos programados utilizando a tecnologia CUDA.

3 DESENVOLVIMENTO

O desenvolvimento do projeto se baseou em programar os algoritmos de multiplicação de matrizes, de produto escalar, o método de Jacobi-Richardson e o método de ordenação bitônica. O desenvolvimento destes algoritmos será utilizando CUDA. Estes mesmos algoritmos serão implementados também da maneira sequencial para posterior comparação entre as duas versões.

Em todos os projetos, foi feita a divisão em três arquivos: um para o algoritmo sequencial (com extensão .cpp), um para o código do *kernel* da execução na GPU, e um para junção de ambos como programa principal.

Os arquivos com extensão .cu são compilados pelo compilador nvcc distribuído pela NVIDIA. A divisão em arquivos é opcional desde que o arquivo tenha a extensão .cu para que o kernel e a chamada a ele sejam compilados pelo nvcc. Neste trabalho, o código foi dividido para maior visibilidade.

3.1 Esquema de divisão das *threads*

Para cada algoritmo foram elaborados esquemas de divisão das *threads.* A seguir, será apresentado como cada algoritmo utiliza cada *thread*.

Na multiplicação de matrizes, cada *thread* ficou responsável por uma parte da divisão feita pelo número de *threads*, ou seja, divide-se o tamanho da matriz pelo número de *threads*. No exemplo, se o tamanho da divisão for igual a 1, cada *thread* ficará responsável por uma linha da matriz A e da matriz resultante C.

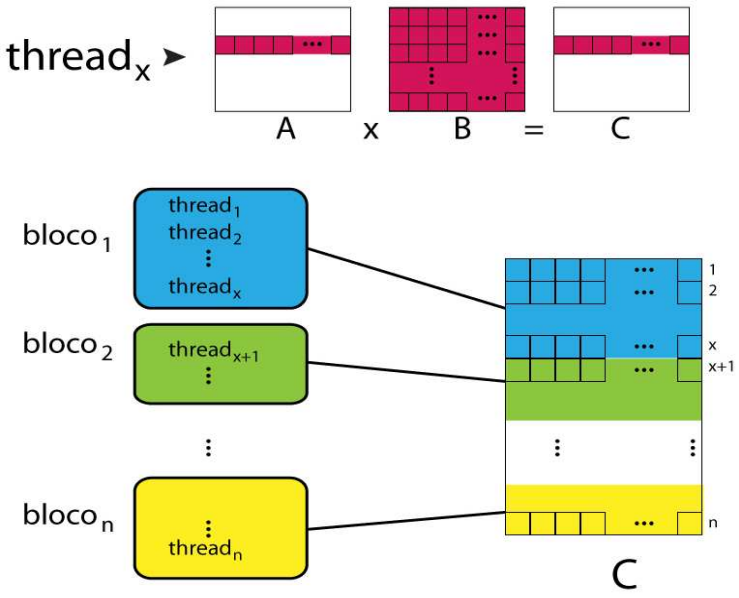


Fig 3.1 Divisão das *threads* para multiplicação de matrizes.

No produto escalar, se for considerado apenas uma dimensão, cada *thread* ficou responsável por cada elemento do vetor (coordenada dos vetores representados).

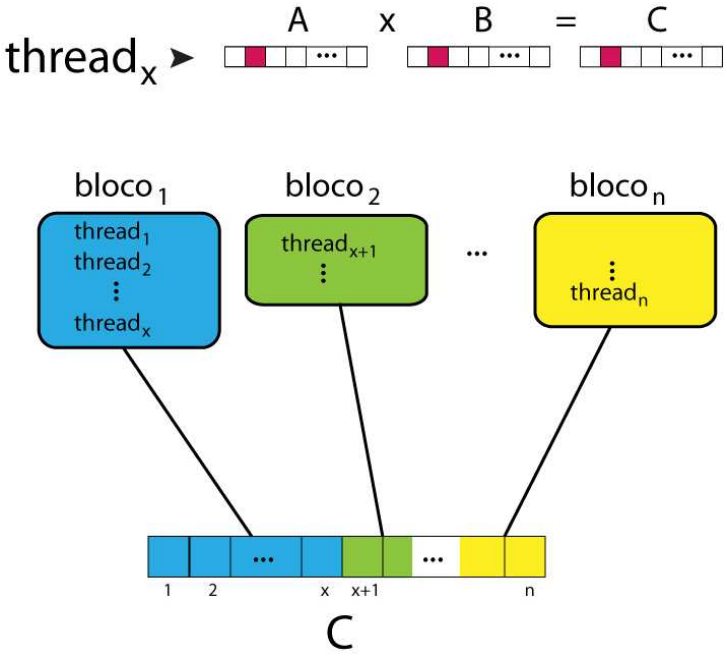


Fig 3.2 Divisão das *threads* para produto escalar.

Para o método de Jacobi-Ricardson cada *thread* ficou incubida de calcular o resultado de cada linha do sistema linear.

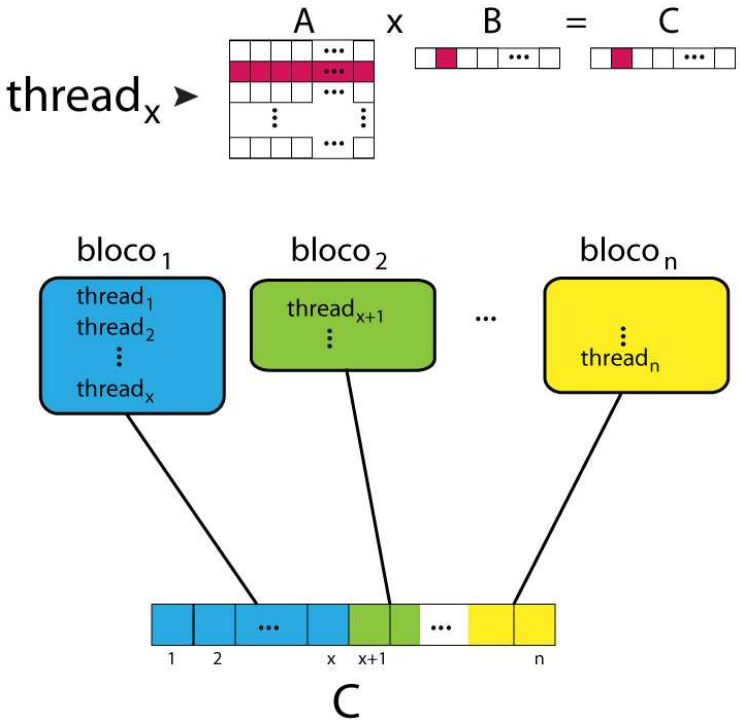


Fig 3.3 Divisão das *threads* para o método de Jacobi-Richardson.

E na ordenação bitônica cada *thread* foi utilizada para acessar cada elemento do vetor a ser ordenado.

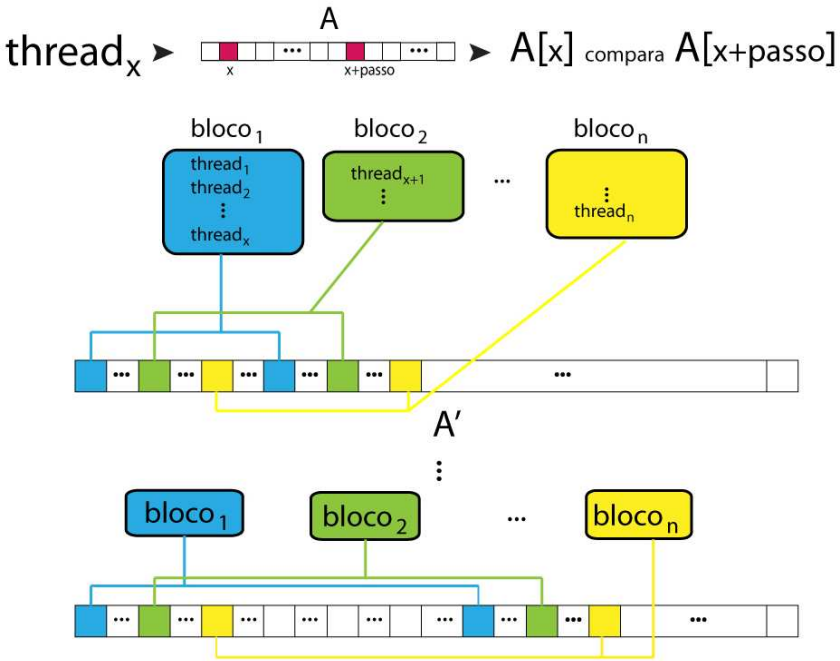


Fig 3.4 Divisão das *threads* para o método de ordenação bitônica.

3.2 Multiplicação de matrizes

•Matriz.cpp

Neste arquivo, contempla-se uma implementação de uma função corriqueira de multiplicação de matrizes. A função matrizMulCPU recebe como parâmetros dois ponteiros para as matrizes que serão multiplicadas (A e B), um ponteiro da matriz resultante (C) e o tamanho das matrizes. A variável soma é do tipo *double* para ter mais precisão durante a soma. No final, esta variável guarda o resultado na matriz C.

```
void
 matrizMulCPU (float* C, float* A, float* B, int tamanho)
{
      for (int i = 0; i < tamanho; ++i)
            for (int j = 0; j < tamanho; ++j) {
             double soma = 0;
                        for (int k = 0; k < tamanho; ++k) {
                         double a = A[i * tamanho + k];
                         double b = B[k * tamanho + j];
                         soma = a * b  + soma;
                        }
            C[i * tamanho + j] = (float)soma;
            }
}
```

Código 3.1: Matriz.cpp

•Matriz_kernel.cu

Este arquivo contém a implementação do núcleo (*kernel*) do funcionamento de CUDA. É nesta função que ocorre a multiplicação de matrizes utilizando os multiprocessadores da plataforma CUDA.

No início do código 3.2, define-se o tamanho das matrizes. O tamanho foi definido com a variável TAMANHO e o número de linhas e de colunas das matrizes é baseado nesta variável.

```
__global__ void matrizMulGPU( float* C, float* A, float* B)
{
int indice = threadIdx.x + blockDim.x * blockIdx.x;
int divisao = TAMANHO/(blockDim.x * gridDim.x);
if(TAMANHO<(blockDim.x * gridDim.x))
        div=1;
for ( int i = (indice); i<divisao; i++)
    for ( int j = 0; j < TAMANHO; j++) {
        double soma = 0;
        for ( int k = 0; k < TAMANHO; ++k) {
                double a = A[i + k];
                double b = B[k * TAMANHO + j];
                soma += a * b;
                }
        C[i + j] = (float)soma;
        }
}
```

Código 3.2: Matriz_kernel.cu

Define-se a variável índice como a junção das variáveis identificadora da *thread* (threadIdx.x), a dimensão do bloco (blockDim.x) e a identificadora do bloco (blockIdx.x), estas três são variáveis auxiliares embutidas na API, descritas na seção 2.6.2. A quantidade de *threads* será definida no outro arquivo Matriz.cu. Cada *thread* ficará responsável por fazer a multiplicação de uma parte da matriz resultante, salvo no caso de apenas uma *thread*, que será responsável por toda a multiplicação.

Assim, ocorre a multiplicação pertencente a *thread* correspondente. Para finalizar, ocorre a atribuição do resultado para a matriz C.

•Matriz.cu

No início do arquivo, após as inclusões das bibliotecas e do *kernel*, há os protótipos das funções de gerar matrizes com dados aleatórios, de imprimir as matrizes resultantes e de efetuar a multiplicação de maneira sequencial.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <cutil_inline.h>
#include <Matriz_kernel.cu>

void InicializarMatriz(float*, int, int);
void imprime(float*, int, int);
void matrizMulCPU(float*,  float*,  float*,  int);
int main(int argc, char** argv)
…
```

Código 3.3: Inicialização do Matriz.cu

A alocação de memória e a chamada para inicializar os dados das matrizes, que ficarão na CPU (h_A, h_B, h_C_CPU e h_C_GPU), acontecem logo a seguir no código 3.4. As matrizes h_C_CPU e h_C_GPU guardarão os resultados das multiplicações que acontecerão na CPU e GPU respectivamente.

```
 ...
float* d_A;
cutilSafeCall(cudaMalloc((void**) &d_A, mem_tamanho_A));
float* d_B;
cutilSafeCall(cudaMalloc((void**) &d_B, mem_tamanho_B));
float* d_C;
cutilSafeCall(cudaMalloc((void**) &d_C, mem_tamanho_C));
cutilSafeCall(cudaMemcpy(d_A, h_A, mem_tamanho_A,
                  cudaMemcpyHostToDevice) );
cutilSafeCall(cudaMemcpy(d_B, h_B, mem_tamanho_B,
                   cudaMemcpyHostToDevice) );
 ...
```

Código 3.4: Alocação de memória do Matriz.cu

Mais adiante, há a declaração, a alocação de memória e a atribuição dos valores gerados para as matrizes que ficarão na memória da GPU (d_A, d_B e d_C), através das funções cudaMalloc e cudaMemcpy.

```
...
const int max_threads_por_bloco = 256;
int blocos = TAMANHO/threads;
 if(max_threads_por_bloco==512)
        blocos=1;
 while(blocos*max_threads_por_bloco> 512)
        blocos--;
...
```

Código 3.5: Definição do tamanho do bloco do Matriz.cu

No código 3.5, é realizada a definição do tamanho máximo de *threads* por bloco (no exemplo, 256). De acordo com o tamanho das matrizes ocorre a divisão das *threads* em blocos. As verificações abaixo desta definição fixam a quantidade máxima de *threads* para 512 no somatório de todos os blocos. No próximo capítulo, será apresentado testes com diferentes configurações de quantidade de *threads* e blocos.

```
...
cutilCheckError(cutCreateTimer(&timer));
cutilCheckError(cutStartTimer(timer));

matrizMulGPU<<< blocos, max_threads_por_bloco >>>(d_C, d_A, d_B);

cutilCheckError(cutStopTimer(timer));
printf("Tempo de processamento da GPU: %f (ms) \n", cutGetTimerValue(timer));
cutilCheckError(cutDeleteTimer(timer));
…
```

Código 3.6: Chamada ao *kernel* do Matriz.cu

Assim, no código 3.6, há a declaração do contador e o inicia para registrar o tempo de processamento utilizando CUDA. O contador utilizado é uma das funções já existentes em CUDA, a função cutCreateTimer após a execução da função matrizMulGPU pertencente ao *kernel*, o contador é parado, impresso na tela e excluído. O mesmo acontece para a função sequencial. Assim, as matrizes são liberadas da memória, tanto da CPU, quanto da GPU para encerrar o programa.

3.3 Produto escalar

•Produto.cpp

Neste arquivo, há o produto escalar programado de maneira sequencial.

```
void ProdutoEscalarSequencial (float *h_C, float *h_A, float *h_B, int vetorN, int
dimensaoN){
   for(int vet = 0; vet < vetorN; vet++){
      int vetorInicio = dimensaoN * vet;
      int vetorFim  = vetorInicio + dimensaoN;
      double soma = 0;
      for(int pos = vetorInicio; pos < vetorFim; pos++)
          soma += h_A[pos] * h_B[pos];
      h_C[vet] = (float)soma;
   }
}
```

Código 3.7: Produto.cpp

A função ProdutoEscalarSequencial recebe como parâmetros três vetores, o número de vetores que participarão do produto e o número de dimensões das coordenadas que definirão os vetores. Um dos vetores recebidos como parâmetro será o responsável por guardar os resultados dos produtos escalares efetuados. Os outros dois vetores recebidos terão as informações referentes às coordenadas em cada dimensão dos vetores.

•Produto_kernel.cu

Este arquivo de *kernel* define como ocorrerá o produto escalar utilizando CUDA. No início, há a definição do tamanho do vetor que guardará o resultado dos produtos escalares.

```
#define TAMANHO_VET 1024
__global__ void ProdutoEscalarGPU(float *d_C,float *d_A,float *d_B,int vetorN,int
dimensaoN){

   __shared__ float resultado[TAMANHO_VET];
   for(int vet = blockIdx.x; vet < vetorN; vet += gridDim.x){
      int vetorInicio = dimensaoN*vet;
      int vetorFim  = vetorInicio + dimensaoN;
      for(int tx = threadIdx.x; tx < TAMANHO_VET; tx += blockDim.x){
         float soma = 0;
         for(int pos = vetorInicio + tx; pos < vetorFim; pos += TAMANHO_VET)
                soma += d_A[pos] * d_B[pos];
         resultado[tx] = soma;
      }

     for(int divisao = TAMANHO_VET / 2; divisao > 0; divisao = divisao/2){
         for(int tx = threadIdx.x; tx < divisao; tx += blockDim.x)
            resultado[tx] += resultado[divisao + tx];
     }
      if(threadIdx.x == 0)
                d_C[vet] = resultado[0];
   }
}
```

Código 3.8: Produto_kernel.cu

A função ProdutoEscalarGPU é bem parecida com a função sequencial. Ela começa com a definição de uma variável do tipo __shared__ que será o vetor para os resultados. E de maneira semelhante a sequencial, calcula-se os passos para o produto escalar dos vetores. Cada thread fica responsável por cada coordenada e assim faz-se o produto escalar. No final, ocorre as somas das partes da multiplicação. É feito um ajuste para que o produto tenha o resultado correto. Este ajuste é necessário porque o resultado da multiplicação das coordenadas de dimensões diferentes se distanciam pelo tamanho do bloco das *threads.*

•Produto.cu

Neste arquivo, de maneira similar ao de multiplicação de matrizes, define-se a quantidade de vetores e a quantidade de dimensões para que se faça a multiplicação (no exemplo, 100 para ambos).

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <cutil_inline.h>
#include <Produto_kernel.cu>

void ProdutoEscalarSequencial(float *h_C, float *h_A, float *h_B, int vetorN,
intdimensaoN);
float InicializarVetor(float min, float max);

const int VETOR_N = 100;
const int DIMENSAO_N = 100;
const int DADOS_N = VETOR_N * DIMENSAO_N;
const int DADOS_TAM = DADOS_N * sizeof(float);
const int RESULT_TAM = VETOR_N  * sizeof(float);
const int max_threads_por_bloco = 512;

int main(int argc, char **argv){
…
```

Código 3.9: Inicialização do Produto.cu

No código 3.10, acontece a declaração, alocação de memória e inicialização dos vetores em que se deseja fazer o produto escalar.

```
…
cutilCheckError(cutCreateTimer(&timer));
cutilCheckError(cutCreateTimer(&timer2));

h_A     = (float *)malloc(DADOS_TAM);
h_B     = (float *)malloc(DADOS_TAM);
h_C_CPU = (float *)malloc(RESULT_TAM);
h_C_GPU = (float *)malloc(RESULT_TAM);

cutilSafeCall(cudaMalloc((void **)&d_A, DADOS_TAM));
cutilSafeCall(cudaMalloc((void **)&d_B, DADOS_TAM));
cutilSafeCall(cudaMalloc((void **)&d_C, RESULT_TAM));

cutilSafeCall(cudaMemcpy(d_A, h_A, DADOS_TAM, cudaMemcpyHostToDevice));
cutilSafeCall(cudaMemcpy(d_B, h_B, DADOS_TAM, cudaMemcpyHostToDevice));
…
```

Código 3.10: Alocação de memória do Produto.cu

Acontece a alocação de memória para os vetores na GPU, e assim, se inicia o contador para execução do produto escalar utilizando CUDA. Após a execução do *kernel* para sua avaliação, o mesmo acontece para a função no modelo sequencial. A impressão dos tempos de processamento dos dois métodos aparece na tela. Finalmente, acontece a limpeza de memória depois da utilização das matrizes e contadores para terminar o programa.

3.4 Método de Jacobi-Richardson

•Jacobi-Richardson.cpp

Este arquivo tem o código do método de Jacobi-Richrdson implementado de maneira sequencial. A função JacobiRichardsonCPU recebe como parâmetros, considerando o sistema linear Ax=b, a matriz A, o vetor x, o vetor b, a quantidade de linhas N e o limite de erro máximo de truncamento.

```
void JacobiRichardsonCPU(float* resultado, float* A, float* x, float* b, int TAMANHO,
float erro)
{
	float numerador=1, denominador=1;
	int indice =0;
	while((numerador/denominador)>erro){
		resultado[indice] = 0;
		numerador=0;
		denominador=0;
		for (int k = 0; k < TAMANHO; ++k)
		{
			if(indice!=k)
				resultado[indice] += A[(indice*TAMANHO)+k] * x[k];
		}
		resultado[indice]=1/A[(indice*TAMANHO)+indice]*(b[indice]-
resultado[indice]);
		if(numerador<abs(abs(resultado[indice])-abs(x[indice])))
			numerador=abs(abs(resultado[indice])-abs(x[indice]));
		if(denominador<abs(resultado[indice]))
			denominador=abs(resultado[indice]);
		x[indice]=resultado[indice];
		indice++;
		if(indice==TAMANHO)
			indice=0;
		}
	}
```

Código 3.11: Jacobi-Richardson.cpp

Assim, é feito o cálculo do novo resultado a partir do antigo resultado (vetor x), e assim, calcula-se a diferença entre o resultado antigo e o novo, que servirá para a comparação com o erro máximo de truncamento.

Finalmente, ocorre a cópia do novo resultado para o vetor x, e começa-se o algoritmo novamente até que se atinja o resultado com a precisão desejada.

•Jacobi-Richardson_kernel.cu

Este arquivo contém a implementação do método itetativo de Jacobi-Richardson utilizando CUDA. No início, acontece a definição dos valores de erro máximo e do tamanho do sistema (neste trabalho, o erro máximo adotado foi 0.00001).

```
#include <stdlib.h>
#include <math.h>
#define TAMANHO 256
#define ERRO 0.00001

__global__ void JacobiRichardsonGPU(float* resultado, float* A, float* x, float* b)
{
int indice = threadIdx.x + blockDim.x * blockIdx.x;
        if(indice<TAMANHO){
            __shared__ float numerador;
            __shared__ float denominador;
            numerador=1;
            denominador=1;
            while((numerador/denominador)>ERRO){
                resultado[indice] = 0;
                numerador=0;
                denominador=0;
                for (int k = 0; k < TAMANHO; ++k)
                {
                        if(indice!=k)
                                resultado[indice] += A[(indice*TAMANHO)+k] * x[k];
                }
                resultado[indice]=1/A[(indice*TAMANHO)+indice]*(b[indice]-
resultado[indice]);
                if(numerador<abs(abs(resultado[indice])-abs(x[indice])))
                        numerador=abs(abs(resultado[indice])-abs(x[indice]));
                if(denominador<abs(resultado[indice]))
                        denominador=abs(resultado[indice]);
                x[indice]=resultado[indice];
                }
            }
}
```

Código 3.12: Jacobi-Richardson_kernel.cu

O método JacobiRichardsonGPU recebe como parâmetros o vetor que servirá para guardar o resultado, a matriz A e os vetores x e b, considerando novamente o sistema linear Ax=b.

De maneira análoga ao algoritmo sequencial, as variáveis numerador e denominador servirão para o cálculo do erro de truncamento, e no decorrer da implementação ocorre o cálculo da nova solução do sistema baseado na solução anterior. Assim, é feito o cálculo da diferença do resultado antigo e novo para posteriormente constatar se atende ao limite de erro máximo de truncamento.

Finalmente, acontece a cópia do resultado novo para o vetor x.

•Jacobi-Richardson.cu

No início deste arquivo, acontece a prototipação das funções de inicializar os valores da matriz A e dos vetores x e b, de testar a convergência de linha e da coluna, e a de cálculo do resultado pelo método executado na CPU.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <cutil_inline.h>
#include <Jacobi-Richardson_kernel.cu>
void InicializarValores(float*, float*,float*);
bool TestarConvergencia(float*);
void JacobiRichardsonCPU(float* , float*, float* , float *, int,float);
int main(int argc, char** argv)
{
   srand(20);

   int size_A = TAMANHO * TAMANHO;
   int mem_size_A = sizeof(float) * size_A;
   float* h_A = (float*) malloc(mem_size_A);

   int size_x = TAMANHO;
   int mem_size_x = sizeof(float) * size_x;
   float* h_x = (float*) malloc(mem_size_x);

   int size_b = TAMANHO;
   int mem_size_b = sizeof(float) * size_b;
   float* h_b = (float*) malloc(mem_size_b);

   int size_y = TAMANHO;
   int mem_size_y = sizeof(float) * size_y;
   float* h_y_CPU = (float*) malloc(mem_size_y);
   float* h_y_GPU = (float*) malloc(mem_size_y);

   InicializarValores(h_A,h_x,h_b);
   …
```

Código 3.13: Inicialização do Jacobi-Richardson.cu

Ainda no código 3.13, acontece a alocação de memória para a matriz h_A e os vetores h_x, h_b, h_y_CPU e h_y_GPU. O vetor h_y_CPU guardará o resultado do cálculo realizado na CPU e o vetor h_y_GPU guardará o resultado da GPU. Finalmente, acontece a chamada para inicializar os valores do sistema.

```
if(TestarConvergencia(h_A))
{
        float* d_A;
        cutilSafeCall(cudaMalloc((void**) &d_A, mem_size_A));
        float* d_x;
        cutilSafeCall(cudaMalloc((void**) &d_x, mem_size_x));
        float* d_b;
        cutilSafeCall(cudaMalloc((void**) &d_b, mem_size_b));
        float* d_y;
        cutilSafeCall(cudaMalloc((void**) &d_y, mem_size_y));

        cutilSafeCall(cudaMemcpy(d_A, h_A, mem_size_A,
            cudaMemcpyHostToDevice) );
        cutilSafeCall(cudaMemcpy(d_x, h_x, mem_size_x,
            cudaMemcpyHostToDevice) );
        cutilSafeCall(cudaMemcpy(d_b, h_b, mem_size_b,
            cudaMemcpyHostToDevice) );
        ...
```

Código 3.14: Alocação de memória do Jacobi-Richardson.cu

Em seguida, no código 3.14, ocorre o teste de convergência. Para o sistema convergir, basta que apenas um dos critérios (linhas ou das colunas) seja satisfeito. Se pelo menos um dos dois for satisfeito, acontece a alocação de memória na GPU para a matriz d_A e os vetores d_x, d_b e d_y e posteriormente, acontece a cópia dos dados da CPU para GPU.

```
        …
        const int max_threads_por_bloco = 512;
        int blocos = TAMANHO / max_threads_por_bloco;
        …
```

Código 3.15: Definição do tamanho do bloco do Jacobi-Richardson.cu

No código 3.15, acontece a definição do tamanho máximo de *threads* por bloco (no exemplo, 512). Como cada *thread*, ficará responsável por uma linha do sistema, assim o número de *threads* é o mesmo que o de linhas. Logo após, inicia-se o contador para as execuções na GPU e CPU, imprime-se os valores dos contadores. Finalmente, acontece a liberação de memória tanto na CPU, quanto na GPU para a finalização do programa.

3.5 Método de ordenação bitônica

•Ordenacao.cpp

Para fazer comparação com o método de ordenação bitônica programado em paralelo, foi implementado o método na maneira sequencial.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
void OrdenacaoCPU(int tamanho, float* V, int passo, int limite)
  {
        for(int indice=0;indice<tamanho;indice++){
              int elem = indice^passo;
              if ((elem)>indice){
                    if ((indice&limite)==0 &&  V[indice]> V[elem]) { //ordenacao crescente
                          int temp =  V[indice];
                          V[indice] = V[elem];
                          V[elem] = temp;
                    }
                   if ((indice&limite)!=0 &&  V[indice]< V[elem]){ //ordenacao decrescente
                          int temp =  V[indice];
                          V[indice] =  V[elem];
                          V[elem] = temp;
                    }
              }
  }}
```

Código 3.16: Ordenacao.cpp

O método consiste em ordenar o vetor de acordo com os parâmetros passo e limite passados. A variável indice tem função de percorrer o vetor. De acordo com a variável passo, o vetor é percorrido de forma que cada elemento seja verificado apenas uma vez, ou seja, através da operação OU exclusivo, o elemento elem do vetor é selecionado para comparação de acordo com a variável passo. A variável limite condiz com o limite entre a ordenação crescente ou decrescente para aquela iteração, ou seja, se a variável índice for comparada com a variável limite bit a bit, com a operação E, ora a ordenação ocorre crescentemente, ora a ordenação ocorre de maneira inversa.

•Ordenacao_kernel.cu

Este arquivo é o *kernel* para execução do método de ordenação bitônica na GPU.

```
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#define TAMANHO 512

__global__ void OrdenacaoGPU(float * V, int passo, int limite)
{
	int indice = threadIdx.x + blockDim.x * blockIdx.x;
	int elem = indice^passo;
	if ((elem)>indice){
		if ((indice&lmite)==0 && V[indice]>V[elem]) {
			int temp = V[indice];
			V[indice] = V[elem];
			V[elem] = temp;
		}
		if ((indice&limite)!=0 && V[indice]<V[elem]){
			int temp = V[indice];
			V[indice] = V[elem];
			V[elem] = temp;
		}
	}
}
```

Código 3.17: Ordenacao_kernel.cu

O algoritmo começa com a definição do tamanho do vetor a ser ordenado. A variável TAMANHO será utilizada somente no arquivo Ordenacao.cu, onde será alocada a memória para o vetor.

A função OrdenacaoGPU recebe como parâmetros o vetor e os índices para acessar os elementos do vetor.

A variável elem é o resultado da operação XOR entre a variável indice e passo. Ela serve para controlar o acesso a determinado elemento do vetor. A seguir, ocorre a ordenação crescente ou descrescente de acordo com as variáveis limite e indice.

•Ordenacao.cu

Este arquivo contém o programa principal das ordenações e as chamadas para as funções descritas acima.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <cutil_inline.h>
#include <Ordenacao_kernel.cu>

void InicializaVetor(float*);
void OrdenacaoCPU(int, float*, int , int );
int  main(int argc, char** argv)
{
```

Código 3.18: Inicialização do Ordenacao.cu

Ele começa declarando a função de gerar um vetor aleatório e a função de ordenação pela CPU.

```
int size_A = TAMANHO;
int mem_size_A = sizeof(float) * size_A;
float* h_A = (float*) malloc(mem_size_A);
float* h_x = (float*) malloc(mem_size_A);

InicializaVetor(h_A);
float* d_x;
cutilSafeCall(cudaMalloc((void**) &d_x, mem_size_A));
cutilSafeCall(cudaMemcpy(d_x, h_A, mem_size_A,
        cudaMemcpyHostToDevice) );
```

Código 3.19: Alocação de memória do Ordenacao.cu

No código 3.19, acontece a declaração e alocação da memória dos vetores na CPU e na GPU. O vetor h_A e h_x são alocados na CPU e d_x é alocado na GPU. O vetor d_x será o vetor ordenado pela GPU. E o vetor h_A será ordenado pela CPU. O vetor h_x guardará o resultado depois da ordenação de d_x.

```
unsigned int timer = 0;
cutilCheckError(cutCreateTimer(&timer));
cutilCheckError(cutStartTimer(timer));

for (int limite = 2; limite <= TAMANHO; limite =limite*2)
        for (int passo=limite/2; passo>0; passo=passo/2)
          OrdenacaoGPU<<<TAMANHO/max_threads_por_bloco,
max_threads_por_bloco>>>(d_x, passo, limite);

cutilCheckError(cutStopTimer(timer));
printf("Tempo de processamento da GPU: %f ms. \n", cutGetTimerValue(timer));
cutilCheckError(cutDeleteTimer(timer));
```

Código 3.20: Chamada ao kernel do Ordenacao.cu

No código 3.20, encontra-se a execução do *kernel*. O método OrdenacaoGPU é executado, e o contador calcula o tempo de sua execução.

```
unsigned int timer2 = 0;
cutilCheckError(cutCreateTimer(&timer2));
cutilCheckError(cutStartTimer(timer2));
for (int limite = 2; limite <= TAMANHO; limite = limite *2)
        for ( int passo= limite /2; passo>0; passo=passo/2)
OrdenacaoCPU(TAMANHO,h_A,passo,limite);

cutilCheckError(cutStopTimer(timer2));
printf("Tempo de processamento da CPU: %f ms. \n", cutGetTimerValue(timer2));
cutilCheckError(cutDeleteTimer(timer2));
```

Código 3.21: Chamada a função sequencial do Ordenacao.cu

No código 3.21, outro contador calcula o tempo para a execução do método de ordenação na CPU. Assim, libera-se os vetores da memória e termina o programa.

Este capítulo apresentou a implementação dos algoritmos de multiplicação de matrizes, do produto escalar e do método iterativo de Jacobi-Richardson, utilizando CUDA e de maneira sequencial. No próximo capítulo, serão apresentados os resultados dos testes de desempenho em relação a estes algoritmos.

4.TESTES E AVALIAÇÃO DE DESEMPENHO

Neste capítulo são analisados os diversos aspectos relevantes aos dados obtidos na utilização de CUDA.

Para a avaliação do desempenho foi utilizada um notebook com processador Intel Core 2 Duo P8700 (2,53 GHz, FSB 1066 MHz, 3 MB Cache), com 4 GB de RAM 800 MHz DDR2, HD SATA 5400RPM, para a medição do algoritmo sequencial. Para a versão paralela foi utilizada uma GPU NVIDIA GeForce GT 130M 1 GB 500 MHz DDR2 / 800 MHz GDDR3 (1500 MHz, 1024MB DDR2@500MHz), a qual possui 4 multiprocessadores de threads, cada qual com 8 processadores escalares, o que resulta em 32 processadores.

Os softwares utilizados foram o Microsoft Windows 7 64 bits, Microsoft Visual Studio 2008 Professional e o SDK versão 2.3 da NVIDIA exclusivamente para programar em CUDA.

Os testes foram elaborados utilizando na forma Release, ou seja, sem emulação e sem o modo debug com e sem emulação.

A seção 4.1 trata da análise da corretude da versão paralela e a seção 4.2 ilustra o ganho obtido com esta versão em relação à versão sequencial.

4.1 Análise das saídas da implementação CUDA

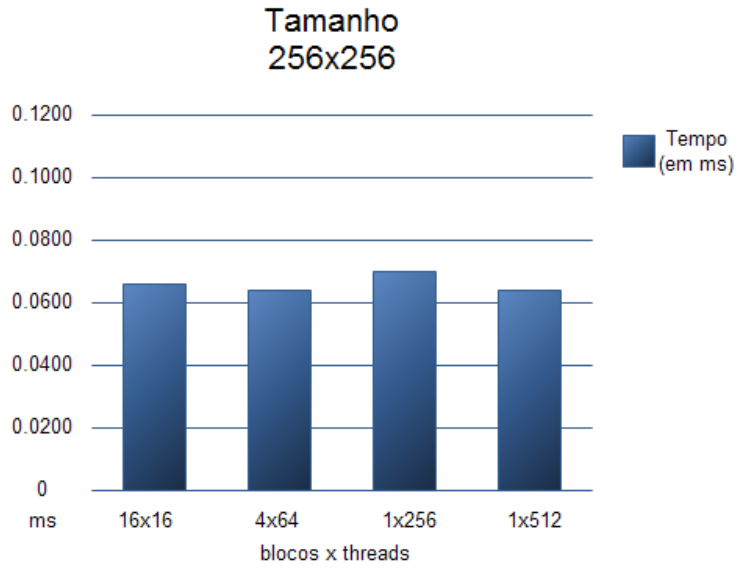
A análise das saídas da implementação CUDA fez-se através da comparação das linhas impressas de saída. Foi feita a comparação linha a linha dos resultados. A implementação paralela foi considerada correta, pois, apresentou apenas diferenças de precisão a partir da quinta casa decimal quando comparada com o resultado da implementação sequencial. Cabe ressaltar que já eram esperadas divergências de precisão devido aos motivos descritos na seção 2.8, considerando que escolheu-se por utilizar a implementação de hardware da notação de ponto flutuante a fim de obter o máximo desempenho possível.

Não foram detectadas divergências em nenhuma saída das diversas configurações de grids e blocos da implementação paralela testadas.

4.2 Comparação de desempenho utilizando CUDA

O método de avaliação do desempenho da implementação CUDA foi o tempo necessário do processamento do kernel. Na versão sequencial, foi o tempo para execução da função responsável para produzir resultado similar. Para a implementação CUDA, diversas configurações de grid foram testadas. Cada algoritmo foi executado diversas vezes e foi considerado o melhor tempo entre as execuções, para todas as situações.

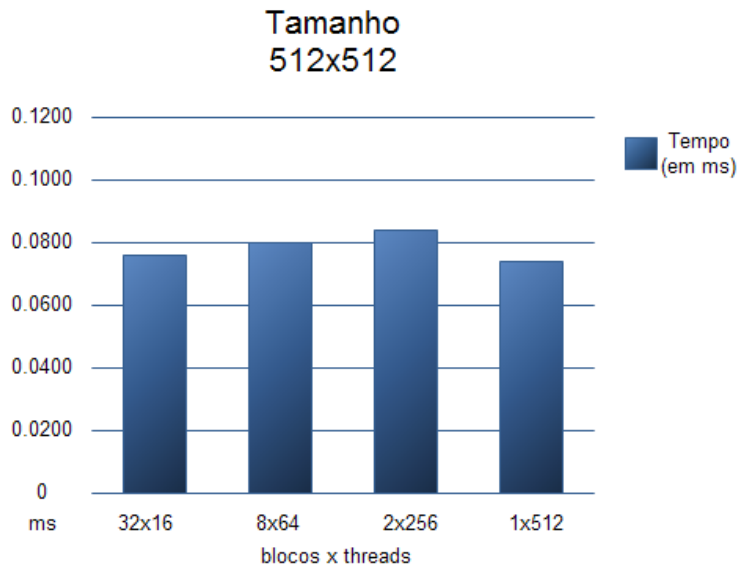
•Na multiplicação de matrizes:



	Sequencial	**16x16**	**4x64**	**1x256**	**1x512**
Tempo (ms):	153,6897	0,0672	0,0668	0,0696	0,0664

	Menor *Speedup*	**Maior *Speedup***
Speedup:	2208,1853	2314,6039

Fig4.1: Desempenho CUDA com matriz 256x256.

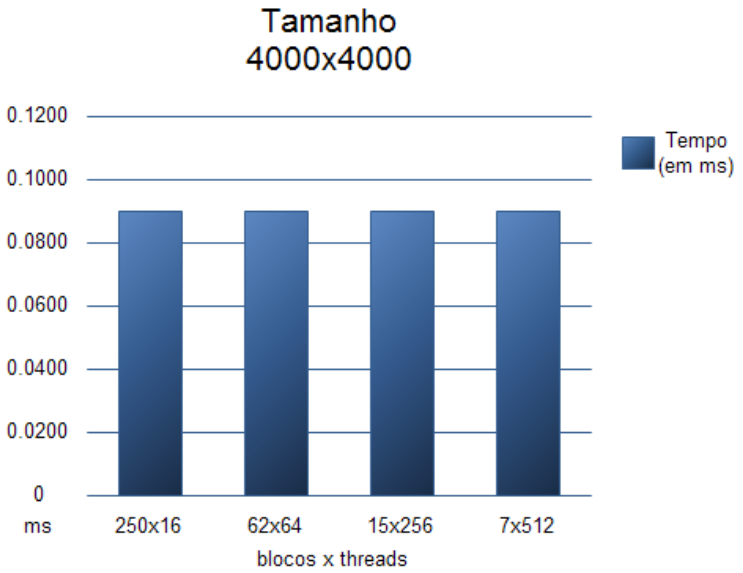


	Sequencial	**32x16**	**8x64**	**2x256**	**1x512**
Tempo (ms):	1279,5688	0,0765	0,0798	0,08266	0,0741

	Menor *Speedup*	**Maior *Speedup***
Speedup:	15479,9032	17268,1349

Fig4.2: Desempenho CUDA com matriz 512x512.

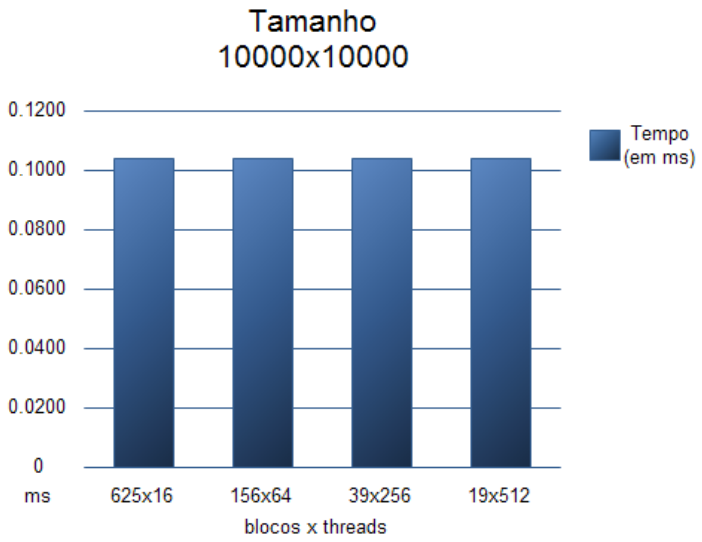
•No produto escalar:



	Sequencial	**250x16**	**62x64**	**15x256**	**7x512**
Tempo (ms):	79,9383	0,0984	0,0972	0,0988	0,0960

	Menor *Speedup*	**Maior *Speedup***
***Speedup*:**	808,5118	832,3879

Fig4.3: Desempenho do produto escalar 4000x4000.

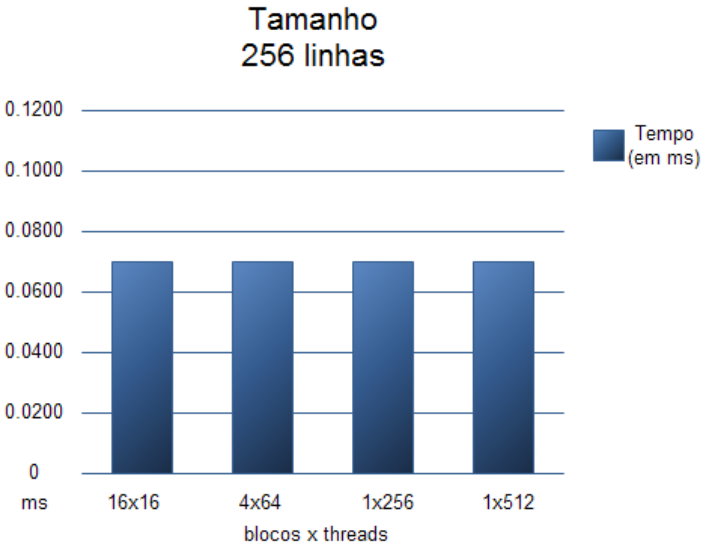


	Sequencial	**625x16**	**156x64**	**39x256**	**19x512**
Tempo (ms):	502,4039	0,1029	0,1045	0,1061	0,1041

	Menor *Speedup*	**Maior *Speedup***
***Speedup*:**	4732, 2931	4881,3095

Fig4.3: Desempenho do produto escalar 4000x4000.

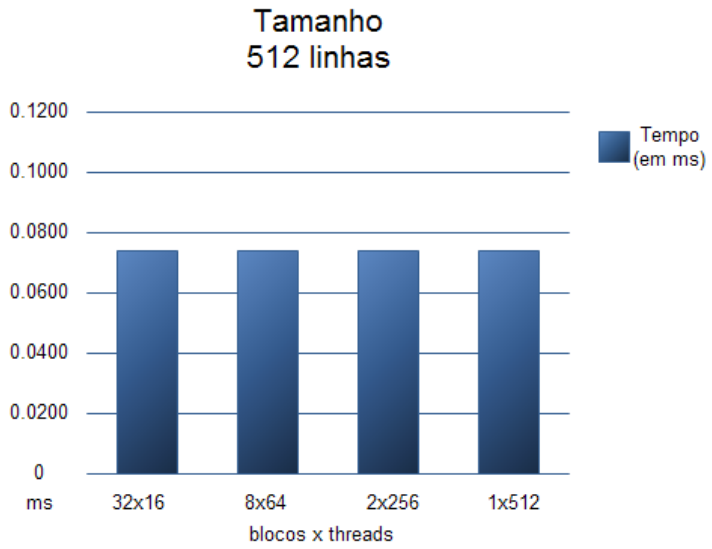
•No método de Jacobi-Richardson:



	Sequencial	**16x16**	**4x64**	**1x256**	**1x512**
Tempo (ms):	5,3808	0,0676	0,068	0,0672	0,0688

	Menor *Speedup*	**Maior *Speedup***
***Speedup*:**	78,2093	80,0714

Fig4.5: Desempenho da resolução de 256 linhas.

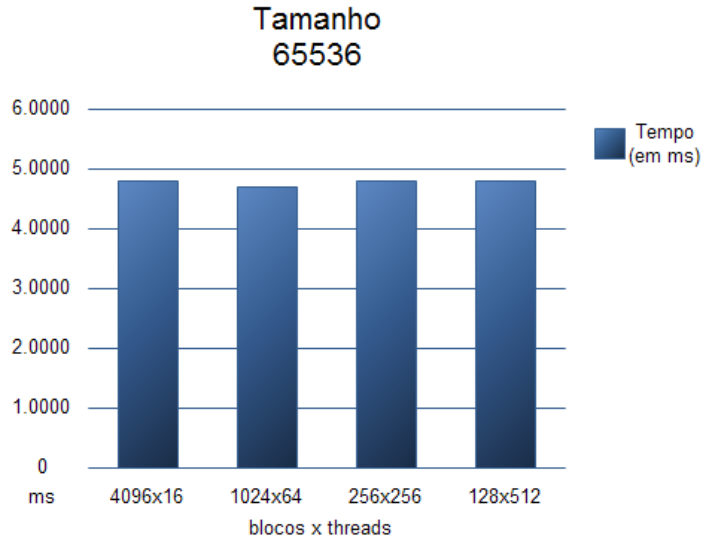


	Sequencial	**32x16**	**8x64**	**2x256**	**1x512**
Tempo (ms):	21,2859	0,0741	0,0721	0,0713	0,0696

	Menor *Speedup*	**Maior *Speedup***
***Speedup*:**	287,2591	305,8318

Fig4.6: Desempenho da resolução de 512 linhas.

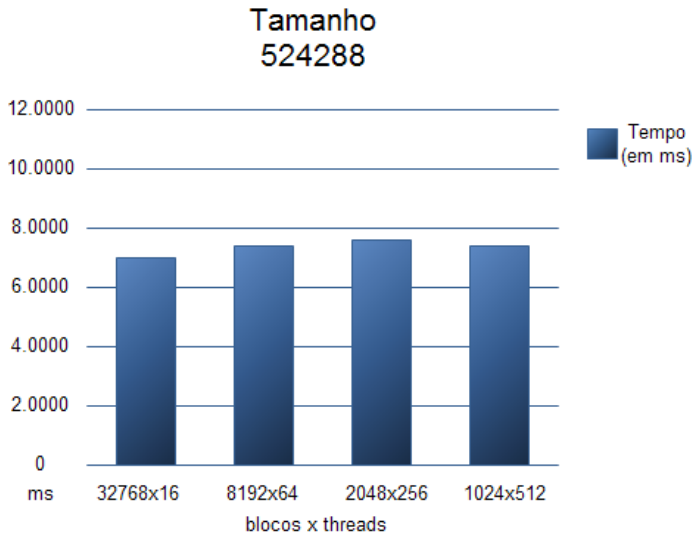
•Na ordenação bitônica:



	Sequencial	**4096x16**	**1024x64**	**256x256**	**128x512**
Tempo (ms):	71,9447	4,9586	4,8666	4,9095	4,9642

	Menor *Speedup*	**Maior *Speedup***
***Speedup*:**	14,4924	14,7832

Fig4.7: Desempenho da ordenação de 65536 elementos.



	Sequencial	**32768x16**	**8192x64**	**2048x256**	**1024x512**
Tempo (ms):	769,433	7,0194	7,2253	7,6033	7,2378

	Menor *Speedup*	**Maior *Speedup***
***Speedup*:**	101,1972	109,6152

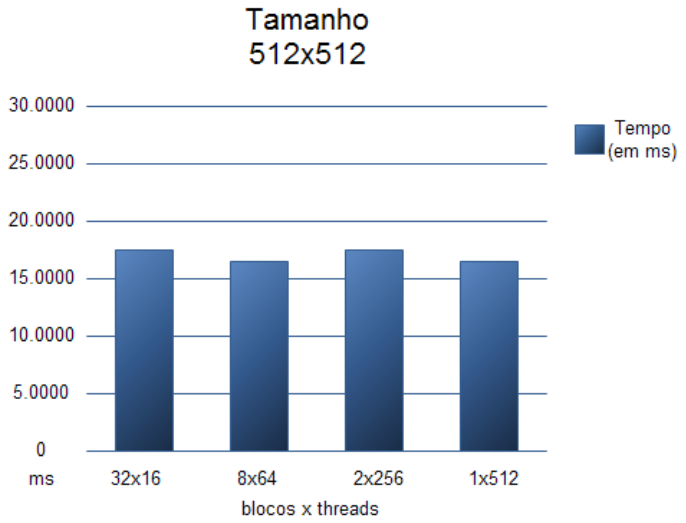
Fig4.8: Desempenho da ordenação de 524288 elementos.

Analisando, os *speedup*s anteriores, temos que todos os algoritmos apresentam *speedup*s próximos, ou no caso, maiores que o número de processadores empregados. Lembrando que a GPU utilizada possui 32 processadores, e que quando o número de blocos excede este número, ele é dividido entre os processadores. Isto se repete em todos os algoritmos, porém na ordenação, em particular, o rendimento não foi igualmente excelente, porém ainda assim bastante satisfatório dependendo do tamanho do vetor a ser ordenado.

Durante os testes, foi constatado que com a restrição de memória que a placa utilizada no trabalho possui, ocorre a restrição do tamanho da quantidade de dados a ser processado. A restrição acontece pelo limite de quantidade de memória utilizada, que é variável de acordo com o modelo de placa de vídeo.

Empiricamente, constatou-se que se, por exemplo, uma matriz tiver tamanho 900x900 ou menos, com esta placa, pode-se utilizar o algoritmo elaborado no trabalho sem problemas. Para se efetuar multiplicações de matrizes maiores, deve-se alterar o algoritmo do *kernel*, de forma a diminuir sua complexidade. Uma alternativa válida é retirar um laço de repetição do código do *kernel* e o colocar na função principal antes da chamada da função do *kernel*, porém esta possibilidade afeta consideravelmente o

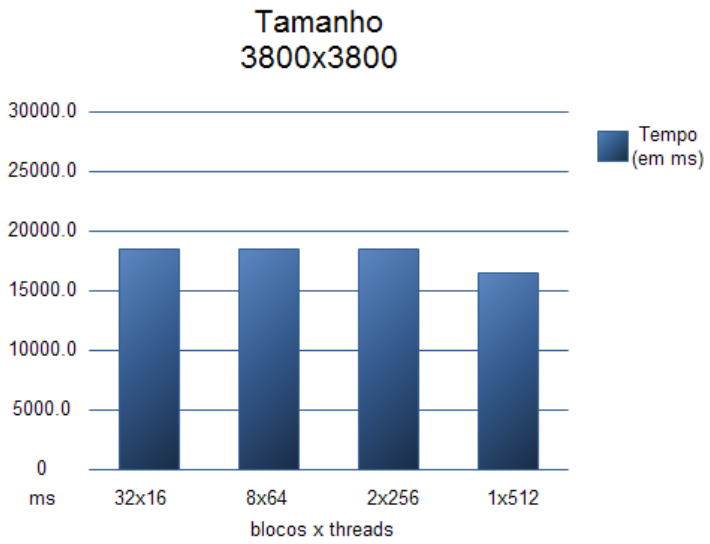
desempenho. Para exemplificar, utilizou-se de uma quantidade fixa de blocos (32 blocos de 16 threads, 8 blocos de 64, 2 blocos de 256 e 1 bloco de 512), para testar uma multiplicação de matriz de tamanho 512x512 e uma de 3800x3800. O código do kernel encontra-se no apêndice V.



	Sequencial	**16x16**	**8x64**	**2x256**	**1x512**
Tempo (ms):	1279,5688	17,5295	16,9257	17,2009	16,8897

	Menor *Speedup*	**Maior *Speedup***
***Speedup*:**	72,9951	75,7603

Fig4.9: Desempenho CUDA com matriz 512x512 com *kernel* modificado.



	Sequencial	**32x16**	**8x64**	**2x256**	**1x512**
Tempo (ms):	1599060,375	17804,9785	17904,3105	17903,2636	16295,5

	Menor *Speedup*	**Maior *Speedup***
***Speedup*:**	89,3114	98,1289

Fig4.10: Desempenho CUDA com matriz 3800x3800 com *kernel* modificado.

No geral, a melhor configuração é utilizando o número máximo de threads num bloco, ou seja, 512. Isto é facilmente percebido porque se utilizarmos um número maior de threads em cada bloco, serão necessários menos blocos e consequentemente menos processadores, e como no desempenho entre as configurações testadas não houve grandes discrepâncias, a configuração com melhor rendimento é com 512 threads em cada bloco.

5. CONCLUSÃO

O uso da computação paralela apresenta uma demanda crescente, visto o número cada vez maior de investimento em placas de vídeo novas, com maior capacidade de processamento.

Ao passo que, as placas de vídeo promovem a popularização do acesso a computação paralela, oferecendo preços mais acessíveis a elas, o mercado fica cada vez mais ansioso pela próxima placa de vídeo e quais seriam suas novidades. Parte dessa busca por aperfeiçoar cada vez mais as placas de vídeo, vem da concorrência entre a ATI e a NVIDIA. Estas duas empresas são as principais produtoras de placas de vídeos, e elas tendem a melhorar ainda mais com as parcerias (Apple e Intel com a NVIDIA e a AMD detentora da ATI).

Neste trabalho, foi utilizado a tecnologia existente nas placas de vídeo recentes da NVDIA, a tecnologia CUDA. CUDA possui uma API com uma boa documentação e já possui diversos trabalhos sobre esta plataforma. Apesar de algumas restrições que atrapalham um pouco o desenvolvimento (limitação de parâmetros para a função do *kernel*, proibido uso de recursão, uso da memória, etc.), é uma ferramenta versátil que atende diversos fins computacionais, exigindo um pouco de dedicação por parte do programador.

É com bastaste satisfação que se elaborou este trabalho, pelo fato de explorar uma ferramenta nova e com um poder computacional muito grande, além da facilidade de aprendizado e de manipulação com as *threads* e com os dados.

O trabalho tratou quatro algoritmos basicamente: a multiplicação de matrizes e o produto escalar, que são amplamente conhecidos, o método iterativo de Jacobi-Richardson para solução de sistemas lineares e o método de ordenação bitônica.

Existem trabalhos relacionados ao algoritmo de Jacobi-Richardson com placas mais atuais do que a usada neste trabalho [ZHANG , 2009] e [WANG, 2009], porém, foram elaborados testes somente neste algoritmo em particular, e na plataforma desktop. Neste trabalho, foi testado apenas o caso com precisão de 5 casas decimais e com tamanho de sistemas menores pelas restrições de memória da placa utilizada no trabalho.

Há idéias novas de se aproveitar o algoritmo de multiplicação de matrizes de forma a ter um desempenho melhor em [FUJIMOTO, 2008] e [CUI, 2009], porém utilizamos a forma mais conhecida e popular da implementação.

Os testes apresentaram resultados bem expressivos. À medida que a quantidade de dados for maior, os resultados se apresentam melhores comparados com a CPU.

CUDA tem uma enorme gama de aplicações e esta ferramenta será de grande valia para resolução de problemas computacionais de grande complexidade. CUDA merece um grande destaque para seu aprendizado, porque, com certeza, fará parte do futuro do processamento de dados.

5.1 Trabalhos futuros

CUDA, como dito anteriormente, possui uma grande possibilidade de campos de pesquisa. Diversos algoritmos paralelizáveis, podem ser migrados para trabalhar com CUDA. Existem demandas nas mais diversas áreas que precisam de um grande poder computacional.

Como propostas de trabalhos futuros, fica a sugestão de pesquisa de quais áreas já utilizam CUDA no cotidiano, como e quanto a utilização de CUDA ajuda/ajudou estas áreas, quanto financeiramente e computacionalmente a utilização de CUDA poupa recursos e qual o fator determinante para a utilização desta tecnologia.

A implementação de algoritmos otimizados para a plataforma CUDA e a análise de comparação com outras plataformas computacionais, paralelas ou não, também são idéias que possam continuar este trabalho.

Há também abstrações de CUDA para outras linguagens como Java (jCUDA), C# (CUDA.NET) e também Python (PyCUDA), que podem ser testadas e avaliadas com seus pontos fortes e fracos em determinadas implementações de algoritmos.

Há muita pesquisa com bioinformática e com processamento digital de imagens, utilizando CUDA.

6. BIBLIOGRAFIA

[BERRILLO, 2008] BERILLO, A. NVIDIA CUDA. *Non-graphic computing with graphics processors*.: iXBT Labs, 2008. Disponível em: <http://www.digit-life.com/articles3/video/cuda-1-p1.html>. Acesso em: fev. 2010.

[BOGGAN et al., 2007] BOGGAN, S.; PRESSEL, D. M. *GPUs: an emerging platform for general-purpose computation*.: Army Research Lab, 2007. Disponível em: <http://www.arl.army.mil/arlreports/2007/ARL-SR-154.pdf>. Acesso em: fev. 2010.

[NVIDIA CUDA, 2007] NVIDIA CUDA for GPU Computing, 2007. Disponível em: <http://news.developer.NVIDIA.com/2007/02/cuda_for_gpu_co.html>. Acesso em: fev. 2010.

[DOWNLOAD CUDA, 2010] DOWNLOAD CUDA Code*: complete and free toolkit for creating derivative works*, 2010 Disponível em: <http://www.NVIDIA.com/object/cuda_get.html>. Acesso em: fev. 2010.

[FLYNN, 2004] FLYNN, 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: fev. 2010.

[FOSTER, 1995] FOSTER, I. T. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Reading: Addison-Wesley, 1995. 379 p.

[HALFHILL, 2008] HALFHILL, T. R. *Parallel Processing With CUDA: NVIDIA's High-Performance Computing Platform Uses Massive Multithreading.*: Microprocessor Report, 2008. Disponível em: <http://www.NVIDIA.com/docs/IO/55972/220401_Reprint.pdf>. Acesso em: fev. 2010.

[INTEL, 2010] INTEL *Microprocessor Quick Reference Guide: Product Family*. Disponível em: <http://www.intel.com/pressroom/kits/quickrefyr.htm>. Acesso em: fev. 2010.

[NICKOLLS, 2008] NICKOLLS, J. et al. *Scalable Parallel Programming With CUDA*. ACM Queue, 2008. Disponível em: <http://www.cs.stevens.edu/~spock/cs385/CUDA-Concepts.pdf>. Acesso em: fev. 2010.

[CUDA PROGRAMMING, 2009] NVIDIA CUDA Programming Guide. NVIDIA, v2.3, 2009. Disponível em:

<http://developer.download.NVIDIA.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf>. Acesso em: fev. 2010.

[NVIDIA TESLA, 2008] NVIDIA TESLA S1070 Datasheet. NVIDIA, 2008. Disponível em: <http://www.NVIDIA.com/docs/IO/43395/NV_DS_Tesla_S1070_US_Jun08_NV_LR_Final.pdf>. Acesso em: fev. 2010.

[OLIVEIRA, 2001] OLIVEIRA, R. S. et al., *Sistemas Operacionais*. 2a ed. Porto Alegre: Sagra Luzzato, 2001.

[TANENBAUM, 2001] TANENBAUM, A. S. *Organização Estruturada de Computadores*. Rio de Janeiro: Livros Técnicos e Científicos Editora, 2001.

[KHATCHATOURIAN, 2006] KHATCHATOURIAN, O. , *Aceleração de Métodos Iterativos para Resolução de Sistemas Lineares de Grande Porte, 2006*, Disponível em: <http://www.sbmac.org.br/eventos/cnmac/xxix_cnmac/PDF/283.pdf> Acesso em: mai. 2010.

[DU, 2007] DU, Y. et al., Application of the Stochastic Second-degree Iterative Method to EM Scattering from Randomly Rough Surfaces, PIERS Online, Vol. 3, nº 5, 2007 Disponível em: <http://piers.mit.edu/piersonline/download.php?file=MDYxMTEzMDQwNzU5fFZvbDNObzVQYWdlNzIzdG83MjYucGRm> Acesso em: mar. 2010.

[BOLDRINI, 1980] BOLDRINI, J. L. et al. *Álgebra Linear*, 3a. edição, Harbra, 1980, 411p., Capítulos 1 e 3.

[LEON, 1998] LEON, S. J. *Álgebra Linear com Aplicações*, 4a. edição, LTC, 1998, 390p., Capítulos 1 e 2.

[RANGER, 2010] RANGER User Guide, Disponível em: <http://services.tacc.utexas.edu/index.php/ranger-user-guide> Acesso em: mar. 2010.

[LIMA, 1995] LIMA, E. L. *Álgebra Linear*, SBM/IMPA, Coleção Matemática Universitária, 1995, 320p., Capítulo 19.

[LIMA, 2001] LIMA, E. L. *Geometria Analítica e Álgebra Linear*, SBM/IMPA, Coleção Matemática Universitária, 2001, 306p., Capítulos 34 a 39.

[BARNEY, 2010] BARNEY, B. *Introduction to Parallel Computing*. Disponível em: <http://computing.llnl.gov/tutorials/parallel_comp/>. Acesso em: mar. 2010.

[GPGPU, 2010] GPGPU.ORG *General-Purpose Computation on Graphics Hardware*. Disponível em: <http://gpgpu.org/tag/NVIDIA-cuda> Acesso em: mar. 2010.

[SILVA, 2004] SILVA, R. E. et al., *Matemática na Computação - Uma Crítica à Visão dos Alunos*, pp. 1-4, 2004, Disponivel em: <http://ensino.univates.br/~chaet/Materiais/Matematica_Computacao.pdf>. Acesso em: mar. 2010

[BAKHODA, 2009] BAKHODA, A. et al., *Analyzing CUDA Workloads Using a Detailed GPU Simulator, University of British Columbia, Vancouver, BC, Canada*, 2009. Disponível em: <http://www.ece.ubc.ca/~aamodt/papers/gpgpusim.ispass09.pdf> Acesso em: mai. 2010.

[LIGOWSKI, 2009] LIGOWSKI, Ł. et al., *An Efficient Implementation of Smith Waterman Algorithm on GPU Using CUDA, for Massively Parallel Scanning of Sequence Databases*, Interdisciplinary Centre for Mathematical and Computational Modelling, University of Warsaw, Warsaw, Poland, 2009. Disponível em: <http://www.hicomb.org/papers/HICOMB2009-10.pdf> Acesso em: mai. 2010.

[BARNEY, 2009] BARNEY, B. *Posix Threads Programming*. Lawrence Livermore National Laboratory, February 2009. Disponivel em: <http://computing.llnl.gov/tutorials/pthreads>. Acesso em: mar. 2010.

[KASIM, 2008] KASIM, H. et al., S. *Survey on Parallel Programming Model*, IFIP International Conference on Network and Parallel Computing (IFIP 2008), 18-20 October 2008, Shanghai, China. Disponivel em: <http://apstc.sun.com.sg/content.php?l1=research&l2=resources&l3=pubs>. Acesso em: mar. 2010

[FRANCO, 2006] FRANCO, N. B., *Cálculo Numérico*, São Paulo: Pearson Prentice Hall, 2006.

[LING, 2009] LING, C. et al., *A parameterisable and scalable Smith-Waterman algorithm implementation on CUDA-compatible GPUs*, sasp, pp.94-100, 2009 IEEE 7th Symposium on Application Specific Processors, 2009 Disponível em: <http://dx.doi.org/10.1109/SASP.2009.5226343> Acesso em: mai. 2010.

[LUEBKE, 2008] LUEBKE, D., *CUDA: Scalable parallel programming for high-performance scientific computing*, NVIDIA Corp., Santa Clara, CA, 2008. Disponível em: <http://dx.doi.org/10.1109/ISBI.2008.4541126> Acesso em: mai. 2010.

[GARLAND, 2008] GARLAND, M. et al., *Parallel Computing Experiences with CUDA*, NVIDIA Corp., Santa Clara, CA, 2008. Disponível em: <http://dx.doi.org/10.1109/MM.2008.57> Acesso em: mai. 2010.

[TAHER, 2009] TAHER, M., *Accelerating scientific applications using GPU's*, Ain Shams Univ., Cairo, Egypt, 2009. Disponível em: <http://dx.doi.org/10.1109/IDT.2009.5404114> Acesso em: mai. 2010.

[SUDA, 2009] SUDA, R. et al., *Aspects of GPU for general purpose high performance computing*, Grad. Sch. of Inf. Sci. & Technol., Univ. of Tokyo, Tokyo, 2009. Disponível em: <http://dx.doi.org/10.1109/ASPDAC.2009.4796483> Acesso em: abr. 2010.

[AMORIM, 2009] AMORIM, R. et al., *Comparing CUDA and OpenGL implementations for a Jacobi iteration*, Inst. for Math. & Sci. Comput., Karl Franzens Univ. of Graz, Graz, Austria, 2009. Disponível em: <http://dx.doi.org/10.1109/HPCSIM.2009.5192847> Acesso em: abr. 2010.

[ZHANG , 2009] ZHANG, Z. et al., *CUDA-Based Jacobi's Iterative Method*, Coll. of Comput. & Commun. Eng., Grad. Univ. of Chinese Acad. of Sci. (GUCAS), Beijing, China, 2009. Disponível em: <http://dx.doi.org/10.1109/IFCSTA.2009.68> Acesso em: abr. 2010.

[FUJIMOTO, 2008] FUJIMOTO, N., *Faster matrix-vector multiplication on GeForce 8800GTX*, Grad. Sch. of Inf. Sci. & Technol., Osaka Univ., Osaka, 2008. Disponível em: <http://dx.doi.org/10.1109/IPDPS.2008.4536350> Acesso em: mar. 2010.

[WANG, 2009] WANG T. et al., *Implementation of Jacobi iterative method on graphics processor unit*, Dept. of Comput. Sci., Zhengzhou Inf. Sci. & Technol. Inst., Zhengzhou, China, 2009. Disponível em: <http://dx.doi.org/10.1109/ICICISYS.2009.5358155> Acesso em: mai. 2010.

[ZOU, 2009] ZOU C. et al., *Numerical Parallel Processing Based on GPU with CUDA Architecture*, Coll. of Comput. Sci. & Technol., Wuhan Univ. of Technol., Wuhan, China, 2009. Disponível em: <http://dx.doi.org/10.1109/WNIS.2009.46> Acesso em: abr. 2010.

[CUI, 2009] CUI, X. et al. *Improving Performance of Matrix Multiplication and FFT on GPU*, Key Lab. of High Confidence Software Technol., Peking Univ., Beijing, China, 2009. Disponível em: <http://dx.doi.org/10.1109/ICPADS.2009.8> Acesso em: mar. 2010.

[CUMMINS, 2008] CUMMINS, G. et al., *Scientific computation through a GPU*, Univ. of the Cumberlands, Williamsburg, 2008. Disponível em: <http://dx.doi.org/10.1109/SECON.2008.4494293> Acesso em: mar. 2010.

[DZIEKONSKI, 2008] DZIEKONSKI, A. et al., Implementation of matrix-type FDTD algorithm on a graphics accelerator, Dept. of Microwave & Antenna Eng., Gdansk Univ. of Technol., Gdansk, 2008. Disponível em: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4630162> Acesso em: mar. 2010.

[SPAMPINATO, 2009] SPAMPINATO, D.G. et al., Linear optimization on modern GPUs, Dept. of Comput. & Inf. Sci., Norwegian Univ. of Sci. & Technol., Trondheim, Norway, 2009. Disponível em: <http://dx.doi.org/10.1109/IPDPS.2009.5161106> Acesso em: abr. 2010.

[MANAVSKI, 2007] MANAVSKI, S. A., *CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography*, ITCIS, Sofia, Bulgaria, 2007. Disponível em: <http://www.manavski.com/downloads/PID505889.pdf> Acesso em: mar. 2010.

[GRAÇA, 2006] GRAÇA, G. et al., *Implementation of float-float operators on graphics hardware.* Université de Perpignan, Perpignan Cedex, France, 2006. Disponível em: <http://hal.archives-ouvertes.fr/docs/00/06/33/56/PDF/float-float.pdf> Acesso em: mar. 2010.

[LOPES, 2008] LOPES, B. C. et al., *CUDA – Compute Unified Device Architecture*, UNICAMP, Brasil, 2008. Disponível em: <http://www.ic.unicamp.br/~rodolfo/cuda/> Acesso em: fev. 2010.

[GRAHAM, 2008] GRAHAM, J. R., *Comparing Parallel Programming Models*, Department of Mathematics & Computer Science, Longwood University, 2008. Disponível em: <http://portal.acm.org/citation.cfm?id=1352383.1352395> Acesso em: abr. 2010.

[GAO, 2008] GAO, H. et al., *Experiencing Various Massively Parallel Architectures and Programming Models for Data-Intensive Applications*, School of EECS, University of Central Florida, 2008. Disponível em: <http://www.eecs.ucf.edu/~zhou/dlp.pdf> Acesso em: mar. 2010.

[RAVELA, 2010] RAVELA, S. C., *Comparison of Shared memory based parallel programming models*, School of Computing, Blekinge Institute of Technology, Sweden 2010. Disponível em: <http://www.bth.se/fou/cuppsats.nsf/all/9841104b73849739c12576b7003d8b98/$file/Comparison%20of%20shared%20memory%20based%20parallel%20programming%20models.pdf> Acesso em: abr. 2010.

[MOGILL, 2010] MOGILL, J. A. et al., *A Comparison Of Shared Memory Parallel Programming Models*, Pacific Northwest National Laboratory, Richland, WA, USA, 2010. Disponível em: <http://www.cug.org/1-conferences/CUG2010/pages/1-program/final_program/CUG10CD/CUG10_Proceedings/pages/authors/11-15Wednesday/14C-mogill-paper.pdf> Acesso em: abr. 2010.

[LI, 2010] LI, G. et al., *A Symbolic Verifier for CUDA Programs*, School of Computing, University of Utah & Lawrence Livermore National Laboratory, USA, 2010. Disponível em: <http://portal.acm.org/citation.cfm?id=1693453.1693512> Acesso em: abr. 2010.

[CÁCERES, 2001] CÁCERES E. N. et al., *Algoritmos Paralelos Usando CGM/PVM: Uma Introdução,* páginas 219–278. XXI Congresso da Sociedade Brasileira de Computação, Jornada de Atualização em Informática, 2001.

[ARCS, 2008] ARCS 2008 GPGPU and CUDA Tutorials. Disponível em: <http://www.mathematik.uni-dortmund.de/~goeddeke/arcs2008/A1_Architecture.pdf> Acesso em: mar. 2010.

[PACHECO, 1997] PACHECO, P. S., Parallel Programming with MPI. Morgan Kaufmann Publishers, Inc, 1997

[FATAHALIAN, 2009] FATAHALIAN K., *Beyond Programmable Shading*, Stanford University, SIGGRAPH, 2009. Disponível em: <http://s09.idav.ucdavis.edu/talks/02_kayvonf_gpuArchTalk09.pdf> Acesso em: abr. 2010.

[PERILS, 2008] The Perils of Parallel : Larrabee vs. NVIDIA, MIMD vs. SIMD. Disponível em: <http://perilsofparallel.blogspot.com/2008/09/larrabee-vs-NVIDIA-mimd-vs-simd.html> Acesso em: mar. 2010.

[HALL, 2010] HALL, M., Parallel Programming for GPUs. Disponível em: <http://www.cs.utah.edu/~mhall/cs6963s09/> Acesso em: mai. 2010.

[GONDA, 2004] GONDA, L., *Algoritmos BSP/CGM para ordenação*, Dissertação (mestrado), DCT, UFMS, 2004.

Apêndice I Multiplicação de matrizes

Matriz.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <cutil_inline.h>
#include <Matriz_kernel.cu>

void InicializarMatriz(float*, int, int);
void imprime(float*, int, int);
void matrizMulCPU(float*,  float*,  float*,  int);

int main(int argc, char** argv)
{
        if( cutCheckCmdLineFlag(argc, (const char**)argv, "device") )
                cutilDeviceInit(argc, argv);
        else
                cudaSetDevice( cutGetMaxGflopsDeviceId() );
  srand(20);
  unsigned int timer = 0;
  unsigned int timer2 = 0;
  int tamanho_A = TAMANHO * TAMANHO;
  int mem_tamanho_A = sizeof(float) * tamanho_A;
  float* h_A = (float*) malloc(mem_tamanho_A);

  int tamanho_B = TAMANHO * TAMANHO;
  int mem_tamanho_B = sizeof(float) * tamanho_B;
  float* h_B = (float*) malloc(mem_tamanho_B);

  int tamanho_C = TAMANHO * TAMANHO;
  int mem_tamanho_C = sizeof(float) * tamanho_C;
  float* h_C_GPU = (float*) malloc(mem_tamanho_C);
  float* h_C_CPU = (float*) malloc(mem_tamanho_C);

  InicializarMatriz(h_A, tamanho_A,TAMANHO);
  InicializarMatriz(h_B, tamanho_B,TAMANHO);


  float* d_A;
  cutilSafeCall(cudaMalloc((void**) &d_A, mem_tamanho_A));
  float* d_B;
  cutilSafeCall(cudaMalloc((void**) &d_B, mem_tamanho_B));
  float* d_C;
  cutilSafeCall(cudaMalloc((void**) &d_C, mem_tamanho_C));

  cutilSafeCall(cudaMemcpy(d_A, h_A, mem_tamanho_A,
                cudaMemcpyHostToDevice) );
  cutilSafeCall(cudaMemcpy(d_B, h_B, mem_tamanho_B,
                cudaMemcpyHostToDevice) );
   const int max_threads_por_bloco = 256;
   int blocos = TAMANHO/max_threads_por_bloco;
   if(max_threads_por_bloco==512)
        blocos=1;
   while(blocos*max_threads_por_bloco> 512)
        blocos--;
```

```
    cutilCheckError(cutCreateTimer(&timer));
    cutilCheckError(cutStartTimer(timer));

                matrizMulGPU<<< blocos, max_threads_por_bloco >>>(d_C, d_A, d_B);

    cutilCheckError(cutStopTimer(timer));
    printf("Tempo de processamento da GPU: %f ms. \n", cutGetTimerValue(timer));
    cutilCheckError(cutDeleteTimer(timer));

    cudaMemcpy(h_C_GPU, d_C, mem_tamanho_C,
                        cudaMemcpyDeviceToHost) ;

    cutilCheckError(cutCreateTimer(&timer2));
    cutilCheckError(cutStartTimer(timer2));
    matrizMulCPU(h_C_CPU, h_A, h_B, TAMANHO);
            cutilCheckError(cutStopTimer(timer2));
    printf("Tempo de processamento da CPU: %f ms. \n", cutGetTimerValue(timer2));
            cutilCheckError(cutDeleteTimer(timer2));

    CUTBoolean res = cutCompareL2fe(h_C_CPU, h_C_GPU, tamanho_C, 1e-6f);

            imprime(h_C_CPU, TAMANHO,TAMANHO);
            imprime(h_C_GPU, TAMANHO,TAMANHO);

    free(h_A);
    free(h_B);
    free(h_C_GPU);
    free(h_C_CPU);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    cudaThreadExit();
    cutilExit(argc, argv);

}

void InicializarMatriz(float* matriz, int tamanho,int linha)
{
            for (int i = 0; i < tamanho; ++i)
                        matriz[i] = rand();
}

void imprime(float* matriz, int tamanho,int linha)
{
            for (int i = 0; i < tamanho; i++)
             {
                        printf("%.0f\t",matriz[i]);
                        if((i+1)%linha==0 && i!=0)
                              printf("\n");
             }
}
```

Apêndice II Produto escalar

Produto.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <cutil_inline.h>
#include <Produto_kernel.cu>

void ProdutoEscalarSequencial(float *h_C, float *h_A, float *h_B, int vetorN, int dimensaoN);
float InicializarVetor(float min, float max);

#define VETOR_N  10000
#define DIMENSAO_N  10000
#define DADOS_N  VETOR_N * DIMENSAO_N
#define DADOS_TAM  DADOS_N * sizeof(float)
#define RESULT_TAM  VETOR_N  * sizeof(float)
#define max_threads_por_bloco  512

int main(int argc, char **argv){
    float *h_A, *h_B, *h_C_CPU, *h_C_GPU;
    float *d_A, *d_B, *d_C;
    unsigned int timer, timer2;
    int i;

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

   srand(20);
              cutilCheckError( cutCreateTimer(&timer) );
               cutilCheckError( cutCreateTimer(&timer2) );

        h_A     = (float *)malloc(DADOS_TAM);
        h_B     = (float *)malloc(DADOS_TAM);
        h_C_CPU = (float *)malloc(RESULT_TAM);
        h_C_GPU = (float *)malloc(RESULT_TAM);

        cutilSafeCall( cudaMalloc((void **)&d_A, DADOS_TAM)   );
        cutilSafeCall( cudaMalloc((void **)&d_B, DADOS_TAM)   );
        cutilSafeCall( cudaMalloc((void **)&d_C, RESULT_TAM) );


        for(i = 0; i < DADOS_N; i++){
            h_A[i] = InicializarVetor(0.0f, 1.0f);
        }
        for(i = 0; i < DADOS_N; i++){
            h_B[i] = InicializarVetor(0.0f, 1.0f);
        }


        cutilSafeCall( cudaMemcpy(d_A, h_A, DADOS_TAM, cudaMemcpyHostToDevice) );
        cutilSafeCall( cudaMemcpy(d_B, h_B, DADOS_TAM, cudaMemcpyHostToDevice) );
```

```
        int blocos = VETOR_N/ max_threads_por_bloco;
        cutilSafeCall( cudaThreadSynchronize() );
        cutilCheckError( cutResetTimer(timer) );
        cutilCheckError( cutStartTimer(timer) );
        ProdutoEscalarGPU<<<blocos, max_threads_por_bloco>>>(d_C, d_A, d_B, VETOR_N,
DIMENSAO_N);
        cutilCheckError( cutStopTimer(timer) );
        cutilCheckMsg("scalarProdGPU() execution failed\n");
         printf("Tempo de processamento da GPU: %f ms.\n", cutGetTimerValue(timer));
        cutilSafeCall( cudaMemcpy(h_C_GPU, d_C, RESULT_TAM, cudaMemcpyDeviceToHost) );

        cutilCheckError( cutResetTimer(timer2) );
        cutilCheckError( cutStartTimer(timer2) );
        ProdutoEscalarSequencial(h_C_CPU, h_A, h_B, VETOR_N, DIMENSAO_N);
        cutilCheckError( cutStopTimer(timer2) );
        printf("Tempo de processamento da CPU: %f ms.\n", cutGetTimerValue(timer2));

        for(i = 0; i < VETOR_N; i++)
           printf("%f",h_C_GPU[i]);
           printf("\n");
        for(i = 0; i < VETOR_N; i++)
           printf("%f",h_C_CPU[i]);

        cutilSafeCall( cudaFree(d_C) );
        cutilSafeCall( cudaFree(d_B)   );
        cutilSafeCall( cudaFree(d_A)   );
        free(h_C_GPU);
        free(h_C_CPU);
        free(h_B);
        free(h_A);
        cutilCheckError( cutDeleteTimer(timer) );
        cutilCheckError( cutDeleteTimer(timer2) );
        cudaThreadExit();
        cutilExit(argc, argv);

}

float InicializarVetor(float min, float max){
   float t = (float)rand() / (float)RAND_MAX;
   return (1.0f - t) * min + t * max;
   return t;
}
```

Apêndice III Método de Jacobi-Richardson

Jacobi-Richardson.cu

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

#include <cutil_inline.h>
#include <Jacobi-Richardson_kernel.cu>


void InicializarValores(float*, float*,float*);
void InicializarValores2(float*, float*,float*);
void InicializarValores3(float*, float*,float*);
bool TestarConvergencia(float*);
void JacobiRichardsonCPU(float* , float*, float* , float *, int,float);


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


   srand(20);
   const int max_threads_por_bloco = 32;

   int size_A = TAMANHO * TAMANHO;
   int mem_size_A = sizeof(float) * size_A;
   float* h_A = (float*) malloc(mem_size_A);

   int size_x = TAMANHO;
   int mem_size_x = sizeof(float) * size_x;
   float* h_x = (float*) malloc(mem_size_x);

   int size_b = TAMANHO;
   int mem_size_b = sizeof(float) * size_b;
   float* h_b = (float*) malloc(mem_size_b);

   int size_y = TAMANHO;
   int mem_size_y = sizeof(float) * size_y;
   float* h_y_CPU = (float*) malloc(mem_size_y);
   float* h_y_GPU = (float*) malloc(mem_size_y);

   InicializarValores2(h_A,h_x,h_b);

   if(TestarConvergencia(h_A))
   {

                  float* d_A;
                  cutilSafeCall(cudaMalloc((void**) &d_A, mem_size_A));
                  float* d_x;
```

```
		cutilSafeCall(cudaMalloc((void**) &d_x, mem_size_x));
		float* d_b;
		cutilSafeCall(cudaMalloc((void**) &d_b, mem_size_b));
		float* d_y;
		cutilSafeCall(cudaMalloc((void**) &d_y, mem_size_y));

		 cutilSafeCall(cudaMemcpy(d_A, h_A, mem_size_A,
		  cudaMemcpyHostToDevice) );
		cutilSafeCall(cudaMemcpy(d_x, h_x, mem_size_x,
		  cudaMemcpyHostToDevice) );
		cutilSafeCall(cudaMemcpy(d_b, h_b, mem_size_b,
		  cudaMemcpyHostToDevice) );

		int blocos = TAMANHO / max_threads_por_bloco;
		unsigned int timer = 0;
		cutilCheckError(cutCreateTimer(&timer));
		cutilCheckError(cutStartTimer(timer));
		JacobiRichardsonGPU<<< blocos, max_threads_por_bloco >>>(d_y, d_A, d_x, d_b);

		cutilCheckError(cutStopTimer(timer));
		printf("Tempo de processamento da GPU: %f ms. \n", cutGetTimerValue(timer));
		cutilCheckError(cutDeleteTimer(timer));

		cudaMemcpy(h_y_GPU, d_y, mem_size_y,cudaMemcpyDeviceToHost);
		 unsigned int timer2 = 0;
		cutilCheckError(cutCreateTimer(&timer2));
		cutilCheckError(cutStartTimer(timer2));
		JacobiRichardsonCPU(h_y_CPU, h_A, h_x, h_b, TAMANHO,ERRO);
		cutilCheckError(cutStopTimer(timer2));
		printf("Tempo de processamento da CPU: %f ms. \n", cutGetTimerValue(timer2));
		cutilCheckError(cutDeleteTimer(timer2));

		for(int h=0; h<TAMANHO; h++)
			printf("%f ",h_y_GPU[h]);
			printf("\n");
		for(int h=0; h<TAMANHO; h++)
			printf("%f ",h_y_CPU[h]);

		free(h_A);
		free(h_b);
		free(h_y_GPU);
		free(h_x);
		free(h_y_CPU);
		cudaFree(d_A);
		cudaFree(d_b);
		cudaFree(d_x);
		cudaFree(d_y);
	}
	else
		printf("Sistema nao converge");
	cudaThreadExit();
	cutilExit(argc, argv);
}


void InicializarValores(float *h_A,float *h_x,float *h_b)
{
	for(int i=0; i<TAMANHO;i++)
	{
		for(int j=0; j<TAMANHO;j++){
```

```
                        if(i!=j)
                        {
                                h_A[i*TAMANHO+j]=1;
                        }
                        else
                        {
                                h_A[i*TAMANHO+j]=j+TAMANHO+60;
                        }
                }
        h_x[i]=TAMANHO+90;
        h_b[i]=TAMANHO+95;
        }
}


bool TestarConvergencia(float *h_A)
{
        float linha_soma, coluna_soma;
        for(int i=0; i<TAMANHO;i++)
        {
                linha_soma=0;
                for(int j=0; j<TAMANHO;j++)
                        if(i!=j)
                                linha_soma += h_A[i*TAMANHO+j];
                linha_soma=linha_soma/h_A[i*TAMANHO+i];
                if(linha_soma>1)
                {
                        for(i=0;i<TAMANHO;i++){
                                coluna_soma=0;
                                for(int j=0;j<TAMANHO;j++)
                                        if(i!=j)
                                                coluna_soma += h_A[j*TAMANHO+i];
                                coluna_soma=coluna_soma/h_A[i*TAMANHO+i];
                                if(coluna_soma>1)
                                        return false;
                        }
                        return true;
                }
        }
return true;
}
```

Apêndice IV Método de ordenação bitônica

Ordenacao.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <cutil_inline.h>
#include <Ordenacao_kernel.cu>


void InicializaVetor(float*);
void OrdenacaoCPU(int, float* int , int);
void imprime(float*,int);

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


    srand(20);
    const int max_threads_por_bloco = 512;


    int size_A = N;
    int mem_size_A = sizeof(float) * size_A;
    float* h_A = (float*) malloc(mem_size_A);
    float* h_x = (float*) malloc(mem_size_A);


    InicializaVetor(h_A);

   float* d_x;
   cutilSafeCall(cudaMalloc((void**) &d_x, mem_size_A));
   cutilSafeCall(cudaMemcpy(d_x, h_A, mem_size_A,cudaMemcpyHostToDevice) );

    unsigned int timer = 0;
    cutilCheckError(cutCreateTimer(&timer));
    cutilCheckError(cutStartTimer(timer));
    for (int limite = 2; limite <= TAMANHO; limite = limite *2)
            for ( int passo = limite /2; passo>0; passo=passo/2)
            OrdenacaoGPU<<<TAMANHO/max_threads_por_bloco,max_threads_por_bloco>>>(d_x,
passo, limite);

   cutilCheckError(cutStopTimer(timer));
   printf("Tempo de processamento da GPU: %f ms. \n", cutGetTimerValue(timer));
   cutilCheckError(cutDeleteTimer(timer));
   cutilCheckMsg("Kernel execution failed");
   cudaMemcpy(h_x, d_x, mem_size_A,cudaMemcpyDeviceToHost);

   unsigned int timer2 = 0;
   cutilCheckError(cutCreateTimer(&timer2));
   cutilCheckError(cutStartTimer(timer2));
```

```
    for (int limite = 2; limite <= TAMANHO; limite = limite *2)
        for ( int passo= limite /2; passo>0; passo=passo/2)
            OrdenacaoCPU(TAMANHO,h_A,passo,limite);
    cutilCheckError(cutStopTimer(timer2));
    printf("Tempo de processamento da CPU: %f ms. \n", cutGetTimerValue(timer2));
    cutilCheckError(cutDeleteTimer(timer2));

    imprime(h_x, TAMANHO);
    imprime(h_A, TAMANHO);
    free(h_A);
    cutilSafeCall(cudaFree(d_x));
    cudaThreadExit();
    cutilExit(argc, argv);
}

void InicializaVetor(float* vetor)
{
        for(int i=0; i<TAMANHO;i++)
            vetor[i]=rand();
}


void imprime(float* matriz, int tamanho)
{
        for (int i = 0; i < tamanho; i++)
   {
            printf("%.0f\t",matriz[i]);

   }
}
```

Apêndice V *Kernel* alternativo para multiplicação de matrizes

Matriz_kernel_alt.cu

```
#define TAMANHO 900

__global__ void
matrizMulGPU( float* C, float* A, float* B, int i)
{
	int indice = threadIdx.x + blockDim.x * blockIdx.x;
	double soma=0;
	for (int j = 0; j < TAMANHO; ++j)
	{
		soma += (double)A[(indice*TAMANHO)+j] * (double)B[(j*TAMANHO)+i];
	}
	C[(indice*TAMANHO)+i]=(float)soma;
}
```

Matriz.cu

Na chamada do *kernel*, basta modificar para executar dentro de um laço.

```
...

    for (int i = 0; i < TAMANHO; ++i)
		matrizMulGPU<<< blocos, max_threads_por_bloco >>>(d_C, d_A, d_B, i);
...
```