

UNIVERSIDADE DE SÃO PAULO

INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

Relatório Trabalho 4 - Programação Concorrente

Prof. Dr. Júlio Cezar Estrella

Julho - 2013

Grupo 9 – Turma B

**Método de Jacobi-Richardson utilizando CUDA**

*D. L. de Souza L. F. A. Prado S. Seraphini*

The contents of this report are the sole responsibility of the authors.

O conteúdo deste relatório é de única responsabilidade dos autores.

Resumo

O projeto visa desenvolver uma aplicação sequencial e uma paralela utilizando CUDA que resolva o sistema linear na forma (Ax = B) utilizando o método iterativo de Jacobi-Richardson. Foram comparados os tempos de execução de cada uma bem como suas diferenças de implementação

**Palavras-Chave:** Programação Concorrente; Método Iterativo; Jacobi-Richardson; Gauss-Jacobi; CUDA; Speedup.

Sumário

1. **Introdução 4**

**1.1 CUDA 4**

**1.1.1 Benefícios ao utilizar CUDA 5**

**1.1.2 Limitações de CUDA 5**

**1.2 Método de Jacobi-Richardson 5**

**1.2.1 Descrição 6**

**1.2.2 Algoritmo 7**

1. **Resultados 7**

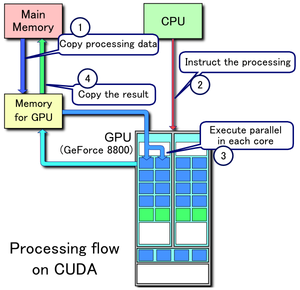
**2.1 Tabelas/Gráficos 8**

1. **Conclusão 10**
2. **Read Me 11**
3. **Referências 12**

1. Introdução

1.1 CUDA

CUDA (Compute Unified Device Architecture) é uma plataforma de computação paralela desenvolvida pela NVidia para tirar proveito das GPUs na resolução de problemas complexos. Graças à arquitetura das GPUs GeForce é possível executar milhares de threads concorrentemente.



*Gráfico 1: fluxo de processamento em CUDA*

Esquematizado no gráfico 1 está o fluxo de processamento em CUDA:

1 - Os dados são copiados na memória principal para a memória da GPU que é muito mais rápida e nesse caso funciona como um cache;

2 - A CPU instrui como deverá ser feito o processamento pela GPU;

3 - A GPU executa paralelamente em cada um de seus núcleos;

4 - Os dados são copiados da memória da GPU para a memória principal.

1.1.1 Benefícios ao utilizar CUDA:

- Leitura paralela (o código pode ler de endereços arbitrários na memória)

- Memória compartilhada entre as threads muito rápida, permitindo grande largura de banda.

- Downloads e readbacks rápidos para a GPU

- Total suporte para operações de números inteiros e bitwise

1.1.2 Limitações de CUDA:

- Não suporta renderização de texturas

- Cópias realizadas entre uma memória e outra podem gerar algum problema na performance das aplicações

- Disponibilidade apenas para placas NVidia

1.2 Método Jacobi-Richardson

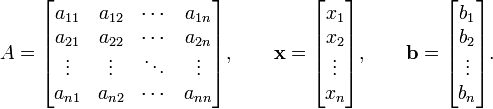
O método iterativo de Jacobi-Richardson trata-se de um algoritmo desenvolvido no final do século XVIII para determinar a solução de um sistema de equações lineares. Essa técnica é raramente utilizada para sistemas de pequenas dimensões pois outras mais eficientes são conhecidas para esses casos. Contudo, para sistema grandes e com grande porcentagem de entradas igual a zero, esse método se mostra eficiente tanto em termo de cálculo como de armazenamento. Com essa informação já é possível perceber o porquê esse método ter tamanha aplicação computacional.

1.2.1 Descrição

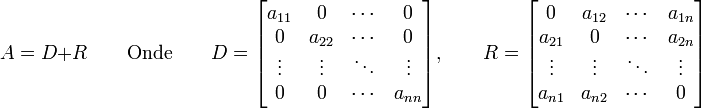
Dada uma matriz quadrada de *n* equações lineares:



em que:



Então *A* pode ser decomposto num componente diagonal *D* e o resto *R*:



O sistema de equações lineares pode ser reescrito como:



O método de Jacobi-Richardson resolve o membro esquerdo da expressão em ordem a *x* ao usar o método resultante da iteração anterior no membro direito. Analiticamente, isso pode ser escrito como:



A expressão pode ser então:



Ao contrário de outros métodos, não se pode reescrever com  pois esse valor é necessário durante a continuação da computação. Esse é o motivo que torna o método de Jacobi Richardson em um algoritmo paralelo.

1.2.2 Algoritmo

O algoritmo utilizado foi:

Definir a primeira tentativa  para a solução

Enquanto o resultado não convergir faça

para *i* de 1 até *n* faça



para *j* de 1 até *n* fala

se *j* é diferente de *i* então







2. Resultados

Para o desenvolvimento dos códigos paralelos foi utilizado o princípio de decomposição de dados, ou seja, decompomos a operação a ser realizada entre os vários blocos de threads e cada uma delas executa as mesmas operações sobre diferentes elementos do conjunto de dados.

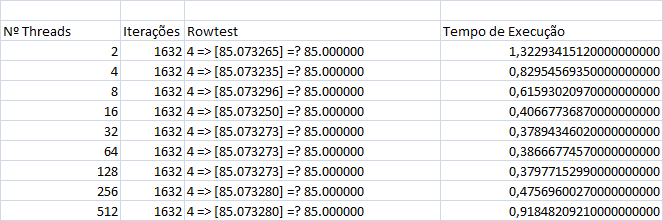
Para comparação de resultados entre os algoritmos sequenciais e paralelos é feito o cálculo do speedup, que consiste na seguinte fórmula:



Onde Sp é o valor do speedup, T1 é o tempo de execução do algoritmo sequencial e Tp é o tempo de execução do algoritmo paralelo. Os programas foram executados no cluster CUDA Lasdpc*.* **Speed Up (CUDA x Sequencial):** . Os dados apresentados a seguir são as médias de dez execuções variando o número de threads em potência de dois, começando em 2 até 512.

2.1 Tabelas/Gráficos

2.1.1 Matriz 250

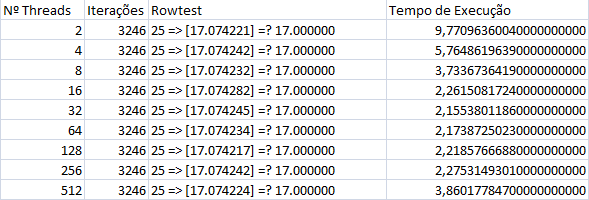


*Resultados da execução paralela da matriz de 250 linhas.*

Notamos speedup de e pudemos perceber que o tempo de execução chegou aos melhores valores ao utilizar entre 16 e 128 threads. Ao diminuir ou aumentar o número de threads, temos também um aumento no tempo de execução.

Um maior tempo de execução era esperado com menos threads porém o aumento com mais que 128 threads provavelmente se deve à limitações da placa.

2.1.2 Matriz 500

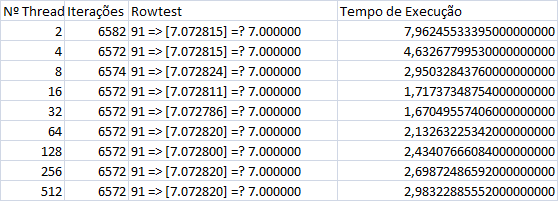


*Resultados da execução paralela da matriz de 500 linhas.*

Novamente tivemos speedup, dessa vez de e pudemos perceber que o tempo de execução chegou aos melhores valores ao utilizar entre 16 e 256threads.

O maior tempo de execução se deve provavelmente aos mesmos fatores do teste com a matriz de 250 linhas: era esperado com menos threads porém o aumento com mais que 256 threads provavelmente se deve à limitações da placa.

2.1.2 Matriz 1000



*Resultados da execução paralela da matriz de 250 linhas.*

Speedup de em relação à execução sequencial e o tempo de execução chegou aos melhores valores ao utilizar entre 16 e 32 threads. Ao diminuir ou aumentar o número de threads, temos também um aumento no tempo de execução. Novamente pudemos perceber que com muitas threads, o tempo foi quase duas vezes maior do que com 16 ou 32 threads.

A impressão geral é que os melhores resultados se encontram em um intervalo de threads e que aumentar a quantidade delas além desse limite ou diminuí-las acarreta em resultados piores. Ainda assim, todos os resultados utilizando CUDA foram muito melhores do que a execução sequencial.

3. Conclusão

Pudemos perceber que a paralelização utilizando CUDA mostrou resultados ótimos. Todos os testes apresentaram speedup muito significativo. A abordagem de CUDA de utilizar várias unidades de processamento trabalhando a velocidades mais baixas do que uma única unidade em alta velocidade é altamente eficiente para as aplicações testadas. Outro fator que também teve impacto positivo foi a relativa simplicidade de programação: aproveita-se grande parte do código sequencial em C++, adicionando apenas as rotinas de CUDA principalmente para a alocação dos dados da memória principal para a memória da GPU, “escondendo” a complexidade de manipulação das threads do usuário.

A impressão geral é que temos no CUDA uma ferramenta muito versátil para aplicações que exigem o uso de processamento paralelo para otimização de tempo de execução.

4. Read Me

|  |
| --- |
| * O programa está na pasta * Para compilar utilize o arquivo Makefile * Para executar |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| 5. Referências  Cuda by Example: An Introduction to General Purpose GPU Programming - Sanders, Jason; Kandrot, Edward - 2010  Cuda-Zone - <https://developer.nvidia.com/category/zone/cuda-zone>  Slides de Aula - Professor Dr. Julio Cézar Estrella |
|  |
|  |
|  |
|  |
|  |
|  |
|  |