INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ANDREAS MUNTE FOERSTER – andreas.foerster@usp.br – #7143997

COMPARAÇÃO DE TEMPO DE EXECUÇÃO SEQUENCIAL E CONCORRENTE DE SMOOT EM IMAGENS EM CPU E GPU:

SÃO CARLOS – SP

DEZ-15

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

1. INTRODUÇÃO

A evolução dos processadores (CPUs) está fornecendo, ao mercado consumidor, CPUs cada vez mais rápidas, com mais transistores e arquiteturas mais eficientes, no entanto, há um limite para a capacidade de processamento para a arquitetura Von Neumann usada nos computadores usuais. Tal limitação deve-se à natureza sequencial da arquitetura que, apesar de ter sido otimizada com a comercialização recente dos processadores com múltiplos núcleos, persiste em cada núcleo do processador. Cada core da CPU ainda é sequencial.

Em contraste, a arquitetura de placas gráficas (GPU) oferece milhares de pequenas unidades de processamento que, apesar de mais simples, possibilitam o processamento de muitos dados simultaneamente. Contanto que o cálculo realizado na GPU não seja muito extenso e possa ser replicado para cada um dos dados a ser processado, a placa gráfica oferece uma alternativa muito mais eficiente que CPUs . Um exemplo para este tipo de otimização está no processamento de imagens.

2. OBJETIVO

Este trabalho tem como objetivo avaliar o ganho de desempenho ao executar um algoritmo simples de smooth de tratamento de imagens em placa gráfica comparado a sua versão sequencial e paralela em CPU.

3. MÉTODOS

3.1. Smooth

O algoritmo de smooth utilizado realiza a média aritmética de cada pixel da imagem com os 8 pixels ao seu redor, um algoritmo facilmente paralelizável.

3.2. Decomposição MPI/OpenMP

Para a divisão de tarefas entre os diferentes nós do cluster, foi utilizada a decomposição de dados na qual a imagem original é ‘cortada’ vertical e horizontalmente em sub imagens procurando fragmentos com o menor perímetro possível. Cada fragmento recebe, então, uma borda adicional contento uma linha de pixels de cada fragmento próximo a ele.

Desta forma, consegue-se produzir fragmentos com o menor overhead e, portanto, tem-se uma menor sobrecarga da rede na hora de transmitir os fragmentos para seus respectivos nós.

Utilizando esta técnica, espera-se, idealmente, obter um tempo de execução inversamente proporcional ao número de processos utilizados.

Para a divisão de tarefas dentro de um mesmo nó, optou-se por uma alternativa mais simples na qual cada thread é responsável por processar um conjunto de linhas da imagem. Uma alternativa seria atribuir cada canal da imagem a uma thread separada, no entanto, isso causaria

um desperdício de recursos uma vez que imagens em preto e branco possuem apenas um canal e imagens coloridas possuem 3 canais, número dificilmente divisível pelo número de núcleos que o processador possuirá.

3.3. Mapeamento MPI/OpenMP

Tendo em vista que a técnica de decomposição escolhida produz fragmentos de tamanhos iguais ou muito próximos, é esperado que não haja grandes diferenças no tempo de execução de cada processo, logo optou-se por um mapeamento estático no qual o número de fragmentos gerados é igual ao número de processos utilizados. O processo mestre encaminha um fragmento para cada processo filho e recebe os fragmentos processados de volta.

3.4. CUDA

Para a versão em CUDA do código, optou-se por mapear um pixel por thread para fragmentos de 16x16 pixels de imagem. No início do kernel CUDA, cada thread é responsável por copiar seu respectivo pixel para a memória compartilhada do bloco antes que o algoritmo de smooth possa ser processado. Tendo em vista que o algoritmo requer os pixels da vizinhança, cada bloco CUDA, na verdade, terá dimensão 18x18. As threads adicionais apenas copiam os pixels para a memória compartilhada e não processam o smooth.

Como estrutura de dados, optou-se por 3 arrays de entrada, cada um contendo um canal da imagem original, e 3 arrays para as respectivas saídas. Como cada thread processa um pixel, no caso de imagens coloridas cada thread processa os três canais.

3.5. Coleta de Dados

Para a coleta de dados foi desenvolvida uma versão sequencial do algoritmos, recebendo como argumentos a imagem a ser processada e o arquivo no qual deve escrever o resultado, e uma versão paralela com argumentos adicionais sendo o número de cortes horizontais e o número de cortes verticais a realizar para produzir os fragmentos.

Foram utilizadas versões coloridas e em escala de cinza e quatro imagens com dimensões definidas por:

- Pequeno (2048x1024)
- Média (4096x2048)
- Grande (8184x4096)
- Muito Grande (16384x8184)

Cada Imagem foi processada usando o algoritmo sequencial, paralelo OpenMP/MPI com 4 e 8 recortes e em placa gráfica. Cada experimento foi realizado 10 vezes para o cálculo dos média e desvio padrão do tempo de execução.

Para medir o tempo de execução, considerou-se a execução do programa todo, incluindo leitura do arquivo e chamada para a rotina em CUDA.

4. RESULTADOS

Para coleta dos dados abaixo foi utilizada uma máquina Linux com as seguintes especificaçõe:

```
Nodes (13 hosts / 104 virtuais) – 12 slaves nodes / 1 master node
Intel® Core™ I7 Processor – LGA -1150 – 4790 3.60GHZ DMI 5GT/S 8MB
32 GB RAM DDR3 Corsair Vegeance
Motherboard Gigabyte GA-Z97X-SLI ATX
Video Nvidia GTX 650 – 1GB
HD 2TB Seagate Sata III 7200RPM
Fonte ATX 650W Real
```

Tabela 1 - Tempo de Execução Sequencial, OpenMP/MPI e CUDA

Versão	Tempo de Execução (s)	Speedup
Imagem Pequena (2048x1024) em Preto e Branco		
Seq.	0.534 +/- 0.016	-
P4	1.858 +/- 0.046	0.287 +/- 0.011
P8	1.913 +/- 0.038	0.279 +/- 0.010
gpu	0.250 +/- 0.008	2.139 +/- 0.095
Imagem Pequena (2048x1024) em Colorida		
Seq.	0.632 +/- 0.004	-
P4	1.964 +/- 0.029	0.322 +/- 0.005
P8	2.002 +/- 0.013	0.316 +/- 0.003
gpu	0.277 +/- 0.005	2.286 +/- 0.042
Imagem Média (4096x2048) em Preto e Branco		
Seq.	2.113 +/- 0.004	-
P4	2.691 +/- 0.022	0.785 +/- 0.007
P8	2.552 +/- 0.029	0.828 +/- 0.010
gpu	0.313 +/- 0.013	6.743 +/- 0.270
Imagem Média (4096x2048) em Colorida		
Seq.	2.660 +/- 0.208	-
P4	3.074 +/- 0.035	0.865 +/- 0.068
P8	3.001 +/- 0.033	0.886 +/- 0.070
gpu	0.566 +/- 0.246	4.700 +/- 2.075
Imagem Grange (8192x4096) em Preto e Branco		
Seq.	8.502 +/- 0.143	-
P4	5.792 +/- 0.028	1.468 +/- 0.026
P8	5.266 +/- 0.048	1.614 +/- 0.031
gpu	1.074 +/- 0.432	7.917 +/- 3.185
Imagem Grange (8192x4096) em Colorida		
Seq.	11.722 +/- 0.764	-
P4	7.540 +/- 0.025	1.555 +/- 0.101
P8	7.097 +/- 0.028	1.652 +/- 0.108
gpu	3.370 +/- 0.842	3.479 +/- 0.898

Imagem Muito Grange (16384x8192) em Preto e Branco		
Seq.	35.943 +/- 1.238	-
P4	18.848 +/- 1.600	1.907 +/- 0.175
P8	15.809 +/- 0.071	2.274 +/- 0.079
gpu	4.591 +/- 0.741	7.829 +/- 1.293
Imagem Muito Grange (16384x8192) em Preto e Branco		
Seq.	47.568 +/- 0.855	-
P4	25.124 +/- 0.087	1.893 +/- 0.035
P8	23.072 +/- 0.129	2.062 +/- 0.039
gpu	13.475 +/- 1.165	3.530 +/- 0.312

5. DISCUSSÕES

Para melhor avaliação dos resultados, estes serão novamente apresentados em forma de gráficos:

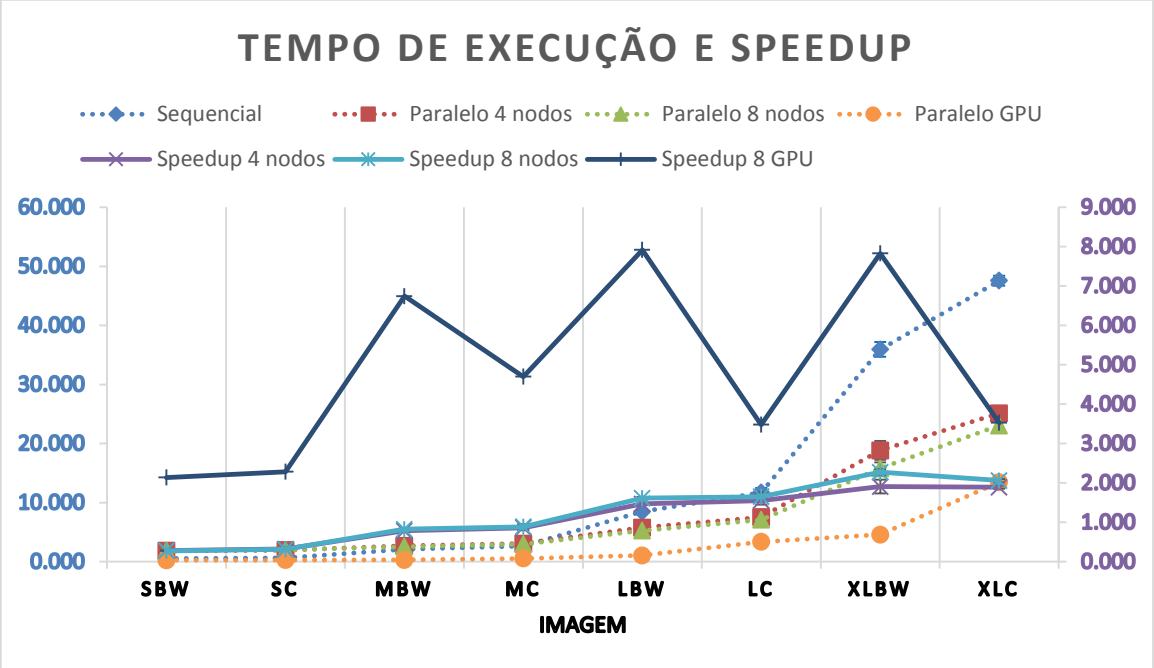

Figura 1 - Gráfico do tempo de execução e speedup para cada Imagem

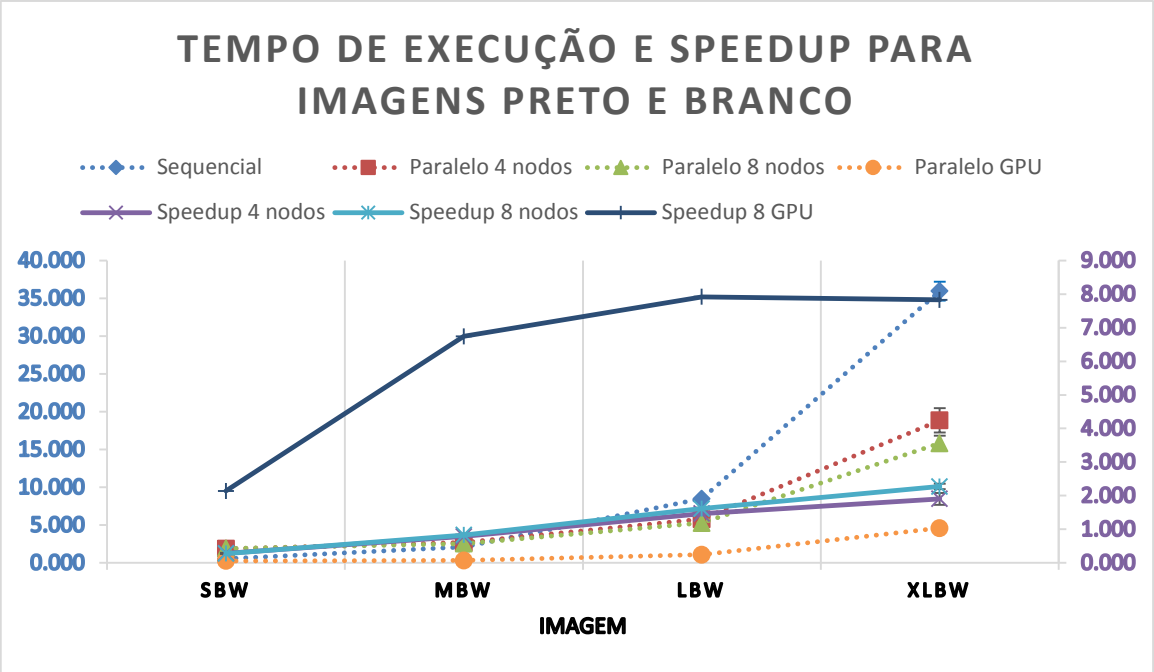

Figura 2 - Gráfico do tempo de execução e speedup para imagens em preto e branco

É notável que houve uma melhora significativa do tempo necessário para processar as imagens em preto e branco. Enquanto com OpenMP/MPI o speedup se aproxima de 2 apenas para as imagens grandes, o código em CUDA beira 8 vezes a velocidade do código sequencial nestes casos. Adicionalmente, para as imagens pequenas, nas quais a versão OpenMP/MPI possuía perda de desempenho, a nova versão para placa gráfica ainda mostra um ganho significativo de desempenho limitado, em grande parte agora, apenas pelas operações de leitura e escrita em disco.

Tal melhora no desempenho, deve-se à ausência da necessidade de transmissão dos dados pela rede, como era o caso do código OpenMP/MPI, e principalmente à subdivisão de tarefas muito superior que a placa gráfica possibilita para o processamento de imagens. Para algumas tarefas de computação científica não tão escaláveis a alternativa de usar vários processadores ainda pode ser mais vantajoso.

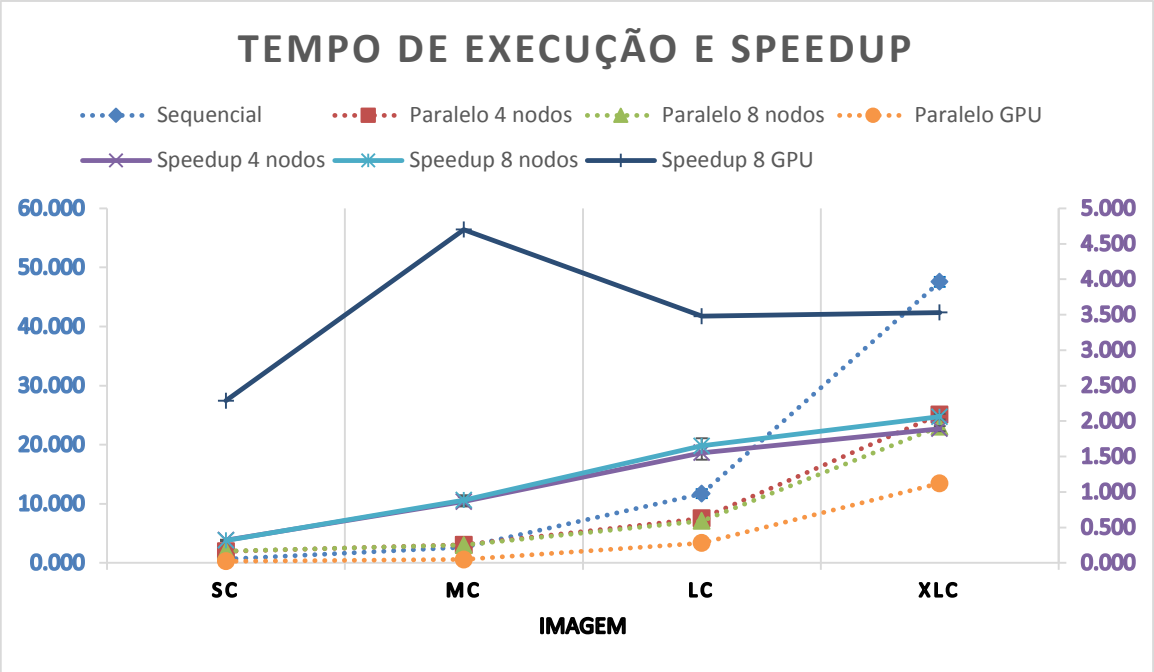

Figura 3 - Gráfico do tempo de execução e speedup para imagens coloridas

Para as imagens coloridas, apesar do speedup não ser tão bom comparado às imagens em preto e branco, ainda há um ganho de desempenho de quase 4 vezes comparado à versão sequencial. Tal aparente perda de eficiência deve-se parcialmente à inclusão das operações em disco na medição do tempo de execução.

Adicionalmente, a estratégia utilizada para leitura e escrita na memória para este caso envolve a leitura/escrita de blocos de memória espalhados dentro de uma mesma thread, o que pode comprometer a eficiência do algoritmo.

Outro ponto relevante é o uso um byte (unsigned char) para cada canal de cada pixel. Considerando que o endereçamento da placa de vídeo é por palavra (word = 4B), há uma perda considerável de eficiência nas operações de escrita tanto para imagens coloridas quanto para em preto e branco. Uma possível melhoria seria a leitura/escrita de blocos de 4x4 pixels por thread (4 Words inteiras). Para imagens coloridas, uma solução alternativa seria o agrupamento dos quatro canais em uma word.

Este efeito também ocorre para a leitura inicial da imagem mas, tendo em vista que cada thread terá que ler pixels vizinhos para processar o filtro da imagem, a única otimização que pode ser feita é estratégia implementada de uso de memória compartilhada para uma cópia local do fragmento da imagem que o bloco da placa de vídeo processará.

6. CONCLUSÃO

De acordo com os dados obtidos, as placas gráficas realmente oferecem uma alternativa muito mais rápida do que em CPUs para processamento de imagens, uma vez que este cálculo envolve poucas operações realizadas para cada pixel da figura. Mesmo quanto utilizados vários

processadores para subdividir os cálculos realizados, a GPU ainda possui um tempo de execução imbatível.

No entanto, apesar do grande aumento de eficiência usando esta técnica, a programação para placas gráficas requer um cuidado muito grande com acessos à memória para a sua otimização.

7. REFERÊNCIAS

Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). *Introduction to Parallel Computing* (2nd ed.). Pearson.

Tanenbaum, A. S. (2006). *Structured Computer Organization* (5ft ed.). Upper Saddle River, New Jersey: Pearson Education.

Young, E., & Jargstorff, F. (n.d.). *Nvidia*. Retrieved December 2015, from http://www.nvidia.com/object/nvision08-ImageVideoCUDA.html

8. APÊNDICE – A: SOBRE O PROGRAMA

O código escrito para este projeto pode ser obtido através de um repositório online do GitHub.

8.1. Projeto GitHub código sequencial e OpenMP/MPI:

https://github.com/maglethong/Trab02-Grupo20-A.git

Para compilar os programas execute:

```
mkdir bin
g++ -o ./bin/sequencial ./src/Image.cc ./src/Pixel.cc ./src/sequencial.cc
mpiCC -o ./bin/parallel ./src/Image.cc ./src/Pixel.cc ./src/parallel.cc -
      fopenmp
```

Para executar o programa sequencial execute:

```
./bin/sequencial arquivo_de_entrada arquivo_de_saida
```

Para executar o programa paralelo execute:

```
mpirun --host lista_de_nodos -np numero_de_processos ./bin/parallel
      arquivo_de_entrada arquivo_de_saida numero_de_cortes_verticais
      numero_de_cortes_horizontais
```

Importante:

```
(V)  numero_de_cortes_verticais
(H) numero_de_cortes_horizontais
(NP) numero_de_processos
NP = V*H
```

NP deve ser igual a V * H. Caso contrário o programa não executará corretamente.

Exemplo:

```
mpirun --host node03,node04,node07,node11 -np 4 ./bin/parallel
      arquivo_de_entrada arquivo_de_saida 2 2
```

8.2. Projeto GitHub código CUDA:

https://github.com/maglethong/Proj-Grupo20-A.git

Para compilar os programas execute:

```
mkdir bin
nvcc src/main.c src/image.c src/gpu.cu –o bin/prog
```

Para executar:

```
./bin/prog arquivo_de_entrada arquivo_de_saida
```

8.3. Imagens de entrada:

As imagens utilizadas para as medições encontram-se em:

```
/home/grupo20a/Imagens/
```