UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação - ICMC-USP

Naylor Garcia Bachiega (NUSP 5567669)

Relatório 3

Algoritmo sequencial e paralelo (Cuda, MPI+OpenMP) para processamento de imagens com Smooth

SÃO CARLOS

2015

SUMÁRIO

[1 Introdução 1](#_Toc437132688)

[1.1 OpenMPI 1](#_Toc437132689)

[1.2 OpenMP 1](#_Toc437132693)

[1.3 CUDA 2](#_Toc437132696)

[2 Desenvolvimento e Metodologia 3](#_Toc437132697)

[2.1 Repositório do Código 3](#_Toc437132698)

[2.2 Hardware 3](#_Toc437132699)

[2.3 Execução dos Códigos (README) 3](#_Toc437132700)

[2.4 Decomposição do Problema 3](#_Toc437132701)

[2.4.1 Algoritmo MPI+OPenMP 4](#_Toc437132702)

[2.4.2 Algoritmo CUDA 5](#_Toc437132703)

[2.5 Imagens PPM e PGM 5](#_Toc437132704)

[3 Resultados e Discussões 6](#_Toc437132705)

[3.2 Algoritmo Sequencial 6](#_Toc437132706)

[3.3 Algoritmo Paralelo: OpenMP e OpenMPI 7](#_Toc437132707)

[3.4 Algoritmo Paralelo: *CUDA* 8](#_Toc437132708)

[3.5 Eficiência do Algoritmo Desenvolvido 9](#_Toc437132709)

[3.6 SpeedUp 10](#_Toc437132710)

[3.7 Dificuldades Encontradas 11](#_Toc437132711)

[3.7.1 Problema de Escrita com Concorrência em Disco 11](#_Toc437132712)

[3.7.2 Disputa por Leitura em Disco por Processos 11](#_Toc437132713)

[3.7.3 Depuração de Software com MPI e cUDA 12](#_Toc437132714)

[3.8 Considerações Finais 12](#_Toc437132715)

[4 Referências 13](#_Toc437132716)

Lista de Figuras

Figura 1 – Comunicação ponto-a-ponto MPI (COULOURIS et al, 2005). 1

Figura 2 – Arquitetura CUDA (OLCF, 2015). 2

Figura 3 – Concorrência por recursos. 4

Figura 4 – Decomposição do problema. 4

Figura 5 – Alocação assíncrona de memória. 5

Figura 6 – Execução das imagens pelo método sequencial. 6

Figura 7 – Algoritmo MPI+OpenMP para imagens pequenas e médias. 7

Figura 8 – Algoritmo MPI+OpenMP para imagens grandes e enormes. 8

Figura 9 – Algoritmo CUDA, memória textura e global. 8

Figura 10 – Eficiência do algoritmo sequencial e paralelo. 9

Figura 11 – SpeedUp entre os algoritmos paralelo e sequencial. 10

Lista de Tabelas

Tabela 1 – Imagens utilizadas nos testes. 5

Tabela 2 – Execução das imagens pelo algoritmo sequencial. 6

Tabela 3 – Execução com cargas aleatórias para o algoritmo MPI+OpenMP. 7

Tabela 4 – Execução das imagens pelo algoritmo CUDA. 9

Tabela 5 – Eficiência dos algoritmos sequencial, MPI e CUDA. 10

Introdução

Este trabalho mostra o desenvolvimento de um algoritmo sequencial e paralelo, usando CUDA e MPI+OpenMP, para o processamento de imagens PPM (*Portable Pixmap Format*) e PGM (*Portable Graymap Format*), utilizando um filtro para suavização de imagens (denominado *smooth*).

## OpenMPI

A interface de passagem de mensagens (MPI) é um padrão desenvolvido para fornecer uma API para um conjunto de operações de transmissão de mensagens com variantes síncronas e assíncronas. O modelo de arquitetura subjacente para MPI é relativamente simples, conforme pode ser visto na Figura 1:

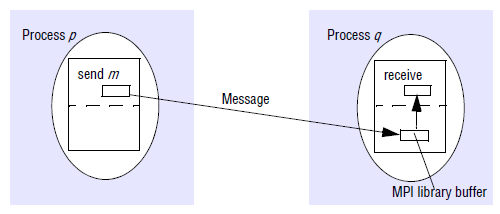


Figura 1 – Comunicação ponto-a-ponto MPI (COULOURIS et al, 2005).

Um processo p envia uma mensagem para um processo q através de uma biblioteca MPI, inicialmente essa mensagem é enviada ao buffer de q, e após, repassada ao processo q (COULOURIS et al, 2005).



## OpenMP

*OpenMP* é uma API (*Application Program Interface*), definida em conjunto por um grupo de grandes fornecedores de *hardware* e *software*. Fornece um modelo portátil, escalável para desenvolvedores de aplicações paralelas de memória partilhada. A API suporta C/C ++ e Fortran em uma ampla variedade de arquiteturas.

Programas *OpenMP* realizam paralelismo exclusivamente através da utilização de *threads*. Um *thread* é a menor unidade de processamento que pode ser programada por um sistema operacional. Normalmente, o número de *threads* coincide com o número de processadores/núcleos. *OpenMP* é um modelo de programação explícita (não automático), oferecendo ao programador controle total sobre a paralelização. (BARNEY, 2015)



## CUDA

CUDA é uma plataforma de computação paralela desenvolvida pela NVIDIA. Ela permite aumentos significativos de desempenho computacional ao aproveitar a potência da Unidade de Processamento Gráfico (GPU). CUDA é uma abstração usada para permitir que o programador ou compilador use mais facilmente os recursos da GPU (NVIDIA, 2015).

Quando o trabalho é enviado para a GPU é sob a forma de uma função, referido como o núcleo, que é para ser executado em paralelo N vezes por threads. Conforme pode ser observado na Figura 2, cada thread é logicamente dividido em 1,2 ou 3 grupos dimensionais referidos como blocos de threads. Cada thread tem acesso sua posição dentro do seu próprio bloco, bem como a posição do bloco dentro da grid (OLCF, 2015).

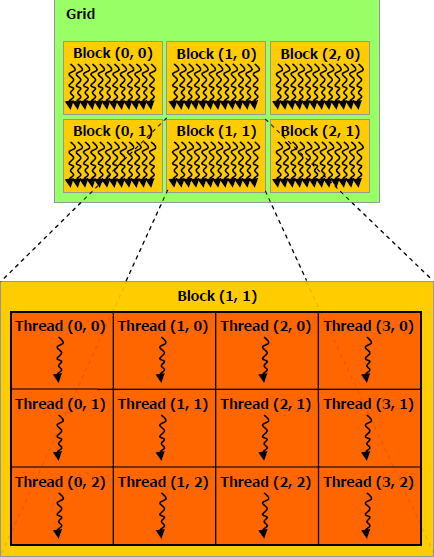


Figura 2 – Arquitetura CUDA (OLCF, 2015).

# Desenvolvimento e Metodologia

## Repositório do Código

Como o trabalho foi desenvolvido em grupo, o GitHub foi utilizado para compartilhar o código e controlar o versionamento. O repositório pode ser acessado pelo link: **https://github.com/naylor/Proj-Grupo11-AePos**

## Hardware

Para esse trabalho foi utilizado o *Cluster* Halley do LaSDPC (*Laboratory of Distributed Systems and Concurrent Programming*). Para o MPI e OpenMP foram utilizados os nós: frontend, rj, mg, pr, sc, rs, go, ms, ba, ms, pa. Para o algoritmo sequencial e CUDA foi utilizado o frontend.

## Execução dos Códigos (README)

No GitHub é possível verificar o README completo para execução dos códigos. Abaixo segue um exemplo dos três algoritmos após clonar o repositório e compilar os códigos:

./PPMcuda -i [IMAGEM] -m [MEMORIA TEXTURA] -c [CARGA TRABALHO(Opcional)] -d [NIVEL DEBUG(Opcional)]

./PPMseq -i [IMAGEM] -d [NÍVEL DEBUG]

mpiexec -n [PROCESSOS] -f [NODES] ./PPMmpi -i [IMAGEM] -t [NÚMERO THREADS] -c [CARGA DE TRABALHO] -r [CARGA ALEATÓRIA] -l [LEITURA INDIVIDUAL] -d [NÍVEL DEBUG]

## Decomposição do Problema

Um algoritmo paralelo é derivado a partir de uma escolha de distribuição de dados. A distribuição de dados deve ser equilibrada, alocar (aproximadamente) o mesmo número de entradas para cada processador; e deverá minimizar a comunicação.

### Algoritmo MPI+OPenMP

O algoritmo foi modelado de forma que o trabalho seja divido logo no início e que os processos recebam os elementos para processamento, estimulando a concorrência de recursos, conforme demonstrado na Figura 3, assim, enquanto um processo aplica o filtro, o outro pode estar gravando ou lendo a imagem.



Figura 3 – Concorrência por recursos.

No algoritmo desenvolvido, os dados em que os cálculos são efetuados são particionados, e em um segundo momento, este particionamento de dados é utilizado para induzir uma compartimentação dos cálculos em tarefas.



Figura 4 – Decomposição do problema.

A Figura 4 demonstra o funcionamento do algoritmo paralelo, em que Rank 0 faz a leitura somente do cabeçalho, pois este contém o tamanho da imagem. Após, ele grava o cabeçalho da nova imagem e envia algumas configurações para os Ranks. O Rank 0 inicia os Threads de 1..N de acordo com o número de processos. Cada thread controla, envia a carga de trabalho e permite acesso a gravação para seu processo.

### Algoritmo CUDA

**Cópia Assíncrona:** para o algoritmo desenvolvido em CUDA, foi utilizada cópia assíncrona de memória entre o *host* e o *device*, isso permite que as chamadas do *kernel* não fiquem bloqueadas, permitindo que o host possa executar outras tarefas, aumentando a concorrência de processos.

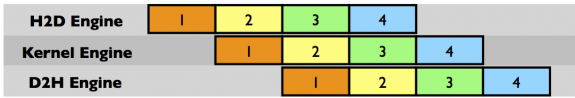


Figura 5 – Alocação assíncrona de memória.

**Memória de Textura:** a memória de textura foi escolhida, pois fornece uma combinação de memória cache, capacidade de processamento local e um modo de interagir com a GPU. O uso de memória de textura pode melhorar o desempenho de largura de banda e latência. Assim, foram criados blocos (16, 16) e a imagem foi particionada em grids de acordo com seu tamanho e a capacidade da memória de textura.

## Imagens PPM e PGM

Para os testes foram utilizadas imagens do site HiRISE (*High Resolution Imaging Science Experiment*): http://www.uahirise.org/katalogos.php, tanto imagens coloridas como imagens em escada de cinza, conforme demostrado na Tabela 1:

Tabela 1 – Imagens utilizadas nos testes.

|  |  |  |  |
| --- | --- | --- | --- |
| **Nome:** ESP\_042084\_1690  **Tamanho:** 34,1MB | **Nome:** ESP\_042084\_1690  **Tamanho:** 33,6MB | **Nome:** ESP\_042433\_1535  **Tamanho:** 66,6MB | **Nome:** ESP\_042433\_1535  **Tamanho:** 55,1MB |
| **Nome:** ESP\_042641\_1900  **Tamanho:** 223MB | **Nome:** ESP\_042641\_1900  **Tamanho:** 375MB | **Nome:** ESP\_024497\_1745  **Tamanho:** 750MB | **Nome:** ESP\_024497\_1745  **Tamanho:** 679MB |

# Resultados e Discussões

## Algoritmo Sequencial

Os testes foram realizados com imagens coloridas e em tons de cinza utilizando o algoritmo sequencial, obtendo os resultados conforme a Tabela 2:

Tabela 2 – Execução das imagens pelo algoritmo sequencial.

|  |  |  |
| --- | --- | --- |
| IMAGEM  *n x n* | MÉDIA  (milissegundos) | DESVIO  (milissegundos) |
| Pequena RGB (7105x1678) | 4214,53 | 26,02 |
| Pequena GRAY 8624x4096 | 10113,12 | 3,03 |
| Média RGB (10275x2268) | 8229,25 | 31,48 |
| Média GRAY (10272x5626) | 27597,24 | 14,26 |
| Grande RGB (16184x4832) | 27597,24 | 14,26 |
| Grande GRAY (21421x18373) | 112713,53 | 69,07 |
| Enorme RGB (36873x7115) | 92515,23 | 34,12 |
| Enorme GRAY (39941x17834) | 201751,45 | 2320,92 |

Foram realizadas as médias das dez execuções para cada imagem e o seu desvio padrão. A Figura 6 mostra os tempos de processamento das imagens coloridas e com tons de cinza.

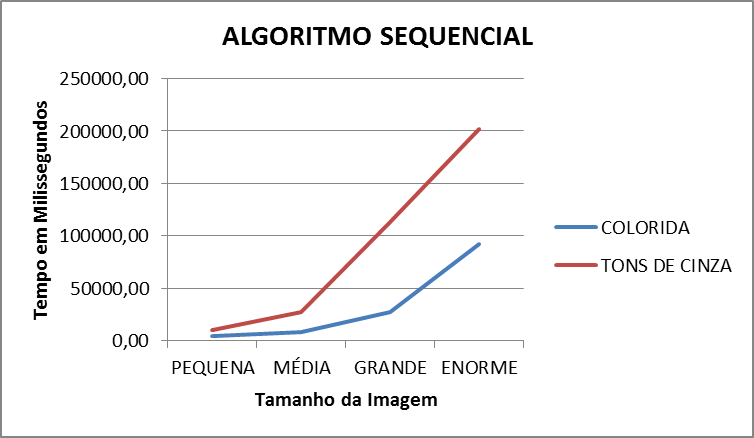
****

Figura 6 – Execução das imagens pelo método sequencial.

## Algoritmo Paralelo: OpenMP e OpenMPI

Uma das formas de equilibrar a disputa quando se tem grandes quantidades de informações, é enviar cargas aleatórias, baseado no algoritmo *QuickSort*. Desse modo, uma das opções de execução do sistema desenvolvido é enviar a *Flag* **a = 1**. Na Tabela 3 é possível verificar a média e o desvio padrão para as melhores médias:

Tabela 3 – Execução com cargas aleatórias para o algoritmo MPI+OpenMP.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| IMAGEM | NODES | PROCESSOS | THREADS | FILTRO  Milissegundos | DESVIO |
| Pequena RGB | 12 | 40 | 8 | 50,01 | 5,91 |
| Pequena GRAY | 12 | 40 | 8 | 160,01 | 6,42 |
| Média RGB | 12 | 40 | 8 | 132,75 | 6,51 |
| Média GRAY | 12 | 40 | 8 | 277,47 | 10,93 |
| Grande RGB | 12 | 40 | 8 | 497,14 | 13,65 |
| Grande GRAY | 12 | 40 | 8 | 2066,12 | 67,46 |
| Enorme RGB | 12 | 40 | 5 | 1839,29 | 31,83 |
| Enorme GRAY | 12 | 40 | 8 | 3728,71 | 88,85 |

A Figura 7 e Figura 8 mostram a evolução do algoritmo de acordo com o número de nós, processos e threads.

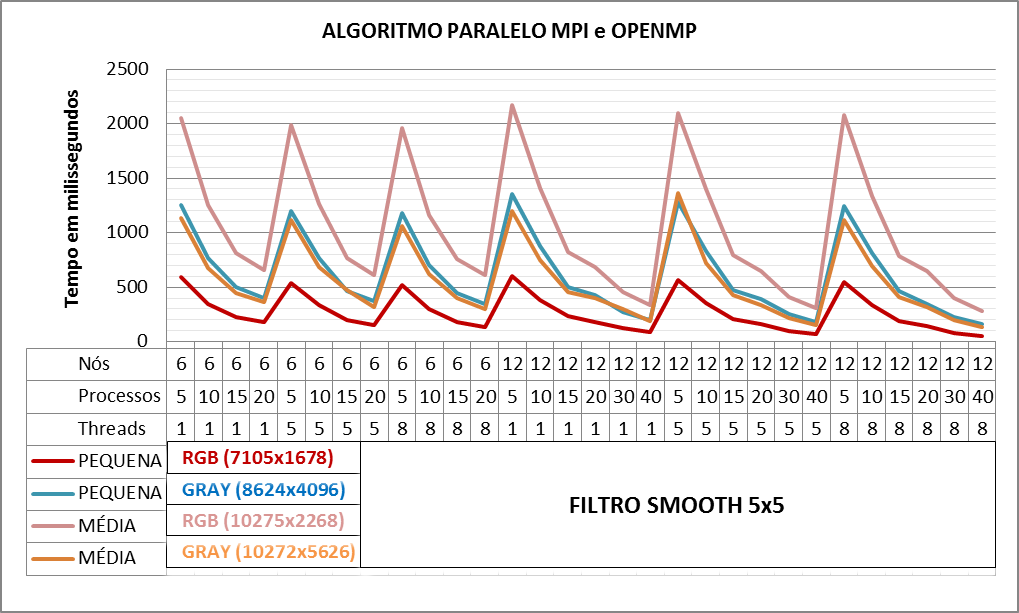


Figura 7 – Algoritmo MPI+OpenMP para imagens pequenas e médias.

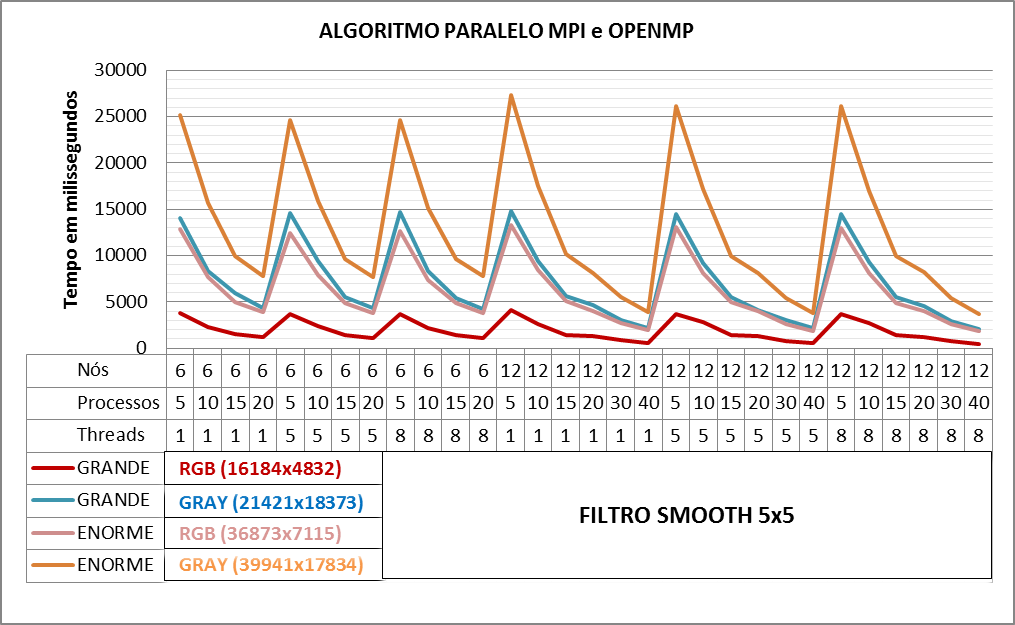


Figura 8 – Algoritmo MPI+OpenMP para imagens grandes e enormes.

## Algoritmo Paralelo: *CUDA*

Foram realizados lançamentos com memória global e textura. A Figura 9 mostra a diferença dos resultados.

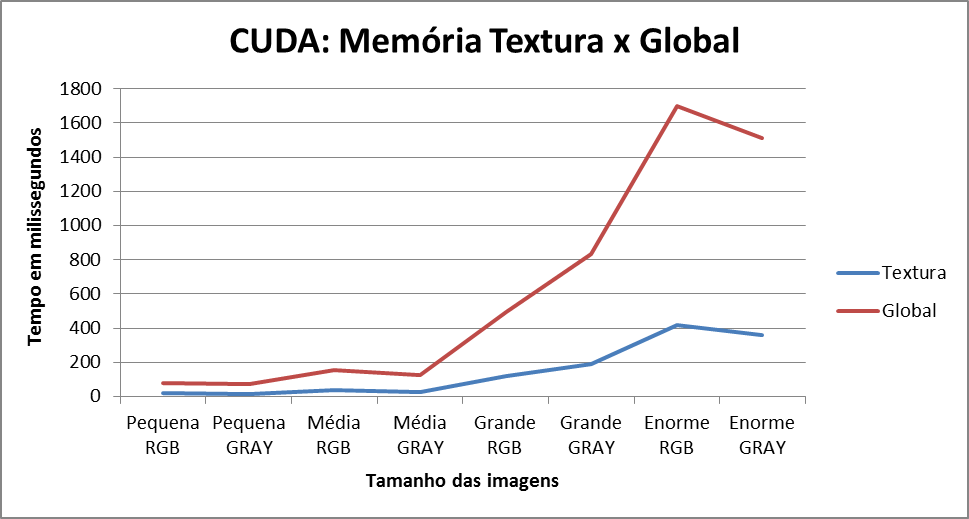


Figura 9 – Algoritmo CUDA, memória textura e global.

A Tabela 4 mostra os valores dos lançamentos com as melhores médias e os seus desvios para cada tipo de lançamento:

Tabela 4 – Execução das imagens pelo algoritmo CUDA.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| IMAGEM  *n x n* | MÉDIA TEXTUA  (milissegundos) | DESVIO  (milissegundos) | MÉDIA GLOBAL | DESVIO |
| Pequena RGB (7105x1678) | 18,88 | 0,02 | 78,93 | 0,00 |
| Pequena GRAY 8624x4096 | 16,29 | 0,06 | 72,54 | 0,00 |
| Média RGB (10275x2268) | 36,54 | 0,03 | 155,89 | 0,01 |
| Média GRAY (10272x5626) | 29,13 | 0,06 | 124,33 | 0,02 |
| Grande RGB (16184x4832) | 119,72 | 0,13 | 495,64 | 0,02 |
| Grande GRAY (21421x18373) | 192,53 | 0,21 | 830,65 | 0,03 |
| Enorme RGB (36873x7115) | 416,12 | 0,18 | 1696,91 | 0,10 |
| Enorme GRAY (39941x17834) | 356,95 | 0,16 | 1509,20 | 0,05 |

## Eficiência do Algoritmo Desenvolvido

A Figura 10 evidencia a diferença da solução paralelizada MPI e CUDA. Os resultados da versão sequencial não foram incluídos na figura pela grandeza dos valores.

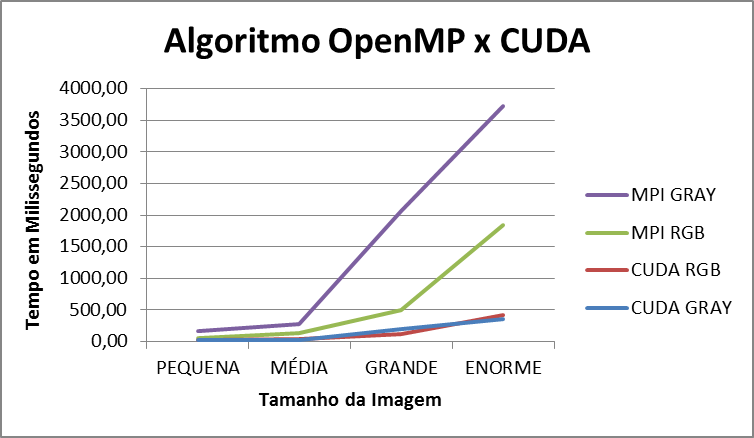


Figura 10 – Eficiência do algoritmo sequencial e paralelo.

A Tabela 5 mostra a eficiência do algoritmo sequencial comparado com os melhores resultados, após a média das dez iterações, dos algoritmos MPI e CUDA:

Tabela 5 – Eficiência dos algoritmos sequencial, MPI e CUDA.

|  |  |  |  |
| --- | --- | --- | --- |
| IMAGEM | MÉDIA SEQUENCIAL  (milissegundos) | MÉDIA MPI  (milissegundos) | MÉDIA CUDA  (milissegundos) |
| Pequena RGB (7105x1678) | 4214,53 | 50,01 | 18,88 |
| Pequena GRAY 8624x4096 | 10113,12 | 160,01 | 16,29 |
| Média RGB (10275x2268) | 8229,25 | 132,75 | 36,54 |
| Média GRAY (10272x5626) | 27597,24 | 277,47 | 29,13 |
| Grande RGB (16184x4832) | 27597,24 | 497,14 | 119,72 |
| Grande GRAY (21421x18373) | 112713,53 | 2066,12 | 192,53 |
| Enorme RGB (36873x7115) | 92515,23 | 1839,29 | 416,12 |
| Enorme GRAY (39941x17834) | 201751,45 | 3728,71 | 356,95 |

## SpeedUp

Para medir o aumento de desempenho do processamento utiliza-se o cálculo chamado de *Speedup* que determina a relação existente entre o código executado em *threads* e sequencial. A medida tem por objetivo determinar a relação existente entre o tempo dispensado para executar um algoritmo em um único processador (T1) e o tempo gasto para executá-lo em p processadores (Tp): Speedup = T1/Tp (ROHDE et al, 2015). A Figura 11 mostra essa relação.

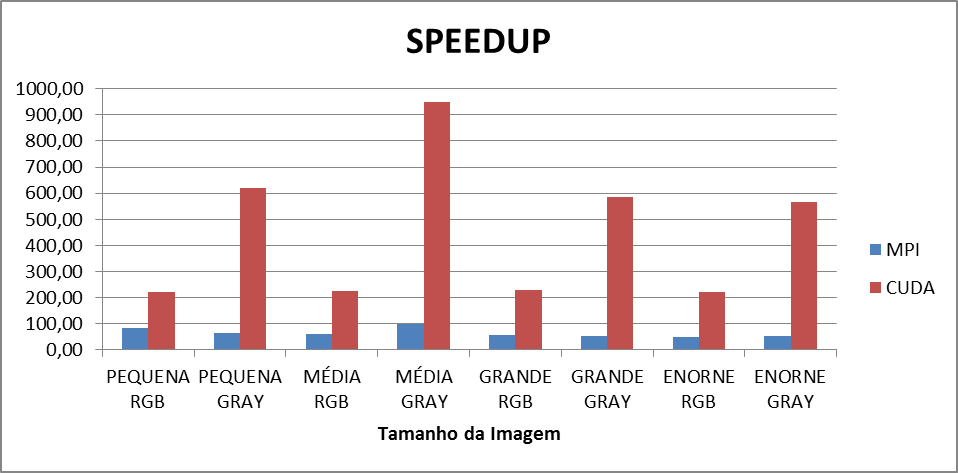
****

Figura 11 – SpeedUp entre os algoritmos paralelo e sequencial.

## Dificuldades Encontradas

Algumas dificuldades foram encontradas no decorrer do trabalho, essa seção aponta quais foram elas e quais as soluções adotadas.

### Problema de Escrita com Concorrência em Disco

Nesse trabalho, foi estabelecido que os nós gravassem diretamente sua parte da imagem no disco, evitando assim, excesso de comunicação na rede. Inicialmente nenhum tipo de semáforo havia sido definido para controlar qual processo poderia gravar sua parte da imagem no disco.

* Problema: nos testes com imagens pequenas nenhum problema foi detectado, porém com imagens maiores foi constatado que algumas partes da imagem não eram gravadas. Isso acontece, pois para um processo gravar uma parte de uma imagem, ele precisa saber o que há no arquivo para conservar o conteúdo existente de outro processo que já tenha gravado. Todavia, se dois processos gravarem ao mesmo tempo, um não saberá o conteúdo do outro e o último processo que finalizar a escrita, conservará apenas o conteúdo anterior da imagem e o seu conteúdo.
* Solução: para contornar esse problema, foi colocado um semáforo no Rank 0, que controla qual o nó que poderá gravar no disco. Assim, quando um processo terminar de aplicar o filtro, este solicita para seu *thread* responsável no Rank 0 para gravar em disco.

### Disputa por Leitura em Disco por Processos

Inicialmente acreditava-se que a disputa em disco por processos, para realizar a leitura da imagem, poderia diminuir o desempenho da aplicação. Nesse caso, também foi criado um semáforo para leitura, porém nos testes ficou claro que a leitura não compromete o desempenho.

Acredita-se que cache de disco e do NFS, a concorrência natural pelos processos e a quantidade pequena de linhas que são lidas, pode justificar esse resultado.

### Depuração de Software com MPI e cUDA

Outro problema encontrado é a forma como o OpenMPI e CUDA apresentam os seus erros e os erros na aplicação. Estes, muitas vezes, são difíceis de interpretar e falta informação, fazendo-se necessário recorrer para aplicações paralelas, como por exemplo, valgrind.

## Considerações Finais

Conforme pode ser observado ao longo desse trabalho, paralelizar aplicações permitiu um ganho de tempo substancial em vista de algoritmos sequenciais. Algumas dificuldades foram encontradas utilizando OpenMPI, OpenMP e CUDA, porém o resultado é bastante satisfatório.

Pelos testes realizados, a aplicação paralela desenvolvida é escalável, tanto MPI quanto a versão em CUDA. Outro fato interessante, é que enviar cargas aleatórias diminuiu visivelmente o tempo de processamento. Uma possibilidade para melhorar o desempenho do algoritmo é integrar o CUDA e MPI para distribuir a imagem entre as GPUs dos nós.

Por fim, ressalta-se que a disciplina foi, extremamente, importante para a exploração do paralelismo e seus desafios. A quantidade de informação adquirida ao longo de quatro meses transpassou a barreira da sala e, eficazmente, perpetuou o conhecimento.

# Referências

BARNEY, Blaise. **OpenMP**. Disponível em: < https://computing.llnl.gov/tutorials/openMP/>. Acesso em: 03 out. 2015.

BROOKSHEAR, J. Glenn. **Ciência da computação: uma visão abrangente**. 11ª ed. Porto Alegre: Bookman, 2013.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. **Distributed Systems - Concepts and Design**. Addison Wesley Publ. Comp., 4th edition, 2005.

GRAMA, A. Gupta, G. Karypis, and V. Kumar. **Introduction to Parallel Computing**. 2nd edition, Addison Wesley; 2003.

HPC. **Examples of MPI Programs**. Disponível em <https://hpcc.usc.edu/support/documentation/examples-of-mpi-programs/>. Acesso em: 23 out. 2015.

KIESSLING, Alina. **An Introduction to Parallel Programming with OpenMP**. Disponível em: <http://www.roe.ac.uk/ifa/postgrad/pedagogy/2009\_kiessling.pdf>. Acesso em: 04 out. 2015

LAMMPS, Molecular Dynamics Simulator. ***GPU and USER-CUDA package benchmarks on Desktop system with Fermi GPUs***. Disponível em: <https://computing.llnl.gov/tutorials/pthreads/#Pthread>. Acesso em: 05 set. 2015

NETBPM. **PPM**. Disponível em: <http://netpbm.sourceforge.net/doc/ppm.html>. Acesso em: 23 de out. 2015.

NVIDIA. **O que é *CUDA*?**. Disponível em: <http://www.nvidia.com.br/object/cuda\_home\_new\_br.html>. Acesso em: 02 set. 2015

ROSETTA CODE. **Bitmap/Write a PPM file**. Disponível em: <http://rosettacode.org/wiki/Bitmap/Write\_a\_PPM\_file>. Acesso em 23 de out. 2015.

SATO, Liria Matsumoto; GUARDIA, Hélio Crestana. **Programando para múltiplos processadores: *Pthreads, OpenMP* e *MPI*.** Disponível em: <http://erad.dc.ufscar.br/mc/eradsp2013-multiproc-3.pdf>. Acesso em: 02 set. 2015.

OLCF. **CUDA Thread Model**. Disponível em: <https://www.olcf.ornl.gov/kb\_articles/cuda-thread-model/>. Acesso em: 05 dez. 2015.

OPENMP. ***What is OpenMP*?**. Disponível em: <http://www.openmp.org/mp-documents/paper/node3.html>. Acesso em: 02 set. 2015.

OPEN-MPI. ***General information about the Open MPI Project***. Disponível em: <https://www.open-mpi.org/faq/?category=general>. Acesso em: 05 set. 2015.

YANO, Luis Gustavo Abe. **Avaliação e comparação de desempenho utilizando tecnologia CUDA**. São José do Rio Preto: 2010.