

Technical Report - SCC0143 - Relatório Técnico

December - 2015 - Dezembro

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

O conteúdo deste relatório é de única responsabilidade dos autores.

##### Trabalho 3 de Programação Concorrente

##### Comparação de performance entre a versão OpenMP e CUDA de uma aplicação de Smooth

*Fernando Gorodscy Leonardo Rebelo*

*7152354 5897894*

UNIVERSIDADE DE SÃO PAULO

INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

**Comparação de performance entre a versão OpenMP e CUDA de uma aplicação de Smooth**

Fernando Gorodscy1, Leonardo Rebelo2

1 Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (USP) - Campus São Carlos, Av. Trabalhador São-carlense, 400. CEP: 13566-590 [fegorodscy@gmail.com](mailto:fegorodscy@gmail.com)

2 Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (USP) - Campus São Carlos, Av. Trabalhador São-carlense, 400. CEP: 13566-590 leonardorebelo88@gmail.com

**Resumo.**Este relatório é fruto do projeto desenvolvido na disciplina de Programação Concorrente e visa o estudo de um algoritmo para aplicar um filtro de Smooth em uma imagem, comparando sua performance em uma execução utilizando a biblioteca OpenMP e outra, utilizando CUDA. A realização deste trabalho contou com o suporte do professor Júlio Cezar Estrella e os estagiários PAE, membros do Laboratório de Sistemas Distribuídos e Programação Concorrente (LaSDPC) USP –Campus de São Carlos. Indo além, o laboratório colaborou com o desenvolvimento deste projeto permitindo acesso ao seu cluster para a realização de todos os testes necessários.

**Palavras-Chave:** Programação Concorrente; Cluster; Algoritmo Paralelo; Smooth; CUDA; OpenMP;

0.README

Repositório git: https://github.com/wakku/trab3\_concorrentes

Para compilar a versão sequencial rode o seguinte comando na pasta com o código fonte:

$ make paralelo\_seq

Para compilar a versão OpenMP rode o seguinte comando na pasta com o código fonte:

$ make paralelo\_openmp

Para compilar a versão CUDA rode o seguinte comando na pasta com o código fonte:

$ make paralelo\_cuda

Para executar rode./bin/smooth\_seq para a versão ./bin/smooth\_openmp para a versão OpenMP e ./bin/smooth\_cuda para a versão CUDA.

Para medir o tempo de execução rode o shell script:

$ ./run\_all.sh

1. Introdução

O problema consiste em aplicar um filtro de smooth em uma imagem JPG e salvá-la em um novo arquivo, com o filtro aplicado.

No decorrer deste relatório será mostrado a abordagem escolhida para a solução deste problema e as diferentes formas utilizadas para paralelizar o mesmo.

O grande diferencial que será visto é o uso de CUDA, uma plataforma de computação paralela que utiliza as placas de processamento gráfico (GPU) NVIDIA para resolver problemas que podem ser paralelizados, tendo um grande aumento de performance.

As sessões a seguir abordam os conceitos necessários para entender o problema e a solução proposta, após será mostrado o problema e o funcionamento dos algoritmos escolhidos para resolver o mesmo, bem como as técnicas para a paralelização dos algoritmos.

Na sessão 4 é explicado o ambiente experimental utilizado para a execução dos testes e, por fim, na sessão 5 e 6 será mostrado a execução e resultados dos programas sendo executados de forma sequencial e de forma paralela utilizando OpenMP e CUDA, juntamente com uma análise dos dados colhidos, concluindo o relatório.

Caminho das imagens: Imagens/

2. Conceitos Envolvidos

2.1 CUDA

A CUDA, sigla para Compute Unified Device Architecture, é uma plataforma sobre a linguagem de programação C que permite o desenvolvimento em computação paralela nas placas de processamento gráfico (GPUs) desenvolvidas pela NVIDIA.

Quando uma tarefa muito complexa é executada, ela pode ser dividida em atividades mais simples, que são distribuídas pelos núcleos do processador, permitido aumentos significativos de performance computacional ao aproveitar esta potência muitas vezes não utilizada da GPU.

A NVIDIA disponibiliza o CUDA Toolkit, que oferece o ambiente de desenvolvimento para desenvolvedores C e C++ e utilizado foi nesse trabalho.

3. Problema Proposto

O objetivo deste programa é receber uma imagem, aplicar um filtro de Smoth (também conhecido como Blur) e salvar a imagem gerada. A técnica utilizada para o nosso filtro implementado consiste em realizar a média aritmética para uma máscara de tamanho 5x5. Ou seja, a partir do pixel central que se deseja suavizar, somamos seus vizinhos dentro desta máscara e dividimos pelo número de elementos da máscara. Como estamos lidando com uma imagem colorida, é necessário dividir esta imagem em canais de cor e aplicar o filtro individualmente para cada canal, juntando-os no final do processo.

Ao lidar com imagens ou vídeos grandes, como por exemplo uma resolução de 20 Mega Pixels, tal filtro mostra-se lento, surgindo dai a necessidade de paraleliza-lo.

3.1 Estratégia de Decomposição e Mapeamento

A estratégia utilizada para a paralelização deste problema foi a Decomposição de Dados Intermediários com Mapeamento Estático dos Dados. Utilizamos um vetor para armazenar todos os canais de cores da matriz da imagem. Este vetor é particionado em três partes: vermelho, verde e azul. Dentro de cada parte de cor, há uma divisão de blocos, e dentro de cada bloco há uma thread para cada pixel que o compõe. O número de blocos define-se pela divisão da área da imagem pelo número de threads a serem usadas. Na figura na pagina seguinte esta exemplificado a decomposição e o mapeamento a ser usado na função paralelizada do filtro de Blur.

A escolha desta estratégia de decomposição deve-se à facilidade de se trabalhar com um vetor dentro da estrutura CUDA, quebrando-o em blocos e threads de maneira simples e intuitiva. A abordagem de mapeamento estático foi escolhida por se tratar de um vetor que não sofrerá alteração de dimensão, pois a imagem é fixa. Além disso, a função de Blur é aplicada independentemente em cada pixel, sendo seu resultado salvo em um vetor de saída. Por isso, a ordem que o filtro é aplicado nos pixels não interfere no resultado final. Por fim, esta estratégia possibilita o uso máximo de processadores da placa gráfica, assim como as threads internas de cada processador, maximizando o desempenho do algoritmo.

3.2 Algoritmo

Os algoritmos utilizados com OpenMP e com CUDA são similares. A principal diferença é que OpenMP utiliza o paralelismo dos cores do processador multi-core (ou no caso de OpenMPI, explora-se o paralelismo dos muitos computadores do cluster), enquanto que o CUDA explora o paralelismo dos muitos cores da placa de Vídeo (GPU).

Primeiramente alocamos em memória o espaço para carregar a imagem. Depois separamos a imagem em seus 3 canais de cores, Vermelho, Verde e Azul. Agora podemos dividir a imagem em blocos de acordo com o número de processadores disponíveis, seja na GPU (caso CUDA) ou na CPU (caso OpenMP).

Com a imagem dividida em blocos e seus canais de cores separados, cada thread (caso OpenMP) ou cada processador/core (Caso CUDA) processa os 3 canais de um bloco da imagem aplicando o filtro.

Foi utilizado um filtro de tamanho 5x5, exceto para os pixel da extremidade da imagem, onde pode não haver uma região 5x5 nas proximidades dos pixels. Neste caso, utilizamos para o calculo apenas os pixels próximos possíveis.

O filtro de smooth consiste em, para cada pixel da imagem exceto a borda, somar os seus 24 vizinhos tanto horizontal quando verticalmente e depois dividi-los pelo numero de vizinhos. Este filtro é aplicado em sequência para as três matrizes de cor e o resultado é salvo nos respectivos canais de cores da imagem de saída, obtendo assim as cores suavizadas da imagem.

O último passo é juntar os três canais resultantes do filtro em um arquivo JPG e salvá-lo no disco.

# **4. Ambiente Experimental**

**Hardware:** Intel Core i7 @ 2.3GHz (4 Cores), Macbook Pro 15, Memory: 16 GB, Disk: 512 GB APPLE SSD SM0512F, Graphics: NVIDIA GeForce GT 750M

**Software:** Mac OS X 10.11.1 El Captain, Compiler: Apple LLVM version 7.0.0 (clang-700.1.76), File-System: OS X Extended, Screen Resolution: 2880x1800

**Placa de Video:**

Name: GeForce GTX 650

Compute capability: 3.0

Clock rate: 1071500

Device copy overlap: Enabled

Kernel execution timeout : Disabled

--- Memory Information for device 0 ---

Total global mem: 1073414144

Total constant Mem: 65536

Max mem pitch: 2147483647

Texture Alignment: 512

--- MP Information for device 0 ---

Multiprocessor count: 2

Shared mem per mp: 49152

Registers per mp: 65536

Threads in warp: 32

Max threads per block: 1024

Max thread dimensions: (1024, 1024, 64)

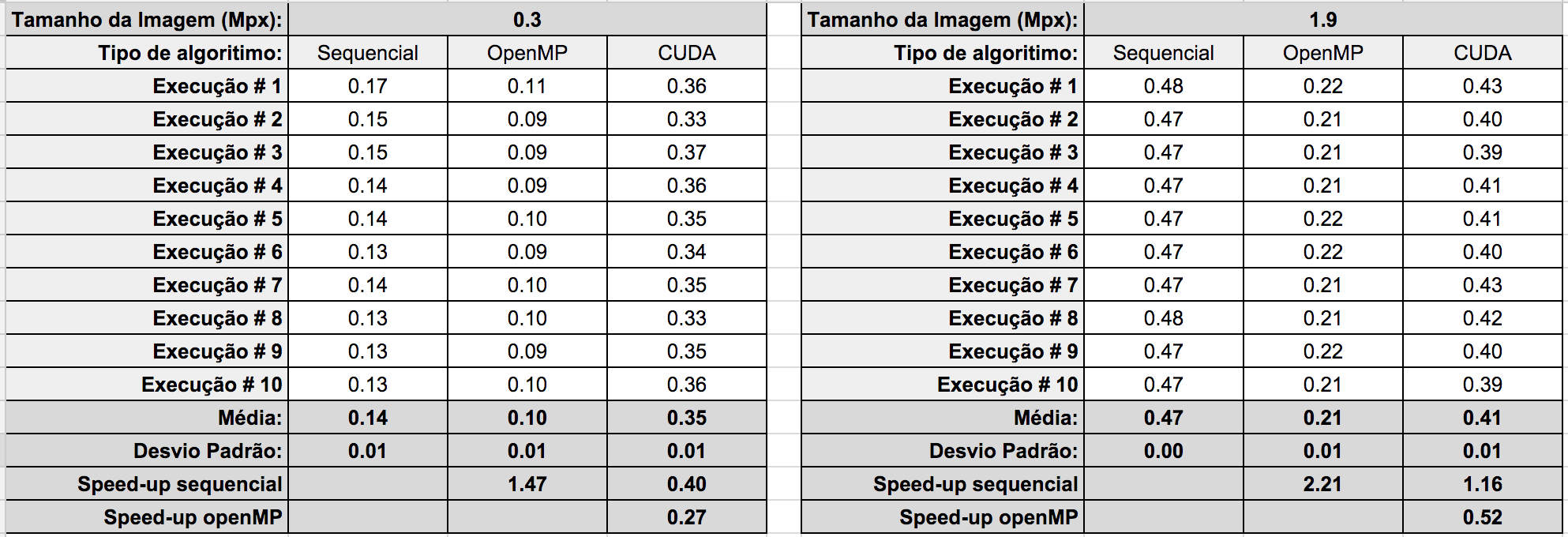
Max grid dimensions: (2147483647, 65535, 65535)

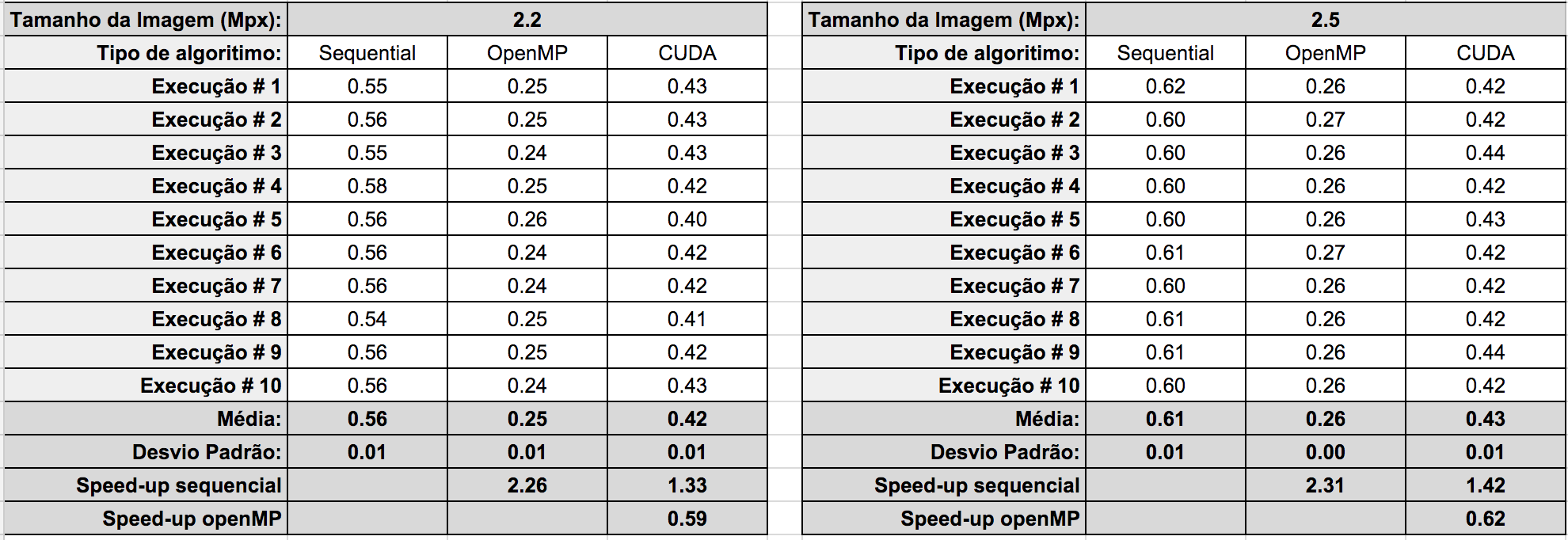
# **5.Resultados**

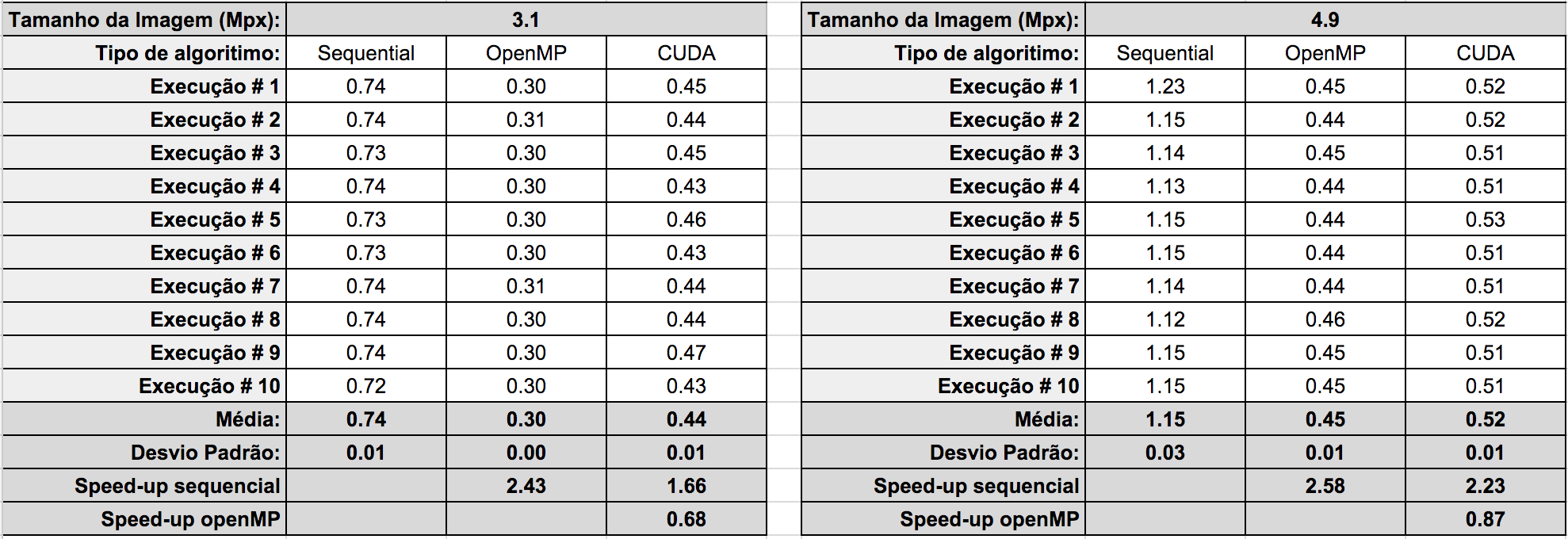
Uma comparação entre a imagem original e a imagem de saída com a aplicação do filtro de Blur é mostrada abaixo:

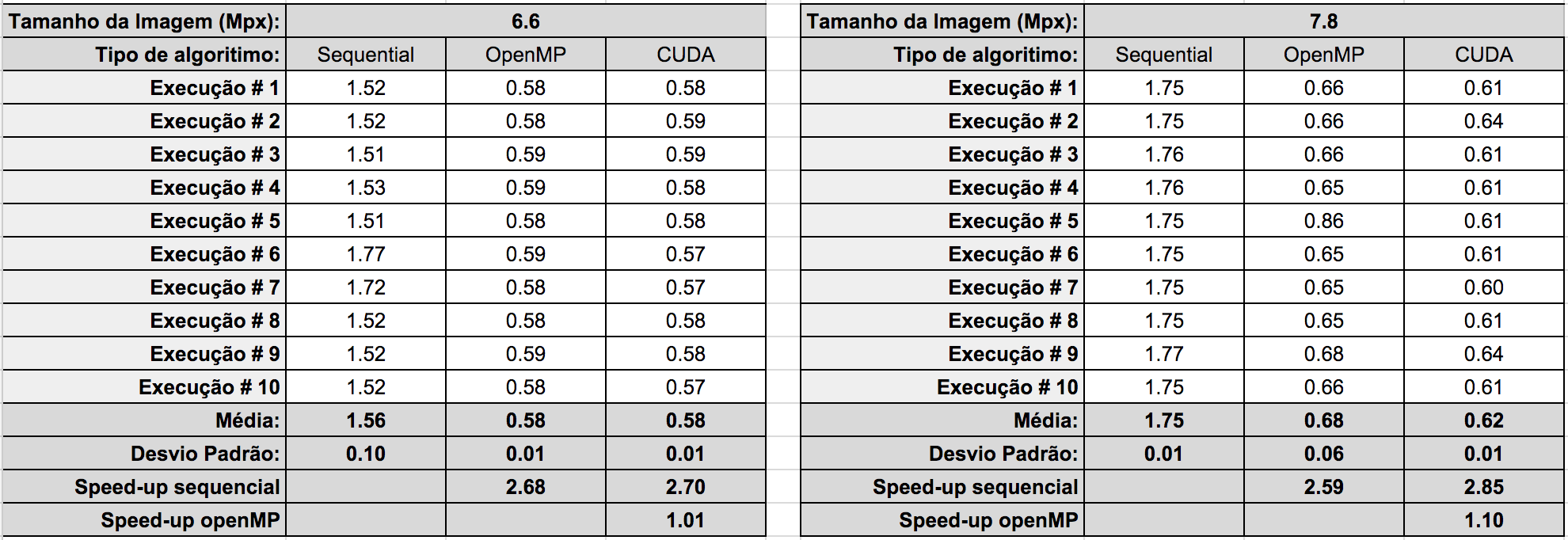


Tabelas com o tempo de execução de cada algoritmo:









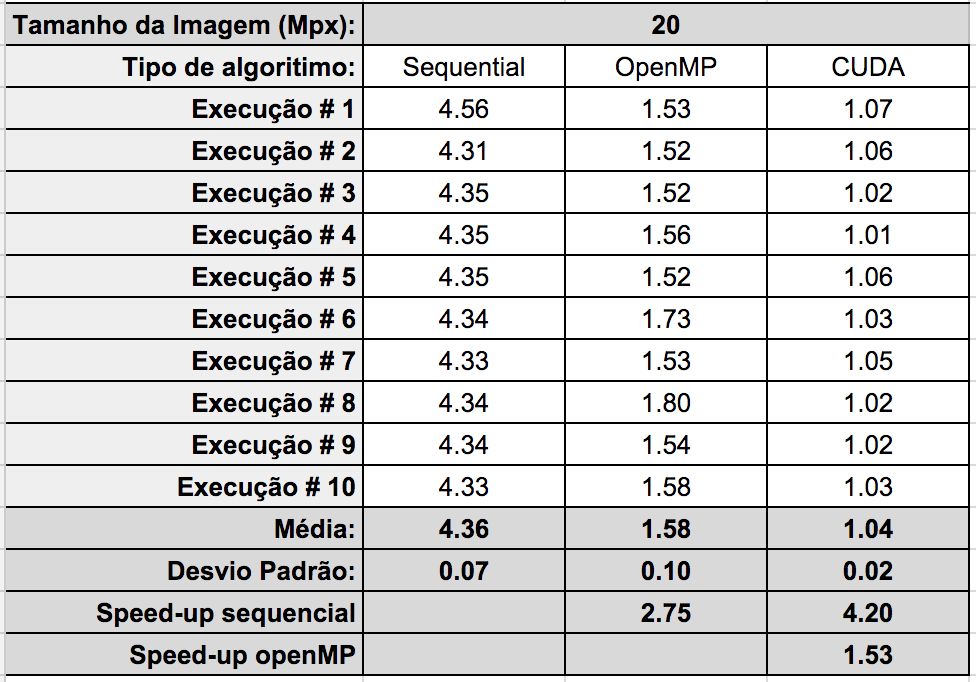


Tabela com o tempo médio de execução de cada algoritmo para cada imagem:

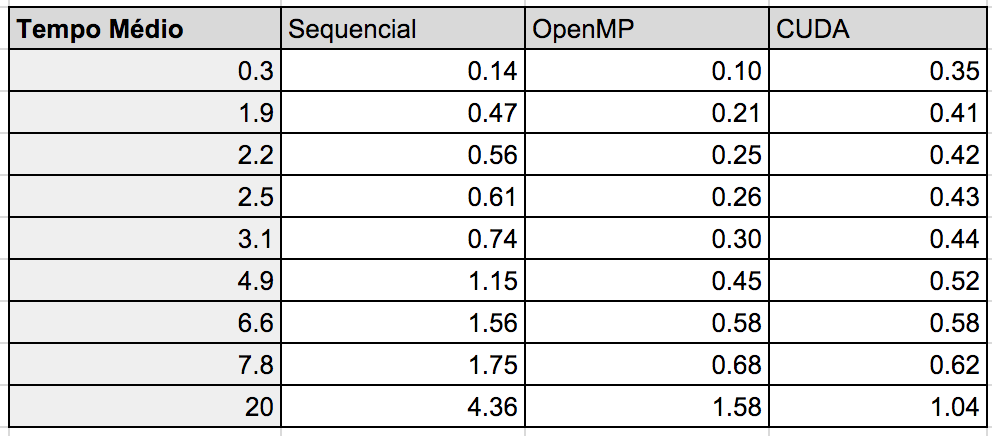


Tabela com o speed-up de cada algoritimo em relação ao outro:

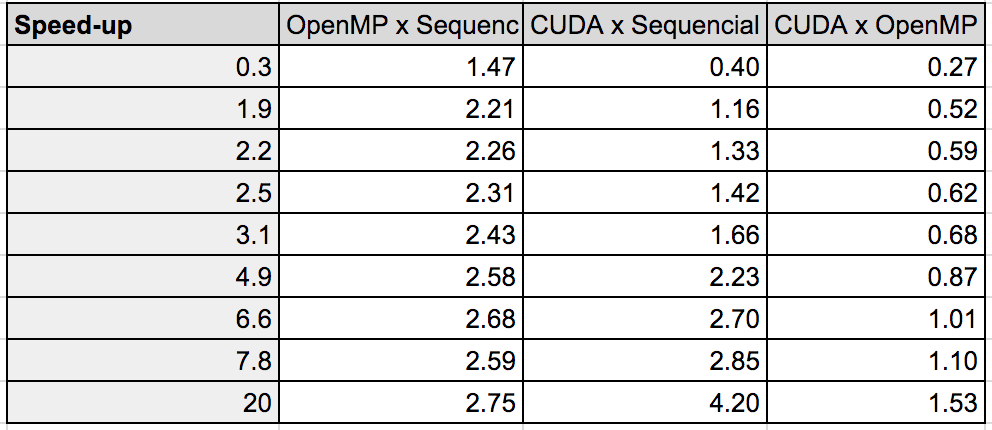


Gráfico com o tempo de execução de cada algoritmo:

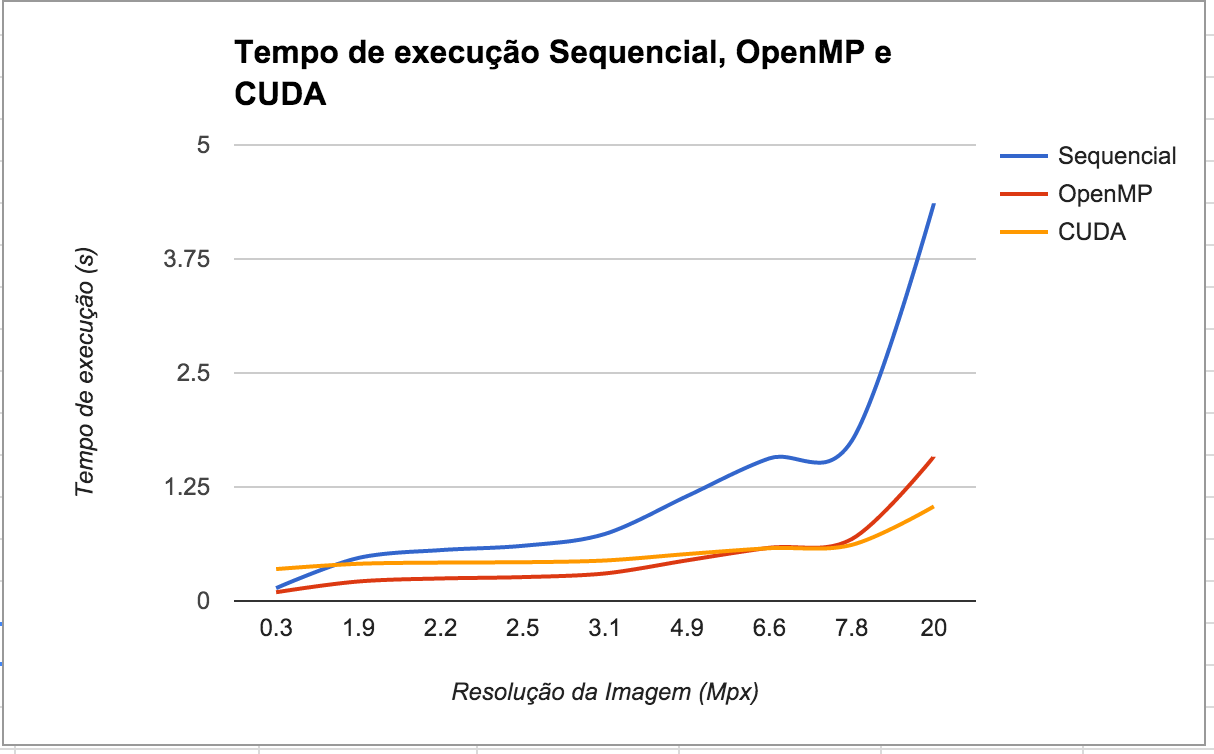
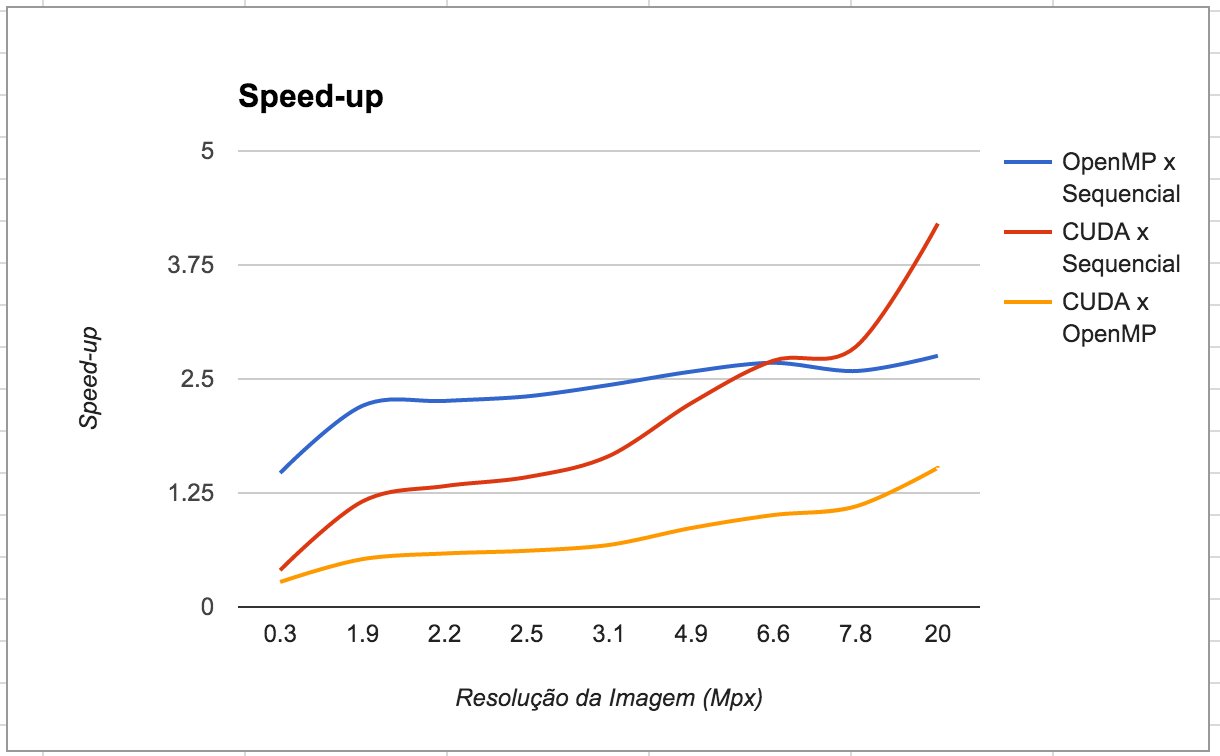


Gráfico com o speed-up de cada algoritmo em relação ao outro:



# **6. Conclusão**

Com base nos resultados e gráficos vistos acima, concluímos que para imagens pequenas o OpenMP se sai melhor, para imagens de até aproximadamente 6 Mega Pixels. Já para imagens maiores o algoritmo CUDA se sai melhor, chegando a ser 53% mais eficiente que o algoritmo OpenMP e 420% melhor que o algoritmo sequencial para imagens de 20 Mega Pixels. Em nenhum caso o algoritmo sequencial se saiu melhor do que ambos os algoritmos paralelos, sendo no melhor dos casos (imagem de 0.3Mpx) 68% inferior do que o algoritimo paralelo OpenMP.

No quesito de uso da capacidade dos processadores da placa de vídeo, com o uso de blocos e threads, não conseguimos encontrar uma vantagem expressiva entre diferentes configurações do mesmo. No entanto, o maior desempenho surgiu com o uso de 512 processadores, como notado nas tabelas de tempo de execução dos algoritmos. Concluímos portanto que a paralelização se vem muito útil para o tratamento de imagens grandes ou pequenas, sendo mais eficaz do que executar o código de maneira sequencial e vital para o futuro da área de processamento de imagens.

# **7. Referências**

<http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/01-intro_to_cuda_c.pdf>

Introduction to Parallel Computing (2nd Edition) [Ananth Grama, George Karypis, Vipin Kumar, Anshul Gupta]