Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação

SSC0143 – Programação Concorrente

Trabalho 3 – *Smooth*

Paralelização com MPI e CUDA

Elias Italiano Rodrigues – 7987251
Rodolfo Megiato de Lima – 7987286
Vinicius Katata Biondo – 6783972

São Carlos, 7 de dezembro de 2014

Repositório: `https://code.google.com/p/pc2014-02-grupo2-turmab/`

Sumário

1 Introdução

1.1 *Smooth* em imagens

Neste trabalho é implementado um algoritmo para fazer *smooth* em um imagem no formato PPM P3. Nesta seção é explicado o formato de arquivo PPM P3 e como funciona o algoritmo para aplicar o efeito *smooth*. Na Seção 2, é explicado o desenvolvimento da modelagem do problema e codificação. Na Seção 3, resultados são apresentados e analisados. Por fim, é descrita uma conclusão sobre a realização deste trabalho na Seção 4.

Este relatório faz parte de um conjunto de arquivos referentes ao trabalho. A documentação do código-fonte encontra-se nos arquivos `.c` dentro de `./src` a partir do diretório raiz.

1.2 O Formato PPM P3

O formato de arquivo PPM P3 consiste em representar os dados de uma imagem em um arquivo ASCII. O cabeçalho do arquivo contém a palavra "P3" seguida de dois valores que representam a largura e comprimento da imagem e de um valor que representa o valor máximo para cada pixel da imagem. Por exemplo, o seguinte cabeçalho diz que a imagem tem 800px de largura, 600px de altura e que os pixels tem valor máximo de 255:

```
P3
800 600
255
```

Após o cabeçalho, seguem os valores das componentes *Red, Green, Blue* (RGB) de cada pixel da imagem. No exemplo acima, após o valor 255, seguem $800 \times 600 \times 3 = 1440000$ valores, sendo a cada três referente a um componente RGB de um pixel.

1.3 O Algoritmo

O algoritmo aplicado consiste em: dada como entrada uma imagem de dimensões $w \times h$, para cada pixel $p_i = (R_i, G_i, B_i)$, $i = 0, 1, \ldots, wh - 1$ da imagem, montar um bloco de pixels $B_i = (b_{i,0}, b_{i,1}, \ldots, b_{i,24})$ de área $5 \times 5 = 25$ tomando $b_{i,12} = p_i$ como pixel central desse bloco e os demais $b_{i,j \neq 12}$ como os pixels ao redor de p_i na imagem de modo a compor a área do bloco. Caso não exista valor para algum $b_{i,j}$, se assume $b_{i,j} = (0, 0, 0)$.

Então é feita a média aritmética dos valores de cada pixel de B_i e o resultado é atribuído ao pixel p_i na imagem. Uma ilustração desse processo é mostrada na Figura 1.

$$p_i = \frac{\sum_{j=0}^{24} b_{i,j}}{25} \ , \ i = 0, 1, \ldots, wh - 1 \tag{1}$$

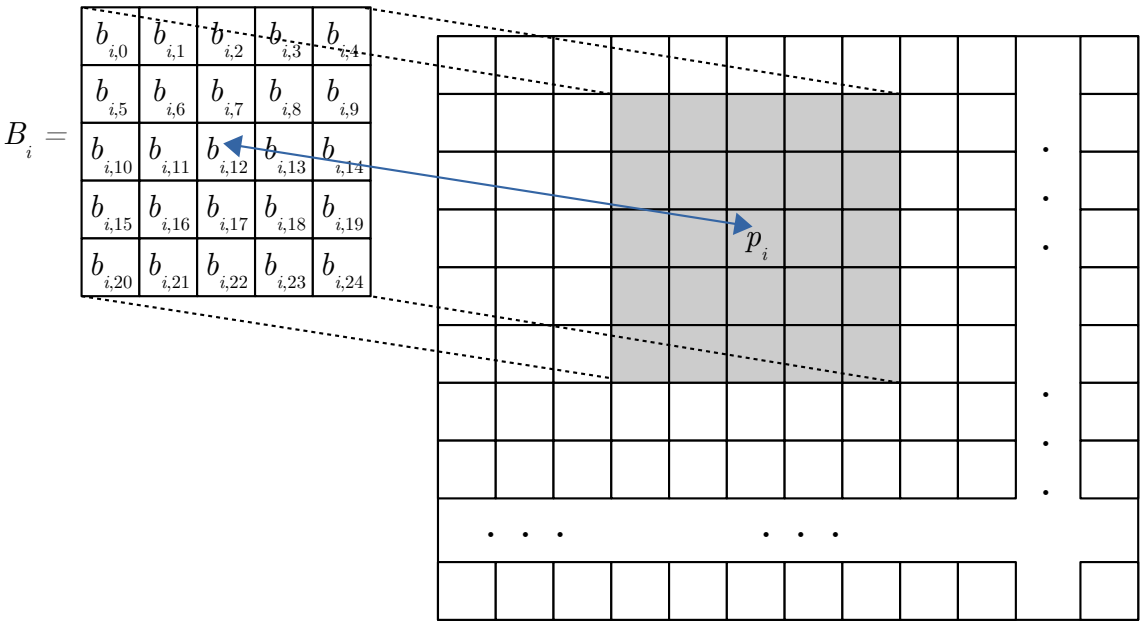

Figura 1: Ilustração da aplicação de *smooth* em um pixel p_i de uma imagem.

2 Desenvolvimento

O programa foi desenvolvido em linguagem `C` em sistema operacional `GNU/Linux`. Além das bibliotecas convencionais `stdlib.h` e `stdio.h`, foi utilizada também a `getopt.h` para leitura de argumentos na linha comando, a `mpi.h` fornecida pelo Open MPI [1] e as funções e tipos de variáveis fornecidas pelo CUDA [2].

Do MPI, usou-se apenas as principais funções como `MPI_Init()`, `MPI_Comm_rank()`, `MPI_Comm_size()`, `MPI_Send()`, `MPI_Recv()` e `MPI_Finalize()`.

Do CUDA, usou-se `__global__` para definir um *kernel* a ser executado por cada *thread* da placa de vídeo associada (*device*) e as variáveis pré-definidas para definir/receber os índices das *threads* e *thread blocks* como `dim3`, `blockIdx`, `blockDim` e `threadIdx`.

2.1 *Smooth* – Sequencial

Na implementação do código sequencial, a imagem PPM P3 é lida totalmente para a memória e armazenada em uma matriz de pixels na variável `pixel_t **img`. O tipo de dado `pixel_t` é definido da seguinte forma:

```
typedef unsigned char pixel_t[3];
```

Sendo o suficiente para armazenar os valores RGB de um pixel. Foi também criada uma estrutura de dados para armazenar o cabeçalho da imagem PPM P3. Todas as estruturas de dados usadas nos programas estão definidas e documentadas no arquivo `./inc/ppm_p3.h`.

Tendo lido a imagem, ela é então passada para a função `void smooth(pixel_t **img, int width, int height)` que, para cada pixel da imagem, pega os valores adequados como explicado na Seção 1.3 e ilustrado na Figura 1, e então é computada a aritmética definida na Equação 1 que é atribuída ao pixel na imagem, ou seja, na variável `img`.

Ao final do processo, os ruídos da imagem terão sidos suavizados. Caso a imagem não contenha ruídos, o efeito *smooth* acaba deixando-a mais embaçada como se percebe no exemplo da Figura 2.

Figura 2: Exemplo da imagem "Lena" [4] antes e depois da aplicação do *smooth*.

2.2 *Smooth* – Paralelo com MPI

Para a implementação da versão paralela com MPI, usou-se as mesmas estruturas de dados da versão sequencial, inclusive a mesma função `smooth()`, porém com uma adaptação para receber uma *flag* que indica o tipo de pedaço da imagem: 0 imagem inteira, 1 último pedaço, 2 primeiro pedaço e 3 pedaço intermediário. Isso foi necessário, pois ao fatiar a imagem horizontalmente, como descrito mais adiante, precisa-se enviar 2 linhas a mais para cima ou para baixo de modo que a aplicação do *smooth* seja correta num pedaço na imagem.

```
void smooth(pixel_t **img, int width, int height, int flag)
```

Como ilustrado no exemplo na Figura 3, optou-se por fazer um fatiamento da imagem em tiras horizontais, proporcionando uma melhor divisão do trabalho qualquer que seja a quantidade de processos. Desse modo, é feito uma partição dos dados de entrada e seguido um modelo mestre-escravo, como explicado no livro-texto da disciplina [3].

Para fazer o fatiamento das regiões da imagem, seguiu-se a mesma aritmética proposta em um tutorial [5] de MPI para distribuir a soma de valores de um vetor.

Considerando um exemplo de um ambiente de rede, como ilustrado na Figura 4, com 4 máquinas h_i, $i = 0, 1, 2, 3$, e criados 8 processos MPI P_j, $j = 0, 1, \ldots, 7$ distribuídos 2 por máquina, a implementação paralela com MPI procede da seguinte forma:

1. O processo *master* P_0 localizado em h_0 é responsável por alocar o tamanho total da imagem em memória e ler para ela os dados que são recebidos da entrada padrão.

2. P_0 faz o fatiamento da imagem em tiras horizontais.

3. Então P_0 envia uma região (quantidade de linhas da imagem) para cada outro processo $P_{j\neq 0}$ localizado nas máquinas h_i.

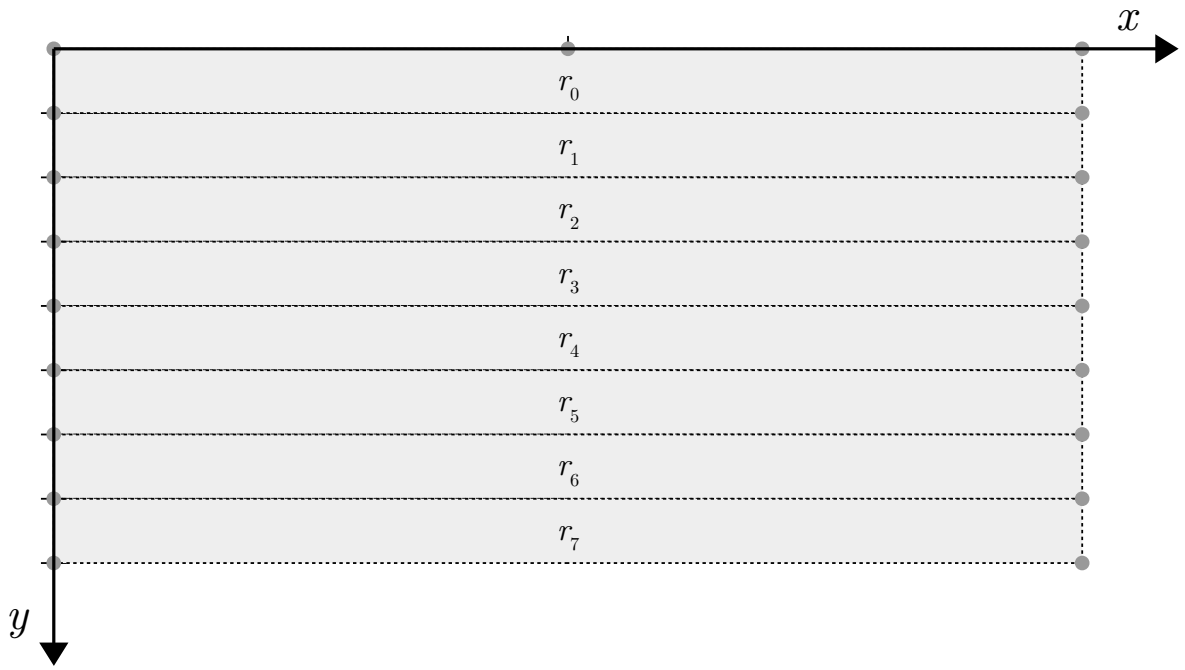

Figura 3: Ilustração do fatiamento de uma imagem em 8 regiões.

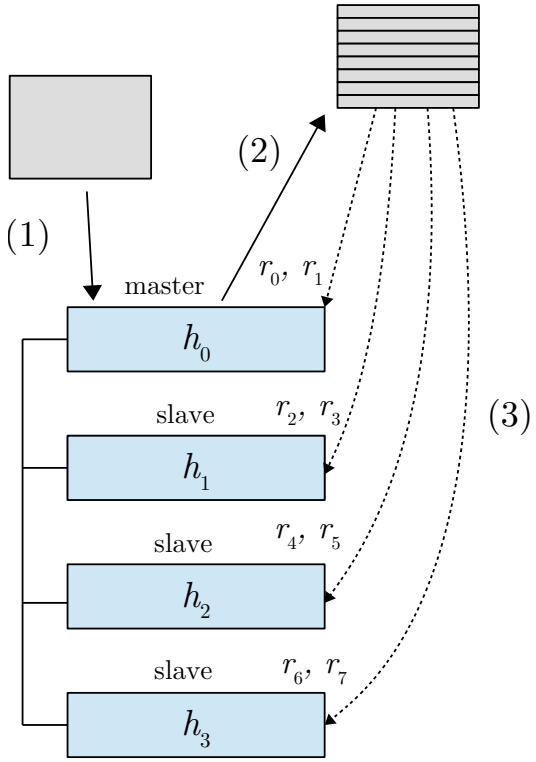

Figura 4: Ilustração da distribuição das tarefas entres as máquinas para o exemplo.

Esse exemplo é ilustrado na Figura 4.

Após o processamento do trabalho por cada processo MPI escravo, o resultados são enviados de volta para o processo master seguindo a ordem inversa dos fluxos da Figura 4. E então o processo master recompõem a imagem e retorna ela inteira com o efeito *smooth* aplicado.

2.3 Paralelo com CUDA

Para a versão paralela com CUDA, usou-se ainda o mesmo tipo de dado `pixel_t` para os pixels da imagem, porém, por facilidade de implementação da parte de cópia da imagem entre *host* e *device*, fez-se a imagem vetorizada e não mais como uma matriz.

O programa em CUDA consiste em:

- Alocação de memória para a imagem no *host* (`pixel_t *img`) e no *device* (`pixel_t *img_in, *img_out`).
- Leitura da imagem no *host* pela entrada padrão.
- Copia da imagem de `img` para `img_in`.
- Definição da quantidade de *threads* por bloco e número de blocos.
- Chamada da função `__global__ void cudaSmooth(pixel_t *img_in, pixel_t *img_out, int width, int height)`
- Copia do resultado de `img_out` para `img`.

Da maneira como foi implementada, cada *thread* fica responsável por calcular o *smooth* de um pixel da imagem o que resulta numa execução muito rápida, pois a GPU contém uma grande quantidade de unidades lógica-aritmética. Para isso, foi usada a quantidade máxima de *threads* por bloco (1024) e fazendo blocos de duas dimensões 32×32. O uso de blocos bidimensionais é de fácil associação com o problema em questão por se tratar de uma imagem que possui duas dimensões.

A função *kernel* para calcular o *smooth* não é a mesma usada na versão sequencial e MPI. Foi necessário criar uma nova função ligeiramente alterada para CUDA. Mais detalhes sobre sua implementação estão no código-fonte `./src/cuda.cu`.

2.4 Cálculo das Estatísticas

Além dos programas para a resolução do problema, foi implementado um outro programa para auxiliar o cálculo das médias, desvios padrão e intervalos de confiança dos tempos das execuções. Esse programa `./bin/stats` recebe como entrada o número de execuções seguido dos tempos de cada execução de um dos programas. O arquivo dos tempos de execução é gerado através de um *shell script* `./run.sh` que roda os programas uma quantidade informada de vezes.

2.5 Compilação e Execução dos Programas

Estando no diretório raiz dos arquivos deste trabalho, é possível compilar e executar os programas seguindo as instruções:

- Para compilar todos programas, execute:
  `make`

- Para rodar a versão sequencial, execute:
  `./bin/sequential < in.ppm > out.ppm`
- Para rodar a versão paralela MPI, execute:
  `mpirun -hostfile ./hosts -np <#> -npernode <#pernode>`
  `./bin/parallel < in.ppm > out.ppm`

  Por exemplo:
  `mpirun -hostfile ./hosts -np 8 -npernode 2 ./bin/parallel < in.ppm > out.ppm`
- Para rodar a versão em CUDA, execute:
  `./bin/cuda < in.ppm > out.ppm`
- Opcionalmente, os programas podem receber os seguinte argumentos:
  `--no-output` ou `-n` para não gerar a saída.
  `--time` ou `-t` para imprimir o tempo em segundos gasto somente com o processamento.
- Para limpar todos os arquivos compilados, execute:
  `make clean`
- Para rodar vários testes paralelo, sequencial ou cuda, execute:
  `./run.sh <program> <#> <#pernode> <file> <#times>`

  Exemplos:
  `./run.sh parallel 8 2 in.ppm 10`
  `./run.sh sequential - - in.ppm 10`
  `./run.sh cuda - - in.ppm 10`
- Para conferir por vazamento de memória, execute:
  `make memcheck`

Caso não consiga compilar e rodar os programas, confira por dependências de uma biblioteca MPI e CUDA, assim como dos programas usados `make`, `gcc` e `valgrind` no seu sistema operacional.

3 Resultados

Os programas foram compilados e executados de modo automatizado por *shell scripts*. Para as versões sequencial e paralela MPI, foram usadas as máquinas do cluster Cosmos que apresentam a seguinte configuração:

```
cosmos.lasdpc.icmc.usp.br
Hardware:
Processor: Intel Core 2 Quad Q9400 @ 2.67GHz (4 Cores)
Motherboard: Gigabyte G41MT-S2P
Chipset: Intel 4 DRAM + ICH7
Memory: 8192MB
Disk: 500GB Western Digital WD5000AACS-0
Graphics: Intel 4 IGP
Network: Realtek RTL8111/8168/8411
```

```
Software:
OS: Ubuntu 14.04, Kernel: 3.13.0-36-generic (x86_64)
```

Para a versão em CUDA, utilizou-se o cluster Andromeda que possui a seguinte configuração:

```
andromeda.lasdpc.icmc.usp.br
*-memory: 31GiB
*-cpu AMD FX(tm)-8350 Eight-Core Processor 4GHz 64 bits
*-display GK107 [GeForce GTX 650] NVIDIA Corporation 64 bits
*-network RTL8111/8168/8411 Gigabit Ethernet Controller 1Gbit/s 64 bits
```

Os resultados de tempos a seguir são provenientes de 10 execuções consecutivas dos programas nos clusters citados acima. Foi considerado para os testes a imagem `big-in.ppm` de dimensões 10720×7120, pois apresenta um tamanho considerável (> 70 Megapixel) e é possível de ser copiada para o *device* na implementação em CUDA sem problemas.

Para a versão MPI, foram escolhidos 19 processos MPI para rodar 1 por máquina nas 19 máquinas disponíveis no momento da execução dos testes no cluster Cosmos. Cada uma das máquinas ficou somente com 1 processo com a intenção de minimizar a quantidade de dados transferidos pela rede, uma vez que pode consideravelmente prejudicar o *speed up*. Os arquivos completos dos resultados aqui apresentados podem ser consultados no diretório `./test` deste trabalho.

Com o objetivo de fazer uma comparação mais honesta, foi calculado para cada execução somente o tempo gasto com processamento. Assim, usou-se das funções `gettimeofday()` na versão sequencial e CUDA, e `MPI_Wtime()` na versão paralela com MPI para saber o tempo de execução e não o programa `/usr/bin/time` como sugerido no enunciado do trabalho. Ignorando a tempo de leitura da entrada e escrita da saída, tem-se os seguintes tempos mostrados nas Tabela 1, 2 e 3.

Execução	**Sequencial**	**Paralelo MPI**	**Paralelo CUDA**
1	22.73	5.07	0.85
2	22.74	4.82	0.83
3	22.77	5.07	0.83
4	29.75	5.04	0.91
5	30.28	4.92	0.83
6	30.32	4.79	0.83
7	30.30	5.04	0.83
8	22.76	5.04	0.84
9	22.73	5.06	0.83
10	30.31	5.09	0.83

Tabela 1: Tempos (s) de 10 execuções dos programas sequencial, paralelo MPI e CUDA.

Programa	**Média**	**Desvio padrão**	**Intervalo de confiança**
Sequencial	26.469000	3.726308	[24.159411, 28.778589]
Paralelo MPI	4.994000	0.104326	[4.929338, 5.058662]
Paralelo CUDA	0.841000	0.023854	[0.826215, 0.855785]

Tabela 2: Estatísticas dos tempos (s) das execuções sequencial, paralelo MPI e CUDA.

Programa	***Speed up***
Sequencial	—
Paralelo MPI	5.300160192
Paralelo CUDA	31.473246136

Tabela 3: *Speed up* das execuções sequencial, paralelo MPI e CUDA.

De acordo com os resultados apresentados, percebe-se que houve um ganho de desempenho nos processamentos paralelos. A versão paralela MPI teve um *speed up* de aproximadamente 5 no processamento dos dados enquanto a versão paralela com CUDA teve *speed up* de 31.

Os resultados do paralelo com MPI são ainda prejudicados por um fator inerente: a transferência de dados pela rede do cluster. Isso tem influência direta no tempo total gasto com processamento e foi um dos motivos de ter sido fixado somente 1 processo MPI por máquina. Com mais testes talvez se possa concluir que mesmo aumentando a quantidade de máquinas e processos, o ganho de *speed up* não cresce proporcionalmente.

Como na versão com CUDA são lançadas uma *thread* para cada pixel, o tempo gasto com o processamento é quase constante em função do tamanho da entrada, pois GPU consegue gerenciar uma grande quantidade de *threads*. Uma vez que a imagem toda e sua cópia com o resultado processado caibam na memória da placa de vídeo (*device*), não é necessário fazer particionamento da imagem. E em comparação com o MPI, o tempo gasto para copiar a imagem inteira para o *device*, e depois o resultado de volta, é muito inferior por se tratar de tranferências entre memórias principais dentro do mesmo computador.

4 Conclusão

Neste trabalho foram desenvolvidos três programas que implementam o algoritmo *smooth*. Uma versão sequencial e duas paralelas sendo uma com MPI e outra com CUDA. A proposta de implementação do algoritmo é simples, porém, dentro do contexto de paralelismo foi necessário estudar o problema para definir como utilizar dos recursos disponíveis visando um melhor desempenho.

Dado que foram ignoramos os tempos gastos com leitura da entrada e escrita da saída, foi possível obter um valor considerável nos *speed ups* calculados como mostrado na Seção 3. Isso mostra que a operação de *smooth*, por mais que uma simples média aritmética, pode ser paralelizada para uma aplicação de modo a buscar uma execução mais rápida.

Referências

[1] Open MPI
<http://www.open-mpi.org/doc/v1.8/>
Acesso em: 6 de novembro de 2014.

[2] Programming Guide :: CUDA Toolkit Documentation
<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
Acesso em: 5 de dezembro de 2014.

[3] GRAMA, A. GUPTA, A. KARYPIS, G. and KUMAR, V.
Introduction to Parallel Computing. Pearson Education. (2rd ed.). p.101, p.141.

[4] The Lenna Story
<http://www.lenna.org/full/len_full.html>
Acesso em: 6 de novembro de 2014.

[5] An introduction to the Message Passing Interface (MPI) using C
<http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml>Acesso em: 9 de novembro de 2014.