Implementação de Operadores de Processamento de Imagens Utilizando Programação Paralela

Chaina S. Oliveira, Íris A. dos Santos, Quelita A. D. S. Ribeiro, Davi M. da Silva

Departamento de Computação
Universidade Federal de Sergipe (UFS) – São Cristóvão, SE - Brasil
{chaina.oliveira, irisandradesantos, quelita.diniz, mrqsdavi}@gmail.com

Abstract. *When we talk about image processing, we are referring to a set of operations that are able to capture, processing and interpret images. Processing that also concerns to the manipulation of a big set of data. So, having in mind that in these operations there are steps which can be done at the same time, in this article we describe an implementation of image processing operators, such as Flip, Blur, Sobel and others, with parallel computing.*

Resumo*. Quando falamos em processamento de imagens, nos referimos a um conjunto de operações capazes de capturar, processar e interpretar imagens. Processamento que diz respeito também, à manipulação de um grande conjunto de dados. Assim, tendo em mente que nestas operações existem passos que podem ser realizados ao mesmo tempo, neste artigo descrevemos a implementação de operadores de processamento de imagens, tais como Flip, Blur, Sobel e outros, com programação paralela.*

1. Introdução

A necessidade de manipular grandes volumes de dados, bem como a cobrança humana imposta à computação, por aplicações cada vez mais velozes e eficientes têm estimulado o desenvolvimento de tecnologias mais sofisticadas. Tais como processadores mais velozes, memórias com maior capacidade de armazenamento, e outras. Pedrini e Schwartz (2010) esclarecem que:

> "As áreas de processamento digital de imagens têm evoluído significativamente nas últimas décadas. Esse crescimento pode ser observado pelo interesse de cientistas e profissionais em diversos domínios de conhecimento, tais como medicina, biologia, automação industrial, sensoriamento remoto, microscopia e artes. Além disso, o avanço contínuo da tecnologia tem permitido o desenvolvimento de recursos computacionais cada vez mais poderosos para a manipulação de imagens."

Uma destes recursos é a unidade de processamento gráfico, GPU (*Graphics Processing Unit*), que tem alto potencial de paralelismo, ou seja, pode processar várias tarefas ao mesmo tempo. Isso faz com o que a complexidade de tempo de algoritmos paralelizáveis seja reduzida se executadas na mesma (GPU). Existem vários métodos de processamento de imagens têm esta característica, de paralelização. Desta forma, estes podem ser explorados neste contexto.

Diante disso, o presente trabalho, intitulado "Implementação de Operadores de Processamento de Imagens Utilizado Programação Paralela", tem o objetivo de explorar o potencial de placas gráficas, GPUs, para implementar tais métodos.

2. GPU (*Graphics Processing Unit*)

GPU, unidade de processamento gráfico, é um hardware responsável pelo processamento de imagens, bem como manipulação de gráficos [Mittmann 2009].

Aplicações gráficas, como jogos de computador, passaram a exigir cada vez mais velocidade e operações em geral [Paes 2008]. Isso fez com o que os projetistas de hardwares gráficos, como GPUs, buscassem novas formas de aperfeiçoá-los. O que fez com que a GPU tivesse alto poder de paralelismo [Yano 2010].

Peres (2008) afirma que:

> "As GPUS são projetadas para acelerar a renderização de gráficos tridimensionais, sendo utilizada, principalmente, em jogos digitais, simuladores, aplicações CAD, entres outras. Tradicionalmente, o conjunto de operações suportadas pela GPU é limitado a transformações simples, instruções de cor e luminosidade. No entanto, o crescimento da demanda para o realismo nos jogos digitais tem permitido a GPU a oferecer maior programabilidade para suportar operações com gráficos."

Diante deste cenário, foi-se percebendo que o poder computacional destas placas poderia ultrapassar as fronteiras das aplicações gráficas [Mittmann 2009]. Que elas poderiam ser usadas em outros campos computacionais, para realizar outros tipos de operações e tarefas paralelas. Este novo campo é chamado de GPGPU (*General-Purpose computations on the GPU*). Uma das plataformas de programação em GPGPU que se destaca é o CUDA (*Compute Unified Device Architecture*), no entanto, podem ser usada apenas em placas da NVIDA [Mamani 2011].

A arquitetura da GPU é dividida em várias unidades aritméticas, diferentemente da CPU (Figura 2.1). Isso faz com o que ela possa executar diversas atividades em paralelo, acelerando, assim o tempo de execução. Esse diferencial computacional entre CPU e GPU é o principal motivo para que se prefira programar para esta placa gráfica.

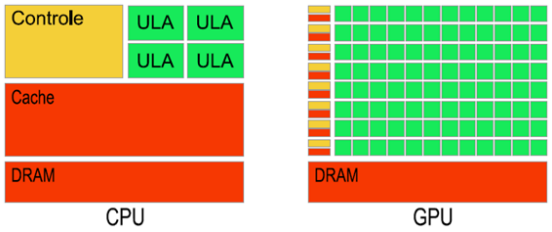

Figura 2.1 CPU X GPU [PAES 2008]

Em outras palavras, a CPU é tipo de hardware em que o processamento de dados ocorre de forma sequencial, enquanto a GPU tem a capacidade de processar os dados de forma paralela [Mamani 2011]. Desta forma, fica claro que o tempo de realização das tarefas na GPU é menor quando se trata de um grande volume de dados.

3. Programação em CUDA (*Compute Unified Device Architecture*)

A NVIDIA nos oferece uma arquitetura, que tem ferramentas para a realização da programação paralela em GPUs. Tal arquitetura foi criada com o propósito de suprir as necessidades da programação de propósito geral (GPGPU). É conhecida como CUDA, sua sintaxe é parecida com a linguagem C/C++, e permite uma computação heterogênea, ou seja, pode usar a CPU e GPU ao mesmo tempo. Explicaremos adiante como isso ocorre.

A execução de um programa na GPU em CUDA pode ser dividida em threads blocos e *grids*. Uma *grid* é formada por um conjunto de blocos. Este, por sua vez é formado por um

conjunto de threads. Threads são segmentos paralelos e possuem um identificador. Para melhor entendimento, visualize esta estrutura na figura 3.1.

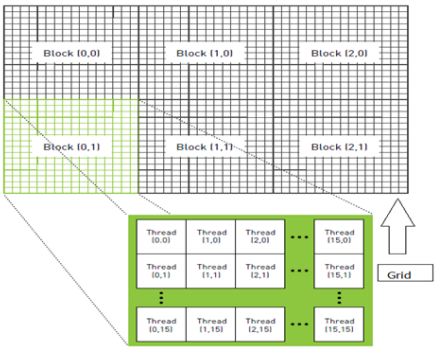

Figura 3.1 Threads, Blocos e Grid [Sandes e Kandort 2011]

Observe que, o identificador do thread também é definido pelo bloco no qual a thread se encontra. Tecnicamente falando, poderíamos definir thread como um grupo de segmentos paralelos que executa um conjunto de instruções (*kernel*).

É importante esclarecer alguns conceitos e terminologias utilizadas pela arquitetura CUDA. O termo *host*, por exemplo, se refere à CPU e sua memória (*host memory*), já o *device*, se refere à GPU e sua memória (*device memory*).

Como mencionado antes, a arquitetura CUDA pode acessar a CPU e GPU, logo alguns identificadores são necessários para sabermos onde a função está sendo executada. São eles: __device__ e __host__ indicam que a função está sendo executada no *device* (GPU) e no *host* respectivamente. E, por último, temos __global__ que indica que a função pode ser chamada pelo *host*, mas executada no device.

Porém, é importante deixar claro que as memórias da CPU e GPU são inteiramente separadas, logo, existem ponteiros tanto do *host*, quanto do *device* que apontam para suas respectivas memórias. Para alocar espaço na memória do dispositivo (*device*), usamos a função cudaMalloc e podemos transferir dados do *host* pro *device*, e vice-versa através cudaMemcpy. Existem ainda, algumas variáveis que indicam as dimensões da *grid* e do bloco, gridDim e bolckDim, respectivamente. Além dessas, existem outras que auxiliam na localização do thread, ou seja, qual informação determinado thread deve processar, blockIdx e threadIdx.

4. Operadores de Processamento de Imagens

Operadores de processamento de imagens digitais são funções responsáveis pela manipulação das mesmas, que na maioria das vezes recebem uma imagem e retornam outra imagem. Como exemplo, tem-se os que fazem a negativa, aqueles que transformam uma imagem colorida em preto e branco (P&B), os que evidenciam as bordas e outros. Descreveremos, aqui, os que foram objetos de estudo neste trabalho.

- **RGB para Escala de Cinza:** Nesta função há a transformação de uma imagem em escala RGB para uma em escala de cinza.

- **Negativo:** É uma operação de mapeamento linear inversa. Onde as regiões mais escuras (com baixos valores de níveis de cinza) tornam-se claras (com altos valores de niveis de cinza).

- **Flip:** Esta operação inverte o eixo da imagem horizontal ou verticalmente, simulando o reflexo de um espelho. Um flip horizontal seria uma rotação de 90º no sentido anti-horário (ou 270º no sentido horário) sob a transposta da imagem, enquanto um flip vertical seria uma rotação de 90º no sentido horário (ou 270º no sentido anti-horário) sob a transposta da imagem.

- **Máscara:** Este operador na verdade deveria se chamar convolução discreta, pois é o seu cálculo que realizamos nesta implementação. A convolução discreta se dá pela seguinte fórmula, da figura 4.1, encontrada em [LEITE 2013]:

$$g(x,y) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} f(i,j)h(x-i,y-j)$$

Figura 4.1 Fórmula da convolução

onde: g: É a imagem resultante; f: Imagem de entrada de tamanho nxn; h: É a máscara, mxm, cujos valores determinarão o tipo de efeito que será encontrado na imagem de saída.

- **Sobel:** É uma operação de detecção de borda, que realça linhas verticais, horizontais e curvas. No caso, as bordas seriam uma fronteira que separa duas regiões onde os níveis de cinza são diferentes. De acordo com [OPENCV 2013], o seu cálculo, aproximado, é feito através da raiz das somas dos quadrados de duas derivadas Gx e Gy.

- **Dilatação:** A dilatação é uma operação que consiste em analisar os pixels vizinhos guardando o maior valor para colocar no pixel central. Na implementação, descrita aqui, serão utilizados dois tipos de máscara:

  - Em cruz: que observa caso o operador esteja em (i,j), serão analisados os vizinhos (i+1,j), (i-1, j), (i, j-1), e (i,j+1).
  - 3x3: Nesta serão analisados, se o operador estiver em (i,j), os vizinhos (i+1, j+1), (i+1, j-1), (i-1, j+1), (i-1, j-1), (i+1,j), (i-1, j), (i, j-1), e (i,j+1).

- **Erosão:** A erosão, de forma semelhante à dilatação é uma operação que consiste em analisar os pixels vizinhos, porém almejando o menor valor para colocar no pixel central. Da mesma forma que a dilatação, na implementação feita, utilizou-se os dois tipos de máscaras descritas no tópico acima (Dilatação).

5. Implementação dos Operadores em CUDA

A implementação dos operadores se deu através da linguagem C com CUDA no ambiente de programação Microsoft Visual Studio. Sendo utilizado ainda o sistema operacional Windows 7 de 64 Bits, 6GB de RAM, processador Intel(R) Core(™) i7- 2670QM CPU @2.20GHz 2.20GHz e GPU com as configurações listadas na tabela 5.1.

Tabela 5.1 Configurações da GPU utilizada no Processamento

CUDA *Driver Version*	5.0
Cuda Compability	2.1

GPU *Clock rate*	1200 Mhz
CUDA *Cores*	(2 Multiprocessors * 48 Cuda Cores) 96
Memory Interface Width	128-bit
Memory BandWidth	28.8 GB/sec
Maximum size of each dimension of a block	1024 x 1024 x 64
Maximum size of each dimension of a grid	65535 x 65535 x 65535

Dadas estas configurações, e o auxílio da biblioteca opencv, utilizada para a captura, exibição e gravação das imagens, foram programadas os operadores de processamento de imagens, detalhados em marcadores adiante, de forma que todos eles possuem a mesma dimensão de Grids e Blocos. O bloco tem dimensão 1x1x1, enquanto a dimensão do grid é igual a numRows x numCols x 1, onde numRows é o número de linhas da imagem e numCols é o número de colunas da imagem.

Para acessar, as threads presentes em cada bloco de cada grid foram utilizados em todos os operadores duas variáveis chamadas i e j, em que i diz respeito a dimensão x do Grid, e j a dimensão y. Note que seria possível ainda definir diversas dimensões para os blocos e as grids, porém foi utilizado neste código uma dimensão de grid suficiente para suprir as necessidades de threads para o processamento de imagens de dimensão 1x1 até 65535x65535.

- **RGB para Grey:**

○ `__global__ void RGBtoGrey (const uchar4* rgbaImage, unsigned char* greyImage, int numRows, int numCols):`

Nesta função ocorre o mapeamento de uma imagem RGB, "modelo de sistema baseado em coordenadas cartesianas com componentes *Red* (Vermelho), *Blue* (Azul) e *Green* (verde)" [LUKAC 2013], para uma imagem Preto e Branco (P&B). A imagem RGB, é parametrizada pela constante rgbaImage do tipo uchar4 (tipo da linguagem C com CUDA) que possui 4 componentes (x,y,z e w), enquanto o retorno é dado pelo ponteiro nomeado greyImage (unsigned char). Os inteiros nunRows e numCols são utilizados nessa e em outras funções para encontrar posições imagens de entrada e saída, como também para limitar possíveis acessos aos tamanhos das imagens.

A codificação da função RGBtoGrey se deu, portanto, pela utilização das variáveis i e j de forma a criar uma relação entre da posição das mesmas na Grid com cada posição dos vetores, cujas informações das imagens de entrada e saída estavam presentes . Por consequência, a combinação dessas duas variáveis para determinar cada posição do vetor faz com que cada pixel da imagem seja acessado e modificado paralelamente por uma thread na grid. Isso acontece em todos os operadores aqui codificados.

- **Negativo:** São dois os operadores implementados para gerar uma imagem negativa a partir de outra, através do cálculo Dynamic Range - Valor do Pixel:

○ `__global__ void Negative_PB(const unsigned char* pbImage, unsigned char* negImage, int numRows, int numCols), e;`

- `__global__ void Negative_RGB(const uchar4* rgbaImage, uchar4 * negImage, int numRows, int numCols).`

Os dois são operadores similares, possuindo como diferenças o tipo da imagem que será utilizada como entrada e aquela pela qual serão retornados os resultados com a negação, unsigned char vs. uchar4. No uchar4, cada componente será acessado/modifica individualmente e a posição do pixel será calculado do mesmo jeito descrito no detalhamento da função RGBtoGrey.

- **Flip:** Foram implementados novamente para o processamento que resultará no Flip da imagem, bem como para os outros processamentos duas funções uma para imagens RGB e outra para imagens P&B. Pelo motivo que, imagens em RGB serão manipuladas nesta implementação através do tipo uchar4 e imagens P&B pelo tipo unsigned char.

Funções Flip, que invertem a imagem na vertical (o que é direita fica na esquerda e vice-versa):

- `__global__ void Flip_PB(const unsigned char* pbImage, unsigned char* flipImage, int numRows, int numCols)`
- `__global__ void Flip_RGB(const uchar4* rgbaImage, uchar4 * flipImage, int numRows, int numCols)`

Nestes operadores, tenha em mente que a imagem de entrada é um vetor, tratado como uma matriz nxm. Sendo assim, o valor do pixel que está na posição 0x0 será colocado através dessa função no pixel 0xm, o que está em 0x1 será colocado em 0xm-1, e assim sucessivamente até que em todas as linhas da imagem os pixels que estariam do lado direito da imagem de entrada sejam encontrados do lado esquerdo da imagem de saída e vice-versa.

- **Mascara:** Na codificação foram criadas duas funções que aplicam através de convolução uma determinada mascara, uma em imagens RGB e outra em imagens P&B. Mascara cujo tamanho é 3x3 e onde os valores são definidos através de um parâmetro intitulado “operador”. De forma que, por exemplo, se operador for igual a 3 a máscara utilizada será a do Blur (Nesta Implementação fazemos uso apenas da máscara do Blur).

As funções citadas acima são:

- `__global__ void Mascara_3x3_RGB(const uchar4* rgbaImage, uchar4 * blurImage, int numRows, int numCols, int operador);`
- `__global__ void Mascara_3x3_PB(const unsigned char* pbImage, unsigned char* blurImage, int numRows, int numCols, int operador);`

Em ambas, foi utilizado o cálculo de convolução para determinar o valor de cada pixel da imagem de saída. Onde o cálculo nas bordas foi feito repetindo o valor central para as posições que ultrapassariam os limites da imagem.

- **Sobel:** Para as duas funções que encontram o valor absoluto do Gradiente X e o Y da imagem, e depois descobrem o valor de cada pixel da imagem resultante, foram utilizados como base a documentação do sobel do [OPENCV 2013]. Nesta

documentação foram vistos todos os passos para se encontrar as bordas de uma imagem. As duas funções foram nomeadas como:

- `__global__ void Sobel_PB(const unsigned char* pbImage, unsigned char* sobelImage, int numRows, int numCols, int operador);`
- `__global__ void Sobel_RGB(const uchar4* rgbaImage, uchar4 * sobelImage, int numRows, int numCols, int operador);`

Apesar de ter como base a documentação do [OPENCV 2013], pequenas mudanças foram feitas já que o opencv não utiliza propriamente a máscara sobel para a sua operação sobel.

- **Dilatação com máscara em cruz:** Foram codificadas novamente duas funções para o operador:

- `__global__ void Dilatacao_PB(const unsigned char* pbImage, unsigned char* dilImage, int numRows, int numCols)`
- `__global__ void Dilatacao_RGB(const uchar4* rgbaImage, uchar4 * dilImage, int numRows, int numCols)`

Nestas funções ocorre a análise dos 4 pixels vizinhos dados pela máscara em cruz, caso haja algum valor maior que todos este será guardado, e posteriormente colocado no pixel de mesma posição do pixel central que está sendo analisado da imagem de saída.

- **Dilatação com máscara 3x3:** Semelhante ao operador acima, se diferenciando apenas na quantidade de pixels observados, que ao invés de 4 serão 8 (os pixels da direita, da esquerda, de cima, de baixo e das diagonais).
- **Erosão com máscara em cruz:** Nas funções, para RGB e P&B, como o operador é de erosão, primeiramente é guardado o valor do pixel central (dado pelas posições das threads) como o novo valor. Em seguida, existe a comparação dos 4 pixels vizinhos, o o novo valor de modo que caso um seja menor, o novo valor será modificado para ele. Ao final, o novo valor é colocado na mesma posição do pixel central na imagem de saída.

  Funções criadas:

- `__global__ void Erosao_PB(const unsigned char* pbImage, unsigned char* dilImage, int numRows, int numCols)`
- `__global__ void Erosao_RGB(const uchar4* rgbaImage, uchar4 * dilImage, int numRows, int numCols)`

- **Erosão com máscara 3x3** Se difere do operador acima, somente na questão da quantidade de pixels observados, que é mais abrangente. Neste caso, o operador olha os valores dos oito pixels que estão acima, abaixo, a direita, a esquerda e nas diagonais.

6. Resultados

Foram obtidos, como resultado da execução da implementação descrita anteriormente, operadores capazes de processar imagens, gerando uma imagem de saída (jpg) referente ao

operador executado. Observe na figura 6.1 os resultados da aplicação das transformações sobre uma imagem em escala de cinza e na figura 6.2, em uma imagem RGB.

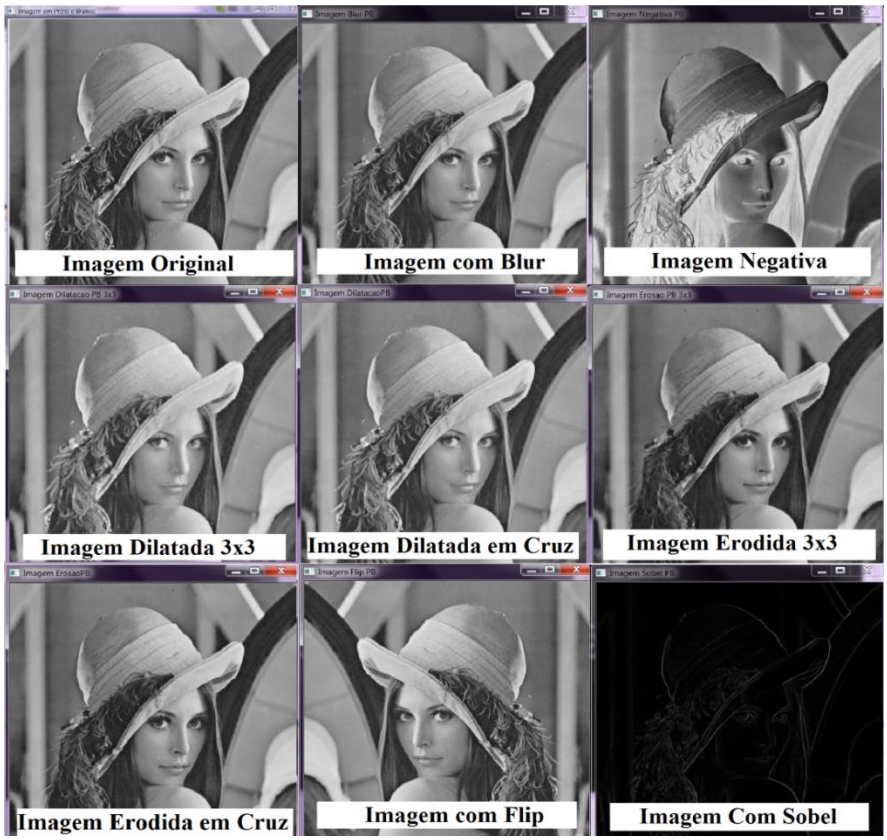

Figura 6.1 Imagem referente às transformações de uma imagem em escala de cinza

Figura 6.2 Imagem referente às transformações de uma imagem RGB

7. Conclusões e Trabalhos Futuros

A implementação de um código de processamento de imagens com a utilização de GPUs, além de possível, constrói operadores capazes de analisar e gerar imagens de forma rápida e precisa o que em outras linguagens que utilizam CPU seriam operações custosas.

Como trabalhos futuros, temos a implementação de outros operadores, da equalização de histogramas, além a implementação dos mesmos operadores em outra linguagem de programação, de forma sequencial. Com isso, poderemos fazer comparações de tempo de execução de cada transformação/mapeamento nas duas linguagens. Assim, observaremos a complexidade de tempo de um operador processado na GPU em relação ao mesmo processado da CPU, vendo, assim o ganho computacional obtido através da programação paralela. Trabalho que seria parecido com o encontrado em [VITOR 2013], só que mais abrangente.

8. Referências

LEITE, Neucimar. **Introdução ao Processamento e Análise de Imagens. Disponível em:** www.ic.unicamp.br/~afalcao/sensremoto/processamento.ppt. Acesso em: 3 jan. 2013.

LUKAC, Rastislav. **Computational Photography: Methods and Applications.** Disponível em: http://migre.me/ghmnM. Acesso em: 3 jan. 2013.

MAMANI, A. V. O. **Soluções aproximadas para algoritmos escaláveis de mineração de dados em domínio de dados complexos usando GPGPU**. São Paulo: USP – São Carlos, 2011.

MITTMANN, Adiel. Tractografia em Tempo Real Através de Unidades de Processamento Gráfico. Florianópolis: Universidade Federal de Santa Catarina, 2009.

OPENCV. Welcome to opencv documentation! Disponível em: http://docs.opencv.org/index.html. Acesso em: 3 jan. 2013.

PAES, M. A. Motor de Física de Corpos Rígidos em GPU com Arquitetura CUDA. Campo Grande: Universidade Federal de Mato Grosso do Sul, 2008.

PEDRINI, Helio; SCHWARTZ, W. R. Análise de Imagens Digitais: Princípios, Algoritmos e Aplicações. Editora Thomson Learning, ISBN 978-85-221-0595-3, 2007.

SANDERS, Jason; KANDROT, Edwad. CUDA by Example. United States: NVIDIA Corporation, 2011.

VITOR, Giovani B. Eficiente Convolução 2D para processamento de imagens, utilizando GPU. Disponível em: http://parati.dca.fee.unicamp.br/media/Attachments/courseIA366F2S2010/gb087390_9/ConvolutionGPU.pdf. Acesso em: 3 jan. 2013.