REVISTA DE **EMPREENDEDORISMO, INOVAÇÃO E TECNOLOGIA**

Conversão Semi-automática de Algoritmos Sequenciais de Processamento Digital de Imagens para Algoritmos Paralelos em CUDA

Semi-Automatic Conversion of Digital Image Processing Sequential Algorithms to Parallel Algorithms in CUDA Architecture

Vagner Gon Furtado(1); Eduardo Max Amaro Amaral(2); Filipe Wall Mutz(3); Flavio Severiano Lamas de Souza(4); Karin Satie Komati(5)

1 Instituto Federal do Espírito Santo (IFES) - Campus Serra, Brasil. E-mail: vagnergfst@gmail.com
2 Instituto Federal do Espírito Santo (IFES) - Campus Serra, Brasil. E-mail: emaxamaral@gmail.com
3 Instituto Federal do Espírito Santo (IFES) - Campus Serra, Brasil. E-mail: filipemtz@gmail.com
4 Instituto Federal do Espírito Santo (IFES) - Campus Serra, Brasil. E-mail: flavio.lamas@gmail.com
5 Instituto Federal do Espírito Santo (IFES) - Campus Serra, Brasil. E-mail: kkomati@ifes.edu.br

Revista de Empreendedorismo, Inovação e Tecnologia, Passo Fundo, vol. 4, n. 2, p. 122-139, Jul.-Dez. 2017 - ISSN 2359-3539

DOI: https://doi.org/10.18256/2359-3539.2017.v4i2.2046

Endereço correspondente / Correspondence address
Karin Satie Komati
Rodovia ES-010 – Km 6,5 – Manguinhos – Serra – ES,
29173-087 - Brasil

Como citar este artigo / How to cite item: clique aqui!/click here!

Resumo

Sistemas de processamento digital de imagens podem ter tempo de execução não compatíveis com o esperado pelo usuário. Uma possível solução é o uso de processamento paralelo para diminuir o tempo de execução de algoritmos de processamento de imagens. A tecnologia CUDA oferece uma interface de desenvolvimento para tirar proveito do processamento paralelo em GPUs, entretanto, possui uma alta curva de aprendizagem e exige conhecimento de recursos específicos, como sua arquitetura e tipos de memória. Este trabalho propõe uma ferramenta semi-automática para converter algoritmos de processamento de imagens sequenciais em uma versão paralela para GPU na qual o programador não precisa conhecer os detalhes da arquitetura, nem os seus comandos de programação específicos. Para tanto, o programador deve adotar a definição da API deste trabalho, seguindo os protótipos de funções e incluir, em seu código fonte, diretivas que identifiquem uma das quatro categorias de processamento: operações pixel a pixel, operações de vizinhança de pixel, operações que reduzem a imagem para um valor escalar e operações que reduzem a imagem para um vetor. O resultado final é o código fonte paralelizado na tecnologia CUDA. Foram realizados experimentos para cada uma das categorias e os resultados mostraram que a versão paralela diminui o tempo de execução para três categorias, exceto a de operações que reduzem a imagem para um vetor devido aos acessos simultâneos ao mesmo endereço de memória da posição do vetor.

Palavras-chave: CUDA, Processamento Digital de Imagens, Algoritmos Paralelos

Abstract

Digital image processing systems may not have execution time compatible with the user's expectation. A possible solution is to use parallel processing to reduce the execution time of image processing algorithms. The CUDA technology offers a development interface to take advantage of parallel processing in GPUs, however, it has a steep learning curve and requires knowledge of specific features such as its architecture and memory types. This work proposes a semi-automatic converter tool for sequential image processing algorithms to a parallel version for GPU in which the programmer does not need to know the details of the architecture or its specific programming commands. Whereupon, the programmer needs to adopt the API definition of this work by following the prototypes of functions and include, in his source code, directives that identify one of four processing categories: pixel to pixel operations, pixel neighborhood operations, operations that reduce the image to a scalar value and operations that reduce the image to a vector. The end result is parallel source code in CUDA technology. Experiments were performed for each one of the categories and the results show that the parallel version decreases the execution time for three categories, except operations that reduces the image to a vector due the simultaneous access to the same memory address on the vector.

Keywords: CUDA, Digital Image Processing, Parallel Algorithms

Introdução

Uma maneira de realizar processamento paralelo consiste no uso de GPUs (*Graphics Processor Unit*, em português Unidade de Processamento Gráfico). GPUs, comumente chamadas de placas de vídeo, tornaram-se muito populares por acelerar o processamento gráfico, principalmente para jogos eletrônicos, devido ao seu *hardware* especializado em processar operações com elementos geométricos, tais como vértices e polígonos num plano tridimensional (Owens et al., 2008).

Até meados do ano de 2003, já se usava GPUs para processamento de dados, porém não havia uma API (*Application Programming Interface*) própria para esse objetivo. Mais tarde, desenvolveram-se soluções que permitiram o uso generalizado destes componentes, possibilitando a utilização de linguagens de programação já conhecidas e difundidas para implementar algoritmos paralelos em GPUs (NVIDIA, 2016). Uma destas soluções é a API NVIDIA CUDA (*Compute Unified Device Architecture*), que dá suporte a métodos que facilitam a alocação de recursos, e execução de aplicações escritas em linguagens de programação nos núcleos de GPUs.

Uma das utilizações da tecnologia CUDA está na área de Processamento Digital de Imagens (PDI). Boa parte desses algoritmos podem ser implementados usando GPU, devido a sua característica de independência de dados, onde regiões de *pixels* podem ser divididas e processadas simultaneamente, agregando os resultados destas divisões num resultado único (Parhami, 1999). Em muitos trabalhos, aplicações de paralelização de algoritmos de PDI resultaram em um aumento considerável de desempenho comparado à versão serial ou de CPU (*Central Processing Unit*) (Kouzinopoulos & Margaritis, 2009).

No entanto, utilizar CUDA é uma tarefa complexa, demandando que o programador conheça a arquitetura e ainda se preocupe em como adaptar da melhor forma possível seu algoritmo sequencial para obter o máximo de desempenho. A utilização explícita de recursos ligados à arquitetura, recursos de baixo nível como memórias para tipos específicos de uso que aceleram o acesso aos dados e o diferente ciclo de execução para realizar o processamento, se corretamente utilizados, alteram drasticamente o desempenho. Assim, a curva de aprendizagem é alta. Há desafios quanto a granularização e a sincronização (Braga, de la Rocha Ladeira, & Mota, 2017). Outro problema é o fato da paralelização ser altamente dependente do *hardware* utilizado. Assim, a portabilidade, adaptabilidade e o processo de desenvolvimento de aplicações paralelas eficientes demandam tempo em excesso comparado ao desenvolvimento das suas aplicações sequenciais correspondentes (Daniel, 2003).

Existem algumas soluções propostas que automatizam o processo de paralelização de software. Entre elas há o OpenACC (http://www.openacc.org/), que utiliza diretivas no código a fim de integrá-lo com GPUs. Entretanto, o OpenACC não

foca especificamente em técnicas de PDI pois é modelado para aplicações em geral e com isso, possui muitas opções e seu uso possui alta curva de aprendizado.

Por outro lado, na área de PDI, foi proposto por Nugteren, Corporaal e Mesman (2011) uma categorização dos algoritmos de PDI em 4 (quatro) tipos:

- P2P – (*Pixel to Pixel*). Onde o resultado de um *pixel* processado depende de apenas um *pixel* de entrada.
- N2P – (*Neighborhood to Pixel*). Onde o *pixel* de saída é formado pelo processamento de um *pixel* e de seus circunvizinhos.
- R2S – (*Reduction to Scalar*). A saída é um valor único ou escalar, calculado a partir de toda a imagem.
- R2V – (*Reduction to Vector*). A saída é uma sequência de valores, um vetor, a partir de toda a imagem.

A proposta Nugteren, Corporaal e Mesman (2011), apesar de focar na área de PDI, se baseia em regras de substituição de índices baseados em expressões regulares que não são facilmente compreendidas.

Neste contexto, este trabalho propõe um sistema semi-automático para a conversão de um código fonte em linguagem C com técnicas de PDI de forma serial para uma versão paralela executável em GPUs com tecnologia CUDA. Esta conversão automática de códigos utiliza esqueletos genéricos pré-desenvolvidos pertencentes às quatro categorias propostas por Nugteren, Corporaal e Mesman (2011). Além disso, será comparado o tempo de execução dos algoritmos transformados pela solução com as implementações originais, analisando as diferenças de desempenho.

Algoritmos de Processamento de Imagens

A Figura 1 demonstra os 4 (quatro) tipos da classificação dos algoritmos de PDI, conforme a proposta de Nugteren, Corporaal e Mesman (2011). Neste trabalho, será usado o algoritmo de binarização como P2P, o algoritmo de filtragem espacial via convolução espacial[1] como N2P, o máximo valor de uma imagem como R2S, enquanto o de histograma será classificado como R2V.

A binarização de imagens baseado em limiar é um método de segmentação, estabelecendo um valor (limiar) que classifica os seus *pixels* em dois valores, gerando uma imagem binária, onde seus *pixels* possuem apenas os valores 0 (zero) e 1 (um), geralmente representando preto e branco. A Figura 2b mostra o resultado da binarização da imagem de moedas sobre a Figura 2a, onde é possível verificar que as moedas se destacaram com relação ao plano de fundo.

1 Convolução é um processamento realizado para cada pixel da imagem de entrada é gerado um novo valor correspondente à soma do resultado da multiplicação posicional de uma máscara sobreposta à imagem de entrada.

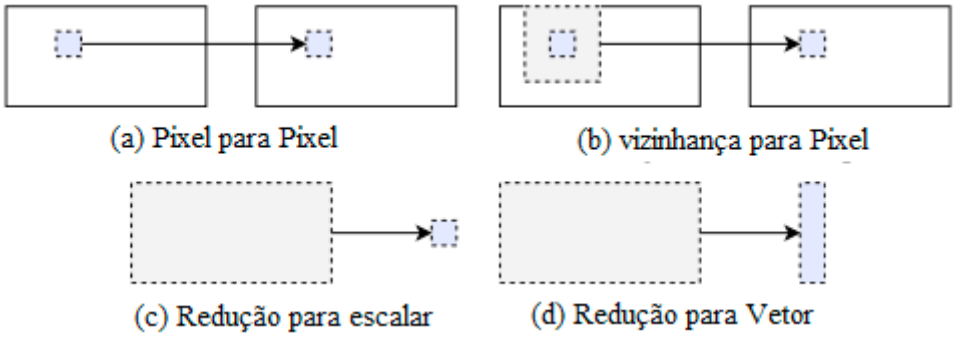

Figura 1. Classificação dos algoritmos de PDI. Retirado e traduzido do trabalho de Nugteren, Corporaal e Mesman (2011).

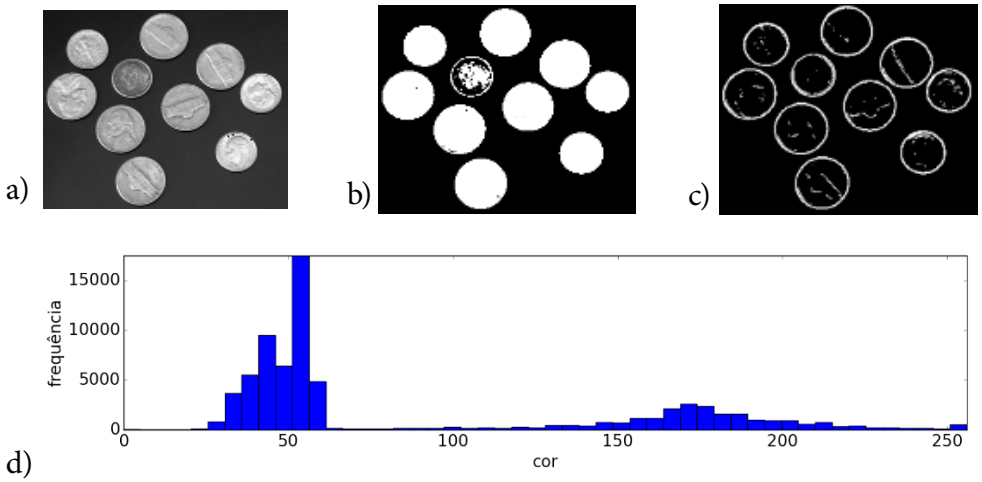

Figura 2. a) imagem original, b) o resultado da imagem após a binarização, c) o resultado após convolução usando máscara de Sobel e d) o histograma da imagem apresentada em a).

A operação de convolução usa pequenas matrizes, chamadas de máscaras, para filtrar uma imagem e extrair ou alterar suas características, como bordas, retirada de ruídos, suavização de imagens, entre outros (Gonzalez & Woods, 2002). A Figura 2c mostra o resultado de uma convolução usando máscara de Sobel sobre a Figura 2a. Já o máximo valor de uma imagem é realmente o retorno do maior valor de todos os seus *pixels*. Em PDI, o histograma é uma função discreta que relaciona as intensidades dos *pixels* com a frequência com que elas aparecem na imagem de entrada. A Figura 2d apresenta o histograma da imagem apresentada na Figura 2a.

A Arquitetura de Computação NVIDIA CUDA

Unidades de processamento gráfico de uso geral (*General Purpose Graphics Processing Units* - GPGPU) são equipamentos que permitem aos programadores usar unidades gráficas de processamento, as GPU, para a realização de computação de propósito geral, e não apenas processamentos gráficos. CUDA é uma plataforma de computação paralela que possui uma API desenvolvida pela NVIDIA para permitir que programadores utilizem GPUs desta marca para computação de propósito geral.

Códigos escritos em CUDA possuem a extensão ".cu" e são compilados em arquivos-objeto utilizando um compilador específico provido pelo *device* (Nvidia, 2017).

Em CUDA, funções são classificadas em três tipos. As funções que são executadas pela CPU são chamadas de funções do *host*. As funções que são executadas na GPU, mas invocadas pela CPU são chamadas de *kernels*. Estas funções devem receber o qualificador __global__ antes do tipo de retorno da função que é sempre **void**. Por fim, as funções que são invocadas e executadas pela GPU são chamadas de funções de *device*. Estas funções devem receber o qualificador __device__ antes do tipo de retorno da função. Funções com este qualificador podem ser invocadas apenas pelo *device*, ao menos que o qualificador __host__ seja incluído (Nvidia, 2016a).

A arquitetura de GPU da NVIDIA foi construída usando como componente central um conjunto de *Streaming Multiprocessors* (SMs) *multi-thread*. SMs usam o modelo de programação paralela *Single Instruction Multiple Thread*. Neste modelo, múltiplas *threads* executam o mesmo trecho de código sobre conjuntos de dados diferentes. O código executado pelas *threads* da GPU é chamado *kernel*. Quando um programa escrito usando CUDA invoca um *kernel*, as operações a serem realizadas são distribuídas para aqueles *multiprocessors* que possuem recursos de computação disponíveis. Várias *threads* são executadas de forma concorrente nos *multiprocessors*, e todas elas executam as instruções do *kernel* sobre trechos diferentes dos dados. Para extrair o máximo desempenho da GPU, é importante conhecer não apenas o conceito de *thread*, mas também os conceitos de bloco, *grid*, e *warp* (Nvidia, 2016a).

Um *grid* é um conjunto com até duas dimensões de blocos. Blocos, por outro lado, são conjuntos com até três dimensões de *threads*. A organização em *grids*, blocos, em *threads*, oferece ao programador a possibilidade de usar todos os níveis de paralelismo disponíveis na GPU. Cada *thread* possui um endereço que é representado por dois vetores. O vetor **threadIdx** com três posições x, y, e z, representa o endereço da *thread* no bloco. O outro vetor de duas dimensões, **blockIdx**, representa o endereço do bloco no *grid*. Dois vetores adicionais de importância fundamental são os vetores **blockDim** e **gridDim** com três e duas posições, respectivamente. O primeiro contém o número de *threads* em cada dimensão do bloco, enquanto o segundo contém o número de blocos em cada dimensão do *grid*. As *threads* de um bloco são enviadas para execução em grupos de 32 *threads* chamados *warps*. Todas as *threads* de um *warp* executam instruções de forma concorrente em um SM (Nvidia, 2016a).

GPUs possuem composições de memória com diferentes características e propósitos. Dois tipos são particularmente importantes para este trabalho, a memória global (também chamada de memória do dispositivo) e a memória compartilhada. A global possui muito espaço de armazenamento (da ordem de GB), mas o acesso às suas informações é ineficiente. Para minimizar o custo de trocar informações com a memória global, é desejado que todos os acessos à esta memória sejam feitos de forma coalescente[2]

2 Endereços de memória estão em partições adjacentes, na mesma região. Logo, o sistema operacional não necessita realizar saltos no acesso de um conjunto de dados na memória.

(i.e., cada *thread* deve acessar dados contíguos e pertencentes ao mesmo bloco). Todas as *threads* de um mesmo bloco compartilham uma memória de pequena escala (da ordem de KB) e baixa latência, chamada memória compartilhada. O acesso à memória compartilhada é significativamente mais eficiente que o acesso à memória global. Por esta razão, a memória compartilhada é frequentemente utilizada como um cache dos dados existentes na memória global. Contudo, a utilização da memória compartilhada não é automática e cabe ao programador gerenciar o seu uso (Bakhoda et al., 2009).

Desenvolvimento do Sistema

O sistema proposto converte códigos fonte de entrada em modelos carregados em código CUDA/C++ de saída. Os modelos são códigos fonte escritos em C++, com áreas específicas a serem alteradas pelo sistema. O desenvolvedor deve seguir algumas regras e inserir diretivas no seu código para que o software possa adaptá-lo e gerar códigos em CUDA. No desenvolvimento, utilizou-se a API CUDA 6.5 e a biblioteca OpenCV 2.4.9 (últimas versões disponíveis na época), uma biblioteca de visão computacional multi-plataforma com código fonte aberto (ITSEEZ, 2016), para auxiliar nas conversões de dados, carregamento e exibição de imagens.

Foram criadas implementações distintas para cada categoria: P2P, N2P, R2S ou R2V. De acordo com a categoria, deve-se implementar um método com uma assinatura específica que processará uma unidade do modelo. Este método deverá ser estático, pois não deve depender de objetos de uma classe e deve estar marcado com uma diretiva explícita, antes de sua declaração. Caso o desenvolvedor precise de parâmetros adicionais para a execução do método, pode-se declarar uma estrutura que contenha os parâmetros necessários. Esta estrutura só poderá conter tipos básicos. Todas as definições usadas e a estrutura de parâmetros devem estar incluídas num arquivo de cabeçalhos (*header)* passado como parâmetro para o conversor. Este *header* será incluído no código da versão paralela.

Com isso, basta executar o sistema conversor, passando o arquivo de código fonte e o de cabeçalhos caso necessário. Será gerado um código fonte compilável para execução em GPUs com tecnologia CUDA. A saída contém um arquivo com extensão ".cu" com a alocação de dados, preparação e declaração do *kernel*. Além de um arquivo ".cpp" que contêm validação de *hardware* e métodos para realizar a chamada ao *kernel*. Como a saída é código fonte, o desenvolvedor está livre para ajustar ou modificar quaisquer partes que lhe convenham. O modelo P2P será descrito de forma mais detalhada, apresentando-se inclusive o código em CUDA produzido pelo sistema. Para os demais modelos, faz-se uma descrição de suas similaridades e diferenças.

P2P

Para o uso do modelo P2P, deve-se implementar um método que recebe como entrada o valor de um *pixel* e a sua posição na imagem, e produz como saída o

valor do *pixel* na imagem resultante. Este método deve ser marcado com a diretiva "#P2P_FUNC". A posição do *pixel* assume que a imagem foi representada como um vetor de *pixels*. Nesta representação, a posição do *pixel* é dada pelo número da linha multiplicada pela altura da imagem somada com a posição da coluna. O Código Fonte 1 ilustra a assinatura de uma função que usa o modelo P2P. Depois do código fonte ser processado, uma classe P2P é criada. Deve-se instanciá-la, passando a imagem de entrada como parâmetro, podendo ser do tipo "IplImage" ou um vetor de *pixels*, e as dimensões da imagem. Feito isto, basta invocar o método "processa()" que fará o processamento na GPU, e armazenará a imagem resultado num objeto dentro da classe, como mostrado no Código Fonte 2.

O *kernel* P2P copiará a imagem para o dispositivo, e cada *thread* será responsável por utilizar o método definido pelo usuário para operar sobre um *pixel* da imagem. A implementação do *kernel* P2P é a mais simples dentre os modelos. A *thread* utiliza seu identificador único para acessar um *pixel* da imagem e escrever o valor resultante. O acesso à memória global é feito de modo coalescente, garantindo uma maior taxa de transferência de dados. A quantidade de paralelismo explorada é igual à quantidade de *pixels* da imagem, pois todas as operações são independentes. O Código Fonte 3 exemplifica a paralelização de um algoritmo que realiza a binarização de uma imagem com uma tolerância passada como parâmetro. A tolerância foi definida como 127 no exemplo. A saída do processador de código é apresentada no Código Fonte 4.

Código Fonte 1. Assinatura do método unitário do modelo P2P

```
//Recebe e retorna um pixel da posição i
#P2P_FUNC
static unsigned char p2pProcessor(unsigned char pixel, int i);
```

Código Fonte 2. – Utilizando o processamento P2P

```
//Instancia a classe P2P, passando a imagem de entrada
P2P* p2p = new P2P((IplImage*)imagem); //IplImage é um tipo do OpenCV
//Realiza o processamento
p2p->processa();
//Acesso ao resultado
p2p->imagemResultante;
```

Código Fonte 3. Código serial para binarização

```
#STRUCT
typedef struct{
        int tolerancia;
} data;

#P2P_FUNC
static unsigned char binarizador(unsigned char pixel, int index, data parametro){
        return pixel + parametro.tolerancia > 127 ? 255 : 0 ;
} (...)
```

Código Fonte 4. Código resultado paralelizado para binarização

```
//Inclusões
#define CHK_ERROR if (erro != cudaSuccess) goto Error;

__device__
static unsigned char binarizador (unsigned char pixel, int index, data parametro) {
        return pixel + parametro.tolerancia > 127 ? 255 : 0 ;
}

__global__ void P2P_kernel(unsigned char* imagem, int tamanho, data param){
        int i = blockIdx.x * blockDim.x + threadIdx.x;
 if(i < tamanho) imagem[i] = subtraction (imagem[i], i, param);
}

extern “C” void kP2P(unsigned char* imagem, unsigned char* imageResult, int tamanho, data
param){
        cudaDeviceProp deviceProp;
        cudaError_t erro;
        unsigned char* d_image;

        erro = cudaMalloc((void**)&d_image,sizeof(unsigned char)*tamanho); CHK_ERROR
        erro = cudaMemcpy(d_image,imagem,tamanho*sizeof(unsigned
char),cudaMemcpyHostToDevice); CHK_ERROR
        cudaGetDeviceProperties(&deviceProp, 0);
        int blockSize = deviceProp.maxThreadsPerBlock;
        int nBlocks = tamanho/blockSize + (tamanho%blockSize == 0 ? 0:1);

        P2P_kernel<<<nBlocks, blockSize>>>(d_image,tamanho, param);

        erro = cudaGetLastError(); CHK_ERROR

        //Espera a GPU terminar o trabalho
        erro = cudaDeviceSynchronize(); CHK_ERROR

        erro = cudaMemcpy(imageResult,d_image,tamanho*sizeof(unsigned char),
cudaMemcpyDeviceToHost); CHK_ERROR
        goto Free;
Error:
        std::cerr << “Error on CUDA: “ << cudaGetErrorString(erro);
Free:
        cudaFree(d_image);
        cudaFree(d_imageResult);
}
```

O sistema proposto usa os índices das *threads* e dos blocos para enviar *pixels* para o método definido pelo usuário. A imagem de entrada é alocada no *device* e os resultados das operações são armazenados na própria imagem. Ao fazer isto, evita-se que uma nova operação de alocação de memória seja feita na GPU. Após o processo, a imagem resultante fica acessível ao usuário no objeto "imagemResultante", descrito no Código Fonte 2. Todos os métodos necessários para o processamento são incorporados ao esqueleto com o identificador "__device__", indicando que este método será apenas utilizado na GPU.

N2P

O processo para utilizar do modelo N2P se assemelha ao P2P, porém o método unitário possui uma assinatura diferente. O Código Fonte 5 ilustra a assinatura de um método N2P. O método recebe como entrada uma matriz de *pixels* que incluem o *pixel* a ser processado, e os seus vizinhos imediatos, e produz como saída o valor do *pixel* na imagem de saída. O acesso à memória e quantidade de paralelismo a ser explorado nos *kernels* N2P se assemelham aos P2P, visto que todas as suas operações são independentes. Dentro do *kernel*, cada *thread* é responsável por processar um *pixel* e sua vizinhança. Para isto, a vizinhança do *pixel* é armazenada em uma matriz e enviada para o método declarado pelo usuário. O resultado deste processamento é armazenado na imagem de saída.

Código Fonte 5. Assinatura do método unitário do modelo N2P

```
//Recebe um conjunto de pixel e retorna um pixel
#N2P_FUNC
static unsigned char n2pProcessor(unsigned char pixel[3][3], int x, int y);
```

R2S

No modelo R2S, um valor escalar é gerado a partir do conjunto de *pixels* da imagem de entrada. Como o processo gera uma cadeia, o paralelismo está num processo *top down* onde partes da imagem geram valores que vão compor novos dados até que se obtenha um único resultado, processo ilustrado na Figura 3. A quantidade de paralelismo é metade da quantidade de *pixels* da imagem no início, e cai pela metade a cada operação, até que seja produzido um valor único como saída. Algoritmos que possuem esta estrutura são usualmente chamados de algoritmos de redução. A memória compartilhada entre o bloco (*shared memory*) é explorada neste caso, fazendo com que o acesso à memória global seja coalescente e feito apenas uma vez. Os resultados intermediários são armazenados nela, até que reste apenas um valor. Este valor é, então, retornado para a memória global.

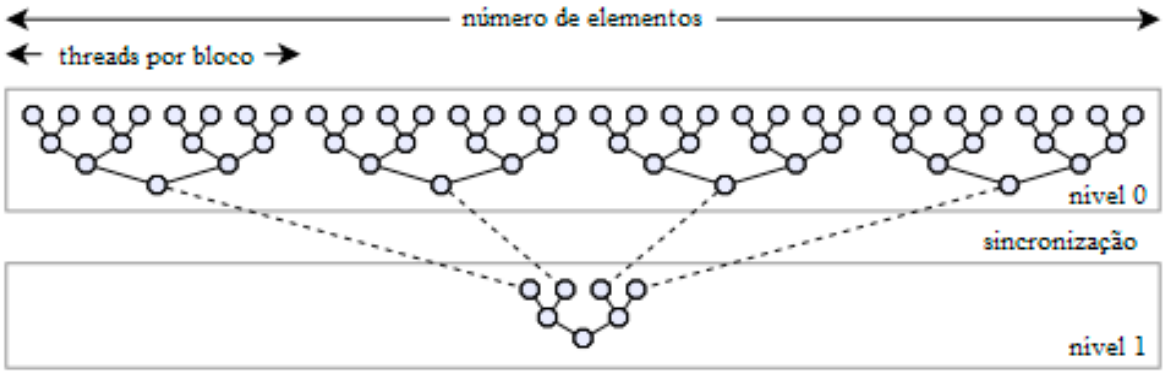

Figura 3. Redução top down onde resultados intermediários compõem novos dados até o valor final. Fonte: Retirado e traduzido de Nugteren, Corporaal e Mesman (2011).

A declaração do método, descrita no Código Fonte 6, deve receber dois *pixels* e retornar um valor como resultado. O método não precisa declarar o *template*, deve apenas usar os tipos E para entrada e o tipo T para a saída. Para obter o valor máximo dos *pixels* de uma imagem, por exemplo, o retorno seria o maior dentre os valores usados como parâmetros. Para utilização, deve-se instanciar um objeto da classe R2S, passando uma imagem como parâmetro, e invocar o método "processa()". Neste caso, o resultado será apenas um valor escalar, cujo tipo será definido no *template* da classe, conforme demonstrado no Código Fonte 7. O Código Fonte 8 mostra a conversão do algoritmo que encontra o valor mínimo de uma imagem para sua versão paralela.

Código Fonte 6. Declaração da função R2S

```
//Recebe dois valores e retorna o resultante entre eles
#R2S_FUNC
static T r2sProcessor(E pixel1, E pixel2);
```

Código Fonte 7. Utilizando o modelo R2S

```
//Instancia a classe R2S, explicitando o tipo resultante
R2S<int> r2s((unsigned char*)imagem, (int) altura, (int) largura);
int resultado = r2s.processa();
```

Código Fonte 8. Declaração de função

```
#R2S_FUNC
static T menor(E pixel1, E pixel2) {
        return pixel1 < pixel2 ? pixel1 : pixel2; }
```

Note que o desenvolvedor deve identificar os tipos genéricos como "T" e "E", obrigatoriamente, pois estas foram as notações utilizadas na declaração do *template* no esqueleto pré desenvolvido. No processo de redução, cada *thread* trabalha com dois valores, e salva o resultado na memória *onchip*. Cada *thead-block* irá operar com uma quantidade de *pixels* igual ao máximo de *threads* suportada por bloco, e retornando um valor escalar deste conjunto. Esta redução realiza o primeiro processamento e

armazena os resultados num vetor da memória compartilhada, repetindo a operação a partir dele até que no final apenas uma posição do vetor contenha o resultado. Ao final da execução do *kernel*, cada bloco conterá o resultado de um valor escalar. Tais resultados são colocados como entrada para novas execuções do *kernel* até que haja apenas um bloco, resultando em apenas um valor escalar.

R2V

O modelo R2V é um processamento de histograma, que recebe como entrada uma matriz e seu resultado é um vetor. A implementação desta aplicação se assemelha ao modelo R2S, de forma *top down*, porém seu resultado é um vetor com o tamanho (número de *bins)* definido pelo utilizador. A implementação do modelo R2V também segue a linha dos algoritmos anteriores, com a implementação de um método que processará uma unidade da imagem. O Código Fonte 9 descreve a assinatura para a categoria R2V, que recebe um *pixel* de entrada, um ponteiro para o vetor de esaída, e o número de elementos (número de *bins)*. Porém, como podem ocorrer situações de concorrência de acesso ao vetor de saída, ao se escrever dados neste vetor deve-se usar a API de operações atômicas.

Depois que todos os blocos são processados, os resultados são mesclados em um único vetor de saída. O nível de paralelização neste tipo de solução é variável, pois em certas situações pode ser necessário que diferentes *threads* realizem operações de escrita nos mesmos endereços. Por exemplo, caso todos os valores da imagem sejam iguais, as *threads* em execução no mesmo *warp* tentarão escrever, simultaneamente, no mesmo endereço de memória compartilhada, levando a uma concorrência de acesso. Nestas situações não haverá ganho de desempenho, pois, devido às operações atômicas, a operação de escrita acontecerá de forma serializada (Podlozhnyuk, 2007).

Código Fonte 9. Declaração do método R2V

```
//Recebe um de pixel de entrada, o número de bins e o vetor de saída
com o //tamanho igual ao número de bins
#R2V_FUNC
static void r2vProcessor(unsigned char entrada, int* saida, int bins);
```

Código Fonte 10. Código de operação unitário da categoria R2V

```
#R2V_FUNC
static void r2vProcessor(unsigned char entrada, int* saida, int bins){
        atomicAdd(&saida[entrada], 1);}
```

Segundo Podlozhnyuk (2007), em uma região de *pixels* de uma mesma imagem, há uma grande chance de encontrar valores repetidos. Com o objetivo de reduzir este *bottleneck* (gargalo) da serialização de dados, ou seja, as múltiplas *threads* tentando acessar a mesma posição de memória simultaneamente, os *pixels* dentro de um bloco são divididos em seções armazenadas na memória embarcada do *chip* (*shared memory*),

diminuindo a chance que mais *threads* acessem o mesmo endereço. Desta forma, cada *warp* possui seu espaço de *shared memory*. Entretanto, isso leva a outro passo para mesclar os resultados. A Figura 4 mostra um esquema do processo.

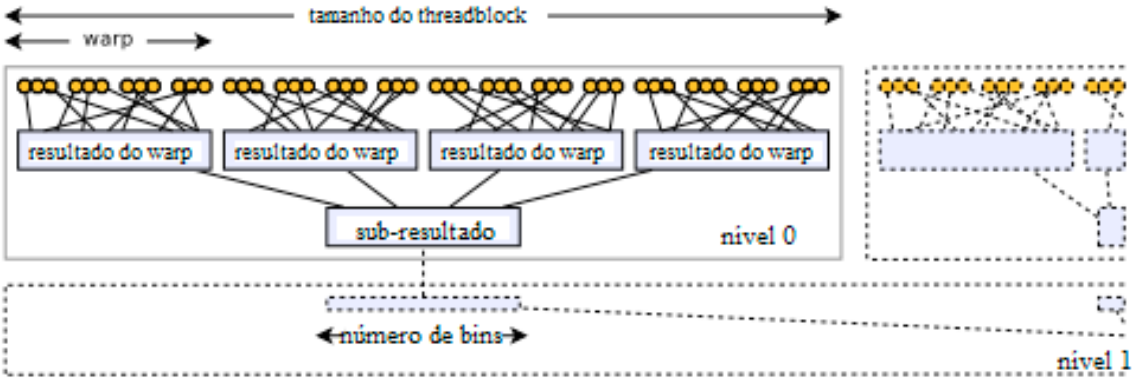

Figura 4. Cada warp trabalha em um espaço exclusivo de memória, que deve ser combinado para o resultado do bloco assim que todas as warps tenham concluído seu processamento.
Fonte: Retirado e traduzido de Nugteren, Corporaal e Mesman (2011).

O Código Fonte 10 apresenta um exemplo de paralelização do histograma gerada pela ferramenta de conversão. A função passada como parâmetro faz uma chamada a operação de adição atômica, da API CUDA, que garante a soma numa posição de memória em que poderia haver concorrência. No código fonte gerado, há dois *kernels*, um para fazer o cálculo do histograma de cada bloco, enquanto outro se encarrega de mesclar os resultados. No primeiro *kernel*, onde é feita a contabilização do histograma, o processo está dividido em 6 partes, conforme o exemplo de Podlozhnyuk (2007). Estas divisões foram criadas para que cada *warp* operasse seu espaço de memória, evitando mais conflitos de escrita. Ao final do cálculo de histograma de todos os *warps*, há uma barreira de sincronização garantindo que o processo seja concluído para todo o bloco. Estes resultados de cada *warp* são mesclados em um único vetor, que se torna o resultado do bloco. Tais vetores são novamente mesclados no segundo *kernel*, resultando no histograma final.

Experimentos, Resultados e Discussão

O *hardware* utilizado foi uma GPU NVIDIA GeForce GTS 450, processador AMD Athlon II x3 435 com 4GB de RAM, no sistema operacional Microsoft Windows 7. Entre as características da GPU utilizada, estão 192 CUDA *cores*, com tamanho máximo de 1024 *threads* por bloco, 1 GB de memória de vídeo GDDR3, *compute capability* 2.1 e arquitetura Fermi. Os testes foram feitos com a mesma imagem em três resoluções diferentes: 7680 x 4320 (8k), 3840 x 2160 (4k) e 1920 x 1080 (1080p).

Os sistemas então são testados com imagens de diferentes resoluções e avaliados quanto a dois requisitos: corretude e tempo. Para avaliar a corretude, comparou-se

os resultados de saída do sistema sequencial e do sistema paralelo, que devem ser idênticos. Caso sejam diferentes, indica falha na conversão, pois a tradução para a versão paralela, de algum modo, alterou o algoritmo original. Em todos os resultados, os resultados foram idênticos e, portanto, corretos.

Quanto ao tempo, os algoritmos foram avaliados de duas formas diferentes: tempo de CPU contra o de GPU (gráficos da coluna esquerda da Figura 5) e o tempo total da CPU contra o total de execução da versão paralela (CPU+GPU) (gráficos da coluna direita da Figura 5). Para se obter o tempo decorrido de processamento pela CPU, foi medido a diferença entre *timestamps* do seu início até sua conclusão. Para a métrica de tempo de GPU, também foi utilizada a diferença entre *timestamps*, obtidos através da API de eventos CUDA, contabilizando apenas o tempo de execução e não de invocação do *kernel* (Harris, 2012).

Os gráficos à esquerda da Figura 5 demonstram a diferença no tempo de execução apenas para o processamento entre a CPU da versão serial e a GPU da versão paralelizada. O eixo vertical contabiliza o tempo em milissegundos decorrido, enquanto o eixo horizontal representa o tamanho da imagem. Os gráficos à direita mostram o resultado de execução levando em conta, no caso da GPU, o tempo de preparação para execução, isto inclui a cópia de dados para a memória do *device*, invocações de *kernels* e barreiras de sincronização.

Para cada resultado, o algoritmo foi executado três vezes e os gráficos apresentam a média destes tempos de execução. O tempo de carregamento da imagem do *hard drive* para a memória não foi contado em nenhum dos casos. Todos os processos e aplicativos não relacionados com os testes foram encerrados antes do início dos experimentos para que o ambiente não fosse influenciado.

Na categoria P2P (gráficos das Figuras 5a e 5b), as operações testadas não consomem muito processamento, pois em apenas um ciclo o resultado de um *pixel* é calculado, assim, a GPU tem um ganho de desempenho pelo paralelismo massivo. A categoria R2S, gráficos das Figuras 5e e 5f, também apresentou melhoria de desempenho. Esse modelo explora a memória *onchip* da GPU, contribuindo para a melhoria do desempenho. O custo computacional por *thread* foi pequeno, resultando em tempos mais longos em alguns casos, em especial com imagens menores.

Para a categoria N2P, gráficos das Figura 5c e 5d mostram a saída da convolução usando máscara de Sobel. O modelo N2P foi o que apresentou maior ganho de desempenho, tanto no tempo de processamento entre CPU e GPU quanto no tempo total de execução, pois ocorrem mais iterações para cada *pixel*, fazendo com que o paralelismo apresente uma diferença de desempenho maior que as outras categorias.

Para o modelo R2V, o resultado pode ser diferente, de acordo com a imagem de entrada. Não há uma definição correta da quantidade de paralelismo a ser obtida, pois as operações podem ser serializadas por conta de acessos simultâneos ao mesmo

endereço de memória. Desta forma, caso uma imagem possua muitos *pixels* da mesma cor, o processo torna-se muito serializado, comprometendo o desempenho do processamento da GPU. A imagem teste possui áreas de céu e grama, aumentando a chance de haver tais *bottlenecks* na paralelização, pois as cores se repetem. Este acesso de escrita serial prejudica a performance, fazendo com que a CPU obtenha valores tão próximos quanto a GPU, já que processa código serial com muito mais velocidade.

No entanto, é notável como a diferença de tempo entre o processamento na CPU e GPU do histograma diminui enquanto o tamanho da imagem aumenta. Visto que mais *pixels* na imagem levam a necessidade de mais ciclos de execução para a CPU do que o necessário para a GPU. Diferente dos outros modelos, o R2V não faz acesso à memória global de forma coalescente, sacrificando este tempo a fim de evitar concorrência à memória compartilhada. Este acesso foi feito de forma deliberada pois é mais provável de se encontrar *pixels* com valores iguais ou semelhantes na mesma área. Desta forma, acessar as áreas da imagem com um deslocamento (*offset*) oferece uma chance maior de encontrar valores diferentes (Podlozhnyuk, 2007).

Considerações Finais

Neste trabalho, apresentamos um sistema para paralelização semi-automática de algoritmos de PDI usando a tecnologia NVIDIA CUDA. Para utilizar o sistema, o usuário deve, por meio de marcações no código fonte, selecionar um tipo de operação e desenvolver uma função que será executada em paralelo pelas *threads* da GPU. A partir destas informações, o sistema produz automaticamente códigos para alocação de recursos na GPU, invocação do *kernel* e gerência de uso da arquitetura de memória da GPU.

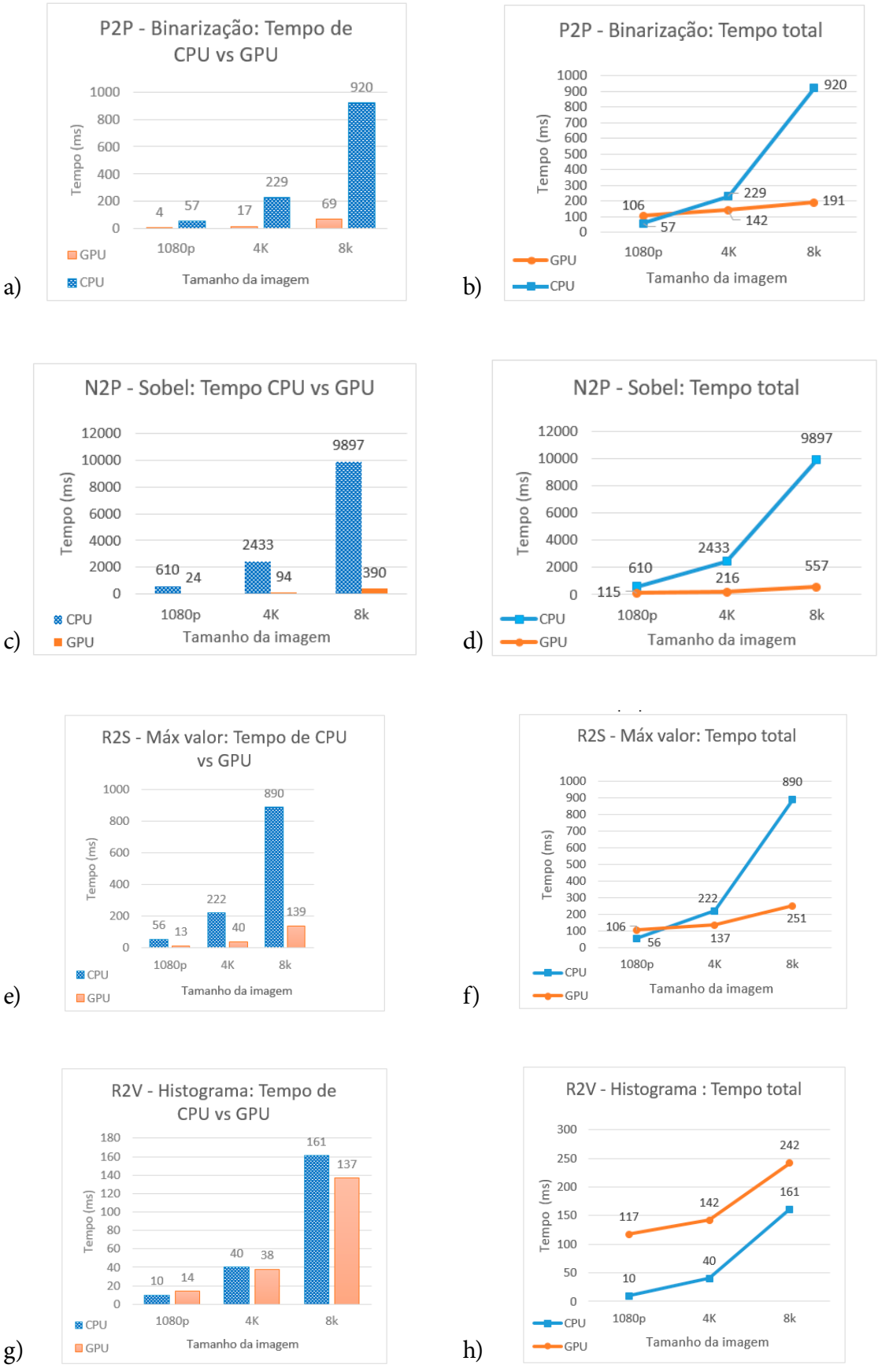

Figura 5. Comparação do tempo de processamento e da aplicação entre CPU e GPU para as categorias P2P (a e b), N2P (c e d), R2S (e e f) e R2V (g e h).

O sistema disponibiliza quatro tipos de operações, operações *pixel* a *pixel* (P2P), operações que computam o valor de um *pixel* dada a sua vizinhança (N2P), operações que reduzem uma imagem em um valor escalar (R2S), operações que reduzem uma imagem em um vetor (R2V). Com o sistema proposto, desenvolver um algoritmo de soma de imagens, por exemplo, se reduz a selecionar as marcações da categoria P2P e desenvolver uma função para soma dos valores dos *pixels* (3 linhas de código).

Analisando os experimentos e seus resultados, observamos que o código paralelo produzido pelo sistema obteve melhor performance que a sua versão serial. Quanto maiores as imagens de entrada, maiores são os ganhos de desempenho obtidos, tanto na comparação de processamento de GPU e CPU como no tempo total da aplicação, exceto na categoria R2V. Destaca-se que o ganho de desempenho é proveniente de um código fonte gerado de forma automática, abstraindo detalhes específicos de desenvolvimento da arquitetura CUDA. Os maiores ganhos foram obtidos nas aplicações que exigiam mais cálculos, como nos casos N2P. Nesses casos, os códigos paralelos chegaram a ser dezenas de vezes mais rápidos que os códigos sequenciais. Na categoria P2P foi possível obter quinze vezes mais desempenho, e na categoria R2S os programas foram executados seis vezes mais rápido que suas versões sequenciais, executadas na CPU.

Em trabalhos futuros, serão estudadas e desenvolvidas novas implementações das operações do tipo R2V que, potencialmente, levarão a melhorias de desempenho tão significativas quanto aquelas alcançadas pelas demais operações. Também **é** possível melhorar o uso da memória compartilhada na versão paralela gerada. O sistema proposto poderá ser estendido para implementação de algoritmos de outros domínios da computação, além de PDI, e pode-se comparar **à** esta ferramenta com aplicações similares, como o OCaml, OpenACC e OpenMP.

Referências

Bakhoda, A., Yuan, G., Fung, W., Wong, H., & Aamodt, T. (2009). *Analyzing CUDA workloads using a detailed GPU simulator*. 2009 IEEE International Symposium on Performance Analysis of Systems and Software, 163-174.

Braga, M. L., de la Rocha Ladeira, R., & Mota, L. D. A. T. (2017). Resolução do Problema das n-Rainhas com Programação Paralela. *Revista de Empreendedorismo, Inovação e Tecnologia*, 3(2), 41-47.

Daniel, H. (2003). *Paralelização automática de algoritmos matriciais.* (Tese de doutorado, Curso de Engenharia Electrônica e Computação na Especialidade de Sistemas de Controlo, Universidade do Algarve).

Gonzalez, R. C., & Woods, R. E. (2005). *Digital image processing*. Upper Saddle River: Prentice-Hall, Inc.

Harris, M. (2012). *How to implement performance metrics in CUDA C/C++*. NVIDIA Developer Zone. Disponível em: <https://goo.gl/qG2RIR>. Acesso em: 5 jun. 2016.

ITSEEZ. (2016). *About OpenCV*. Disponível em: <http://opencv.org/about.html>. Acesso em: 03 maio 2016.

Kouzinopoulos, C., & Margaritis, K. (2009). *String Matching on a Multicore GPU Using CUDA*. *2009 13th* Panhellenic Conference on Informatics, 14-18.

NVIDIA. (2016). *CUDA Programação Paralela Facilitada*. Disponível em: <http://www.nvidia.com.br/object/cuda_home_new_br.html>. Acesso em: 22 abr.

NVIDIA. (2017). *Parallel Thread Execution ISA Version 6.0*. Disponível em: <http://docs.nvidia.com/cuda/parallel-thread-execution/index.html>. Acesso em: 22 out.

NVIDIA (a) (2016). CUDA C *Programming Guide*. 2015. Disponível em: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html >. Acesso em: 21 abr. 2016.

Nugteren, C., Corporaal, H., & Mesman, B. (2011, July). *Skeleton-based automatic parallelization of image processing algorithms for GPUs*. In 2011 IEEE International Conference on Embedded Computer Systems (SAMOS), 25-32.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE*, 96(5), 879-899.

Parhami, C. (1999). Introduction to parallel processing. 1. ed. New York: Plenum Pub Corp, 1999. 557 p.

Podlozhnyuk, V. (2007). *Histogram calculation in CUDA*. NVIDIA Corporation, White Paper. Disponível em: <https://goo.gl/FwK1nN>. Acesso em: 27 maio 2016.