Detecção de Bordas com CUDA

Paralelização do Algoritmo de Canny para Detecção de Bordas de Imagens com CUDA

Disciplina MO-644A

Universidade Estadual de Campinas - Unicamp
Instituto de Computação - IC
Professor Ph.D. Guido Araujo
Co-orientador: Rafael Cardoso Fernandes Sousa

Andrius Henrique Sperque	RA: 189918
Bruno Juliani Martho	RA: 208502

Campinas, 26 de Junho de 2017

1. Introdução

Desenvolvido por John F. Canny em 1986, o detector de bordas Canny é um algoritmo de múltiplos estágios utilizado para detecção de bordas em imagens. O processamento do algoritmo de detecção pode ser compreendido em cinco etapas principais: aplicação de um filtro gaussiano, encontro da intensidade do gradiente da imagem, aplicação de supressão não-máxima, aplicação de limites duplos e rastreamento de borda por histereses.

A proposta deste trabalho é minimizar o tempo para detecção de borda de imagem pelo Canny realizando a paralelização de etapas deste processo através da plataforma CUDA da NVidia, aprendida em sala de aula.

Esta seção tem por objetivo justificar e apontar as mudanças realizadas no programa serial Canny de detecção de bordas pela adição de códigos paralelos em substituição de códigos de execução serial.

1.1. Região de código sequencial

Através da utilização da ferramenta de perfilamento *Perf*, foi possível verificar e priorizar as funções (etapas do algoritmo) que possuem maiores tempos de processamento. A **Tabela 1** apresenta os resultados obtidos na primeira execução do *profiling*. As funções em negrito foram paralelizadas:

```
# Samples: 88K of event 'cpu-clock:u'
# Event count (approx.): 22100500000
#
# Overhead  Command      Symbol
#
  74.68%  canny_edge  [.] gaussian_smooth
   8.94%  canny_edge  [.] derrivative_x_y
   6.51%  canny_edge  [.] non_max_supp
   4.98%  canny_edge  [.] apply_hysteresis
   2.41%  canny_edge  [.] magnitude_x_y
   1.83%  canny_edge  [.] follow_edges
   0.12%  canny_edge  [.] sqrt@plt
```

Tabela 1. *Perf* aplicado para o código serial.

Com o resultado da Tabela 1, passa-se para a análise de viabilidade de paralelização dos métodos do algoritmo.

1.1.1. Gaussian Smooth

Como *gaussian_smooth* é a função que demanda mais de 70% de todo o tempo de execução do algoritmo, foi priorizada para realização da paralelização. A função em questão é responsável pela filtragem Gaussiana, que é usada para desfocar imagens e remover ruídos e detalhes. Através da análise dos loops que a compõem, verificou-se que os laços são DOALL e, portanto, podem ser paralelizados. A Figura 1 apresenta a versão serial do código:

```
/*******************************************************************************
* Blur in the x - direction.
*******************************************************************************/
if(VERBOSE) printf("   Bluring the image in the X-direction.\n");
for(r=0;r<rows;r++){
   for(c=0;c<cols;c++){
      dot = 0.0;
      sum = 0.0;
      for(cc=(-center);cc<=center;cc++){
         if(((c+cc) >= 0) && ((c+cc) < cols)){
            dot += (float)image[r*cols+(c+cc)] * kernel[center+cc];
            sum += kernel[center+cc];
         }
      }
      tempim[r*cols+c] = dot/sum;
   }
}

/*******************************************************************************
* Blur in the y - direction.
*******************************************************************************/
if(VERBOSE) printf("   Bluring the image in the Y-direction.\n");
for(c=0;c<cols;c++){
   for(r=0;r<rows;r++){
      sum = 0.0;
      dot = 0.0;
      for(rr=(-center);rr<=center;rr++){
         if(((r+rr) >= 0) && ((r+rr) < rows)){
            dot += tempim[(r+rr)*cols+c] * kernel[center+rr];
            sum += kernel[center+rr];
         }
      }
      (*smoothedim)[r*cols+c] = (short int)(dot*BOOSTBLURFACTOR/sum + 0.5);
   }
}
```

Figura 1. Código serial da etapa Gaussian Smooth

1.1.2. Função Derivative X e Y

Função que calcula a primeira derivada da imagem nas direções x e y. Sendo os filtros diferenciais usados:

$$dx = \begin{bmatrix} -1 & 0 & +1 \end{bmatrix} \quad \text{and} \quad dy = \begin{bmatrix} -1 \\ 0 \\ +1 \end{bmatrix}$$

A função **derivative_x_y()** apresenta o segundo maior tempo de execução segundo o perfilamento inicial realizado. Sua paralelização foi possível por se tratar de um outro exemplo de DOALL. A Figura 2 apresenta sua versão serial:

```
/****************************************************************************
* Compute the x-derivative. Adjust the derivative at the borders to avoid
* losing pixels.
****************************************************************************/
if(VERBOSE) printf("   Computing the X-direction derivative.\n");
for(r=0;r<rows;r++){
   pos = r * cols;
   (*delta_x)[pos] = smoothedim[pos+1] - smoothedim[pos];
   pos++;
   for(c=1;c<(cols-1);c++,pos++){
      (*delta_x)[pos] = smoothedim[pos+1] - smoothedim[pos-1];
   }
   (*delta_x)[pos] = smoothedim[pos] - smoothedim[pos-1];
}

/****************************************************************************
* Compute the y-derivative. Adjust the derivative at the borders to avoid
* losing pixels.
****************************************************************************/
if(VERBOSE) printf("   Computing the Y-direction derivative.\n");
for(c=0;c<cols;c++){
   pos = c;
   (*delta_y)[pos] = smoothedim[pos+cols] - smoothedim[pos];
   pos += cols;
   for(r=1;r<(rows-1);r++,pos+=cols){
      (*delta_y)[pos] = smoothedim[pos+cols] - smoothedim[pos-cols];
   }
   (*delta_y)[pos] = smoothedim[pos] - smoothedim[pos-cols];
}
```

Figura 2. Código serial da função Derivative X e Y

1.1.3. Função Magnitude X e Y

Calcula a magnitude do gradiente da imagem pelo cálculo da raiz quadrada da soma das derivadas primeiras de X e Y ao quadrado. Essa função não demanda muito tempo de execução, mas também pôde ser paralelizada por ser um DOALL. A Figura 3 apresenta sua versão serial:

```
/* Part of the function that computes the magnitude of the gradient. */
for(r=0,pos=0;r<rows;r++){
   for(c=0;c<cols;c++,pos++){
      sq1 = (int)delta_x[pos] * (int)delta_x[pos];
      sq2 = (int)delta_y[pos] * (int)delta_y[pos];
      (*magnitude)[pos] = (short)(0.5 + sqrt((float)sq1 + (float)sq2));
   }
}
```

Figura 3. Código serial da função Magnitude X e Y

1.1.4. Outras funções do programa serial

Após a análise completa do algoritmo, foi constatado que as outras funções não poderiam ser paralelizadas por serem DOACROSS. Porém, fora possível paralelizar as duas funções que mais demandavam tempo e uma outra função extra.

2. Imagens e Explicações

Esta seção apresenta imagens relativas aos trechos de códigos paralelizados e explicações detalhadas sobre a codificação de cada trecho. É importante ressaltar que para todas as funções paralelizadas e descritas a seguir foram implementados códigos semelhantes para inicialização e chamada das funções na GPU (Figura 4), sendo estes: alocação de memória no dispositivo (GPU), transferência de dados para a GPU, chamada da função, transferência de dados de volta para o HOST e liberação da memória alocada.

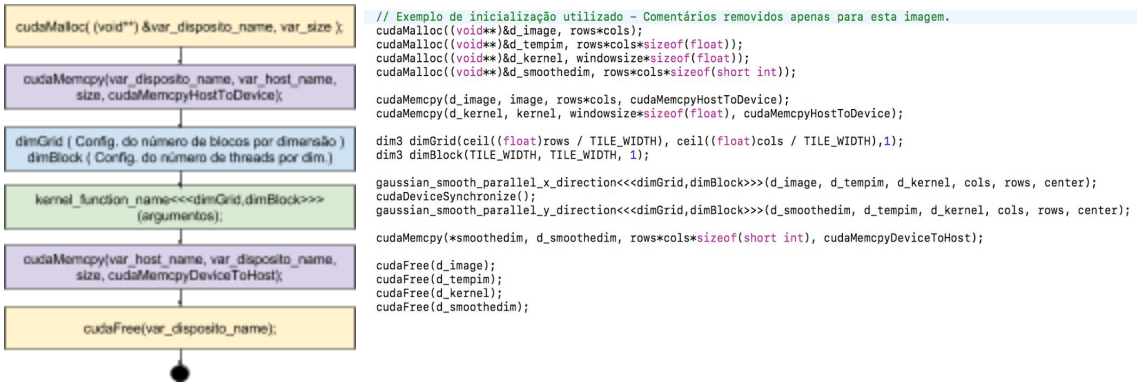

Figura 4. Processo comum para as chamadas de funções do Kernel

No desenvolvimento deste trabalho foi utilizado o servidor Parsusy do IC da Unicamp, o qual possui instalada uma placa gráfica aceleradora modelo NVidia K40c. Este dispositivo permite 2048 threads por multiprocessador, 1024 threads por bloco, 1024 blocos por dimensão e 2147483647 threads por grid de uma dimensão.

2.1. Gaussian Smooth

Para o Gaussian Smooth foram utilizadas duas dimensões, X e Y, possibilitando uma coerência com a forma em que uma imagem é representada, por largura e altura (em pixels), sendo cada pixel de uma imagem encontrado por endereçamento em x e y, representados no código do kernel (função a ser executada na GPU) pelas variáveis locais 'r' e 'c', respectivamente. Ambas 'r' e 'c' são geradas por: número de identificação do bloco vezes sua dimensão (tamanho) somado ao número identificador da thread, gerando um ID único para cada pixel em sua dimensão.

Para também maximizar a performance da GPU e obter escalabilidade quanto ao tamanho da imagem, utilizou-se a estrutura de dados Dim3 da plataforma CUDA.

```
// TILE_WIDTH = 16
dim3 dimGrid(ceil((float)rows / TILE_WIDTH), ceil((float)cols/TILE_WIDTH),1);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
```

A figura 5 apresenta as funções que processam a imagem nas direções X e Y. O propósito do Kernel para Gaussian Smooth é que cada pixel seja processado paralelamente por uma *thread*, reduzindo o tempo de aplicação do efeito. A variável ***d_image*** é a imagem vetorizada a ser processada e ao final do processo, ***d_smoothedim*** é a imagem com a redução de ruídos e detalhes.

As funções ***__fadd_rn*** e ***__fmul_rn*** foram utilizadas pois adicionam e multiplicam valores de ponto flutuante no modo *round-to-closer-even*.

```
/*******************************************************************************************
 * PROCEDURE: gaussian_smooth_parallel_x_direction and gaussian_smooth_parallel_y_direction
 * PURPOSE: Blur an image with a gaussian filter using CUDA platform
 * DATE: 6/15/2017
 *******************************************************************************************/
__global__ void gaussian_smooth_parallel_x_direction(unsigned char *d_image, float *d_tempim, float *d_kernel, int cols, int
    rows, int center) {
    int r, c, cc;

    //Calcula o i e j da imagem, correspondente ao for do serial
    r = blockIdx.x * blockDim.x + threadIdx.x;
    c = blockIdx.y * blockDim.y + threadIdx.y;

    float dot = 0.0;
    float sum = 0.0;
    if(r < rows && c < cols){
         for(cc=(-center);cc<=center;cc++){
            if(((c+cc) >= 0) && ((c+cc) < cols)){
                // operações precisas nos pontos flutuantes
                dot = __fadd_rn(dot, __fmul_rn((float)d_image[r*cols+(c+cc)], d_kernel[center+cc]));
                sum = __fadd_rn(sum, d_kernel[center+cc]);
            }
         }
         d_tempim[r*cols+c] = __fdiv_rn(dot,sum);
    }
}

__global__ void gaussian_smooth_parallel_y_direction(short int *d_smoothedim, float *d_tempim, float *d_kernel, int cols, int
    rows, int center) {
    int r, c, rr;

    //Calcula o i e j da imagem, correspondente ao for do serial
    r = blockIdx.x * blockDim.x + threadIdx.x;
    c = blockIdx.y * blockDim.y + threadIdx.y;

    float dot = 0.0;
    float sum = 0.0;
    if(r < rows && c < cols){
      for(rr=(-center);rr<=center;rr++){
         if(((r+rr) >= 0) && ((r+rr) < rows)){
                // operações precisas nos pontos flutuantes
                dot = __fadd_rn(dot, __fmul_rn(d_tempim[(r+rr)*cols+c], d_kernel[center+rr]));
                sum = __fadd_rn(sum, d_kernel[center+rr]);
         }
      }

      d_smoothedim[r*cols+c] = (short int)(dot*BOOSTBLURFACTOR/sum + 0.5);
    }
}
```

Figura 5. Kernel the Gaussian Smooth.

2.2. Função Derivative X e Y

Assim como para a função Gaussian Smooth descrita no tópico 2.1, uma mesma inicialização foi utilizada para o cálculo e aplicação da primeira derivada sobre a imagem, exceto que neste caso apenas uma dimensão é utilizada para cada Kernel (Figura 6), pois o algoritmo primeiro varre a imagem na vertical e depois na horizontal, conforme apresentado na Figura 7:

```
//Calcula os dims referentes a imagem
dim3 dimGrid(ceil((float)rows / TILE_WIDTH), 1,1);
dim3 dimBlock(TILE_WIDTH, 1, 1);
dim3 dimGrid2(1, ceil((float)cols / TILE_WIDTH),1);
dim3 dimBlock2(1, TILE_WIDTH, 1);


/*******************************************************************************
* Compute the x-derivative. Adjust the derivative at the borders to avoid
* losing pixels.
*******************************************************************************/
 if(VERBOSE) printf("   Computing the X-direction derivative.\n");
compute_derivative_x_direction<<<dimGrid,dimBlock>>>(d_smoothedim, d_delta_x, cols, rows);
```

Figura 6. Inicialização para o compute_derivative_x_y.

```
/*******************************************************************************
 * PROCEDURE: derrivative_x_y
 * PURPOSE: Compute the first derivative of the image in both the x any y
 * directions using CUDA concept.
 * DATE: 6/15/2017
 *******************************************************************************/

__global__ void compute_derivative_x_direction(short int *d_smoothedim, short int *d_delta_x, int cols, int rows) {
    int r, c, pos;
    r = blockIdx.x * blockDim.x + threadIdx.x;

    if(r < rows) {
        pos = r * cols;
        d_delta_x[pos] = d_smoothedim[pos+1] - d_smoothedim[pos];
        pos++;
        for(c=1;c<(cols-1);c++,pos++) {
            d_delta_x[pos] = d_smoothedim[pos+1] - d_smoothedim[pos-1];
        }
        d_delta_x[pos] = d_smoothedim[pos] - d_smoothedim[pos-1];
    }
}

__global__ void compute_derivative_y_direction(short int *d_smoothedim, short int *d_delta_y, int cols, int rows) {
    int r, c, pos;
    c = blockIdx.y * blockDim.y + threadIdx.y;

    if(c < cols) {
        pos = c;
        d_delta_y[pos] = d_smoothedim[pos+cols] - d_smoothedim[pos];
        pos += cols;
        for(r=1;r<(rows-1);r++,pos+=cols){
            d_delta_y[pos] = d_smoothedim[pos+cols] - d_smoothedim[pos-cols];
        }
        d_delta_y[pos] = d_smoothedim[pos] - d_smoothedim[pos-cols];
    }
}
```

Figura 7. Kernel de compute_derivative_x_y.

2.3. Função Magnitude X e Y

A função compute_magnitude_x_y utiliza a mesma forma de inicialização e características de gaussian_smooth, pois gera um valor de magnitude para cada pixel.

```
/*******************************************************************************
* PROCEDURE: compute_magnitude_x_y
* PURPOSE: Compute the magnitude of the gradient using CUDA.
* This is the square root of the sum of the squared derivative values.
* DATE: 6/15/2017
*******************************************************************************/
__global__ void compute_magnitude_x_y(short int *d_magnitude, short int *d_delta_x, short int *d_delta_y, int cols, int rows) {
    int r, c, pos, sq1, sq2;
    r = blockIdx.x * blockDim.x + threadIdx.x;
    c = blockIdx.y * blockDim.y + threadIdx.y;

    if(r < rows && c < cols) {
         // faz o calculo de pos
         pos = r*cols + c;
         sq1 = (int)d_delta_x[pos] * (int)d_delta_x[pos];
         sq2 = (int)d_delta_y[pos] * (int)d_delta_y[pos];
         d_magnitude[pos] = (short)(0.5 + sqrt((float)sq1 + (float)sq2));
    }
}
```

Figura 8. Kernel de compute_magnitude_x_y.

3. Gráfico de Desempenho

É apresentado nesta seção o gráfico que mostra a variação de tempo para cada teste que é feito do arquivo makefile, comparando o código paralelizado com relação ao serial.

Os testes 1 e 2 foram realizados com uma figura de 8 MB de tamanho, havendo variação nos parâmetros de entrada. Os testes 3 e 4 foram realizados com uma figura de 15 MB, havendo variações nos parâmetros de entrada. Finalmente, os testes 5 e 6 foram realizados com uma figura de 80 MB com diferentes parâmetros de entrada. Os tempos de execução são representados no Gráfico 1.

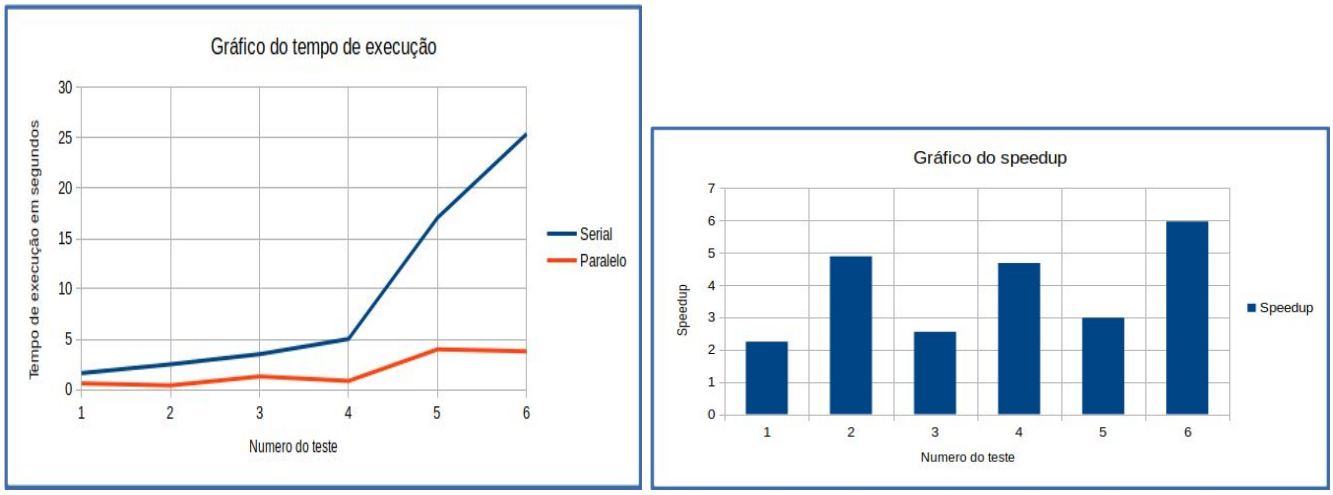

Gráfico 1. Tempo de execução geral e speedup por testes executados.

4. Dificuldades Encontradas

Ao longo do desenvolvimento do projeto, algumas dificuldades foram enfrentadas, dentre elas:

- Após a implementação da versão paralela da função *Gaussian Smooth*, notou-se que ao aplicar o comando ***diff*** do linux sobre as imagens geradas pela versão serial e paralela, o resultado mostrava diferenças entre elas. Essa diferença foi causada pela diferença entre a precisão do ponto flutuante utilizada pela GPU (Device) e o CPU (Host). A solução para esse problema foi utilizar a função do CUDA **"*__fadd_rn*"** e "***__fmul_rn***".
- Assim como em experimentos em aula, verificamos que após a compilação de um código CUDA, era necessário executar uma vez uma chamada ao dispositivo para que suas *libraries* fossem carregadas em memória. Assim, utilizou-se uma função sem conteúdo chamada *__global__ void discard() { }, para que os drivers fossem carregados quando essa fosse chamada, sem a necessidade de re-execução do programa.*
- Por usarmos o servidor Parsusy para realização deste trabalho, o que é destinado para alunos do IC, assistimos alguns problemas de instabilidade nos serviços, assim como sua total indisponibilidade durante mais de 2 dias, ocasionando na suspensão das atividades deste trabalho.