

Relatório do Projeto 2

Canny Edge Detector using CUDA

Arquitetura de Computadores Avançada

2014/2015

Daniel Silva 51908
Ivo Silva 64833

Índice

Tarefa 1

Convolution function (global memory)

Convolution function (texture memory)

Convolution function (shared memory)

Tarefa 2

Non_maximum_supression function (global memory)

Non_maximum_supression function (texture memory)

Tarefa 3

First_edges function (global memory)

First_edges function (texture memory)

First_edges function (shared memory)

Tarefa 4

Hysteresis_edges function (global memory)

Hysteresis_edges function (texture memory)

Hysteresis_edges function (shared memory)

Conclusão

Bibliografia

Tarefa 1

Convolution function (global memory)

➤ Arquitetura da Solução:

Na implementação desta solução foi criada uma função auxiliar de forma a que o código presente no `canny_device` seja mais legível. O que essa função faz é instanciar novas estruturas e alocar a memória para as mesmas serem usadas nas placas gráficas, copiar blocos de memória necessários do processador para a placa gráfica, configurar as dimensões da grid e dos blocos, a invocação da função de convolução em CUDA e no final copiar da placa gráfica para o host o conteúdo desejado depois de processado.

Neste caso enviamos para a placa gráfica a imagem à qual se pretende aplicar a convolução, a matriz de convolução e as dimensões da imagem e da matriz.

Recebe-se um ponteiro para um array de `pixel_t` que corresponde à imagem após a operação de convolução.

➤ Estruturas e Algoritmos:

O primeiro passo nesta função foi descobrir o id da thread responsável por processar cada pixel da imagem. O id será então dado por $m + n * nx$, onde:

$$m = \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x}$$

e

$$n = \text{threadIdx.y} + \text{blockIdx.y} * \text{blockDim.y}$$

‘m’ equivale à posição no eixo x da thread e ‘n’ à posição y. Como é usado um array unidimensional é necessário multiplicar a posição y (n) pela largura da imagem (nx) de maneira a obter o id correto, visto que uma deslocação de uma unidade no eixo y corresponde a adicionar nx ao id atual no array da imagem.

Depois foi calculado o `khalf` para se saber quantos “vizinhos” eram necessários ter em conta nesta operação.

Foi adicionada uma condição que evita o processamento dos pixels das pontas e dos cantos devido a não existirem os pixels vizinhos necessários.

Após as verificações, a operação em em tudo semelhante à praticada no processador à exceção da travessia de todos os elementos do array. Neste caso a operação é feita apenas para o pixel correspondente ao id da thread visto que os outros pixels são tratados por outras threads. A variável ‘pixel’ resulta no somatório dos vizinhos do pixel correspondente à thread multiplicado com o valor do kernel correspondente àquele vizinho.

No final, é atribuído ao mesmo pixel da nova imagem o valor de ‘pixel’.

➤ **Resultados Obtidos:**

Estes resultados foram obtidos com as seguintes funções em CUDA:

- CUDA_convolution_kernel

```
aca0310@nikola:~/Projeto2/aca_canny$ sh fuuu.sh ex1/canny_ex1  
  
Image: lena.pgm  
gaussian_filter: kernel size 7, sigma=1  
gaussian_filter: kernel size 7, sigma=1  
Host processing time: 178.167175 (ms)  
Device processing time: 148.345886 (ms)  
images are identical  
  
Image: house.pgm  
gaussian_filter: kernel size 7, sigma=1  
gaussian_filter: kernel size 7, sigma=1  
Host processing time: 50.101376 (ms)  
Device processing time: 43.058849 (ms)  
images are identical  
  
Image: chessRotatel.pgm  
gaussian_filter: kernel size 7, sigma=1  
gaussian_filter: kernel size 7, sigma=1  
Host processing time: 20.025185 (ms)  
Device processing time: 17.153984 (ms)  
images are identical
```

➤ **Instruções de Compilação e Execução:**

Para compilar:

```
nvcc -arch=sm_13 -O -Icommon/inc canny_ex1.cu -Llib -lcutil_x86_64 -o canny_ex1
```

Para executar:

```
./canny_ex1 -i imagem.pgm -o imagemOut.pgm
```

Convolution function (texture memory)

➤ Arquitetura da Solução:

Antes de fazer qualquer declaração identificámos onde poderíamos fazer o melhor uso da memória de textura, e por isso nesta função, a variável *A* do tipo *pixel_t**, como apenas é lida e é lida bastantes vezes, é a indicada para ser colocada na memória de texturas, de forma a termos um ganho na velocidade de acesso aos dados de *A*.

Para usarmos a memória de textura, foi necessário declararmos uma texture reference. Após termos feito a a declaração anterior, ligámos os dados de *A*, já alocados na memória global da placa gráfica, ao local declarado na memória de textura.

Já no kernel da nossa função, para obtermos os dados de *A*, usávamos a função *tex1Dfetch()*; que devolve o valor da posição indicado por nós, mas que está residente na memória de textura.

➤ Estruturas e Algoritmos:

Declaração de uma referencia de textura:

```
texture<pixel_t,1,cudaReadModeElementType> A_texture;
```

Fazer a ligação entre dados na placa gráfica da memória global para a memória de textura:

```
cudaBindTexture(0, A_texture, A, memsize);
```

Ler da memória de textura:

```
pixel += tex1Dfetch(A_texture,(n + j) * nx + m + i) * kernel[c];
```

➤ Resultados Obtidos:

Estes resultados foram obtidos com as seguintes funções em CUDA:

- CUDA_convolution_kernel

```
aca0310@nikola:~/Projeto2/aca_canny$ sh fuuu.sh ex1/canny_ex1_texture

Image: lena.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 177.481567 (ms)
Device processing time: 147.697021 (ms)
images are identical

Image: house.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 51.533920 (ms)
Device processing time: 44.587746 (ms)
images are identical

Image: chessRotatet1.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 20.015615 (ms)
Device processing time: 17.055649 (ms)
images are identical
```

➤ Instruções de Compilação e Execução:

Para compilar:

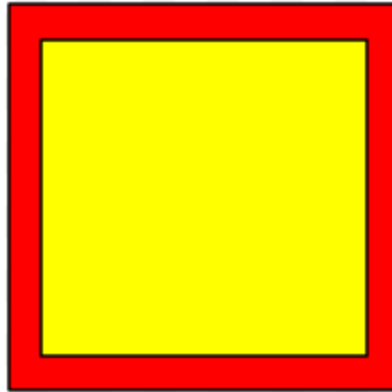
```
nvcc -arch=sm_13 -O -Icommon/inc canny_ex1_texture.cu -Llib -lcutil_x86_64 -o  
canny_ex1_texture
```

Para executar:

```
./canny_ex1_texture -i imagem.pgm -o imagemOut.pgm
```

Convolution function (shared memory)

➤ Estruturas e Algoritmos:



Na imagem acima vemos o que representa um bloco da grid, a amarelo, e o que representa um array de memória partilhada nesse mesmo bloco, a vermelho. Acontece que a função de convolução funciona utilizando os vértices vizinhos do vértice que se encontra a processar. Então para isto o array da memória partilhada a usar irá ter que ser do seguinte tamanho:

$$(blockDim.x + 2) * (blockDim.y + 2)$$

Desta maneira o array de memória partilhada irá ter mais 2 colunas e mais 2 linhas do que o bloco de processamento em si. Estas linhas e colunas servirão para colocar os pixels vizinhos do pixel a processar para a operação poder ser executada com sucesso.

Neste caso começámos por calcular o id da thread, tal como nos exercícios de memória global mas também o sid que é o identificador da posição do pixel no array de memória partilhada. Este sid é dado por:

$$sid = (threadIdx.x + 1) + ((threadIdx.y + 1) * sdimx),$$

onde sdimx é a largura da matriz de memória partilhada.

No início deste algoritmo é atribuído ao array de memória partilhada, na posição sid, o valor do input, na posição id. Ou seja, à posição que representa o pixel a ser processado na matriz de memória partilhada é atribuído o valor desse mesmo pixel na imagem de input.

Depois disto, se o pixel a ser processado não tiver os vizinhos necessários, ou seja, se for um pixel dos extremos da imagem, a execução termina ali para ele.

Neste momento as posições do array de memória partilhada correspondente ao bloco da imagem já estará preenchido e resta preencher com os vizinhos dos pixels dos extremos de cada bloco.

Se não se tratar de um pixel dos extremos da imagem temos que verificar mais condições.

Elas são:

Será um pixel dos cantos de um bloco?
Será um pixel de um extremo de um bloco?

Se for um pixel dos cantos de um bloco é necessário obter o valor dos pixels acima, ao lado (do lado da extremidade) e acima desse para preencher no array da memória partilhada.

Se for um pixel de um extremo (excluindo os cantos), é necessário obter o pixel ao lado (do lado da extremidade).

Note-se que estes pixels necessários nestes casos especiais já serão pertencentes a outros blocos.

Depois disto é necessário utilizar a diretiva 'syncthread' de modo a que o array da memória partilhada já esteja completamente preenchido quando for necessário para a leitura.

A partir destas alterações o algoritmo é o mesmo que foi usado na implementação com memória global, com a alteração do valor utilizado para o cálculo do pixel passar a ser obtido através do array de memória partilhada em vez do array de input que era utilizado na implementação em memória global. Desta maneira cada bloco terá a informação necessária para o processamento da função, possibilitando a execução de mais instruções em paralelo. Isto resulta numa melhoria dos tempos de processamento significativa.

➤ Resultados Obtidos:

```
Image: lena.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 183.591675 (ms)
Device processing time: 161.298431 (ms)
0 diffs of 1
0 diffs of 2
0 diffs of 3
0 diffs of 4
0 diffs of 5
74 diffs > 5

Image: house.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 51.543839 (ms)
Device processing time: 44.637280 (ms)
0 diffs of 1
0 diffs of 2
0 diffs of 3
0 diffs of 4
0 diffs of 5
58 diffs > 5

Image: chessRotatel.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 20.043615 (ms)
Device processing time: 17.200865 (ms)
0 diffs of 1
0 diffs of 2
0 diffs of 3
0 diffs of 4
0 diffs of 5
21 diffs > 5
```

Este teste foi efetuado com todas as funções do host à exceção da convolução, que é a da implementação com shared memory.

Esta função não funciona perfeitamente. As imagens processadas por esta função ficam com alguns pontos de diferença, normalmente nas extremidades da imagem. Não conseguimos resolver este problema, no entanto a imagem gerada está muito perto de ser igual à imagem de referência.

➤ Instruções de Compilação e Execução:

Para compilar:

```
nvcc -arch=sm_13 -O -Icommon/inc canny_ex1_shared.cu -Llib -lcutil_x86_64 -o canny_ex1_shared
```

Para executar:

```
./canny_ex1_shared -i imagem.pgm -o imagemOut.pgm
```

Tarefa 2

Non_maximum_supression function (global memory)

➤ Arquitetura da Solução:

Começámos por alocar no device espaço para os arrays correspondentes às imagens resultantes dos gradientes sobre o eixo x e y e a imagem resultante da nossa função.

Copiámos os blocos de memória desses mesmos arrays para para os arrays correspondentes na placa gráfica e fizemos um MemSet do array da imagem de retorno para que todos os seus elementos sejam iguais ao valor 0 de modo a não deixar as posições do array não preenchidas com bits aleatórios que podem prejudicar o output da imagem.

A dimensão de cada bloco será de 16x16 e a dimensão da grid é dada pela seguinte expressão:

$$\text{ceil}(\text{float}(\text{nx})/16) \times \text{ceil}(\text{float}(\text{ny})/16)$$

Desta maneira é atribuído o número de blocos necessário ao processamento da totalidade da imagem.

Depois do processamento feito é copiado o resultado da placa gráfica para o array nms e esse será o resultado final da função.

➤ Estruturas e Algoritmos:

Inicialmente calculámos o id da thread correspondente a cada pixel da imagem da mesma maneira que foi feito no exercício anterior com memória global.

Colocámos uma condição para que os pixeis dos extremos da imagem não sejam processados uma vez que não têm todos os vizinhos necessários para executar o algoritmo.

A partir deste ponto o procedimento é em tudo semelhante ao da função `non_maximum_supression_kernel` do host: são encontrados os vizinhos e são feitas as comparações necessárias para saber se o pixel em causa faz parte do contorno.

➤ **Resultados Obtidos:**

Estes resultados foram obtidos com as seguintes funções em CUDA:

- CUDA_convolution_kernel
- CUDA_non_maximum_supression_kernel

```
aca0310@nikola:~/Projeto2/aca_canny$ sh fuuu.sh ex2/canny_ex2  
  
Image: lena.pgm  
gaussian_filter: kernel size 7, sigma=1  
gaussian_filter: kernel size 7, sigma=1  
Host processing time: 178.771484 (ms)  
Device processing time: 121.122368 (ms)  
images are identical  
  
Image: house.pgm  
gaussian_filter: kernel size 7, sigma=1  
gaussian_filter: kernel size 7, sigma=1  
Host processing time: 51.617058 (ms)  
Device processing time: 38.846336 (ms)  
images are identical  
  
Image: chessRotatel.pgm  
gaussian_filter: kernel size 7, sigma=1  
gaussian_filter: kernel size 7, sigma=1  
Host processing time: 20.067841 (ms)  
Device processing time: 15.426528 (ms)  
images are identical
```

➤ **Instruções de Compilação e Execução:**

Para compilar:

```
nvcc -arch=sm_13 -O -Icommon/inc canny_ex2.cu -Llib -lcutil_x86_64 -o canny_ex2
```

Para executar:

```
./canny_ex2 -i imagem.pgm -o imagemOut.pgm
```

Non_maximum_supression function (texture memory)

➤ Arquitetura da Solução:

Mantivemos a mesma arquitetura, já que o muda é apenas a forma como lê alguns dos dados, que em vez de os ir buscar á memória global da placa gráfica, vai á memória de textura que foi alocada e obtém de lá os dados.

➤ Estruturas e Algoritmos:

Declaração da referencia de textura:

```
texture<pixel_t,1,cudaReadModeElementType> C_texture;
```

Fazer a ligação entre dados na placa gráfica da memória global para a memória de textura:

```
cudaBindTexture(0,C_texture,C,memsize);
```

Ler da memória de textura:

```
C_texture_fetched[0] = tex1Dfetch(C_texture,nn);
```

...

Esta variável, é carregada com 8 posições a toda a volta do pixel em que está num dado momento da execução, que está alocado na memória de textura, para mais tarde ser utilizado em condições no fluxo do kernel.

➤ **Resultados Obtidos:**

Estes resultados foram obtidos com as seguintes funções em CUDA:

- CUDA_convolution_kernel
- CUDA_non_maximum_supression_kernel

```
aca0310@nikola:~/Projeto2/aca_canny$ sh fuuu.sh ex2/canny_ex2_texture

Image: lena.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 177.815720 (ms)
Device processing time: 118.640099 (ms)
images are identical

Image: house.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 51.537247 (ms)
Device processing time: 38.852673 (ms)
images are identical

Image: chessRotatel.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 20.016928 (ms)
Device processing time: 15.370144 (ms)
images are identical
```

➤ **Instruções de Compilação e Execução:**

Para compilar:

```
nvcc -arch=sm_13 -O -Icommon/inc canny_ex2_texture.cu -Llib -lcutil_x86_64 -o canny_ex2_texture
```

Para executar:

```
./canny_ex2_texture -i imagem.pgm -o imagemOut.pgm
```

Tarefa 3

First_edges function (global memory)

➤ **Arquitetura da Solução:**

Começámos novamente por alocar e copiar memória para a placa gráfica e fazer o MemSet do array correspondente ao resultado da operação a 0.

As dimensões da grid e dos blocos são atribuídos de acordo com a mesma fórmula da função da alínea anterior em memória global.

O output da função é copiado da placa gráfica para o array reference.

➤ **Estruturas e Algoritmos:**

Calculámos o id da thread correspondente ao pixel a processar da mesma maneira em todos os exercícios de memória global.

Se o pixel processado pela thread não for um pixel dos extremos da imagem e o pixel seguinte tiver um valor superior ao tmax, então a esse pixel é atribuído o valor máximo possível (MAX_BRIGHTNESS).

➤ **Resultados Obtidos:**

Estes resultados foram obtidos com as seguintes funções em CUDA:

- CUDA_convolution_kernel
- CUDA_non_maximum_supression_kernel
- CUDA_first_edges_kernel

```
aca0310@nikola:~/Projeto2/aca_canny$ sh fuuu.sh ex3/canny_ex3  
  
Image: lena.pgm  
gaussian_filter: kernel size 7, sigma=1  
gaussian_filter: kernel size 7, sigma=1  
Host processing time: 178.282883 (ms)  
Device processing time: 121.898048 (ms)  
images are identical  
  
Image: house.pgm  
gaussian_filter: kernel size 7, sigma=1  
gaussian_filter: kernel size 7, sigma=1  
Host processing time: 51.636032 (ms)  
Device processing time: 39.359550 (ms)  
images are identical  
  
Image: chessRotatel.pgm  
gaussian_filter: kernel size 7, sigma=1  
gaussian_filter: kernel size 7, sigma=1  
Host processing time: 20.016577 (ms)  
Device processing time: 15.804256 (ms)  
images are identical
```

➤ **Instruções de Compilação e Execução:**

Para compilar:

```
nvcc -arch=sm_13 -O -Icommon/inc canny_ex3.cu -Llib -lcutil_x86_64 -o canny_ex3
```

Para executar:

```
./canny_ex3 -i imagem.pgm -o imagemOut.pgm
```


First_edges function (texture memory)

➤ Arquitetura da Solução:

Utiliza exatamente o mesmo procedimento descrito para a memória global, mas neste caso faz uso de dados que estão alocados na memória de textura.

➤ Estruturas e Algoritmos:

Declaração da referencia de textura:

```
texture<pixel_t,1,cudaReadModeElementType> nms_texture;
```

Fazer a ligação entre dados na placa gráfica da memória global para a memória de textura:

```
cudaBindTexture(0, nms_texture, A, memsize);
```

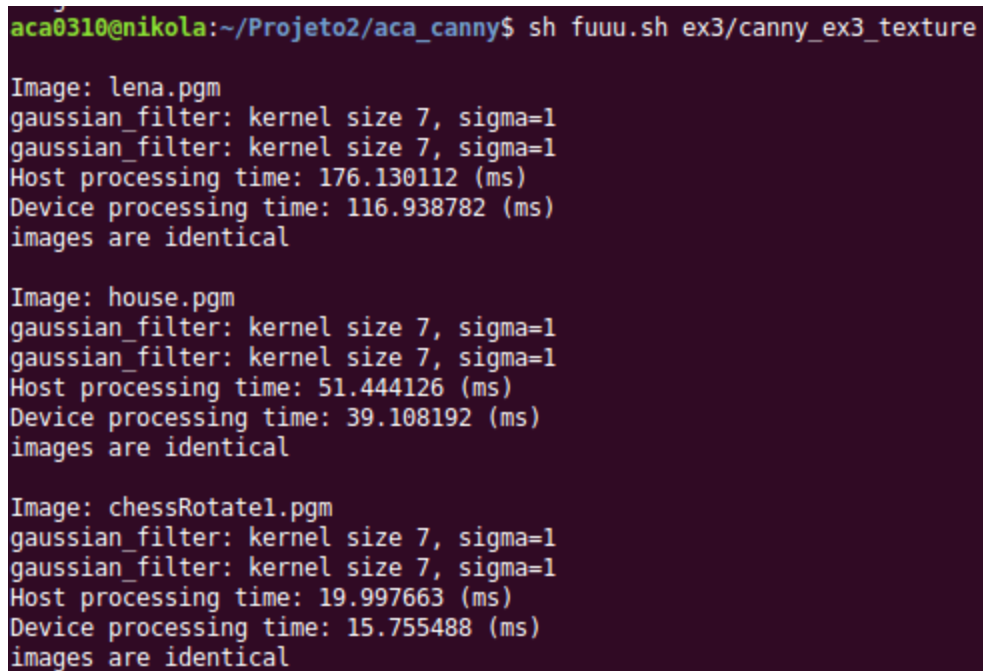
Ler da memória de textura:

```
pixel_t nms_texture_fetched = tex1Dfetch(nms_texture,id+1);
```

➤ Resultados Obtidos:

Estes resultados foram obtidos com as seguintes funções em CUDA:

- CUDA_convolution_kernel
- CUDA_non_maximum_supression_kernel
- CUDA_first_edges_kernel



```
aca0310@nikola:~/Projeto2/aca_canny$ sh fuuu.sh ex3/canny_ex3_texture

Image: lena.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 176.130112 (ms)
Device processing time: 116.938782 (ms)
images are identical

Image: house.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 51.444126 (ms)
Device processing time: 39.108192 (ms)
images are identical

Image: chessRotat1.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 19.997663 (ms)
Device processing time: 15.755488 (ms)
images are identical
```

➤ Instruções de Compilação e Execução:

Para compilar:

```
nvcc -arch=sm_13 -O -Icommon/inc canny_ex3_texture.cu -Llib -lcutil_x86_64 -o  
canny_ex3_texture
```

Para executar:

```
./canny_ex3_texture -i imagem.pgm -o imagemOut.pgm
```

First_edges function (shared memory)

Esta função não foi implementada, dado que não compensa de todo, fazer uso da memória partilhada (shared memory).

Como apenas executa uma vez por bloco, se fizéssemos a passagem dos dados da memória global para a memória partilhada e depois no fim, da memória partilhada para a memória global, seria ainda maior, o tempo necessário para a execução da função, do que usar apenas a memória global.

Tarefa 4

Hysteresis_edges function (global memory)

➤ **Arquitetura da Solução:**

Começámos por alocar e copiar blocos de memória para a placa gráfica com os arrays originários da função non-maximum supression, first_edges e ainda o valor booleano pc.

O dimensionamento da grid e dos blocos é feito da mesma maneira das funções anteriores.

Depois de executado o processamento é copiado o valor resultante para a variável reference que é o resultado da função. Também é copiado o valor para o changed de modo a transparecer quando houve alguma alteração.

➤ **Estruturas e Algoritmos:**

É calculado o id da thread correspondente ao pixel a ser processado da mesma maneira que foi nas funções de memória global anteriores.

Se não for um pixel dos extremos são calculados os índices dos seus pixeis vizinhos e colocados num array.

Depois desta operação o algoritmo é semelhante ao do implementado no host com a exceção de que o índice do pixel é dado pelo id da thread e cada thread se encarrega de processar um pixel.

➤ **Resultados Obtidos:**

Estes resultados foram obtidos com as seguintes funções em CUDA:

- CUDA_convolution_kernel
- CUDA_non_maximum_supression_kernel
- CUDA_first_edges_kernel
- CUDA_hysteresis_edges_kernel

```
aca0310@nikola:~/Projeto2/aca_canny$ sh fuuu.sh ex4/canny_ex4

Image: lena.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 177.465927 (ms)
Device processing time: 116.355103 (ms)
images are identical

Image: house.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 61.901760 (ms)
Device processing time: 46.452255 (ms)
images are identical

Image: chessRotatet1.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 24.022560 (ms)
Device processing time: 16.802273 (ms)
images are identical
```

➤ **Instruções de Compilação e Execução:**

Para compilar:

```
nvcc -arch=sm_13 -O -Icommon/inc canny_ex4.cu -Llib -lcutil_x86_64 -o canny_ex4
```

Para executar:

```
./canny_ex4 -i imagem.pgm -o imagemOut.pgm
```

Hysteresis_edges function (texture memory)

➤ **Arquitetura da Solução:**

Igual á implementação com apenas a memória global, mas os valores dos pixeis que vai ler estão todos alocados na memória de textura.

➤ **Estruturas e Algoritmos:**

Declaração da referencia de textura:

```
texture<pixel_t,1,cudaReadModeElementType> nbs_texture;
```

Fazer a ligação entre dados na placa gráfica da memória global para a memória de textura:

```
cudaBindTexture(0,nbs_texture,out,memsize);
```

Ler da memória de textura:

```
tex1Dfetch(nbs_texture,id)
```

```
tex1Dfetch(nbs_texture,nbs[k])
```

A variável *nbs* é um array com os vizinhos em redor.

➤ Resultados Obtidos:

Estes resultados foram obtidos com as seguintes funções em CUDA:

- CUDA_convolution_kernel
- CUDA_non_maximum_supression_kernel
- CUDA_first_edges_kernel
- CUDA_hysteresis_edges_kernel

```
aca0310@nikola:~/Projeto2/aca_canny$ sh fuuu.sh ex4/canny_ex4_texture

Image: lena.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 178.159576 (ms)
Device processing time: 123.821182 (ms)
images are identical

Image: house.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 51.544254 (ms)
Device processing time: 41.678368 (ms)
images are identical

Image: chessRotatel.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 20.018784 (ms)
Device processing time: 16.804319 (ms)
images are identical
```

➤ Instruções de Compilação e Execução:

Para compilar:

```
nvcc -arch=sm_13 -O -Icommon/inc canny_ex4_texture.cu -Llib -lcutil_x86_64 -o canny_ex4_texture
```

Para executar:

```
./canny_ex4_texture -i imagem.pgm -o imagemOut.pgm
```

Hysteresis_edges function (shared memory)

➤ Estruturas e Algoritmos:

A estrutura é em tudo semelhante à implementação da convolução em shared memory: o tamanho da matriz da shared memory é o mesmo, é calculado o id e o ids e a matriz de memória partilhada é preenchida. O casos especiais dos extremos dos blocos mantêm-se exatamente iguais.

Se o pixel que está a ser processado não for um pixel do extremo da imagem, é preenchido um array com os seus vizinhos.

O algoritmo de processamento é igual ao da função implementada com memória global com a diferença de que onde dantes se acedia ao reference para ler, passa a ser ao array de memória partilhada.

➤ Resultados Obtidos:

```
Image: lena.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 239.479813 (ms)
Device processing time: 166.822983 (ms)
0 diffs of 1
0 diffs of 2
0 diffs of 3
0 diffs of 4
0 diffs of 5
370 diffs > 5

Image: house.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 51.516800 (ms)
Device processing time: 49.501793 (ms)
0 diffs of 1
0 diffs of 2
0 diffs of 3
0 diffs of 4
0 diffs of 5
85 diffs > 5

Image: chessRotatel.pgm
gaussian_filter: kernel size 7, sigma=1
gaussian_filter: kernel size 7, sigma=1
Host processing time: 20.010656 (ms)
Device processing time: 20.476065 (ms)
images are identical
```

Estes resultados foram obtidos usando todas as funções do host à exceção da Hysteresis_edges function em que foi usada a sua versão com memória partilhada.

A função não está a funcionar totalmente, havendo pequenas diferenças indetetáveis a olho humano entre a imagem gerada e a imagem de referência. Não conseguimos descobrir a causa destas diferenças.

➤ **Instruções de Compilação e Execução:**

Para compilar:

```
nvcc -arch=sm_13 -O -Icommon/inc canny_ex4_shared.cu -Llib -lcutil_x86_64 -o  
canny_ex4_shared
```

Para executar:

```
./canny_ex4_shared -i imagem.pgm -o imagemOut.pgm
```

Conclusão

Para uma implementação simples, chegamos á conclusão de que usar a memória global da placa gráfica, melhora bastante os tempos de processamento, comparativamente ao que o processador necessita para realizar o mesmo trabalho, de forma relativamente simples.

Mas em contra partida, a memória global não é de todo a mais eficaz nem a indicada para todo o tipo de situações. Portanto, implementamos dois outros tipos de memórias disponibilizadas, a memória partilhada e a memória de textura.

A memória partilhada como está localizada na placa gráfica e condicionada a individualmente a cada bloco, permitiu-nos melhorar muito a velocidade de processamento das funções, apesar de não termos conseguido implementar desta forma 100% corretamente.

A memória de textura sendo apenas de leitura e também localizada na placa gráfica, mas esta, já sendo acessível por todos os blocos da grid, causa um melhoramento da performance ao permitir alocar blocos de dados na memória de textura para que diferentes blocos na grid acedam aos mesmos dados, sem ser necessário esperar por sincronizações.

Bibliografia

- <http://docs.nvidia.com/cuda/cuda-driver-api/index.html>
- <http://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/>
- http://www.math.ntu.edu.tw/~wwang/mtxcomp2010/download/cuda_04_ykhung.pdf
- <http://cuda-programming.blogspot.pt/2013/02/texture-memory-in-cuda-what-is-texture.html>
- http://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/html/group__CUDART__MEMORY.html