Universidade de Aveiro

departamento de electrónica, telecomunicações e informática

47022- Arquitectura de Computadores Avançada

Home group assignment 2

Semi-Global Matching stereo processing using CUDA

8240 - mestrado integrado em engenharia de computadores e telemática

António Rafael da Costa Ferreira
NMec: 67405

Rodrigo Lopes da Cunha
NMec: 67800

Docentes: Nuno Lau e José Luís Azevedo

Janeiro de 2016
2015-2016

Conteúdos

1 Introdução

O trabalho proposto para a unidade curricular de Arquitetura de Computadores Avançada foi a implementação em CUDA para o processamento de um Semi-Global Matching.

Este programa tem como objetivo determinar a imagem de disparidade entre duas imagens idênticas mas de posições diferentes, como se de dois olhos se tratasse, uma vista com o olho da esquerda e outra com o olho da direita.

O relatório reflete todas as geometrias de kernel implementadas, formas de pensamento, métodos de como foram implementados os algoritmos, resultados, tutorial para correr o código elaborado, e por último a conclusão deste mesmo trabalho.

2 Exercício 1

2.1 Cuda Kernel da função "determine_costs()"

Neste primeiro exercício, era pedido que se desenvolvesse um kernel em CUDA que substituísse a função *determine_costs()*. Este exercicío foi ainda realizado de duas maneira, uma utilizando a *global memory*, e outra onde se coloca as imagens e o valor de COSTS na *texture memory*.

Versão 1 - Global Memory

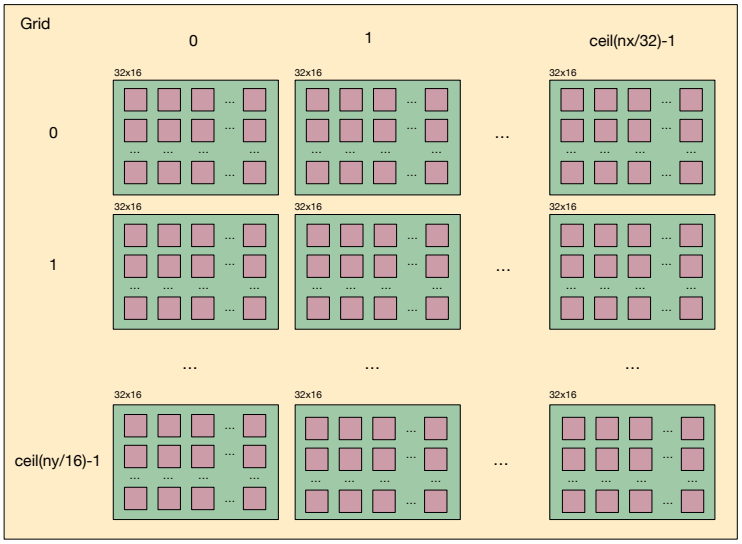

Figura 1:
Geometria do Kernel para a função determine_costs()

Nesta versão do kernel optou-se por uma geometria (Figura 1 constituída por uma grid de tamanho (ceil(nx32) x ceil(ny16)) com blocos de 32 x 16 threads cada. Neste kernel, cada thread corresponde a um pixel da imagem, e cada um calcula o valor de custo, sendo este a diferença entre as imagens num determinado pixel.

Para a *global memory* utilizou-se o seguinte algoritmo para desenvolver o kernel:

```
__global__ void determine_costs_device(const int *left_image, const int *right_image,
 int *costs,
const int nx, const int ny, const int disp_range)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  if (i < nx && j < ny)
```

```
  {
    for ( int d = 0; d < disp_range; d++ ) {
      if(i >= d){
        COSTS(i,j,d) = abs( LEFT_IMAGE(i,j) - RIGHT_IMAGE(i-d,j));
      }
    }
  }
}
```

Com esta implementação obtiveram-se os seguintes resultados:

```
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm$ ./sgm
Host processing time: 5160.187500 (ms)
Device processing time: 5048.691406 (ms)
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm$ !./te
./testDiffs h_dbull.pgm d_dbull.pgm
images are identical
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm$ ./sgm -p 64
Host processing time: 19739.484375 (ms)
Device processing time: 19562.093750 (ms)
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm$ !./te
./testDiffs h_dbull.pgm d_dbull.pgm
images are identical
```

Figura 2:
Resultados obtidos utilizando global memory - versão 1

Versão 2 - Global Memory

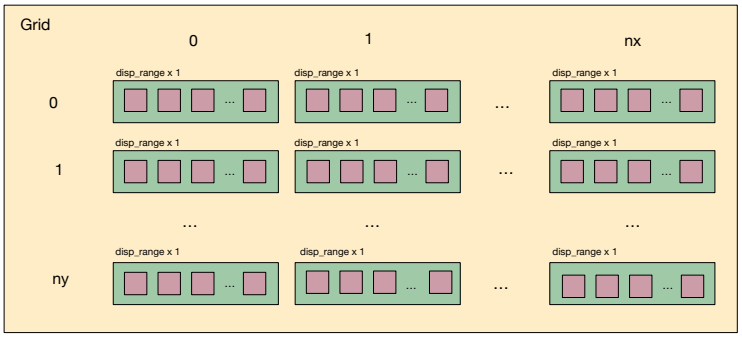

Figura 3:
Geometria do Kernel para a função determine_costs()

Na segunda versão deste kernel optou-se por uma geometria (Figura 3 constituída por uma grid de tamanho nx x ny com blocos de disp_range x 1 threads cada. Neste kernel, cada thread corresponde a um valor de disparidade diferente e cada bloco corresponde a um pixel da imagem.

Os resultados desta nova implementação foram:

```
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm$ ./sgm
Host processing time: 5057.642578 (ms)
Device processing time: 5044.949219 (ms)
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm$ !./te
./testDiffs h_dbull.pgm d_dbull.pgm
images are identical
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm$ ./sgm -p 64
Host processing time: 19576.736328 (ms)
Device processing time: 19501.994141 (ms)
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm$ !./te
./testDiffs h_dbull.pgm d_dbull.pgm
images are identical
```

Figura 4:
Resultados obtidos utilizando global memory - versão 2

Como os resultados das duas versões não diferem muito, e há casos em que um é melhor e outros em que é pior, então, nos próximos exercícios que utilizam este kernel em memória global, será utilizada a versão 1.

Texture Memory

Neste exercício foi ainda possível a utilização de *texture memory*, pois seria interessante colocar as imagens em cache, de forma a que o acesso a elas fosse mais rápido, notaram-se algumas melhorias, mas não como se estava à espera.

Para que isto fosse possível foi necessário introduzir algumas configurações novas:

```
texture<int, cudaTextureType2D, cudaReadModeElementType> devTex_leftImage;
texture<int, cudaTextureType2D, cudaReadModeElementType> devTex_rightImage;


__global__ void determine_costs_device(int *costs,
                                       const int nx, const int ny, const int disp_range)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  if (i < nx && j < ny)
  {
    for ( int d = 0; d < disp_range; d++ ) {
      if(i >= d){
        COSTS(i,j,d) = abs( tex2D(devTex_leftImage, i, j) - tex2D(devTex_rightImage, i-d, j));
      }
    }
  }
}

void sgmDevice( const int *h_leftIm, const int *h_rightIm,
                int *h_dispImD,
                const int w, const int h, const int disp_range )
{
...
  cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<int>();
```

```
    cudaArray* cuArrayLeftImage;
    cudaArray* cuArrayRightImage;
    cudaMallocArray(&cuArrayLeftImage, &channelDesc, nx, ny);
    cudaMallocArray(&cuArrayRightImage, &channelDesc, nx, ny);
    cudaMemcpyToArray(cuArrayLeftImage, 0, 0, h_leftIm, imageSize, cudaMemcpyHostToDevice);
    cudaMemcpyToArray(cuArrayRightImage, 0, 0, h_rightIm, imageSize, cudaMemcpyHostToDevice);

    devTex_leftImage.addressMode[0] = cudaAddressModeClamp;
    devTex_leftImage.addressMode[1] = cudaAddressModeClamp;
    devTex_leftImage.filterMode     = cudaFilterModePoint;
    devTex_leftImage.normalized     = false;
    devTex_rightImage.addressMode[0] = cudaAddressModeClamp;
    devTex_rightImage.addressMode[1] = cudaAddressModeClamp;
    devTex_rightImage.filterMode     = cudaFilterModePoint;
    devTex_rightImage.normalized     = false;

    cudaBindTextureToArray(devTex_leftImage, cuArrayLeftImage, channelDesc);
    cudaBindTextureToArray(devTex_rightImage, cuArrayRightImage, channelDesc);
...
}
```

Os resultados que se obtiveram foram os seguintes:

```
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm/ex1_p2_67405_67800_texture$ ./sgm
Host processing time: 5108.825195 (ms)
Device processing time: 5037.208984 (ms)
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm/ex1_p2_67405_67800_texture$ !./te
./testDiffs h_dbull.pgm d_dbull.pgm
images are identical
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm/ex1_p2_67405_67800_texture$ ./sgm -p 64
Host processing time: 19654.699219 (ms)
Device processing time: 19552.791016 (ms)
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm/ex1_p2_67405_67800_texture$ !./te
./testDiffs h_dbull.pgm d_dbull.pgm
images are identical
```

Figura 5:
Resultados obtidos utilizando texture memory

3 Exercício 2

3.1 Cuda Kernel(s) da função "iterate_direction_dirxpos_dev()" e das funções correspondestes a outras direcções

Para este exercicío foram implementadas duas versões para a utilização de *global memory*, sendo a versão 2 (otimizada) utilizada na utilização da *shared memory*.

Global Memory - Versão 1

Nesta versão, foram criadas duas geometrias apenas, sendo que uma diz respeito às iterações nas direções em x, e outra em y, visto que tanto para o lado positivo como para o negativo a geometria era idêntica.

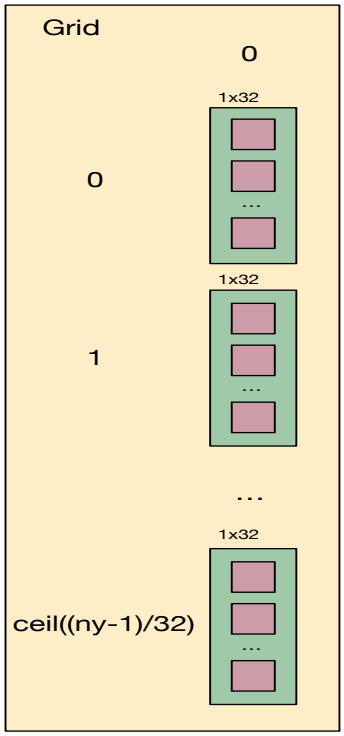

Figura 6:
Geometria do Kernel para as funções iterate_direction_dirxpos() e iterate_direction_dirxneg()

Como podemos ver na figura 6, a grid é composta por ceil(ny/32) blocos, cada bloco composto por 32 threads, sendo cada uma responsável pela linha em x onde está inserida para cálculo dos respetivos paths.

Esta operação tem de ser efetuada sequencialmente pois o pixel seguinte depende sempre do anterior, pelo que se recorreu à seguinte implementação para o kernel *iterate_direction_dirxpos()* e para o kernel *iterate_direction_dirxneg()*:

```
__global__ void iterate_direction_dirxpos_dev(const int dirx, const int *left_image,
                        const int* costs, int *accumulated_costs,
                        const int nx, const int ny, const int disp_range ){

      int i = 0;
      int j = blockIdx.y * blockDim.y + threadIdx.y;
      if(j < ny){

        for ( int d = 0; d < disp_range; d++ ) {
          ACCUMULATED_COSTS(0,j,d) += COSTS(0,j,d);
        }

        for(i = 1; i<nx; i++){
          evaluate_path_dev( &ACCUMULATED_COSTS(i-dirx,j,0),
                             &COSTS(i,j,0),
                             abs(LEFT_IMAGE(i,j)-LEFT_IMAGE(i-dirx,j)) ,
                             &ACCUMULATED_COSTS(i,j,0), nx, ny, disp_range);
        }
      }
}

__global__ void iterate_direction_dirxneg_dev(const int dirx, const int *left_image,
                        const int* costs, int *accumulated_costs,
                        const int nx, const int ny, const int disp_range )
{
      int i = nx-1;
      int j = blockIdx.y * blockDim.y + threadIdx.y;

      if(j < ny){

        for ( int d = 0; d < disp_range; d++ ) {
            ACCUMULATED_COSTS(nx-1,j,d) += COSTS(nx-1,j,d);
        }

        for(i = nx-2; i >= 0; i--){
            evaluate_path_dev( &ACCUMULATED_COSTS(i-dirx,j,0),
                             &COSTS(i,j,0),
                             abs(LEFT_IMAGE(i,j)-LEFT_IMAGE(i-dirx,j)),
                             &ACCUMULATED_COSTS(i,j,0), nx, ny, disp_range );
        }
      }
}
```

No caso da direção ser em y, então seguiu-se o mesmo pensamento que em x, obtendo a seguinte geometria:

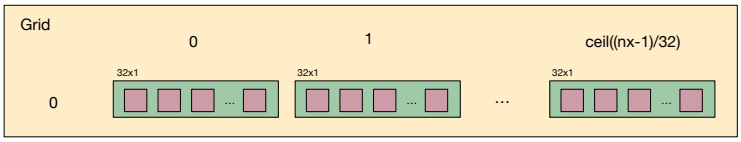

Figura 7:
Geometria do Kernel para as funções iterate_direction_dirypos() e iterate_direction_diryneg()

Tal como apresentado na figura, neste caso a geometria é composta por uma grid de tamanho ceil(ny/32) blocos, cada um composto por 32 threads, onde cada uma volta a ser responsável pelo cálculo do respetivo caminho de todos os pixeis daquela coluna.

Esta geometria volta a aplicar-se às direções positivas e negativa da mesma maneira tal como em x.

Foi então desenvolvido o seguinte código para os kernels *iterate_direction_dirypos()* e *iterate_direction_diryneg()*:

```
__global__ void iterate_direction_dirypos_dev(const int diry, const int *left_image,
                        const int* costs, int *accumulated_costs,
                        const int nx, const int ny, const int disp_range )
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = 0;
    if(i < nx){
        for ( int d = 0; d < disp_range; d++ ) {
            ACCUMULATED_COSTS(i,0,d) += COSTS(i,0,d);
        }
        for(j = 1; j<ny; j++){

          evaluate_path_dev( &ACCUMULATED_COSTS(i,j-diry,0),
                        &COSTS(i,j,0),
                        abs(LEFT_IMAGE(i,j)-LEFT_IMAGE(i,j-diry)),
                        &ACCUMULATED_COSTS(i,j,0), nx, ny, disp_range );
      }
    }
}

__global__ void iterate_direction_diryneg_dev(const int diry, const int *left_image,
                        const int* costs, int *accumulated_costs,
                        const int nx, const int ny, const int disp_range )
{

      int i = blockIdx.x * blockDim.x + threadIdx.x;
      int j = ny-1;
      if(i < nx){
        for ( int d = 0; d < disp_range; d++ ) {
            ACCUMULATED_COSTS(i,ny-1,d) += COSTS(i,ny-1,d);
        }
```

```
        for(j = ny-2; j >= 0; j--){

            evaluate_path_dev( &ACCUMULATED_COSTS(i,j-diry,0),
                               &COSTS(i,j,0),
                               abs(LEFT_IMAGE(i,j)-LEFT_IMAGE(i,j-diry)),
                               &ACCUMULATED_COSTS(i,j,0) , nx, ny, disp_range);
          }
       }
}
```

Nesta primeira versão, os resultados obtidos foram os seguintes:

```
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm/ex2_p2_67405_67800_global$ ./sgm
Host processing time: 5044.548340 (ms)
Device processing time: 4754.518066 (ms)
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm/ex2_p2_67405_67800_global$ !./te
./testDiffs h_dbull.pgm d_dbull.pgm
images are identical
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm/ex2_p2_67405_67800_global$ ./sgm -p 64
Host processing time: 19286.228516 (ms)
Device processing time: 17583.138672 (ms)
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm/ex2_p2_67405_67800_global$ !./te
./testDiffs h_dbull.pgm d_dbull.pgm
images are identical
```

Figura 8:
Resultados obtidos utilizando a versão 1 com global memory

Notaram-se algumas melhorias, contudo é possível melhorar o speedup, e para isso recorreu-se a uma segunda versão, desenvolvida com o apoio da leitura do artigo [1] fornecido pelos professores.

[1] Real-time Stereo Vision: Optimizing Semi-Global Matching, Matthias Michael, Jan Salmen, Johannes Stallkamp, and Marc Schlipsing, IEEE Intelligent Vehicles Symposium pp 1197-1202, 2013

Global Memory - Versão 2

Nesta segunda versão, decidiu-se alterar a geometria do kernel, de forma a que agora cada thread fosse responsável por um único valor de disparidade num path. Para isso a geometria criada para x foi a seguinte:

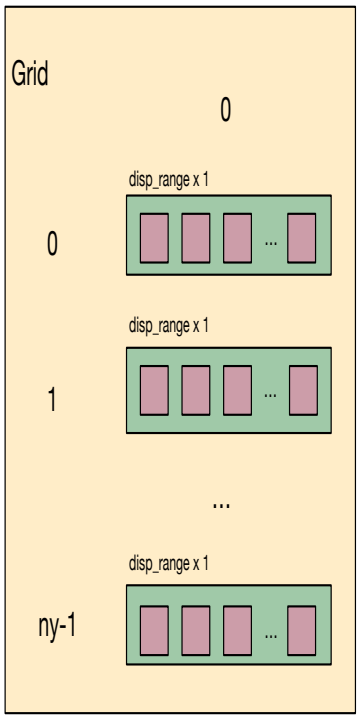

Figura 9:
Geometria do Kernel para as funções iterate_direction_dirxpos() e iterate_direction_dirxneg()

A grid passa a ser composta por ny blocos, cada um com um número de threads igual ao *disparity range*. Passa então a existir um bloco para cada linha em x, composto por threads, onde cada uma corresponde a um valor de disparidade diferente.

A implementação destes dois kernels foi efetuada através do seguinte algoritmo:

```
__global__ void iterate_direction_dirxpos_dev(const int dirx, const int *left_image,
                        const int* costs, int *accumulated_costs,
                        const int nx, const int ny, const int disp_range ){

    int i = threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if(i < disp_range && j<ny){
      ACCUMULATED_COSTS(0,j,i) += COSTS(0,j,i);
```

```
        __syncthreads();


        for(int l = 1; l<nx;l++){
          evaluate_path_dev( &ACCUMULATED_COSTS(l-dirx,j,0),
                             &COSTS(l,j,0),
                             abs(LEFT_IMAGE(l,j)-LEFT_IMAGE(l-dirx,j)) ,
                             &ACCUMULATED_COSTS(l,j,0), nx, ny, disp_range, i);
          __syncthreads();

        }
      }
}

__global__ void iterate_direction_dirxneg_dev(const int dirx, const int *left_image,
                          const int* costs, int *accumulated_costs,
                          const int nx, const int ny, const int disp_range )
{
        int i = threadIdx.x;
        int j = blockIdx.y * blockDim.y + threadIdx.y;

        if(i < disp_range && j < ny){

          ACCUMULATED_COSTS(nx-1,j,i) += COSTS(nx-1,j,i);

          __syncthreads();


          for(int l = nx-2; l >= 0; l--){
              evaluate_path_dev( &ACCUMULATED_COSTS(l-dirx,j,0),
                                 &COSTS(l,j,0),
                                 abs(LEFT_IMAGE(l,j)-LEFT_IMAGE(l-dirx,j)),
                                 &ACCUMULATED_COSTS(l,j,0), nx, ny, disp_range, i);
              __syncthreads();
          }
        }
}
```

No caso da direção ser em y, a geometria utilizada foi a seguinte:

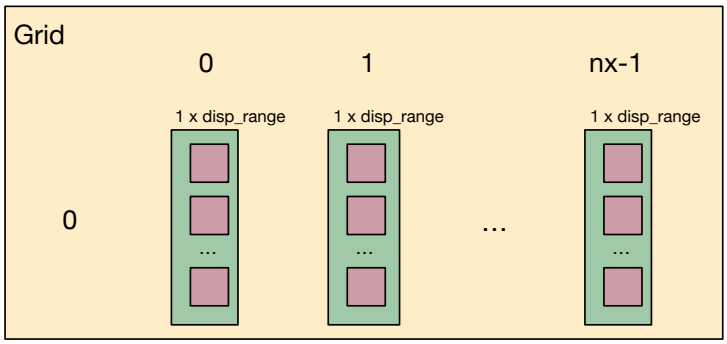

Figura 10:
Geometria do Kernel para as funções iterate_direction_dirypos() e iterate_direction_diryneg()

Nesta situação, a grid é composta por nx blocos, cada um um número de threads igual ao disparity range, onde cada thread, tal como em x, é responsável por um valor de disparidade diferente.

A implementação dos kernels correspondestes a esta geometria é a seguinte:

```
__global__ void iterate_direction_dirypos_dev(const int diry, const int *left_image,
                        const int* costs, int *accumulated_costs,
                        const int nx, const int ny, const int disp_range )
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = threadIdx.y;
    if(j < disp_range && i < nx){

        ACCUMULATED_COSTS(i,0,j) += COSTS(i,0,j);
        __syncthreads();

        for(int l = 1; l<ny; l++){

          evaluate_path_dev( &ACCUMULATED_COSTS(i,l-diry,0),
                             &COSTS(i,l,0),
                             abs(LEFT_IMAGE(i,l)-LEFT_IMAGE(i,l-diry)),
                             &ACCUMULATED_COSTS(i,l,0), nx, ny, disp_range, j);
          __syncthreads();

      }
    }
}

__global__ void iterate_direction_diryneg_dev(const int diry, const int *left_image,
                        const int* costs, int *accumulated_costs,
                        const int nx, const int ny, const int disp_range )
{

      int i = blockIdx.x * blockDim.x + threadIdx.x;
      int j = threadIdx.y;
      if(j < disp_range && i < nx){
```

```
        ACCUMULATED_COSTS(i,ny-1,j) += COSTS(i,ny-1,j);
        __syncthreads();


        for(int l = ny-2; l >= 0; l--){

            evaluate_path_dev( &ACCUMULATED_COSTS(i,l-diry,0),
                               &COSTS(i,l,0),
                               abs(LEFT_IMAGE(i,l)-LEFT_IMAGE(i,l-diry)),
                               &ACCUMULATED_COSTS(i,l,0) , nx, ny, disp_range, j);
            __syncthreads();

         }
      }
}
```

Para que a implementação desta segunda versão funcionasse foi necessário recorrer ao comando *__syncthreads()*, de forma a que todas as threads esperassem umas pelas outras quando chegavam ao ponto onde este comando se encontra colocado, garantindo assim que tudo era feito sequencialmente, e posteriormente utilizado de maneira correta quando se recorresse à *shared memory*. Foi ainda necessário efetuar alterações no código da função *evaluate_path()* para que agora dentro deste apenas calculasse o valor necessário para aquele valor de disparidade, ficando assim:

```
__device__ void evaluate_path_dev(const int *prior, const int *local,
                    int path_intensity_gradient, int *curr_cost ,
                    const int nx, const int ny, const int disp_range, const int d)
  {
    memcpy(&curr_cost[d], &local[d], sizeof(int));

    int e_smooth = NPP_MAX_16U;

    for ( int d_p = 0; d_p < disp_range; d_p++ ) {
      if ( d_p - d == 0 ) {
        // No penality
        e_smooth = MMIN(e_smooth,prior[d_p]);

      } else if ( abs(d_p - d) == 1 ) {
        // Small penality
        e_smooth = MMIN(e_smooth,prior[d_p]+PENALTY1);

      } else {
        // Large penality
        e_smooth =
          MMIN(e_smooth,prior[d_p] +
                    MMAX(PENALTY1,
                     path_intensity_gradient ? PENALTY2/path_intensity_gradient : PENALTY2));
      }
    }

    curr_cost[d] += e_smooth;

    int min = NPP_MAX_16U;

    for ( int d_s = 0; d_s < disp_range; d_s++ ) {
      if (prior[d_s]<min) min=prior[d_s];
    }
```

```
    curr_cost[d]-=min;
}
```

Com esta nova implementação, a melhoria no speedup foi brutal, melhorando bastante os resultados:

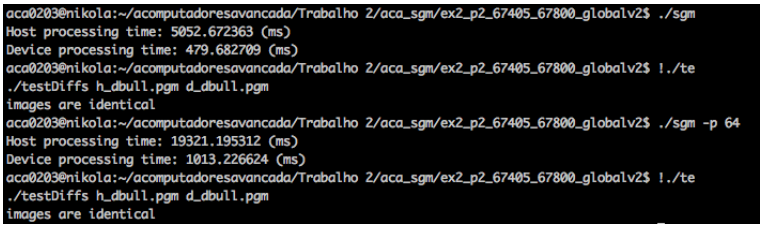

Figura 11:
Resultados obtidos utilizando a versão 2 com global memory

Para melhorar ainda mais estes resultados passou-se a utilizar a *shared memory* para o cálculo do path.

Esta versão 2 será a utilizada nos exercicíos seguintes quando se utilizar apenas memória global.

Shared Memory

Visto que as threads na versão 2 eram executadas em paralelo, e que existiam valores que todas partilhavam e necessitavam umas das outras para o cálculo dos mínimos, então decidiu-se que seria mais produtivo que as pesquisas fossem feitas na *shared memory*. Para isso, foi necessário criar um array de *shared memory* com shared_memory_size igual a disparity_range*sizeof(int), onde cada índice do array corresponde a um valor de disparidade. Agora em vez de a pesquisa ser efetuada no array prior, passou a ser efetuada no shmem, como mostrado no código seguinte.

A maioria das alterações foi feita na função *evaluate_path()*, visto ser nesta que se efetuam todas as pesquisas necessárias para determinar o current cost e o minimo. Com isto a função ficou da seguinte forma:

```
__device__ void evaluate_path_dev(const int *prior, const int *local,
                    int path_intensity_gradient, int *curr_cost ,
                    const int nx, const int ny, const int disp_range, const int d, int shmem[])
  {
    memcpy(&curr_cost[d], &local[d], sizeof(int));

    int e_smooth = NPP_MAX_16U;

    for ( int d_p = 0; d_p < disp_range; d_p++ ) {
      if ( d_p - d == 0 ) {
        // No penality
        e_smooth = MMIN(e_smooth,shmem[d_p]);

      } else if ( abs(d_p - d) == 1 ) {
        // Small penality
        e_smooth = MMIN(e_smooth,shmem[d_p]+PENALTY1);

      } else {
        // Large penality
        e_smooth =
          MMIN(e_smooth,shmem[d_p] +
                    MMAX(PENALTY1,
                     path_intensity_gradient ? PENALTY2/path_intensity_gradient : PENALTY2));
      }
    }

    curr_cost[d] += e_smooth;

    int min = NPP_MAX_16U;

    for ( int d_s = 0; d_s < disp_range; d_s++ ) {
      if (shmem[d_s]<min) min=shmem[d_s];
    }

    curr_cost[d]-=min;

    __syncthreads();

    shmem[d] = curr_cost[d];

}
```

Aqui foi também necessário colocar outra vez o comando *__syncthreads()*,

para que todas as threads apenas escrevessem na memória partilhada quando todas tivessem calculado o mínimo valor nesta.

Mais uma vez com esta nova implementação, os resultados voltaram a melhorar muito em relação à memória global:

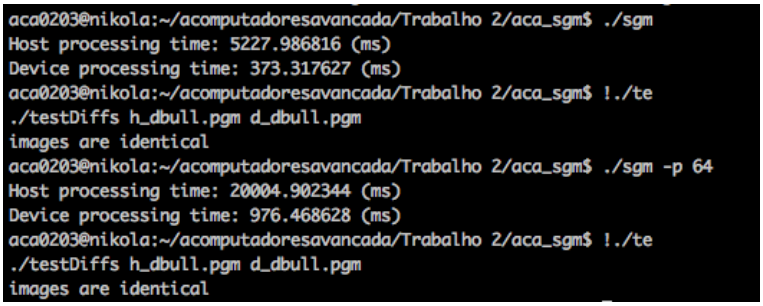

Figura 12:
Resultados obtidos utilizando shared memory

Texture Memory

Neste segundo exercício foi também possível recorrer às imagens que se encontravam em cache, pois em todas as funções das várias direções recorre-se às imagens, pelo que decidiu-se optar por recorrer às mesmas, através deste tipo de memória.

As alterações efetuadas foram as seguintes, sendo que as alterações efetuadas no exercício 1 se mantêm:

```
__global__ void iterate_direction_dirxpos_dev(const int dirx,
                        const int* costs, int *accumulated_costs,
                        const int nx, const int ny, const int disp_range ){

      int i = threadIdx.x;
      int j = blockIdx.y * blockDim.y + threadIdx.y;
      extern __shared__ int shmem[];

      if(i < disp_range && j<ny){
        ACCUMULATED_COSTS(0,j,i) += COSTS(0,j,i);
        shmem[i] = COSTS(0,j,i);
      __syncthreads();


      for(int l = 1; l<nx;l++){
        evaluate_path_dev( &ACCUMULATED_COSTS(l-dirx,j,0),
                           &COSTS(l,j,0),
                           abs(tex2D(devTex_leftImage, l, j)-tex2D(devTex_leftImage,l-dirx,j)) ,
                           &ACCUMULATED_COSTS(l,j,0), nx, ny, disp_range, i, shmem);

        __syncthreads();

      }
    }
}

__global__ void iterate_direction_dirypos_dev(const int diry,
                        const int* costs, int *accumulated_costs,
                        const int nx, const int ny, const int disp_range )
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = threadIdx.y;
    extern __shared__ int shmem[];

    if(j < disp_range && i < nx){
        shmem[j] = COSTS(i,0,j);
        ACCUMULATED_COSTS(i,0,j) += COSTS(i,0,j);
        __syncthreads();

        for(int l = 1; l<ny; l++){

          evaluate_path_dev( &ACCUMULATED_COSTS(i,l-diry,0),
                           &COSTS(i,l,0),
                           abs(tex2D(devTex_leftImage, i, l)-tex2D(devTex_leftImage,i,l-diry)) ,
                           &ACCUMULATED_COSTS(i,l,0), nx, ny, disp_range, j,shmem);
          __syncthreads();

      }
    }
}
```

```
__global__ void iterate_direction_dirxneg_dev(const int dirx,
                        const int* costs, int *accumulated_costs,
                        const int nx, const int ny, const int disp_range )
{
      int i = threadIdx.x;
      int j = blockIdx.y * blockDim.y + threadIdx.y;
      extern __shared__ int shmem[];

      if(i < disp_range && j < ny){
        shmem[i] = COSTS(nx-1,j,i);

        ACCUMULATED_COSTS(nx-1,j,i) += COSTS(nx-1,j,i);

        __syncthreads();


        for(int l = nx-2; l >= 0; l--){
            evaluate_path_dev( &ACCUMULATED_COSTS(l-dirx,j,0),
                        &COSTS(l,j,0),
                        abs(tex2D(devTex_leftImage, l, j)-tex2D(devTex_leftImage,l-dirx,j)) ,
                        &ACCUMULATED_COSTS(l,j,0), nx, ny, disp_range, i, shmem);
            __syncthreads();


        }
      }
}

__global__ void iterate_direction_diryneg_dev(const int diry,
                        const int* costs, int *accumulated_costs,
                        const int nx, const int ny, const int disp_range )
{

      int i = blockIdx.x * blockDim.x + threadIdx.x;
      int j = threadIdx.y;
      extern __shared__ int shmem[];

      if(j < disp_range && i < nx){
        shmem[j] = COSTS(i,ny-1,j);

        ACCUMULATED_COSTS(i,ny-1,j) += COSTS(i,ny-1,j);
        __syncthreads();


        for(int l = ny-2; l >= 0; l--){

            evaluate_path_dev( &ACCUMULATED_COSTS(i,l-diry,0),
                       &COSTS(i,l,0),
                       abs(tex2D(devTex_leftImage, i, l)-tex2D(devTex_leftImage,i,l-diry)) ,
                       &ACCUMULATED_COSTS(i,l,0) , nx, ny, disp_range, j, shmem);
            __syncthreads();

         }
      }
}
```

Os resultados obtidos foram os seguintes, que como podemos ver, não houve melhorias extraordinárias:

```
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm/ex2_p2_67405_67800_texture$ ./sgm
Host processing time: 5294.922363 (ms)
Device processing time: 372.182068 (ms)
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm/ex2_p2_67405_67800_texture$ !./te
./testDiffs h_dbull.pgm d_dbull.pgm
images are identical
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm/ex2_p2_67405_67800_texture$ ./sgm -p 64
Host processing time: 19992.228516 (ms)
Device processing time: 976.926697 (ms)
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm/ex2_p2_67405_67800_texture$ !./te
./testDiffs h_dbull.pgm d_dbull.pgm
images are identical
```

Figura 13:
Resultados obtidos utilizando texture memory

4 Exercício 3

4.1 Cuda Kernel da função "inplace_sum_views()"

A função *inplace_sum_views*, tem como objectivo a soma de pixeis de duas imagens. Para esta a geometria utilizada foi idêntica à da determinação de custos (Exercício 1), contudo possui uma pequena alteração, pois o número de colunas é agora correspondente a ceil((nx*disp_range)/32):

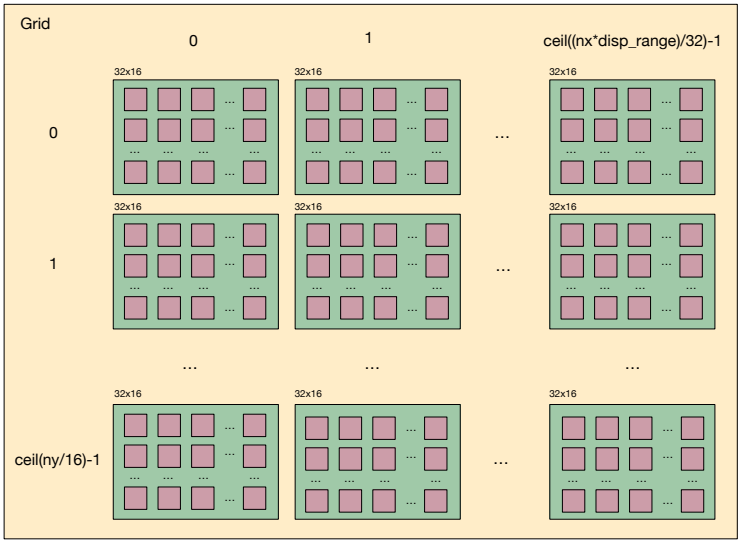

Figura 14:
Geometria do Kernel para a função inplace_sum_views()

Para a implementação deste kernel criou-se o seguinte algoritmo:

```
__global__ void inplace_sum_views_dev(int * im1, const int * im2,
                                       const int nx, const int ny, const int disp_range){

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
```

```
        int id = i + (j * (nx*disp_range));

        if(i < nx*disp_range && j < ny){
          int *im1_init = im1;
          im1 += id;
          im2 += id;
          if(im1 != (im1_init + (nx*ny*disp_range))  ){
            *im1 += *im2;
          }
        }
}
```

Posteriormente a esta implementação obtiveram-se os seguintes resultados com a utilização do exercício 2 em *global memory*, em *shared memory* e *texture memory* :

```
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm/ex3_p2_67405_67800_globalv2$ ./sgm
Host processing time: 5163.062500 (ms)
Device processing time: 383.577179 (ms)
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm/ex3_p2_67405_67800_globalv2$ !./te
./testDiffs h_dbull.pgm d_dbull.pgm
images are identical
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm/ex3_p2_67405_67800_globalv2$ ./sgm -p 64
Host processing time: 19342.464844 (ms)
Device processing time: 826.313293 (ms)
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm/ex3_p2_67405_67800_globalv2$ !./te
./testDiffs h_dbull.pgm d_dbull.pgm
images are identical
```

Figura 15:
Resultados obtidos utilizando global memory

```
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm$ ./sgm
Host processing time: 5176.985840 (ms)
Device processing time: 276.664124 (ms)
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm$ !./te
./testDiffs h_dbull.pgm d_dbull.pgm
images are identical
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm$ ./sgm -p 64
Host processing time: 19944.400391 (ms)
Device processing time: 790.564941 (ms)
aca0203@nikola:~/acomputadoresavancada/Trabalho 2/aca_sgm$ !./te
./testDiffs h_dbull.pgm d_dbull.pgm
images are identical
```

Figura 16:
Resultados obtidos utilizando shared memory

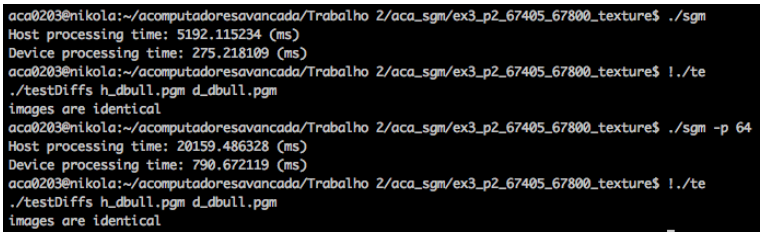

Figura 17:
Resultados obtidos utilizando texture memory

Mais uma vez, é possível verificar-se as melhorias nos resultados tanto a nível de memória global como de memória partilhada.

5 Exercício 4

5.1 Cuda Kernel da função "create_disparity_view()"

No 4º e último exercício era proposto a implementação de um kernel para a função *create_disparity_view*, que tem como objectivo a criação da imagem de disparidade originada pelo programa.

Para este kernel pensou-se numa geometria idêntica à da determinação de custos:

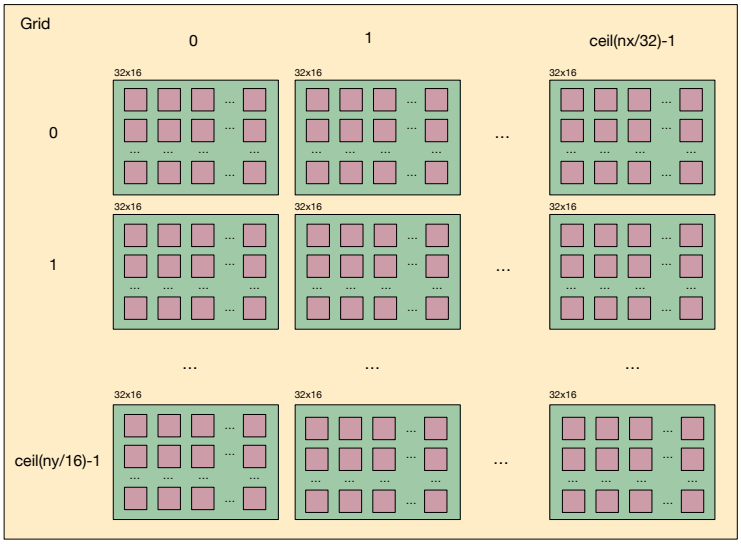

Figura 18:
Geometria do Kernel para a função create_disparity_view()

Constituída por uma grid de ceil(nx/32) por ceil(ny/16) blocos, cada um constituído por 32x16 threads, onde cada uma corresponde a um pixel da imagem final e calcula o índice do array de custos daquele pixel, multiplicando-o por 4.

Com a implementação deste exercício 4, todo o programa passou a ser corrido no device, não ficando nenhuma função a ser executada no host no que diz respeito à função *sgmDevice()*. Nesta fase começa-se a obter os valores finais de toda a implementação, contudo, estes valores variam consoante o tipo de memórias utilizadas. Iremos então mostrar os resultados para os vários tipos de memória, sendo que o que possui a *texture memory*, será o que contém os três tipos de memória e será o mais otimizado.

Utilizando *global memory*, *shared memory* e *texture memory*, obtiveram-se os seguintes resultados:

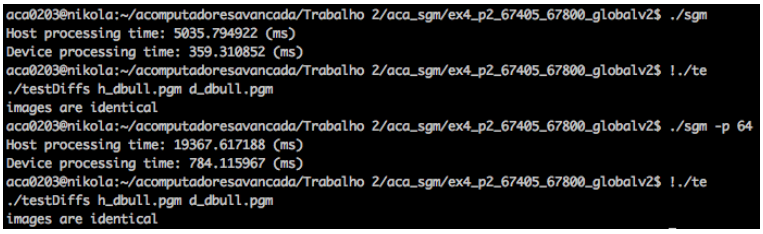

Figura 19:
Resultados obtidos utilizando global memory

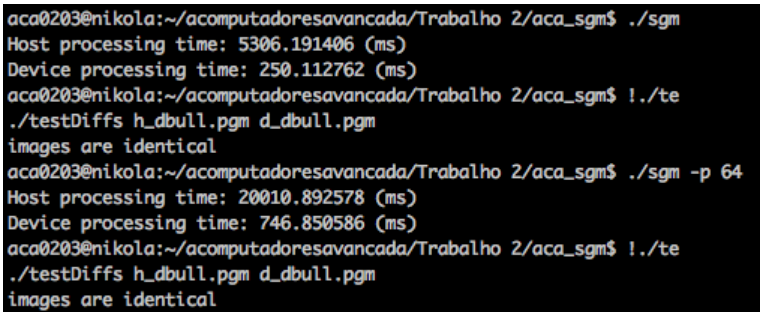

Figura 20:
Resultados obtidos utilizando shared memory

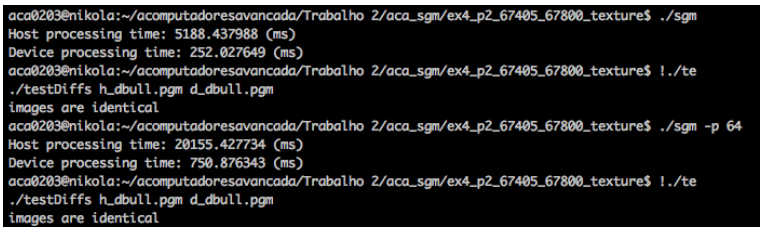

Figura 21:
Resultados obtidos utilizando texture memory

Todos os exemplos nos exercícios anteriores com *texture memory*, possuem também *shared memory*, sendo então estes exemplos os com melhores tempos e versões finais de cada exercício.

6 Intruções de execução

Apenas nos exercícios 1 e 2 existem duas versões na memória global.

No exercício 1, a versão 2 é possível de ser utilizada pelo que é necessário descomentar a linha 429 e comentar a linha 430, correspondente à versão 1 do exercício.

No exercício 2, para utilizar a versão 2 é necessário descomentar as linhas 819 e 828 e comentar as linhas 818 e 827, correspondentes à versão 1.

Nos exercícios seguintes e nos diferentes tipos de memória foram utilizadas as versões 1 do exercício 1 e a versão 2 do exercício 2.

Para utilização dos exercícios, basta apenas fazer *make*, para que tudo compile, o resto é como os professores ensinaram, pelo que não foi alterado nada na maneira de executar os programas.

7 Conclusão

Este trabalho foi útil para assentar conhecimentos que não foram muito abordados nas aulas teóricas e práticas, contudo é um tema bastante interessante, pelo que se deveria considerar a possibilidade de dedicar mais uma aula prática para adquirir os conhecimentos necessários para efetuar este trabalho com menos dificuldades.

É também interessante ver a diferença dos tempos de execução do mesmo programa no device e no host, pois não se tinha a noção que seria um speedup tão elevado e que uma geometria, como foi o caso da versão 1 e 2 do exercicío 2, pudesse ter tanto impacto neste mesmo speedup.

Foi um projeto que deu bastante gosto a realizar, devido ao desafio de conseguir reduzir sempre os tempos de execução e pela aprendizagem e conhecimentos adquiridos.

Por fim, agradecer aos professores, por todo o apoio disponibilizado ao longo da realização do trabalho e do semestre, pois foi muito importante para o nosso progresso académico.