



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	2
2	Exercício 1	3
	2.1 Cuda Kernel da função "determine_costs()"	3
3	Exercício 2	5
	3.1 Cuda Kernel(s) da função "iterate_direction_dirxpos_dev()" e das funções correspondentes a outras direcções	5
4	Conclusão	14

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()*.

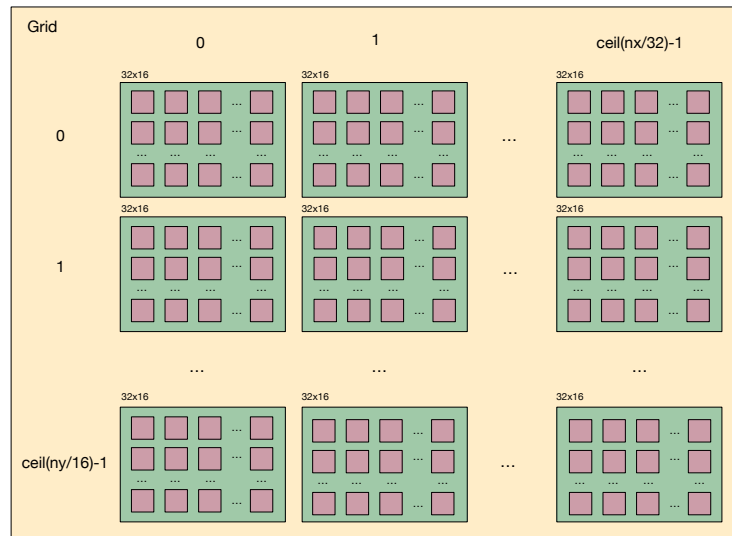


Figura 1:
Geometria do Kernel para a função *determine_costs()*

Neste kernel optou-se por uma geometria (Figura 1 constituída por uma grid de tamanho $(\text{ceil}(\text{nx}/32) \times \text{ceil}(\text{ny}/16))$ com blocos de 32×16 threads cada. Nesta função, 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.

Este exercício 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*.

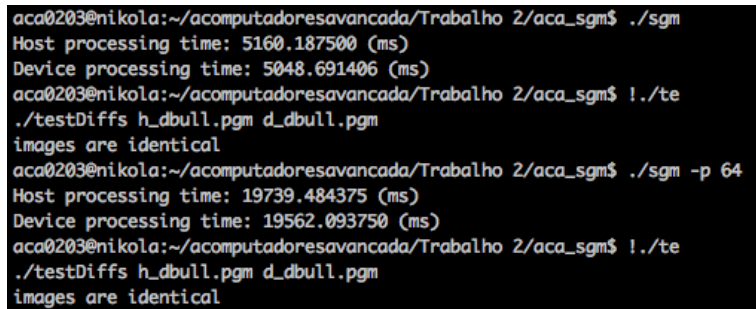
Para a *global memory* utilizou-se o seguinte algoritmo para desenvolver o kernel:

```
__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

TEXTURE MEMORY CODIGO E RESULTADOS

3 Exercício 2

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

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

Versão 1

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

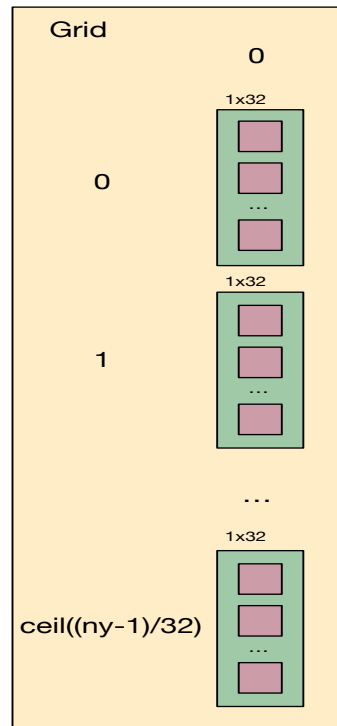


Figura 3:

Geometria do Kernel para as funções `iterate_direction_dirxpos()` e `iterate_direction_dirxneg()`

Como podemos ver na figura 3, a grid é composta por $\text{ceil}(ny/32)$ 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:

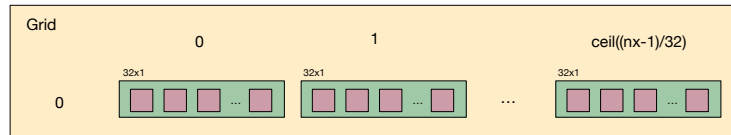


Figura 4:

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 $\text{ceil}(ny/32)$ 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-dir_y,0),
                           &COSTS(i,j,0),
                           abs(LEFT_IMAGE(i,j)-LEFT_IMAGE(i,j-dir_y)),
                           &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 5:

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

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:

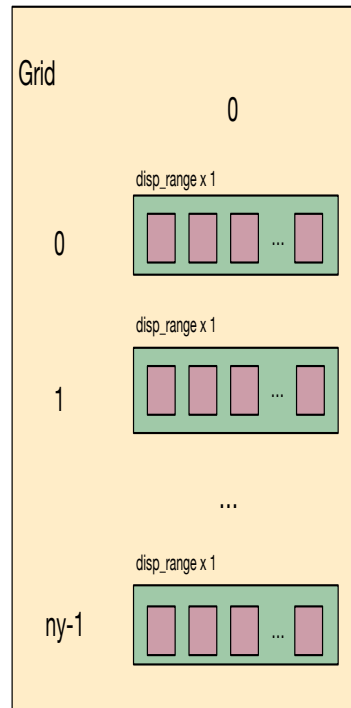


Figura 6:
Geometria do Kernel para as funções `iterate_direction_dirxpos()` e `iterate_direction_dirxneg()`

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:

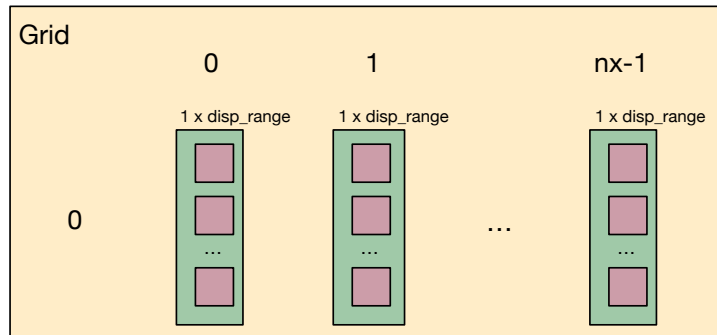


Figura 7:

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 correspondentes 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-dir_y , 0) ,
                      &COSTS(i , l , 0) ,
                      abs(LEFT_IMAGE(i , l)-LEFT_IMAGE(i , l-dir_y)) ,
                      &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 , local , sizeof(int)*disp_range);

    int e_smooth = NPP_MAX_16U;

    for ( int d_p = 0; d_p < disp_range; d_p++ ) {
        if ( d_p - d == 0 ) {
            // No penalty
            e_smooth = MMIN(e_smooth, prior[d_p]);
        } else if ( abs(d_p - d) == 1 ) {
            // Small penalty
            e_smooth = MMIN(e_smooth, prior[d_p]+PENALTY1);
        } else {
            // Large penalty
            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;  
}
```

4 Conclusão

Este trabalho foi útil para assentar todos os conhecimentos que se foi obtendo ao longo destes anos, tanto em Arquitectura de Computadores Avançada como em Arquitectura de Computadores I e II. Ambas as entregas foram primeiro planeadas em papel antes de se avançar para a implementação o que resultou em bons resultados a nível de tempo despendido, para ambos as entregas fez-se uso de dois dias de trabalho.

Na primeira entrega houve uma falha na implementação, esqueceu-se de realizar o shift left 2 do program counter para calcular o endereço para onde é suposto saltar a instrução jump.

Já na segunda entrega teve-se os máximos cuidados para que tudo funcionasse perfeitamente.