Lab 5 - CUDA

Marta Nunes, nº2017246232

1 Exercício 1

Neste primeiro exercício queremos converter uma imagem RGB para uma imagem a preto e branco. Para isso já nos é fornecido o código da kernel, sendo que vamos aplicar o mesmo às várias imagens e comparar o resultado com a versão sequencial. De notar que por uma questão de simplicidade as imagens estão a ser simuladas por matrizes criadas por mim com valores RGB entre 0 e 255. Usámos então o seguinte código.

```
#include <stdio.h>
#define CHANNELS 3

// Device code
__global__ void colorToGreyScaleConvertion(unsigned char * grayImage,
                unsigned char * rgbImage, int width, int height)
{
    int Col = threadIdx.x + (blockIdx.x * blockDim.x);
    int Row = threadIdx.y + (blockIdx.y * blockDim.y);
    if(Col < width && Row < height)
    {
        //get 1D coordinate for the grayscale image
        int greyOffset = Row * width + Col;
        //one can think of the RGB image having CHANNEL times columns
        //of the gray scale image
        int rgbOffset = greyOffset * CHANNELS;
        unsigned char r = rgbImage[rgbOffset];
        //printf("r = %u\n", r);
        //red value for pixel
        unsigned char g = rgbImage[rgbOffset + 1];
        //green value for pixel
        unsigned char b = rgbImage[rgbOffset + 2];
        //blue value for pixel
```

```
        //perform the rescaling and store it
        //we multiply by floating point constants
        grayImage[greyOffset] = (unsigned char)(0.21f * r + 0.71f * g + 0.07f * b);
    }
}

int main()
{
    int width = 255;
    int height = 255;
    int N = width * height;
    //inicialização da imagem
    unsigned char aux[N * CHANNELS];
    for (int i=0; i<N; i++)
    {
        aux[i*CHANNELS] = 255 * sin(i) * sin(i);
        aux[i*CHANNELS + 1] = 255 * sin(i) * sin(i);
        aux[i*CHANNELS + 2] = 255 * sin(i) * sin(i);
    }
    //alocação de memória para a GPU
    unsigned char *rgbImage = NULL;
    unsigned char *grayImage = NULL;
    int err1 = 0;
    int err2 = 0;
    err1 = cudaMalloc(&rgbImage, sizeof(unsigned char)*N*CHANNELS);
    err2 = cudaMalloc(&grayImage, sizeof(unsigned char)*N);
    if(err1 != cudaSuccess || err2 != cudaSuccess)
    {
        printf("Error allocating device memory.\n");
    }
    //CLOCK_PROCESS_CPUTIME_ID - Profiling the execution time of loop
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);
    err1=cudaMemcpy(rgbImage,&aux,sizeof(unsigned char)*N*CHANNELS,cudaMemcpyHostToDevice);
    if(err1 != cudaSuccess)
    {
        printf("Error transfering data to device memory.\n");
    }
```

```
    // Launch device function
    dim3 threadsPerBlock(16, 16, 1);
    dim3 blocksPerGrid(width/16, height/16, 1);
    colorToGreyScaleConvertion<<<blocksPerGrid,threadsPerBlock>>>(grayImage,
                            rgbImage, width, height);
    // Copy data from device memory to host memory
    unsigned char * host_buffer = (unsigned char *)malloc(sizeof(unsigned char)*N);
    err1=cudaMemcpy(host_buffer,grayImage,sizeof(unsigned char)*N,cudaMemcpyDeviceToHost);
    if(err1 != cudaSuccess)
    {
        printf("Error transfering data from device memory.\n");
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
    double initialTime=(start.tv_sec*1e3)+(start.tv_nsec*1e-6);
    double finalTime=(end.tv_sec*1e3)+(end.tv_nsec*1e-6);
    printf("host_buffer[%i][%i]=%u\n", width/2, height/2,
                    host_buffer[((width/2) * width) + height/2]);
    printf("host_buffer[%i][%i]=%u\n", width/4, height/4,
                    host_buffer[((width/4) * width) + height/4]);
    printf("host_buffer[%i][%i]=%u\n", 3*width/4, 3*height/4,
                    host_buffer[((3*width/4) * width) + 3*height/4]);
    printf("Cuda (%i*%i elemens):\t%f ms\n", width, height, (finalTime - initialTime));
    // Free device buffers
    cudaFree(rgbImage);
    cudaFree(grayImage);
    return 0;
}
```

Para analisarmos a melhoria de performance entre os códigos sequencial e CUDA vamos executar os mesmos para vários tamanhos das matrizes. Comecemos por ver os tempos para uma matriz de 255x255.

```
uc2017246232@student.uc.pt@mf-w10-k18:~/lab5/ex1$ ./lab5_ex1_seq.out 255 255
img_grey[127][127]=27
img_grey[63][63]=163
img_grey[191][191]=14
Sequential (255x255 elemens):   0.450927 ms
```

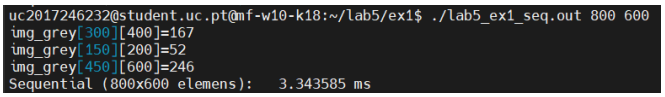
(a) Código sequencial.

```
uc2017246232@student.uc.pt@mf-w10-k18:~/lab5/ex1$ ./lab5_ex1_255x255
host_buffer[127][127]=27
host_buffer[63][63]=163
host_buffer[191][191]=14
Cuda (255*255 elemens): 0.146865 ms
```

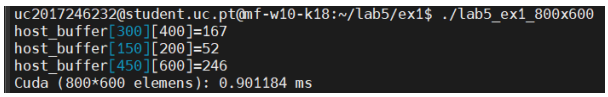
(b) Código CUDA

Figure 1: Tempo gasto na execução dos códigos sequencial e CUDA para uma matriz de 255x255.

De seguida vejamos os tempos de execução para uma matriz de 800x600.



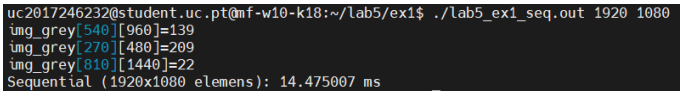
(a) Código sequencial.



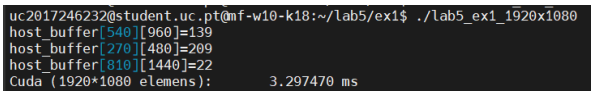
(b) Código CUDA

Figure 2: Tempo gasto na execução dos códigos sequencial e CUDA para uma matriz de 800x600.

Por fim, vejamos os tempos de execução para uma matriz de 1920x1080.



(a) Código sequencial.



(b) Código CUDA

Figure 3: Tempo gasto na execução dos códigos sequencial e CUDA para uma matriz de 1920x1080.

Usando os tempos obtidos temos então os seguintes valores de speedup para os vários tamanhos das matrizes:

- 255x255 - $Speedup = 3.07035$
- 800x600 - $Speedup = 3.71021$
- 1920x1080 - $Speedup = 4.38973$

Concluímos então que o código CUDA melhora a performance do nosso programa, no entanto, para além disso, é possível observar que quanto maior é o tamanho da nossa matriz, maior é o speedup. Na verdade esta conclusão já era expectável uma vez que quanto maior for o tamanho da nossa matriz mais operações temos disponíveis para paralelizar, ou seja, mais conseguimos diminuir o nosso tempo de execução em relação ao tempo de execução sequencial.

2 Exercício 2

Neste exercício queremos criar um programa que realize a soma de todos os elementos de um vetor com N elementos, sendo que por uma questão de simplificação escolhemos para o N o valor de 9984. Pretendemos ainda implementar uma versão otimizada que faça uso da memória partilhada, sendo que a mesma é apresentada no seguinte código.

```
#include <stdio.h>
#define N 9984

// Device code
__global__ void reduce(int *g_idata, int *g_odata)
```

```
{
    __shared__ int s_data[N];
    unsigned int tid = threadIdx.x;
    unsigned int gid = threadIdx.x + (blockIdx.x * blockDim.x);

    s_data[tid] = g_idata[gid];
    __syncthreads();

    //reduction in shared memory
    for(unsigned int s=1; s<blockDim.x; s*=2)
    {
        if (tid % (2*s) == 0)
        {
            s_data[tid] += s_data[tid+s];
        }
        __syncthreads();
    }

    //write result to global memory
    if (tid == 0)
        g_odata[blockIdx.x] = s_data[tid];
}

int main()
{
    //inicialização do vetor
    int aux[N];
    int final = 0;
    for (int i=0; i<N; i++)
    {
        aux[i] = 1;
    }
    int *v1 = NULL;
    int *v2 = NULL;
    int err1 = 0;
    int err2 = 0;
    err1 = cudaMalloc(&v1, sizeof(int)*N);
    err2 = cudaMalloc(&v2, sizeof(int)*(N/256));
```

```
if(err1 != cudaSuccess || err2 != cudaSuccess)
{
    printf("Error allocating device memory.\n");
}
//CLOCK_PROCESS_CPUTIME_ID - Profiling the execution time of loop
struct timespec start, end;
clock_gettime(CLOCK_MONOTONIC, &start);

err1 = cudaMemcpy(v1,&aux,sizeof(int)*N,cudaMemcpyHostToDevice);
if(err1 != cudaSuccess)
{
    printf("Error transfering data to device memory.\n");
}
// Launch device function
int threadsPerBlock = 256;
int blocksPerGrid = N/256;
reduce<<<blocksPerGrid,threadsPerBlock>>>(v1, v2);

// Copy data from device memory to host memory
int * host_buffer = (int *)malloc(sizeof(int)*(N/256));
err1 = cudaMemcpy(host_buffer,v2,sizeof(int)*(N/256),cudaMemcpyDeviceToHost);
if(err1 != cudaSuccess)
{
    printf("Error transfering data from device memory.\n");
}
for (int i=0; i<N/256; i++)
{
    final += host_buffer[i];
}
clock_gettime(CLOCK_MONOTONIC, &end);
double initialTime=(start.tv_sec*1e3)+(start.tv_nsec*1e-6);
double finalTime=(end.tv_sec*1e3)+(end.tv_nsec*1e-6);
printf("final = %i\n", final);
printf("Cuda (%i elemens):\t%f ms\n", N, (finalTime - initialTime));
// Free device buffers
cudaFree(v1);
cudaFree(v2);
return 0;
```

```
}
```

Implementámos ainda uma versão CUDA não otimizada, ou seja, que não recorria à memória partilhada, sendo que apenas alterámos o código da kernel para o código apresentado de seguida.

```
__global__ void reduce(int *g_idata, int *g_odata)
{
    unsigned int tid = threadIdx.x;
    unsigned int gid = threadIdx.x + (blockIdx.x * blockDim.x);

    for(unsigned int s=1; s<blockDim.x; s*=2)
    {
        if (tid % (2*s) == 0)
        {
            g_idata[gid] += g_idata[gid+s];
        }
        __syncthreads();
    }

    if (tid == 0)
        g_odata[blockIdx.x] = g_idata[tid];
}
```

Ao executarmos estes códigos juntamente com o código sequencial, obtemos os seguintes tempos de execução.

```
uc2017246232@student.uc.pt@mf-w10-k18:~/lab5/ex2$ ./lab5_ex2_seq.out
final = 9984
Sequential (9984 elemens):      0.088104 ms
```

(a) Código sequencial.

```
uc2017246232@student.uc.pt@mf-w10-k18:~/lab5/ex2$ ./lab5_ex2_1
final = 9984
Cuda (9984 elemens):    0.060260 ms
```

(b) Código CUDA sem shared memory.

```
uc2017246232@student.uc.pt@mf-w10-k18:~/lab5/ex2$ ./lab5_ex2_2
final = 9984
Cuda (9984 elemens):    0.053582 ms
```

(c) Código CUDA com shared memory.

Figure 4: Tempo gasto na execução dos códigos sequencial e CUDA.

Da análise destes tempos percebemos mais uma vez que a versão CUDA do programa otimiza a performance do nosso código, no entanto, com a memória partilhada, uma vez que não estamos sempre a perder tempo com os acessos à memória, melhoramos o nosso tempo total de execução do programa.

3 Exercício 3

Neste exercício temos como objetivo implementar uma função que devolve a transposta de uma matriz dada. Comecemos por implementar uma versão simples em CUDA sem otimizações, sendo que o código da kernel é o apresentado de seguida.

```
// Device code
__global__ void reduce(int *g_idata, int *g_odata, int width, int height)
{
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    // CODE
    g_odata[col * height + row] = g_idata[row * width + col];
}
```

No entanto podemos ainda otimizar mais o nosso código recorrendo à shared memory, como implementado no seguinte código.

```
#include <stdio.h>
#define TILE_DIM 32
#define BLOCK_ROWS 8
#define N 262144

// Device code
__global__ void reduce(int *g_idata, int *g_odata)
{
    __shared__ int tile[TILE_DIM][TILE_DIM];

    int x = threadIdx.x + blockIdx.x * TILE_DIM;
    int y = threadIdx.y + blockIdx.y * TILE_DIM;
    int width = gridDim.x * TILE_DIM;

    for (int j=0; j<TILE_DIM; j+=BLOCK_ROWS)
        tile[threadIdx.y+j][threadIdx.x] = g_idata[(y+j)*width + x];

    __syncthreads();

    x = blockIdx.y * TILE_DIM + threadIdx.x;
    y = blockIdx.x * TILE_DIM + threadIdx.y;
```

```
    for (int j=0; j<TILE_DIM; j+=BLOCK_ROWS)
        g_odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}

int main()
{
    int width = 512;
    int height = 512;

    //inicialização do vetor
    int aux[N];
    for(int i=0; i<height; i++)
    {
        for (int j=0; j<width; j++)
        {
            aux[i*width + j] = i;
        }
    }
    int *v1 = NULL;
    int *v2 = NULL;
    int err1 = 0;
    int err2 = 0;
    err1 = cudaMalloc((void **)&v1, sizeof(int)*N);
    err2 = cudaMalloc((void **)&v2, sizeof(int)*N);
    if(err1 != cudaSuccess || err2 != cudaSuccess)
    {
        printf("Error allocating device memory.\n");
    }
    //CLOCK_PROCESS_CPUTIME_ID - Profiling the execution time of loop
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    err1 = cudaMemcpy(v1,aux,sizeof(int)*N,cudaMemcpyHostToDevice);
    if(err1 != cudaSuccess)
    {
        printf("Error transfering data to device memory.\n");
    }
    // Launch device function
```

```
    dim3 threadsPerBlock(TILE_DIM/4, TILE_DIM/4, 1);
    dim3 blocksPerGrid(width/TILE_DIM, height/TILE_DIM, 1);
    reduce<<<blocksPerGrid,threadsPerBlock>>>(v1, v2);

    // Copy data from device memory to host memory
    int * host_buffer = (int *)malloc(sizeof(int)*N);
    err1 = cudaMemcpy(host_buffer,v2,sizeof(int)*N,cudaMemcpyDeviceToHost);
    if(err1 != cudaSuccess)
    {
        printf("Error transfering data from device memory.\n");
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
    double initialTime=(start.tv_sec*1e3)+(start.tv_nsec*1e-6);
    double finalTime=(end.tv_sec*1e3)+(end.tv_nsec*1e-6);
    printf("host_buffer[%i][%i]=%u\n", width/2, height/4,
                host_buffer[((width/2) * height) + height/4]);
    printf("host_buffer[%i][%i]=%u\n", width/4, 3*height/4,
                host_buffer[((width/4) * height) + 3*height/4]);
    printf("host_buffer[%i][%i]=%u\n", 3*width/4, height/2,
                host_buffer[((3*width/4) * height) + height/2]);
    printf("Cuda (%ix%i elemens):\t%f ms\n", width, height, (finalTime - initialTime));
    // Free device buffers
    cudaFree(v1);
    cudaFree(v2);
    return 0;
}
```

Vamos então analisar os tempos de execução dos códigos sequencial e CUDA.

```
uc2017246232@student.uc.pt@mf-w10-k18:~/lab5/ex3$ ./lab5_ex3_seq.out 512 512
m_transposta[256][128]=128
m_transposta[128][384]=384
m_transposta[384][256]=256
Sequential (512x512 elemens):   8.797525 ms
```

(a) Código sequencial.

```
uc2017246232@student.uc.pt@mf-w10-k18:~/lab5/ex3$ ./lab5_ex3_1
host_buffer[256][128]=128
host_buffer[128][384]=384
host_buffer[384][256]=256
Cuda (512x512 elemens): 1.156261 ms
```

(b) Código CUDA sem shared memory.

```
uc2017246232@student.uc.pt@mf-w10-k18:~/lab5/ex3$ ./lab5_ex3_2
host_buffer[256][128]=128
host_buffer[128][384]=384
host_buffer[384][256]=256
Cuda (512x512 elemens): 1.047340 ms
```

(c) Código CUDA com shared memory.

Figure 5: Tempo gasto na execução dos códigos sequencial e CUDA.

Da análise destes percebemos mais uma vez que com o código CUDA estamos a melhorar consider-

avelmente a performance do programa. Com shared memory, uma vez que diminuimos a quantidade de vezes que acedemos à memória, existe ainda uma melhoria de tempo de execução em relação ao código CUDA não otimizado.

4 Exercício 4

Neste exercício queremos implementar um programa que devolva o histograma de intensidades de uma imagem. Uma vez que não era especificado no enunciado, decidi usar uma imagem de 256x256 pixeis e ter um histograma de 16 bins. De seguida apresento o código da kernel da implementação do histograma não usando memória partilhada.

```
__global__ void histogram(unsigned char * image, int * hist, int size)
{
    int Col = threadIdx.x + (blockIdx.x * blockDim.x);
    int Row = threadIdx.y + (blockIdx.y * blockDim.y);

    int i = Row * size + Col;
    unsigned char pixel = image[i];
    int index = (int)(pixel/bin);
    atomicAdd(&hist[index],1);
}
```

Usando memória partilhada temos a seguinte implementação.

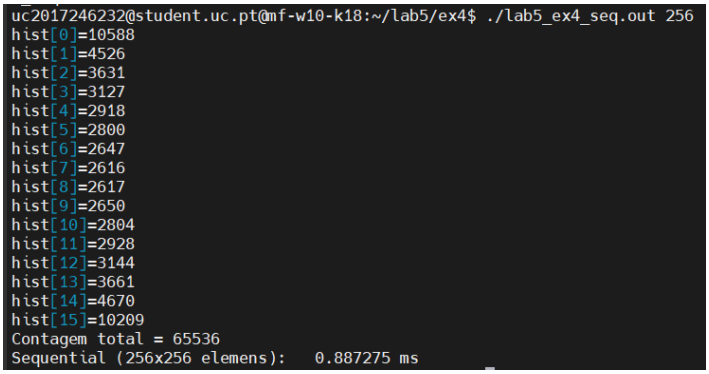
```
#include <stdio.h>
#define bin 16
#define N 65536

// Device code
__global__ void histogram(unsigned char * image, int * hist, int size)
{
    __shared__ unsigned char s_data[256];
    __shared__ int s_hist[bin];

    int Col = threadIdx.x + (blockIdx.x * blockDim.x);
    int Row = threadIdx.y + (blockIdx.y * blockDim.y);
    int i = Row * size + Col;
    int i_thread = threadIdx.y * blockDim.x + threadIdx.x;

    s_data[i_thread] = image[i];
```

```
    if (i_thread < bin)
    {
        s_hist[i_thread] = 0;
    }
    __syncthreads();
    atomicAdd(&s_hist[(int)(s_data[i_thread]/bin)],1);
    __syncthreads();

    if (i_thread < bin)
    {
        atomicAdd(&hist[i_thread + blockIdx.x * bin], s_hist[i_thread]);
    }
}

int main()
{
    int size = 256;

    //inicialização da imagem
    unsigned char aux[N];
    for (int i=0; i<N; i++)
    {
        aux[i] = 255 * sin(i) * sin(i);
    }

    //alocação de memória para a GPU
    unsigned char *image = NULL;
    int *hist = NULL;

    int err1 = 0;
    int err2 = 0;
    err1 = cudaMalloc(&image, sizeof(unsigned char)*N);
    err2 = cudaMalloc(&hist, sizeof(int)*bin*bin);
    if(err1 != cudaSuccess || err2 != cudaSuccess)
    {
        printf("Error allocating device memory.\n");
    }
```

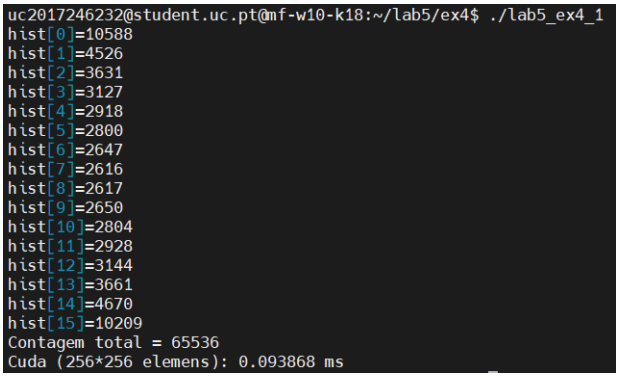
```
//CLOCK_PROCESS_CPUTIME_ID - Profiling the execution time of loop
struct timespec start, end;
clock_gettime(CLOCK_MONOTONIC, &start);

err1 = cudaMemcpy(image,&aux,sizeof(unsigned char)*N,cudaMemcpyHostToDevice);
if(err1 != cudaSuccess)
{
    printf("Error transfering data to device memory.\n");
}

// Launch device function
dim3 threadsPerBlock(16, 16, 1);
dim3 blocksPerGrid(size/16, size/16, 1);
histogram<<<blocksPerGrid,threadsPerBlock>>>(image, hist, size);

// Copy data from device memory to host memory
int * host_buffer = (int *)malloc(sizeof(int)*bin*bin);
err1 = cudaMemcpy(host_buffer,hist,sizeof(int)*bin*bin,cudaMemcpyDeviceToHost);
if(err1 != cudaSuccess)
{
    printf("Error transfering data from device memory.\n");
}
int histograma[bin];
for (int i=0; i<bin; i++)
{
    histograma[i] = 0;
    for (int j=0; j<bin; j++)
    {
        histograma[i] += host_buffer[j*bin + i];
    }
}

clock_gettime(CLOCK_MONOTONIC, &end);
double initialTime=(start.tv_sec*1e3)+(start.tv_nsec*1e-6);
double finalTime=(end.tv_sec*1e3)+(end.tv_nsec*1e-6);
int contador = 0;
for(int i=0; i<bin; i++)
{
```

```
        printf("hist[%i]=%u\n", i, histograma[i]);
        contador += histograma[i];
    }
    printf("Contagem total = %i\n", contador);
    printf("Cuda (%i*%i elemens):\t%f ms\n", size, size, (finalTime - initialTime));
    // Free device buffers
    cudaFree(image);
    cudaFree(hist);
    return 0;
}
```

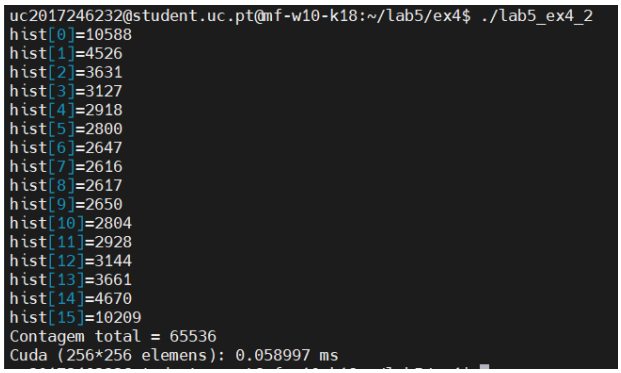
Da observação destas implementações percebemos que o que o código está a fazer é a confirmar o valor de cada pixel e a incrementar o bin que corresponde ao valor desse pixel. De notar, no entanto, que estamos a usar algo que ainda não usámos nos códigos anteriores, as operações atómicas. Estas são operações que executam sem interrupções, ou seja, podemos alterar o valor da mesma variável a partir de várias threads com estas operações preservando a integridade do código, o que de outra maneira não era possível.



(a) Código sequencial.



(b) Código CUDA sem shared memory.



(c) Código CUDA com shared memory.

Figure 6: Tempo gasto na execução dos códigos sequencial e CUDA.

Tal como nos programas anteriores, a versão sequencial é a que demora mais tempo a executar e a versão CUDA com shared memory é a mais eficiente, ou seja, mesmo com as operações atómicas que implicam que as threads leiam todas os mesmos dados ao mesmo tempo, a performance do programa é melhorada.

5 Exercício 5

Neste exercício queremos implementar a multiplicação de duas matrizes. Por uma questão de simplicidade, ambas as matrizes tem o mesmo tamanho, 256x256 pixeis. Implementei então a seguinte função na kernel.

```
__global__ void reduce(int *m1, int *m2, int *m3, int width, int height)
{
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    int row = threadIdx.y + blockIdx.y * blockDim.y;

    for (int k=0; k<width; k++)
    {
        m3[row * width + col] += m1[row*width + k] * m2[k*width + col];
    }
}
```

Como se pode observar, a função apresentada faz uso do paralelismo, mas não aproveita a memória partilhada. Para fazer uso desta alterei o código para seguir as ideias discutidas nas aulas teóricas, ou seja, para cada pixel vamos aproveitar todas as threads de um bloco para irmos buscar um valor por por thread a cada matriz sucessivamente até chegarmos ao fim da matriz, usando todos os valores para realizar a operação pretendida.

```
#include <stdio.h>
#define TILE 16

// Device code
__global__ void reduce(int *m1, int *m2, int *m3, int width, int height)
{
    __shared__ int sub1[TILE][TILE];
    __shared__ int sub2[TILE][TILE];

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
```

```
    int ty = threadIdx.y;

    int col = tx + bx * TILE;
    int row = ty + by * TILE;
    int vfinal = 0;

    // Loop que corre todos os sub-blocos necessários para calcular 1 elemento
    for (int m=0; m<width/TILE; ++m)
    {
        sub1[ty][tx] = m1[row*width + m*TILE + tx];
        sub2[ty][tx] = m2[(m*TILE + ty)*width + col];
        __syncthreads();
        for (int k=0; k<TILE; ++k)
            vfinal += sub1[ty][k] * sub2[k][tx];
        __syncthreads();
    }

    m3[row * width + col] = vfinal;
}

int main()
{
    int width = 256;
    int height = 256;
    int N = width * height;

    //inicialização do vetor
    int aux1[N];
    int aux2[N];
    for(int i=0; i<N; i++)
    {
        aux1[i] = 1;
        aux2[i] = 2;
    }

    int *m1 = NULL;
    int *m2 = NULL;
    int *m3 = NULL;
```

```
int err1 = 0;
int err2 = 0;
int err3 = 0;
err1 = cudaMalloc(&m1, sizeof(int)*N);
err2 = cudaMalloc(&m2, sizeof(int)*N);
err3 = cudaMalloc(&m3, sizeof(int)*N);
if(err1 != cudaSuccess || err2 != cudaSuccess || err3 != cudaSuccess)
{
    printf("Error allocating device memory.\n");
}
//CLOCK_PROCESS_CPUTIME_ID - Profiling the execution time of loop
struct timespec start, end;
clock_gettime(CLOCK_MONOTONIC, &start);

err1 = cudaMemcpy(m1,&aux1,sizeof(int)*N,cudaMemcpyHostToDevice);
err2 = cudaMemcpy(m2,&aux2,sizeof(int)*N,cudaMemcpyHostToDevice);
if(err1 != cudaSuccess || err2 != cudaSuccess)
{
    printf("Error transfering data to device memory.\n");
}
// Launch device function
dim3 threadsPerBlock(16, 16, 1);
dim3 blocksPerGrid(width/16, height/16, 1);
reduce<<<blocksPerGrid,threadsPerBlock>>>(m1, m2, m3, width, height);

// Copy data from device memory to host memory
int * host_buffer = (int *)malloc(sizeof(int)*N);
err1 = cudaMemcpy(host_buffer,m3,sizeof(int)*N,cudaMemcpyDeviceToHost);
if(err1 != cudaSuccess)
{
    printf("Error transfering data from device memory.\n");
}
clock_gettime(CLOCK_MONOTONIC, &end);
double initialTime=(start.tv_sec*1e3)+(start.tv_nsec*1e-6);
double finalTime=(end.tv_sec*1e3)+(end.tv_nsec*1e-6);
printf("host_buffer[%i][%i]=%u\n", width/2, height/4,
                host_buffer[((width/2) * height) + height/4]);
printf("host_buffer[%i][%i]=%u\n", width/4, 3*height/4,
```

```
                    host_buffer[((width/4) * height) + 3*height/4]);
    printf("host_buffer[%i][%i]=%u\n", 3*width/4, height/2,
                    host_buffer[((3*width/4) * height) + height/2]);
    printf("Cuda (%ix%i elemens):\t%f ms\n", width, height, (finalTime - initialTime));
    // Free device buffers
    cudaFree(m1);
    cudaFree(m2);
    cudaFree(m3);
    return 0;
}
```

De seguida apresenta-se os tempos de execução dos programas sequencial e CUDA implementados para resolução do problema apresentado. De notar que por uma questão de simplicidade, as matrizes a multiplicar tem os valores 1 e 2, respetivamente, pelo que o resultado final da multiplicação irá ser 512 para qualquer valor da matriz final. Por essa razão, o código sequencial faz apenas debug para um valor, enquanto que os códigos CUDA fazem debug para 3 valores, pois não precisamos de mais valores para confirmar que o resultado final é o esperado.

```
uc2017246232@student.uc.pt@mf-w10-k18:~/lab5/ex5$ ./lab5_ex5_seq.out 256
m3[64][64] = 512
Sequential (256x256 elemens):   88.821794 ms
```

(a) Código sequencial.

```
uc2017246232@student.uc.pt@mf-w10-k18:~/lab5/ex5$ ./lab5_ex5_1
host_buffer[128][64]=512
host_buffer[64][192]=512
host_buffer[192][128]=512
Cuda (256x256 elemens): 1.049738 ms
```

(b) Código CUDA sem shared memory.

```
uc2017246232@student.uc.pt@mf-w10-k18:~/lab5/ex5$ ./lab5_ex5_2
host_buffer[128][64]=512
host_buffer[64][192]=512
host_buffer[192][128]=512
Cuda (256x256 elemens): 0.543476 ms
```

(c) Código CUDA com shared memory.

Figure 7: Tempo gasto na execução dos códigos sequencial e CUDA.

Da análise dos resultados concluímos que existe uma melhoria enorme de performance com a inclusão de paralelismo no código. É ainda visível uma melhoria nos tempos de execução no código com shared memory, o que já era expectável.

6 Exercício 6

Por fim, no último exercício pretendemos implementar um programa CUDA que aplique a operação de convolução a uma imagem. Para isso usámos um filtro 3x3 de média dado por

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

e uma imagem de 256x256 pixeis. Esta imagem foi criada por mim, por uma questão de simplicidade, e consiste em pixeis alternados com o valor 0 e com o valor 255, como mostra a seguinte figura para

uma matriz semelhante mas de tamanho 10x10.

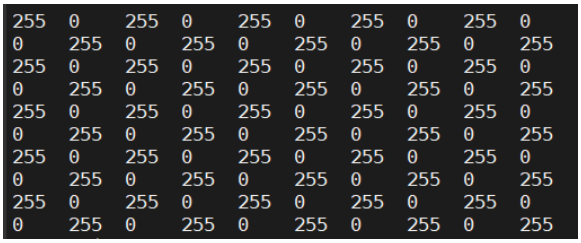


Figure 8: Exemplo da matriz inicial.

Tendo em conta a matriz e o filtro usado, na imagem seguinte é apresentado um exemplo da matriz final para uma imagem de tamanho 10x10.

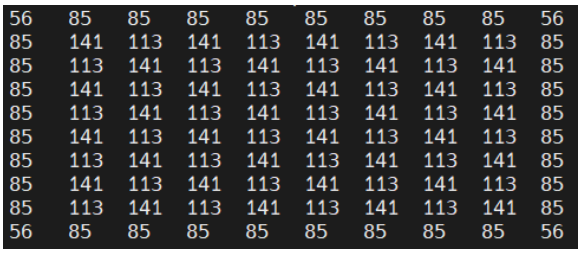


Figure 9: Exemplo da matriz final.

Tendo isto em conta, implementei esta operação realizando uma versão simples em CUDA.

```
__global__ void convolution(unsigned char *img, unsigned char *res, float *filtro, int size)
{
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    int row = threadIdx.y + blockIdx.y * blockDim.y;

    int aux[t_filtro*t_filtro];
    float valor = 0;

    for (int i=0; i<t_filtro; i++)
    {
        for (int j=0; j<t_filtro; j++)
        {
            if(row+i-1 < 0 || row+i-1 == size || col+j-1 < 0 || col+j-1 == size)
                aux[i*t_filtro + j] = 0;
            else
                aux[i*t_filtro + j] = img[(row+i-1)*size + (col+j-1)];
        }
```

```
    }
    for (int n=0; n<t_filtro*t_filtro; n++)
    {
        valor += aux[n] * filtro[n];
    }
    res[row*size + col] = (unsigned char)valor;
}
```

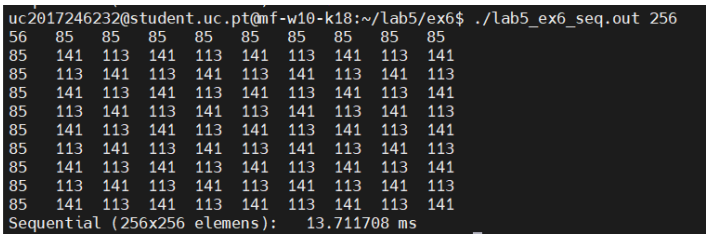
De seguida procedi à melhoria do código anterior, fazendo uso da shared memory.

```
#include <stdio.h>
#define t_filtro 3
#define N 65536
#define TILE 16

// Device code
__global__ void convolution(unsigned char *img, unsigned char *res, float *filtro, int size)
{
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    int tcol = threadIdx.x;
    int trow = threadIdx.y;

    float valor = 0;

    __shared__ unsigned char image[TILE][TILE];
    __shared__ float fil[t_filtro][t_filtro];

    if (col < 3 && row < 3)
    {
        fil[row][col] = filtro[row*3 + col];
    }

    image[trow][tcol] = img[row*size + col];

    __syncthreads();

    for (int i=0; i<t_filtro; i++)
    {
```

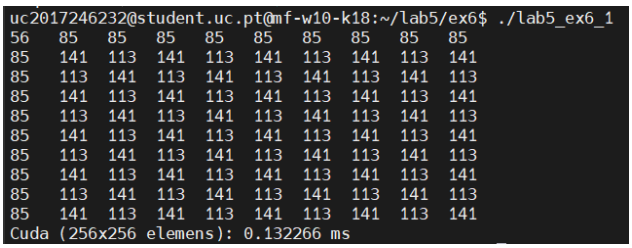
```
        for (int j=0; j<t_filtro; j++)
        {
            if ((trow+i-1)<0  || (trow+i-1)>15 || (tcol+j-1)<0 || (tcol+j-1)>15)
            {
                if ((row+i-1)>=0  || (row+i-1)<=255 || (col+j-1)>=0 || (col+j-1)<=255)
                    valor += img[(row + i - 1)*size + (col + j - 1)];
            }
            else
                valor += image[trow + i - 1][tcol + j - 1] * fil[i][j];
        }
    }
    res[row*size + col] = (unsigned char)valor;
}

int main()
{
    int size = 256;

    //inicialização do vetor
    unsigned char aux1[N];
    float aux2[t_filtro*t_filtro];
    for(int i=0; i<size; i++)
    {
        for(int j=0; j<size; j++)
        {
            if (i%2 == j%2)
                aux1[i*size + j] = 255;
            else
                aux1[i*size + j] = 0;
        }
    }

    for(int i=0; i<t_filtro*t_filtro; i++)
    {
        aux2[i] = 1;
        aux2[i] = aux2[i]/9;
    }
```

```
unsigned char *img = NULL;
float *filtro = NULL;
unsigned char *res = NULL;
int err1 = 0;
int err2 = 0;
int err3 = 0;
err1 = cudaMalloc(&img, sizeof(unsigned char)*N);
err2 = cudaMalloc(&filtro, sizeof(float)*t_filtro*t_filtro);
err3 = cudaMalloc(&res, sizeof(unsigned char)*N);
if(err1 != cudaSuccess || err2 != cudaSuccess || err3 != cudaSuccess)
{
    printf("Error allocating device memory.\n");
}
//CLOCK_PROCESS_CPUTIME_ID - Profiling the execution time of loop
struct timespec start, end;
clock_gettime(CLOCK_MONOTONIC, &start);

err1 = cudaMemcpy(img,&aux1,sizeof(unsigned char)*N,cudaMemcpyHostToDevice);
err2 = cudaMemcpy(filtro,&aux2,sizeof(float)*t_filtro*t_filtro,cudaMemcpyHostToDevice);
if(err1 != cudaSuccess || err2 != cudaSuccess || err3 != cudaSuccess)
{
    printf("Error transfering data to device memory.\n");
}
// Launch device function
dim3 threadsPerBlock(16, 16, 1);
dim3 blocksPerGrid(size/16, size/16, 1);
convolution<<<blocksPerGrid,threadsPerBlock>>>(img, res, filtro, size);

// Copy data from device memory to host memory
unsigned char * host_buffer = (unsigned char *)malloc(sizeof(unsigned char)*N);
err1 = cudaMemcpy(host_buffer,res,sizeof(unsigned char)*N,cudaMemcpyDeviceToHost);
if(err1 != cudaSuccess)
{
    printf("Error transfering data from device memory.\n");
}
clock_gettime(CLOCK_MONOTONIC, &end);
double initialTime=(start.tv_sec*1e3)+(start.tv_nsec*1e-6);
double finalTime=(end.tv_sec*1e3)+(end.tv_nsec*1e-6);
```

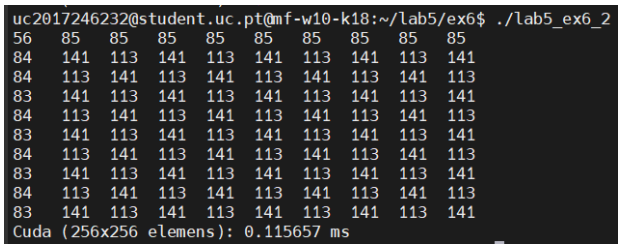
```
    for(int i=0; i<10; i++)
    {
        for(int j=0; j<10; j++)
        {
            printf("%-4u ", host_buffer[i*size + j]);
        }
        printf("\n");
    }
    printf("Cuda (%ix%i elemens):\t%f ms\n", size, size, (finalTime - initialTime));
    // Free device buffers
    cudaFree(img);
    cudaFree(filtro);
    cudaFree(res);
    return 0;
}
```



(a) Código sequencial.



(b) Código CUDA sem shared memory.



(c) Código CUDA com shared memory.

Figure 10: Tempo gasto na execução dos códigos sequencial e CUDA.

Da observação dos tempos de execução percebemos mais uma vez que a paralelização otimizou bastante o código, no entanto a utilização de memória partilhada não melhorou tanto a performance como esperado. Isto pode dever-se às condições existentes no código da kernel que não consegui otimizar.