Computação Heterogénea de Alto Desempenho - Lab 6

Manuel Santos - 2019231352

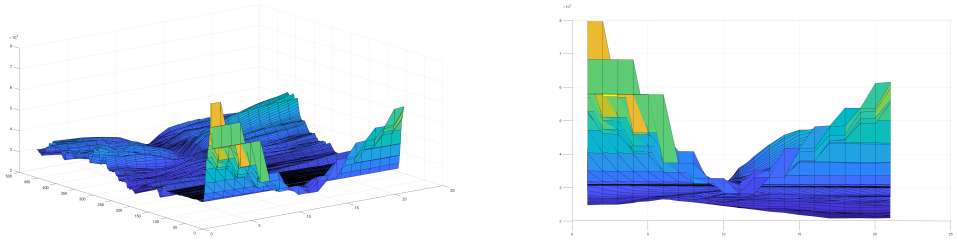
19 Novembro 2023

Exercício 1

No exercício 1, segui as indicações e explicações dadas para correr com sucesso o código disponibilizado.

```
# -> Run
./Lab6_ex1.sh
```

De seguida, visualizei os resultados obtidos recorrendo ao *Matlab*, como é possível verificar na figura 1.



(a) Vista Panorâmica (b) Vista Frontal

Figure 1: Representação gráfica dos resultados obtidos

Pela análise dos resultados obtidos, concluo que os feeds de vídeo têm um atraso de aproximadamente 11 frames um em relação ao outro, uma vez que no gráfico o vale (valor mais baixo) situa-se sensivelmente em t = 11.

Exercício 2

Neste exercício, o objetivo é criar um programa que calcule um valor aproximado de π, recorrendo à biblioteca CURAND.

A estratégia utilizada passa por obter pontos aleatórios dentro de um quadrado onde está inscrita um circulo. Relacionando as áreas do quadrado e do círculo, como demonstrado na figura 2.

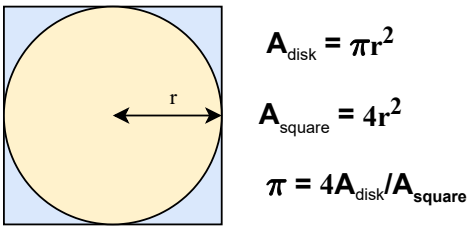


Figure 2: Relação utilizada para obter π

Na prática, o algoritmo passa por gerar pontos aleatórios dentro do quadrado e contabilizar quantos calham dentro do círculo e quantos são gerados no total, ou seja, no quadrado.

$$\frac{4 \times Points_{Circle}}{Points_{Circle} + Points_{Square}} \tag{1}$$

De forma a obter este resultado, o seguinte kernel foi produzido.

```
__global__ void get_pi(unsigned long long int *count_circle,
                       unsigned long long int *count_square, int seed)
{
    // ===== Calculate global index
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    // ===== Initialize random number generator
    curandState state;
    curand_init(seed, idx, 0, &state);

    // ===== Initialize counters
    count_circle[idx] = 0;
    count_square[idx] = 0;

    // ===== Generate random numbers and count
    for(int i = 0; i < triesPerThread; i++)
    {
        float x = curand_uniform(&state)*2.0 - 1.0;
        float y = curand_uniform(&state)*2.0 - 1.0;

        if (x*x + y*y <= 1.0)
            count_circle[idx]++;
        else
            count_square[idx]++;
    }
}
```

Este kernel recebe dois arrays para registar os pontos dentro do círculo, os pontos dentro do quadrado e uma *seed* para inicializar o gerador de números aleatórios. Para correr o programa, basta correr o script bash associado.

```
# -> Run
./Lab6_ex2.sh
```

O resultado obtido pode ser observado na figura 3.

```
=========== Running Lab6 Exercise 2 ===========
Threads per block: 256 || Blocks per grid: 1024
--> Pi = 3.1415892361
Points: Circle = 205887192175
        Square = 56256807825
```

Figure 3: Resultado do cálculo do valor aproximado de π

O objetivo de calcular o valor de π foi alcançado com sucesso.

Exercício 3

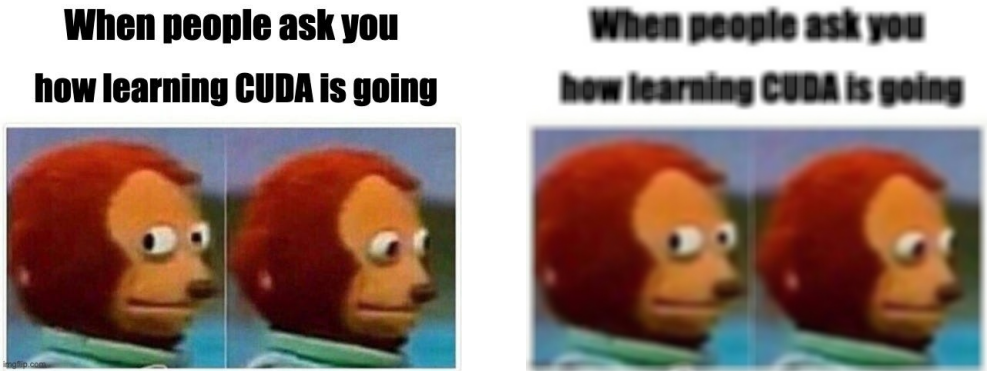
O exercício 3 pede para desenvolver um *kernel* CUDA que permita fazer o *blur* de uma imagem.

Para a alínea A, onde não é necessária nenhuma otimização, desenvolvi o seguinte *kernel*:

```
__global__ void blurKernel(unsigned char* in, unsigned char* out,
int width, int height, int num_channel)
{
    // ===== Pixel Variables
    int pixSum, numPixels;

    // ===== Global Pixel Position
    int col_global = blockIdx.x * blockDim.x + threadIdx.x;
    int row_global = blockIdx.y * blockDim.y + threadIdx.y;

    // ===== Check if pixel is inside image
    if(col_global > -1 && col_global < width &&
        row_global > -1 && row_global < height )
    {
        // ===== Iterate over all channels
        for(int channel = 0; channel < num_channel; channel++)
        {
            // ===== Initialize Pixel Variables
            pixSum = 0;
            numPixels = 0;

            // ===== Iterate over row_global
            for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE + 1;
                ++blurRow)
            {
```

```
                    // ===== Iterate over column
                    for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE + 1;
                        ++blurCol)
                    {
                        // ===== Current Pixel Position
                        int curRow = row_global + blurRow;
                        int curCol = col_global + blurCol;

                        // ===== Check if pixel is inside filter kernel
                        if(curRow > -1 && curRow < height && curCol > -1
                            && curCol < width)
                        {
                            // ===== Add Pixel Value
                            pixSum += in[curRow * width * num_channel
                                + curCol * num_channel + channel];
                            numPixels++;
                        }
                    }
                }

                // ===== Save Pixel Value
                out[row_global * width * num_channel
                    + col_global * num_channel + channel]
                    = (unsigned char)(pixSum/numPixels);
            }
        }
    }
```

Da execução deste programa com um *kernel* de blur 5 por 5 resultaram as figuras 4 e 5.



(a) Original (b) Processada

Figure 4: Resultado alínea A - Imagem 1

(a) Original (b) Processada

Figure 5: Resultado alínea A - Imagem 2

Já para a alínea B, que pedia que pedia para recorrer ao uso de *shared memory*, desenvolvi o seguinte *kernel*:

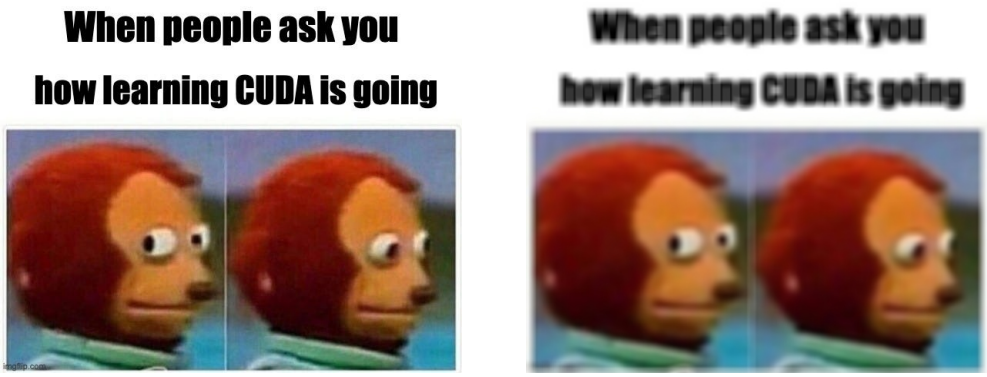
```
__global__ void blurKernel(unsigned char* in, unsigned char* out,
int width, int height, int num_channel)
{
    // ===== Global Pixel Position
    int idx_Global = blockIdx.x * blockDim.x + threadIdx.x;

    // ===== Shared Memory
    __shared__ unsigned char shared_Image[TILE_SIZE*TILE_SIZE]
    [BLUR_SIZE*BLUR_SIZE];

    // ===== Work on each color channel
    for (int channel = 0; channel < num_channel; channel++)
    {
        // ===== Shared Memory Position
        int local_row = 0, local_col = 0, count = 0;

        // ===== Horizontal offset from Global Pixel
        for (int i = -BLUR_SIZE / 2; i <= BLUR_SIZE / 2; i++)
        {
            // ===== Vertical offset from Global Pixel
            for (int j = -BLUR_SIZE / 2; j <= BLUR_SIZE / 2; j++)
            {
```

```
                    // ===== Out of Bounds of Image
                    if (idx_Global + i*width + j < 0 ||
                        idx_Global + i*width + j >= width*height)
                    {
                        shared_Image[threadIdx.x]
                                    [local_row*BLUR_SIZE + local_col] = 0;
                    }
                    // ===== In Bounds of Image
                    else
                        shared_Image[threadIdx.x]
                                    [local_row*BLUR_SIZE + local_col]
                                    = (unsigned char)in[(idx_Global +
                                    i*width + j)
                                    * num_channel + channel];
                        count++;

                    // ===== Update Local Position (in kernel)
                    local_col++;
                }
                // ===== Update Local Position (in kernel)
                local_row++;
                local_col = 0;
            }
            __syncthreads();

            // ===== Blur Pixel
            int sum = 0;
            for (int i = 0; i < BLUR_SIZE; i++)
            {
                for (int j = 0; j < BLUR_SIZE; j++)
                {
                    sum += shared_Image[threadIdx.x][i*BLUR_SIZE + j];
                }
            }
            __syncthreads();

            // ===== Save Blurred Pixel
            out[idx_Global * num_channel + channel] =
                (unsigned char)(sum/count);
            __syncthreads();
        }
    }
```

Para correr o programa, basta correr o script bash associado.

```
    # -> Run
    ./Lab6_ex2.sh
```

Da execução deste programa com um *kernel* de blur 5 por 5 resultaram as figuras 6 e 7.



(a) Original (b) Processada

Figure 6: Resultado alínea B - Imagem 1

(a) Original (b) Processada

Figure 7: Resultado alínea B - Imagem 2

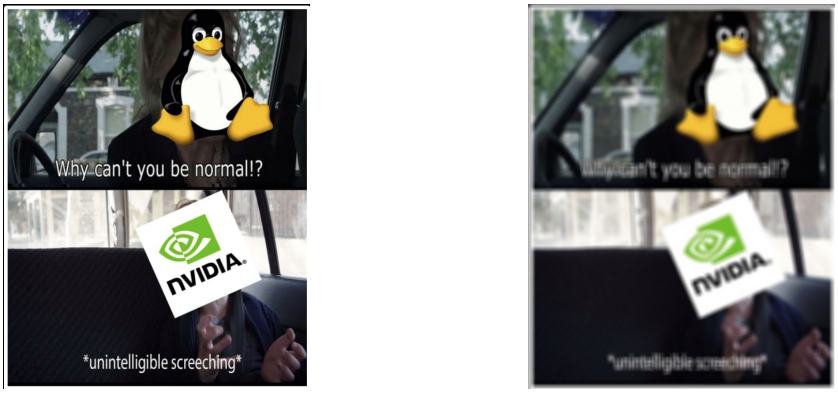
Como é possível constatar pelos resultados, o programa criado provou ser bem sucedido.

Anexo

Mais imagens (*why not?*).

(a) Original (b) Processada

Figure 8: Imagem 3



(a) Original (b) Processada

Figure 9: Imagem 4

(a) Original (b) Processada

Figure 10: Imagem 5

(a) Original

(b) Processada

Figure 11: Imagem 6

(a) Original

(b) Processada

Figure 12: Imagem 7

(a) Original

(b) Processada

Figure 13: Imagem 8

(a) Original (b) Processada

Figure 14: Imagem 9

(a) Original (b) Processada

Figure 15: Imagem 10

(a) Original (b) Processada

Figure 16: Imagem 11