Computo Paralelo

Tarea 2

Rubén Pérez Palacios Lic. Computación Matemática
Profesor: Dr. Francisco Javier Hernández López

9 de marzo de 2022

Reportes

Se explica la solución de los ejercicios, así como la implementación en secuencial, paralelo en cpu y en gpu.

Para todas las soluciones se uso la siguiente librería para el manejo de imagenes:

```
        #include <opencv2/highgui/highgui.hpp>
        using namespace cv;
        cv::Mat frame;//almacenar imagenes
        imread();//leer imagenes
        frame.create(/*...*/);//alocar memoria para una cv::Mat
```

Para la paralelización en cpu se usola siguiente librería e instrucción:

```
        #include <omp.h>
        omp_set_num_threads(8);//cantidad de hilos a paralelizar
```

Para la paralelización en gpu se usola siguiente librería e instrucción:

```
        #include <cuda_runtime.h>
        #include <device_launch_parameters.h>
        cudaSetDevice(0);
        cudaMemcpy(/*...*/);//copiar memoria entre la gpu y algo mas
        cudaMalloc(/*...*/);//alocar memoria en la gpu
        cudaDeviceSynchronize();//esperar a que la gpu termine de ejecutar
↪   kernels.
        cudaFree(/*...*/);//desalojar memoria en la gpu
```

Ejercicio 1

Descripción

Combinación de dos imágenes usando una máscara (alpha matting).

Solución

Para cada pixel de la imagen resultado se realizo la operación indicada:

$$Image\_Res[i][j] = Image\_Alpha[i][j] * Image\_source\_1[i][j] + (Image\_Alpha[i][j] - 1) * Image\_source\_2[i][j],$$

donde

$$Image\_Alpha[i][j] = \begin{cases} 1 & Image\_Mask[i][j] > 255/2 \\ 0 & \text{Si no} \end{cases}.$$

Cuyo código es, con $idx = i * columns + j$:

```
        void AlphaMatting(Mat &frame_source_1, Mat &frame_source_2, Mat
↪   &frame_mask, Mat &frame_result, int N)
        {
                for(int idx = 0; idx < N; idx++)
                {
                        int alpha = (frame_mask.data[idx] > 255 / 2) ? 1 : 0;
                        frame_result.data[idx * 3 + 0] = frame_source_1.data[idx *
↪   3 + 0] * alpha + (1 - alpha) * frame_source_2.data[idx * 3 + 0];
                        frame_result.data[idx * 3 + 1] = frame_source_1.data[idx *
↪   3 + 1] * alpha + (1 - alpha) * frame_source_2.data[idx * 3 + 1];
                        frame_result.data[idx * 3 + 2] = frame_source_1.data[idx *
↪   3 + 2] * alpha + (1 - alpha) * frame_source_2.data[idx * 3 + 2];
                }
        }
```

Paralelización

1. CPU:

   Se uso *omp parallel for* con la directiva *default(shared)* ya que las variables de iteración fueron declaradas localmente y todas puedan acceder a las matrices por multiplicar y al resultado.

```
        void AlphaMatting(Mat &frame_source_1, Mat &frame_source_2, Mat
↪   &frame_mask, Mat &frame_result, int N)
        {
                #pragma omp parallel for default(shared)
                for(int idx = 0; idx < N; idx++)
                //...
        }
```

2. GPU:

   Para ello se hizo la alocación y copias de memoria para las imagenes a combinar, la imagen de la mascara y la imagen del resultado:

```
        cudaMalloc((void**)&image_source_1, N * sizeof(uchar3));
        cudaMalloc((void**)&image_source_2, N * sizeof(uchar3));
        cudaMalloc((void**)&image_result, N * sizeof(uchar3));
        cudaMalloc((void**)&image_mask, N * sizeof(uchar));
        cudaMemcpy(image_source_1, frame_source_1.data, N * sizeof(uchar3),
↪   cudaMemcpyHostToDevice);
        cudaMemcpy(image_source_2, frame_source_2.data, N * sizeof(uchar3),
↪   cudaMemcpyHostToDevice);
        cudaMemcpy(image_mask, frame_mask.data, N * sizeof(uchar),
↪   cudaMemcpyHostToDevice);
```

   Después se uso una función kernel para resolver el problema, de tipo *__global__*, con número de bloques y tamaño de malla

```
        int threads = 512, grid = divUp(N, threads);
```

   la cual es

```
        __global__ void AlphaMatting(uchar3* image_source_1, uchar3*
↪   image_source_2, uchar* image_mask, uchar3* image_result, int N) {
                int idx = blockIdx.x * blockDim.x + threadIdx.x;
                if (idx < N) {
```

```
                        int alpha = (image_mask[idx] > 255 / 2) ? 1 : 0;
                        image_result[idx].x = image_source_1[idx].x * alpha +
↪  (1 - alpha) * image_source_2[idx].x;
                        image_result[idx].y = image_source_1[idx].y * alpha +
↪  (1 - alpha) * image_source_2[idx].y;
                        image_result[idx].z = image_source_1[idx].z * alpha +
↪  (1 - alpha) * image_source_2[idx].z;
                }
        }
```

Resultado

La imagen resultado es:

Ejercicio 1

Descripción

Detección de bordes en la imagen.

Solución

Primero se cáculo las matrices de derivadas D_x, D_y de cada pixel como se describio en la tarea, usando la convolución de la aproximación de las derivadas con las matrices k_1, k_2, cuyo código es:

```
        const int k[2][3][3] = {
                {
                        {-1, 0, 1},
                        {-2, 0, 2},
                        {-1, 0, 1}
                },
                {
                        {-1, -2, -1},
                        {0, 0, 0},
                        {1, 2, 1}
                }
        };
        void Derivates(uchar* image, int* Dx, int* Dy, int imageW, int imageH, int
↪  N)
        {
                int index, index1;
                for (int idx = 1; idx < imageW - 1; idx++)
                {
                        for (int idy = 1; idy < imageH - 1; idy++)
                        {
                                index = (idy)*imageW + (idx);
                                Dx[index] = 0;
                                Dy[index] = 0;
                                for (int k_i = -1; k_i < 2; k_i++)
                                {
                                        for (int k_j = -1; k_j < 2; k_j++)
                                        {
                                                index1 = (idy + k_i) * imageW +
↪  (idx + k_j);
                                                Dx[index] += image[index1] *
↪  k[0][k_i + 1][k_j + 1];
                                                Dy[index] += image[index1] *
↪  k[1][k_i + 1][k_j + 1];
```

```
                                        }
                                }
                        }
                }
        }
```

Y después se caculo la "norma" de las matrices D_x, D_y para cada pixel para obtener la matriz MG (hay que tener cuidado porque puede ser que este número sea mayor a 255 por lo que se debe topar) y se usa un umbral de $255/2$ en esta para obtener la matriz MGT, con el código:

```
        void BordersinImage(int* Dx, int* Dy, uchar* MG, uchar* MGT, int imageW,
↪   int imageH, int N)
        {
                int index;
                for (int idx = 0; idx < imageW; idx++)
                {
                        for (int idy = 0; idy < imageH; idy++)
                        {
                                index = (idy)*imageW + (idx);
                                if (0 < idx && idx < imageW - 1 && 0 < idy && idy
↪   < imageH - 1)
                                        MG[index] = min(sqrt(Dx[index] * Dx[index]
↪   + Dy[index] * Dy[index]), 255.0);
                                else
                                        MG[index] = 0;
                                MGT[index] = (MG[index] > 255 / 2 ? 255 : 0);
                        }
                }
        }
```

Paralelización

1. CPU:

   Para la paralelización primero se uso $omp\ parallel\ for\ collapse(2)$, con las directivas $private(index, index1)$ porque cada iteración tiene su propio indice y $default(shared)$ ya que las variables de iteración fueron declaradas localmente y todas puedan accerder a las matrices por multiplicar y al resultado, así los primeros dos for anidados se paralelizaran.

```
        void Derivates(uchar* image, int* Dx, int* Dy, int imageW, int imageH,
↪  int N)
        {
                int index, index1;
                #pragma omp parallel for collapse(2) default(shared)
↪  private(index, index1)
                for (int idx = 1; idx < imageW - 1; idx++)
                //...
        }
```

   Después se uso $omp\ parallel\ for\ collapse(2)$, con las directivas $private(index)$ porque cada iteración tiene su propio indice y $default(shared)$ ya que las variables de iteración fueron declaradas localmente y todas puedan accerder a las matrices por multiplicar y al resultado, así los primeros dos for anidados se paralelizaran.

```
        void BordersinImage(int* Dx, int* Dy, uchar* MG, uchar* MGT, int
↪  imageW, int imageH, int N)
        {
                int index;
                #pragma omp parallel for collapse(2) default(shared)
↪  private(index)
                for (int idx = 0; idx < imageW; idx++)
                //...
        }
```

2. GPU:

   Para ello se hizo la alocación y copias de memoria para la imagen fuente, las derivadas de la imagen, y las imágenes MG y MGT:

```
        cudaMalloc((void**)&image, N * sizeof(uchar));
        cudaMalloc((void**)&Dx, N * sizeof(int));
        cudaMalloc((void**)&Dy, N * sizeof(int));
        cudaMalloc((void**)&MG, N * sizeof(uchar));
```

```
        cudaMalloc((void**)&MGT, N * sizeof(uchar));
        cudaMemcpy(image, frame.data, N * sizeof(uchar),
↪  cudaMemcpyHostToDevice);
```

Después se uso dos función kernel para calcular las derivadas y las imagenes finales, de tipo *__global__*, con número de bloques y tamaño de malla

```
                dim3 threads(16, 16, 1), grid(divUp(imageW, 16), divUp(imageH,
↪  16), 1);
```

la cuales son

```
        __global__ void Derivates(uchar* image, int* Dx, int* Dy, int imageW,
↪  int imageH, int N)
        {
                int idx = blockIdx.x * blockDim.x + threadIdx.x;
                int idy = blockIdx.y * blockDim.y + threadIdx.y;
                int index, index1;

                if (0 < idx && idx < imageW - 1 && 0 < idy && idy < imageH -
↪  1)
                {
                        index = (idy) * imageW + (idx);
                        Dx[index] = 0;
                        Dy[index] = 0;
                        for (int k_i = -1; k_i < 2; k_i++)
                        {
                                for (int k_j = -1; k_j < 2; k_j++)
                                {
                                        index1 = (idy + k_i) * imageW + (idx +
↪  k_j);
                                        Dx[index] += image[index1] * k[0][k_i
↪  + 1][k_j + 1];
                                        Dy[index] += image[index1] * k[1][k_i
↪  + 1][k_j + 1];
                                }
                        }
                }
        }
```

```
        __global__ void BordersinImage(int* Dx, int* Dy, uchar* MG, uchar*
↪  MGT, int imageW, int imageH, int N)
        {
                int idx = blockIdx.x * blockDim.x + threadIdx.x;
                int idy = blockIdx.y * blockDim.y + threadIdx.y;
                int index = (idy)*imageW + (idx);

                if (0 < idx && idx < imageW - 1 && 0 < idy < imageH - 1)
                        MG[index] = fminf(sqrtf(Dx[index] * Dx[index] +
↪  Dy[index] * Dy[index]), 255);
                else if (0 <= idx && idx < imageW && 0 <= idy && idy < imageH)
                        MG[index] = 0;
                MGT[index] = (MG[index] > 255 / 2 ? 255 : 0);
        }
```

Resultados

La imagenes resultados son:

Figura 1: MG

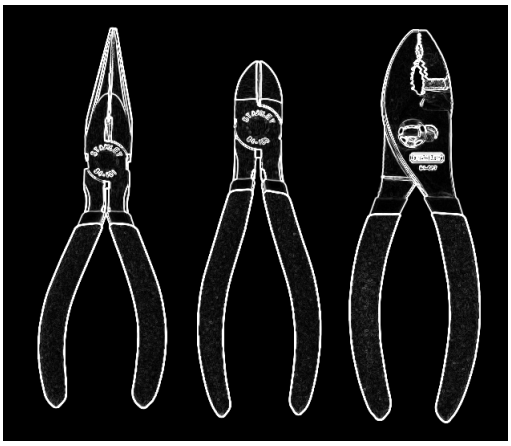

Figura 2: MGT