**REPORTE DE PRÁCTICA**

**IDENTIFICACIÓN DE LA PRÁCTICA**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Práctica** | **11** | **Nombre de la práctica** | | **Operaciones de pixel en imágenes RGB** |
| **Fecha** | **06/10/2021** | **Nombre del profesor** | | **Alma Nayeli Rodríguez Vázquez** |
| **Nombre del estudiante** | | | **Mariana Ávalos Arce** | |

**OBJETIVO**

|  |
| --- |
| El objetivo de esta práctica consiste en implementar un kernel para modificar el brillo y otro para modificar el contraste de una imagen en formato RGB. |

**PROCEDIMIENTO**

|  |
| --- |
| Realiza la implementación siguiendo estas instrucciones. |
| Realiza un programa en C/C++ utilizando CUDA en el que implementes un kernel para calcular el complemento de una imagen, otro para modificar el brillo de una imagen RGB y otro kernel para modificar el contraste. Para ello, considera los siguientes requerimientos:   * Considera los siguientes prototipos de kernel para el brillo y contraste   \_\_global\_\_ void complemento(uchar\* B, uchar\* G, uchar\* R);  \_\_global\_\_ void brillo(uchar\* B, uchar\* G, uchar\* R, float fb );  \_\_global\_\_ void contraste(uchar\* B, uchar\* G, uchar\* R, float fc);   * Para el lanzamiento de los kernel deberás considerar la siguiente configuración: * Bloques de 32 por 32 hilos en 2D * Una malla de 14 por 24 bloques * Incluir manejo de errores usando la siguiente función:   \_\_host\_\_ void check\_CUDA\_Error(const char\* mensaje) |

**IMPLEMENTACIÓN**

|  |
| --- |
| Agrega el código de tu implementación aquí. |
| #include "cuda\_runtime.h"  #include "device\_launch\_parameters.h"  #include <stdio.h>  #include <stdlib.h>  #include <opencv2/opencv.hpp>  \_\_host\_\_ void checkCUDAError(const char\* msg) {  cudaError\_t error;  cudaDeviceSynchronize();  error = cudaGetLastError();  if (error != cudaSuccess) {  printf("ERROR %d: %s (%s)\n", error, cudaGetErrorString(error), msg);  }  }  \_\_global\_\_ void complement(uchar\* R, uchar\* G, uchar\* B) {  // locate my current block row  int threads\_per\_block = blockDim.x \* blockDim.y;  int threads\_per\_row = threads\_per\_block \* gridDim.x;  int row\_offset = threads\_per\_row \* blockIdx.y;  // locate my current block column  int block\_offset = blockIdx.x \* threads\_per\_block;  int threadId\_inside = blockDim.x \* threadIdx.y + threadIdx.x;  int gId = row\_offset + block\_offset + threadId\_inside;  R[gId] = 255 - R[gId];  G[gId] = 255 - G[gId];  B[gId] = 255 - B[gId];  }  \_\_global\_\_ void contrast(uchar\* R, uchar\* G, uchar\* B, float fc) {  // locate my current block row  int threads\_per\_block = blockDim.x \* blockDim.y;  int threads\_per\_row = threads\_per\_block \* gridDim.x;  int row\_offset = threads\_per\_row \* blockIdx.y;  // locate my current block column  int block\_offset = blockIdx.x \* threads\_per\_block;  int threadId\_inside = blockDim.x \* threadIdx.y + threadIdx.x;  int gId = row\_offset + block\_offset + threadId\_inside;  if (fc \* R[gId] > 255) { R[gId] = 255; }  else { R[gId] = fc \* R[gId]; }  if (fc \* G[gId] > 255) { G[gId] = 255; }  else { G[gId] = fc \* G[gId]; }  if (fc \* B[gId] > 255) { B[gId] = 255; }  else { B[gId] = fc \* B[gId]; }  }  \_\_global\_\_ void brightness(uchar\* R, uchar\* G, uchar\* B, float fb) {  // locate my current block row  int threads\_per\_block = blockDim.x \* blockDim.y;  int threads\_per\_row = threads\_per\_block \* gridDim.x;  int row\_offset = threads\_per\_row \* blockIdx.y;  // locate my current block column  int block\_offset = blockIdx.x \* threads\_per\_block;  int threadId\_inside = blockDim.x \* threadIdx.y + threadIdx.x;  int gId = row\_offset + block\_offset + threadId\_inside;  if (fb >= 0) {  if (fb + R[gId] > 255) { R[gId] = 255; }  else { R[gId] = fb + R[gId]; }  if (fb + G[gId] > 255) { G[gId] = 255; }  else { G[gId] = fb + G[gId]; }  if (fb + B[gId] > 255) { B[gId] = 255; }  else { B[gId] = fb + B[gId]; }  }  if (fb < 0) {  if (fb + R[gId] < 0) { R[gId] = 0; }  else { R[gId] = fb + R[gId]; }  if (fb + G[gId] < 0) { G[gId] = 0; }  else { G[gId] = fb + G[gId]; }  if (fb + B[gId] < 0) { B[gId] = 0; }  else { B[gId] = fb + B[gId]; }  }  }  using namespace cv;  int main() {  Mat img = imread("antenaRGB.jpg");  const int R = img.rows;  const int C = img.cols;  Mat imgComp(img.rows, img.cols, img.type());  Mat imgCont(img.rows, img.cols, img.type());  Mat imgBright(img.rows, img.cols, img.type());  uchar\* host\_r, \* host\_g, \* host\_b, \* dev\_r1, \* dev\_g1, \* dev\_b1, \* dev\_r2, \* dev\_g2, \* dev\_b2, \* dev\_r3, \* dev\_g3, \* dev\_b3;  host\_r = (uchar\*)malloc(sizeof(uchar) \* R \* C);  host\_g = (uchar\*)malloc(sizeof(uchar) \* R \* C);  host\_b = (uchar\*)malloc(sizeof(uchar) \* R \* C);  cudaMalloc((void\*\*)&dev\_r1, sizeof(uchar) \* R \* C);  checkCUDAError("Error at malloc dev\_r1");  cudaMalloc((void\*\*)&dev\_g1, sizeof(uchar) \* R \* C);  checkCUDAError("Error at malloc dev\_g1");  cudaMalloc((void\*\*)&dev\_b1, sizeof(uchar) \* R \* C);  checkCUDAError("Error at malloc dev\_b1");  cudaMalloc((void\*\*)&dev\_r2, sizeof(uchar) \* R \* C);  checkCUDAError("Error at malloc dev\_r2");  cudaMalloc((void\*\*)&dev\_g2, sizeof(uchar) \* R \* C);  checkCUDAError("Error at malloc dev\_g2");  cudaMalloc((void\*\*)&dev\_b2, sizeof(uchar) \* R \* C);  checkCUDAError("Error at malloc dev\_b2");  cudaMalloc((void\*\*)&dev\_r3, sizeof(uchar) \* R \* C);  checkCUDAError("Error at malloc dev\_r3");  cudaMalloc((void\*\*)&dev\_g3, sizeof(uchar) \* R \* C);  checkCUDAError("Error at malloc dev\_g3");  cudaMalloc((void\*\*)&dev\_b3, sizeof(uchar) \* R \* C);  checkCUDAError("Error at malloc dev\_b3");  // matrix as vector  for (int i = 0; i < R; i++) {  for (int j = 0; j < C; j++) {  Vec3b pix = img.at<Vec3b>(i, j);  host\_r[i \* C + j] = pix[2];  host\_g[i \* C + j] = pix[1];  host\_b[i \* C + j] = pix[0];  }  }  cudaMemcpy(dev\_r1, host\_r, sizeof(uchar) \* R \* C, cudaMemcpyHostToDevice);  checkCUDAError("Error at memcpy host\_r -> dev\_r1");  cudaMemcpy(dev\_g1, host\_g, sizeof(uchar) \* R \* C, cudaMemcpyHostToDevice);  checkCUDAError("Error at memcpy host\_r -> dev\_g1");  cudaMemcpy(dev\_b1, host\_b, sizeof(uchar) \* R \* C, cudaMemcpyHostToDevice);  checkCUDAError("Error at memcpy host\_r -> dev\_b1");  cudaMemcpy(dev\_r2, host\_r, sizeof(uchar) \* R \* C, cudaMemcpyHostToDevice);  checkCUDAError("Error at memcpy host\_r -> dev\_r2");  cudaMemcpy(dev\_g2, host\_g, sizeof(uchar) \* R \* C, cudaMemcpyHostToDevice);  checkCUDAError("Error at memcpy host\_r -> dev\_g2");  cudaMemcpy(dev\_b2, host\_b, sizeof(uchar) \* R \* C, cudaMemcpyHostToDevice);  checkCUDAError("Error at memcpy host\_r -> dev\_b2");  cudaMemcpy(dev\_r3, host\_r, sizeof(uchar) \* R \* C, cudaMemcpyHostToDevice);  checkCUDAError("Error at memcpy host\_r -> dev\_r3");  cudaMemcpy(dev\_g3, host\_g, sizeof(uchar) \* R \* C, cudaMemcpyHostToDevice);  checkCUDAError("Error at memcpy host\_r -> dev\_g3");  cudaMemcpy(dev\_b3, host\_b, sizeof(uchar) \* R \* C, cudaMemcpyHostToDevice);  checkCUDAError("Error at memcpy host\_r -> dev\_b3");  dim3 block(32, 32);  dim3 grid(C / 32, R / 32);  complement << < grid, block >> > (dev\_r1, dev\_g1, dev\_b1);  cudaDeviceSynchronize();  checkCUDAError("Error at kernel complement");  cudaMemcpy(host\_r, dev\_r1, sizeof(uchar) \* R \* C, cudaMemcpyDeviceToHost);  checkCUDAError("Error at memcpy host\_r <- dev\_r1");  cudaMemcpy(host\_g, dev\_g1, sizeof(uchar) \* R \* C, cudaMemcpyDeviceToHost);  checkCUDAError("Error at memcpy host\_r <- dev\_g1");  cudaMemcpy(host\_b, dev\_b1, sizeof(uchar) \* R \* C, cudaMemcpyDeviceToHost);  checkCUDAError("Error at memcpy host\_r <- dev\_b1");  for (int i = 0; i < R; i++) {  for (int j = 0; j < C; j++) {  imgComp.at<Vec3b>(i, j)[0] = host\_b[i \* C + j];  imgComp.at<Vec3b>(i, j)[1] = host\_g[i \* C + j];  imgComp.at<Vec3b>(i, j)[2] = host\_r[i \* C + j];  }  }  contrast << < grid, block >> > (dev\_r2, dev\_g2, dev\_b2, 0.5);  cudaDeviceSynchronize();  checkCUDAError("Error at kernel contrast");  cudaMemcpy(host\_r, dev\_r2, sizeof(uchar) \* R \* C, cudaMemcpyDeviceToHost);  checkCUDAError("Error at memcpy host\_r <- dev\_r2");  cudaMemcpy(host\_g, dev\_g2, sizeof(uchar) \* R \* C, cudaMemcpyDeviceToHost);  checkCUDAError("Error at memcpy host\_r <- dev\_g2");  cudaMemcpy(host\_b, dev\_b2, sizeof(uchar) \* R \* C, cudaMemcpyDeviceToHost);  checkCUDAError("Error at memcpy host\_r <- dev\_b2");  for (int i = 0; i < R; i++) {  for (int j = 0; j < C; j++) {  imgCont.at<Vec3b>(i, j)[0] = host\_b[i \* C + j];  imgCont.at<Vec3b>(i, j)[1] = host\_g[i \* C + j];  imgCont.at<Vec3b>(i, j)[2] = host\_r[i \* C + j];  }  }  brightness << < grid, block >> > (dev\_r3, dev\_g3, dev\_b3, 100);  cudaDeviceSynchronize();  checkCUDAError("Error at kernel brightness");  cudaMemcpy(host\_r, dev\_r3, sizeof(uchar) \* R \* C, cudaMemcpyDeviceToHost);  checkCUDAError("Error at memcpy host\_r <- dev\_r3");  cudaMemcpy(host\_g, dev\_g3, sizeof(uchar) \* R \* C, cudaMemcpyDeviceToHost);  checkCUDAError("Error at memcpy host\_r <- dev\_g3");  cudaMemcpy(host\_b, dev\_b3, sizeof(uchar) \* R \* C, cudaMemcpyDeviceToHost);  checkCUDAError("Error at memcpy host\_r <- dev\_b3");  for (int i = 0; i < R; i++) {  for (int j = 0; j < C; j++) {  imgBright.at<Vec3b>(i, j)[0] = host\_b[i \* C + j];  imgBright.at<Vec3b>(i, j)[1] = host\_g[i \* C + j];  imgBright.at<Vec3b>(i, j)[2] = host\_r[i \* C + j];  }  }  imshow("Image", img);  imshow("Image Complement", imgComp);  imshow("Image Contrast", imgCont);  imshow("Image Brightness", imgBright);  waitKey(0);  free(host\_r);  free(host\_g);  free(host\_b);  cudaFree(dev\_r1);  cudaFree(dev\_g1);  cudaFree(dev\_b1);  cudaFree(dev\_r2);  cudaFree(dev\_g2);  cudaFree(dev\_b2);  cudaFree(dev\_r3);  cudaFree(dev\_g3);  cudaFree(dev\_b3);  return 0;  } |

**RESULTADOS**

|  |  |
| --- | --- |
| Agrega las imágenes obtenidas en los espacios indicados. | |
|  |  |
| Imagen original en RGB | Complemento |
|  |  |
| Imagen con más brillo fb=100 | Imagen con menos brillo fb=-100 |
|  |  |
| Imagen con más contraste fc=1.5 | Imagen con menos contraste fc=0.5 |

**CONCLUSIONES**

|  |
| --- |
| Escribe tus observaciones y conclusiones. |
| Esta práctica abarcó bastantes temas: primero, refuerza la idea de utilizar configuraciones de bloques múltiplo de 32 para que la imagen se procese sin hilos desperdiciados; después, el utilizar una imagen RGB requería mayor memoria, pero tampoco se trataba de hacer lo mismo 3 veces, sino que podíamos ahorrarnos el hacer nueve vectores en el host si utilizábamos el mismo para copiar los resultados del device, aunque eso significaba que cada kernel debía ejecutarse sincronizadamente para evitar que el mismo arreglo sea accedido por múltiples hilos. |