**REPORTE DE PRÁCTICA**

**IDENTIFICACIÓN DE LA PRÁCTICA**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Práctica** | **13** | **Nombre de la práctica** | | **Validación del kernel** |
| **Fecha** | **20/10/2021** | **Nombre del profesor** | | **Alma Nayeli Rodríguez Vázquez** |
| **Nombre del estudiante** | | | **Mariana Ávalos Arce** | |

**OBJETIVO**

|  |
| --- |
| El objetivo de esta práctica consiste en implementar la validación de un kernel para verificar el procesamiento correcto de datos masivos. |

**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 RGB. El kernel deberá ser verificado utilizando la validación de kernel. Para ello deberás atender los siguientes requerimientos:   * La operación complemento está definida como:      * Bloques de 768 x 1 hilos * Una malla de 1 x 448 x 3 bloques * El kernel para el cálculo del complemento como:   \_\_global\_\_ void complementoGPU(uchar\* vectorImg)   * Implementa la función complemento en el host:   \_\_host\_\_ void complementoCPU(Mat\* Img)   * Incluir validación del kernel usando la siguiente función:   \_\_host\_\_ void validacionKernel(Mat img1, Mat img2)   * 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>  using namespace cv;  \_\_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\* RGB) {  // locate my current block row  int threads\_per\_block = blockDim.x;  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;  // locate my current grid row  int thread\_per\_grid = (gridDim.x \* gridDim.y \* threads\_per\_block);  int gridOffset = blockIdx.z \* thread\_per\_grid;  int gId = gridOffset + row\_offset + block\_offset + threadIdx.x;  RGB[gId] = 255 - RGB[gId];  }  \_\_host\_\_ void complementCPU(Mat\* original, Mat\* comp) {  for (int i = 0; i < original->rows; i++) {  for (int j = 0; j < original->cols; j++) {  comp->at<Vec3b>(i, j)[0] = 255 - original->at<Vec3b>(i, j)[0];  comp->at<Vec3b>(i, j)[1] = 255 - original->at<Vec3b>(i, j)[1];  comp->at<Vec3b>(i, j)[2] = 255 - original->at<Vec3b>(i, j)[2];  }  }  }  \_\_host\_\_ bool validationKernel(Mat img1, Mat img2) {  Vec3b\* pImg1, \* pImg2;  for (int k = 0; k < 3; k++) {  for (int i = 0; i < img1.rows; i++) {  pImg1 = img1.ptr<Vec3b>(i);  pImg2 = img2.ptr<Vec3b>(i);  for (int j = 0; j < img1.cols; j++) {  if (pImg1[j][k] != pImg2[j][k]) {  printf("Error at kernel validation\n");  return true;  }  }  }  }  printf("Kernel validation successful\n");  return false;  }  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 imgCompCPU(img.rows, img.cols, img.type());  uchar\* host\_rgb, \* dev\_rgb;  host\_rgb = (uchar\*)malloc(sizeof(uchar) \* R \* C \* 3);  cudaMalloc((void\*\*)&dev\_rgb, sizeof(uchar) \* R \* C \* 3);  checkCUDAError("Error at malloc dev\_r1");  // matrix as vector  for (int k = 0; k < 3; k++) {  for (int i = 0; i < R; i++) {  for (int j = 0; j < C; j++) {  Vec3b pix = img.at<Vec3b>(i, j);  host\_rgb[i \* C + j + (k \* R \* C)] = pix[k];  }  }  }  cudaMemcpy(dev\_rgb, host\_rgb, sizeof(uchar) \* R \* C \* 3, cudaMemcpyHostToDevice);  checkCUDAError("Error at memcpy host\_rgb -> dev\_rgb");  //dim3 block(32, 32);  //dim3 grid(C / 32, R / 32, 3); // 24 14  dim3 block(C, 1, 1); // 768  dim3 grid(1, R, 3); // 448  complement << < grid, block >> > (dev\_rgb);  cudaDeviceSynchronize();  checkCUDAError("Error at kernel complement");  cudaMemcpy(host\_rgb, dev\_rgb, sizeof(uchar) \* R \* C \* 3, cudaMemcpyDeviceToHost);  checkCUDAError("Error at memcpy host\_rgb <- dev\_rgb");  for (int k = 0; k < 3; k++) {  for (int i = 0; i < R; i++) {  for (int j = 0; j < C; j++) {  imgComp.at<Vec3b>(i, j)[k] = host\_rgb[i \* C + j + (k \* R \* C)];  }  }  }  complementCPU(&img, &imgCompCPU);  bool error = validationKernel(imgCompCPU, imgComp);  if (error) {  printf("Check kernel operations\n");  return 0;  }    imshow("Image", img);  imshow("Image Complement CPU", imgCompCPU);  imshow("Image Complement GPU", imgComp);  waitKey(0);  free(host\_rgb);  cudaFree(dev\_rgb);  return 0;  } |

**RESULTADOS**

|  |  |
| --- | --- |
| Agrega las imágenes obtenidas en los espacios indicados. | |
|  | |
| Imagen original en RGB | |
|  |  |
| Imagen complemento GPU | Imagen complemento CPU |
|  | |
| Imagen de la consola en la que se muestre el mensaje de validación del kernel | |

**CONCLUSIONES**

|  |
| --- |
| Escribe tus observaciones y conclusiones. |
| Esta práctica es útil para separar lo que es el desdoblamiento para transformar la imagen de entrada en un vector, y el desdoblamiento de la configuración para poder acceder a este vector de entrada que tiene el kernel como parámetro. No importa la configuración, una imagen RGB se transforma en un vector lineal de la misma manera, cosa que al principio me confundió. Sólo hay que preocuparse por acceder a estos índices de alguna forma, y para eso se utiliza el global ID. La validación del kernel también es útil cuando uno quiere comparar con lo que en teoría debería ser. |