UNIVERSIDAD PANAMERICANA

Universidad Panamericana

Campus Guadalajara

Escuela de ingenierías

Fundamentos de programación en paralelo

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:
  $$I_p(x,y) = 255 - I(x,y)$$
- 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.

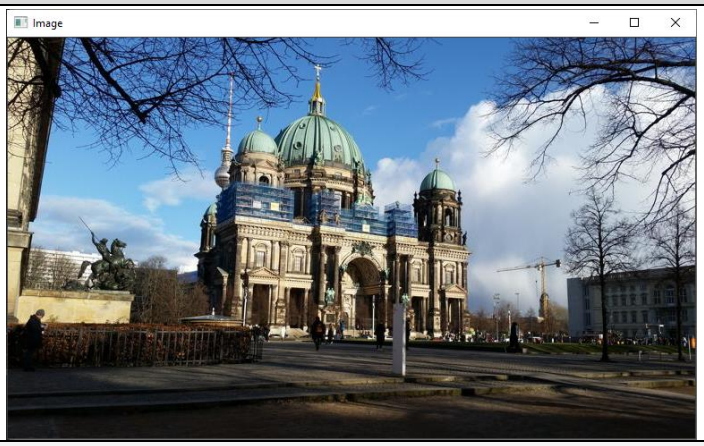

Imagen original en RGB

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.