UNIVERSIDAD PANAMERICANA

Universidad Panamericana

Campus Guadalajara

Escuela de ingenierías

Fundamentos de programación en paralelo

REPORTE DE PRÁCTICA

IDENTIFICACIÓN DE LA PRÁCTICA

Práctica	**8**	**Nombre de la práctica**	**Operaciones de vecindad con matrices**
Fecha	**24/09/2021**	**Nombre del profesor**	**Alma Nayeli Rodríguez Vázquez**
Nombre del estudiante		**Mariana Ávalos Arce**	

OBJETIVO

El objetivo de esta práctica consiste en realizar la operación de vecindad promedio utilizando matrices.

PROCEDIMIENTO

Realiza la implementación siguiendo estas instrucciones.

Realiza un programa en C/C++ utilizando CUDA en el que implementes un kernel que calcule los valores de una matriz B considerando el promedio de los vecinos en base a la información de una matriz A atendiendo los siguientes requerimientos:

- 36 hilos
- Un bloque 2D de 6 por 6 hilos
- Las matrices A y B de tamaño 6x6
- La matriz A deberá ser inicializada con valores enteros aleatorios entre 0 y 9
- El kernel debe ser como el siguiente:
  __global__ void operacionVecindad(int* A, int* B)
- Incluir manejo de errores usando la siguiente función:
  __host__ void check_CUDA_Error(const char* mensaje)

IMPLEMENTACIÓN

Agrega el código de tu implementación aquí.

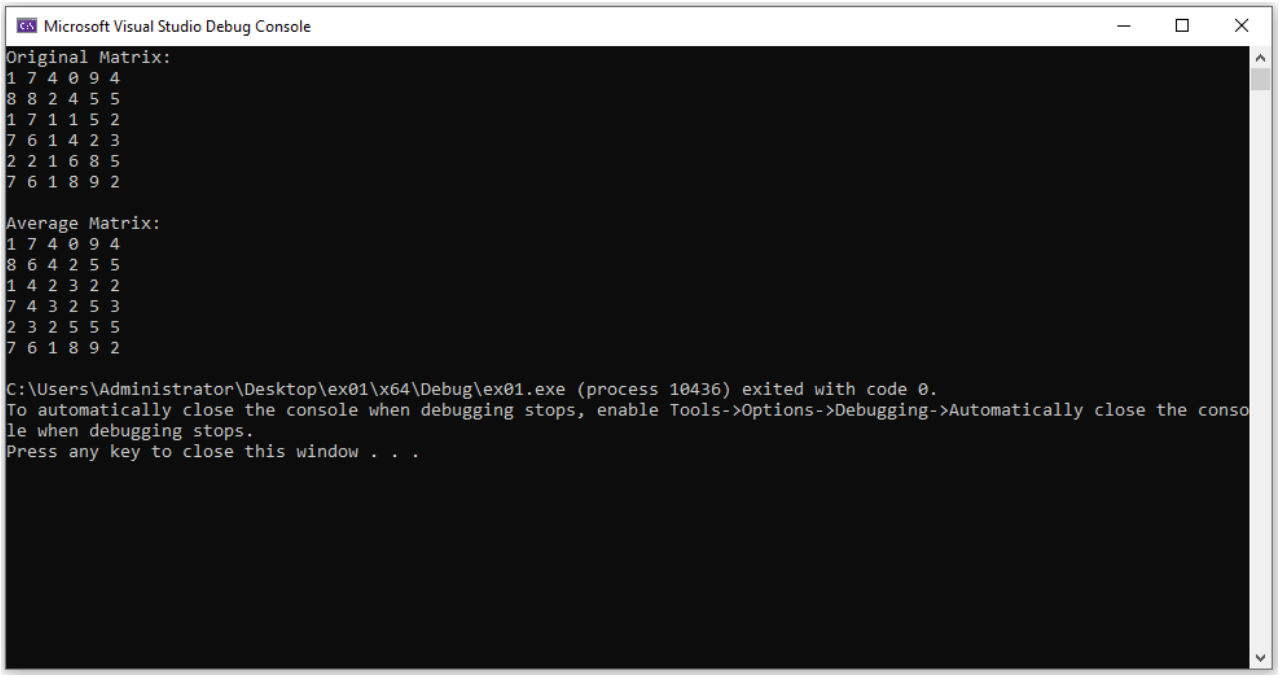
```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>

__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 kernel(int* m, int* r) {
	int gId = threadIdx.x + threadIdx.y * blockDim.x;
	int n1 = gId - 1;
	int n2 = gId + 1;
	int n3 = gId - blockDim.x;
	int n4 = gId + blockDim.x;
```

```
	if (threadIdx.x == 0 || threadIdx.x == (blockDim.x - 1) || threadIdx.y == 0 || threadIdx.y == (blockDim.y - 1)) {
		r[gId] = m[gId];
	}
	else {
		int avg = (m[n1] + m[n2] + m[n3] + m[n4]) / 4;

		r[gId] = avg;
	}
}

int main() {
	const int size = 6;

	int m[size][size] = { 0 };
	int r[size][size] = { 0 };
	int m_vec[size * size] = { 0 };
	int r_vec[size * size] = { 0 };

	int* dev_m, * dev_r;
	cudaMalloc((void**)&dev_m, sizeof(int) * size * size);
	checkCUDAError("Error at cudaMalloc for dev_m");
	cudaMalloc((void**)&dev_r, sizeof(int) * size * size);
	checkCUDAError("Error at cudaMalloc for dev_r");

	for (int i = 0; i < size; i++) {
		for (int j = 0; j < size; j++) {
			m[i][j] = (int)(rand() % 10);
			m_vec[j + i * size] = m[i][j];
		}
	}

	printf("Original Matrix:\n");
	for (int i = 0; i < size; i++) {
		for (int j = 0; j < size; j++) {
			printf("%d ", m[i][j]);
		}
		printf("\n");
	}

	for (int i = 0; i < size * size; i++) {
		//printf("%d ", m_vec[i]);
	}
	printf("\n");

	cudaMemcpy(dev_m, m_vec, sizeof(int) * size * size, cudaMemcpyHostToDevice);
	checkCUDAError("Error at cudaMemcpy Host -> Device");

	dim3 grid(1);
	dim3 block(size, size);
	kernel << < grid, block >> > (dev_m, dev_r);
	checkCUDAError("Error at kernel");

	cudaMemcpy(r_vec, dev_r, sizeof(int) * size * size, cudaMemcpyDeviceToHost);
	checkCUDAError("Error at cudaMemcpy Device -> Host");

	printf("Average Matrix:\n");
	for (int i = 0; i < size; i++) {
		for (int j = 0; j < size; j++) {
			r[i][j] = r_vec[j + i * size];
```

```
                    printf("%d ", r[i][j]);
                }
                printf("\n");
        }

        cudaFree(dev_m);
        cudaFree(dev_r);
}
```

RESULTADOS

Agrega la imagen de la consola con el despliegue de los resultados obtenidos.



CONCLUSIONES

Escribe tus observaciones y conclusiones.

Esta práctica fue bastante útil paa entender y diferenciar entre los datos de entrada y la configuración del kernel: puede que parezcan fáciles de diferenciar, pero al trabajar con matrices se vuelve confuso por todo el desdoblamiento de matrices que hay que hacer. Concluyo pues que el kernel posee datos de entrada que tienen que ser accesados por medio del global ID y este a su vez tiene que desdoblarse debido a la configuración de matriz en el kernel, y además se debe hacer otro procesado para que la entrada se mande como vector. Por lo que se requieren dos procesos para desdoble de matriz: uno para la entrada y otro para el globalID.