**REPORTE DE PRÁCTICA**

**IDENTIFICACIÓN DE LA PRÁCTICA**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Práctica** | **20** | **Nombre de la práctica** | | **Memoria constante** |
| **Fecha** | **15/11/2021** | **Nombre del profesor** | | **Alma Nayeli Rodríguez Vázquez** |
| **Nombre del estudiante** | | | **Mariana Ávalos Arce** | |

**OBJETIVO**

|  |
| --- |
| El objetivo de esta práctica consiste en implementar la traspuesta de una matriz cuadrada utilizando memoria constante con un bloque bidimensional. |

**PROCEDIMIENTO**

|  |
| --- |
| Realiza la implementación siguiendo estas instrucciones. |
| Realiza un programa en C/C++ en CUDA en el que implementes un kernel para trasponer una matriz cuadrada utilizando memoria constante. El kernel deberá ser verificado utilizando la validación de kernel. Para ello deberás atender los siguientes requerimientos:   * El tamaño de la matriz es de NxN con valores enteros cualesquiera, en donde N=32 * Lanzar el kernel utilizando 1 bloque de NxN hilos * Implementar el kernel como:   \_\_global\_\_ void GPU\_traspuesta(int\* vector)   * Implementa la misma función en el host:   \_\_host\_\_ void CPU\_traspuesta(int\* vector)   * Incluir validación del kernel usando la siguiente función:   \_\_host\_\_ void validacion(int result\_CPU, int result\_GPU)   * 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 <iostream>  #define N 32  \_\_constant\_\_ int dev\_A[N \* N];  using namespace std;  \_\_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);  }  }  \_\_host\_\_ void validate(int\* result\_CPU, int\* result\_GPU) {  for (int i = 0; i < N \* N; i++) {  if (\*result\_CPU != \*result\_GPU) {  printf("[FAILED] The results are not equal.\n");  return;  }  }  printf("[SUCCESS] Kernel validation.\n");  return;  }  \_\_host\_\_ void CPU\_transpose(int\* vector, int\* res) {  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++) {  res[(i \* N) + j] = vector[(N \* j) + i];  }  }  }  \_\_global\_\_ void GPU\_transpose(int\* res) {  int gId = threadIdx.x + (blockDim.x \* threadIdx.y);  res[gId] = dev\_A[N \* threadIdx.x + threadIdx.y];  }  \_\_host\_\_ void printMtx(int\* mtx) {  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++) {  cout << mtx[(i \* N) + j] << " ";  }  cout << endl;  }  }  int main() {  int\* dev\_B;  int\* host\_B = (int\*)malloc(sizeof(int) \* N \* N);  int\* cpu\_B = (int\*)malloc(sizeof(int) \* N \* N);  int\* host\_A = (int\*)malloc(sizeof(int) \* N \* N);  cudaMalloc((void\*\*)&dev\_B, sizeof(int) \* N \* N);  checkCUDAError("Error at cudaMalloc: dev\_B");  for (int i = 0; i < N \* N; i++) {  host\_A[i] = i + 1;  }  cudaMemcpyToSymbol(dev\_A, host\_A, sizeof(int) \* N \* N);  checkCUDAError("Error at MemcpyToSymbol");  dim3 grid(1);  dim3 block(N, N);  GPU\_transpose << < grid, block >> > (dev\_B);  checkCUDAError("Error at kernel");  cudaMemcpy(host\_B, dev\_B, sizeof(int) \* N \* N, cudaMemcpyDeviceToHost);  checkCUDAError("Error at Memcpy host\_B <- dev\_B");  CPU\_transpose(host\_A, cpu\_B);  printf("Input: \n");  printMtx(host\_A);  printf("CPU: \n");  printMtx(cpu\_B);  printf("GPU: \n");  printMtx(host\_B);  validate(cpu\_B, host\_B);  free(host\_B);  free(cpu\_B);  free(host\_A);  cudaFree(dev\_B);  return 0;  } |

**RESULTADOS**

|  |
| --- |
| Agrega las imágenes obtenidas en los espacios indicados. |
|  |
| Código del kernel |
|  |
| Imagen de la consola en la que se muestren los resultados y el mensaje de validación del kernel |

**CONCLUSIONES**

|  |
| --- |
| Escribe tus observaciones y conclusiones. |
| En esta práctica es bastante evidente cómo las operaciones necesarias para el cálculo de transpuesta son exactamente las mismas en GPU y CPU, sólo que en el CPU se utilizan los ciclos para generar y recorrer cada índice i,j y usar estos para transponer. En cambio, para GPU, estos índices ya vienen incluidos al lanzar los hilos en paralelo, por lo que se omiten los ciclos y se acelera el proceso. |