

REPORTE DE PRÁCTICA

IDENTIFICACIÓN DE LA PRÁCTICA

Práctica	18	Nombre de la práctica	Memoria compartida
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 el método de reducción paralela utilizando memoria compartida con múltiples bloques.

PROCEDIMIENTO

Realiza la implementación siguiendo estas instrucciones.

Realiza un programa en C/C++ en CUDA en el que implementes un kernel para reducción paralela utilizando memoria compartida. El kernel deberá ser verificado utilizando la validación de kernel. Para ello deberás atender los siguientes requerimientos:

- El tamaño del vector es de $N = 8192$ con valores de 1
- Lanzar el kernel utilizando 8 bloques de 1024 hilos por bloque
- Implementar el kernel como:
`__global__ void GPU_reduccion(int* vector, int* suma)`
- Implementa la misma función en el host:
`__host__ void CPU_reduccion(int* vector, int* suma)`
- 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 numBlocks 8
#define threadsPerBlock 1024

using namespace std;

__host__ void checkCUDAError(const char* msg) {
    cudaError_t error;
    cudaDeviceSynchronize();
}
```



Fundamentos de programación en paralelo

```
error = cudaGetLastError();
if (error != cudaSuccess) {
    printf("ERROR %d: %s (%s)\n", error, cudaGetErrorString(error), msg);
}
}

__host__ void validate(int* result_CPU, int* result_GPU) {
    if (*result_CPU != *result_GPU) {
        printf("[FAILED] Kernel validation.\n");
        return;
    }
    printf("[SUCCESS] Kernel validation.\n");
    return;
}

__host__ void CPU_reduction(int* v, int* sum) {
    for (int i = 0; i < numBlocks * threadsPerBlock; i++) {
        *sum += v[i];
    }
}

__global__ void GPU_reduction(int* v, int* sum) {
    __shared__ int vector[numBlocks * threadsPerBlock];
    int gId = threadIdx.x + blockDim.x * blockIdx.x;

    vector[gId] = v[gId];
    __syncthreads();
    int step = threadsPerBlock / 2;
    while (step) {
        if (threadIdx.x < step) {
            vector[gId] = vector[gId] + vector[gId + step];
            __syncthreads();
        }
        step = step / 2;
    }
    __syncthreads();
    if (threadIdx.x == 0) { // copy the partial results to the global mem vector
        //printf("SM->vector[%d]: %d\n", gId, vector[gId]);
        v[gId] = vector[gId];
        __syncthreads();
        //printf("GM->v[%d]: %d\n", gId, v[gId]);
    }
    if (gId < numBlocks) { // choose the first 4 threads to copy in the first 4 cells the
partial sums
        v[gId] = v[gId * threadsPerBlock]; // 0 <- 0*8, 1 <- 1*8 ...
        __syncthreads();
        //printf("%d<-%d\n", v[gId], v[gId * threadsPerBlock]);
    }
    int new_step = numBlocks / 2;
    while (new_step) {
        if (gId < new_step) {
            v[gId] += v[gId + new_step];
        }
        new_step = new_step / 2;
    }
    __syncthreads();
    if (gId == 0) {
        *sum = v[gId];
    }
}
```



Fundamentos de programación en paralelo

```
int main() {  
  
    int* dev_a, * dev_sum;  
    int host_sum = 0, CPU_sum = 0;  
    int* host_a = (int*)malloc(sizeof(int) * numBlocks * threadsPerBlock);  
    cudaMalloc((void**)&dev_a, sizeof(int) * numBlocks * threadsPerBlock);  
    cudaMalloc((void**)&dev_sum, sizeof(int));  
  
    for (int i = 0; i < numBlocks * threadsPerBlock; i++) {  
        host_a[i] = 1;  
    }  
  
    cudaMemcpy(dev_a, host_a, sizeof(int) * numBlocks * threadsPerBlock,  
cudaMemcpyHostToDevice);  
  
    dim3 grid(numBlocks, 1, 1);  
    dim3 block(threadsPerBlock, 1, 1);  
    GPU_reduction << < grid, block >> > (dev_a, dev_sum);  
    cudaDeviceSynchronize();  
    checkCUDAError("Error at kernel");  
  
    printf("N: %d\n", numBlocks * threadsPerBlock);  
    cudaMemcpy(&host_sum, dev_sum, sizeof(int), cudaMemcpyDeviceToHost);  
    printf("GPU result: %d\n", host_sum);  
  
    CPU_reduction(host_a, &CPU_sum);  
    printf("CPU result: %d\n", CPU_sum);  
  
    validate(&CPU_sum, &host_sum);  
  
    return 0;  
}
```



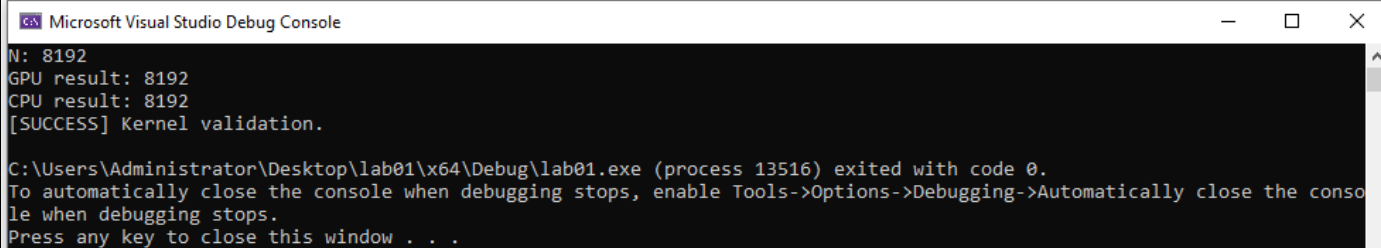
RESULTADOS

Agrega las imágenes obtenidas en los espacios indicados.

```
__global__ void GPU_reduction(int* v, int* sum) {
    __shared__ int vector[numBlocks * threadsPerBlock];
    int gId = threadIdx.x + blockDim.x * blockIdx.x;

    vector[gId] = v[gId];
    __syncthreads();
    int step = threadsPerBlock / 2;
    while (step) {
        if (threadIdx.x < step) {
            vector[gId] = vector[gId] + vector[gId + step];
            __syncthreads();
        }
        step = step / 2;
    }
    __syncthreads();
    if (threadIdx.x == 0) { // copy the partial results to the global mem vector
        //printf("SM->vector[%d]: %d\n", gId, vector[gId]);
        v[gId] = vector[gId];
        __syncthreads();
        //printf("GM->v[%d]: %d\n", gId, v[gId]);
    }
    if (gId < numBlocks) { // choose the first 4 threads to copy in the first 4 cells the partial sums
        v[gId] = v[gId * threadsPerBlock]; // 0 <- 0*8, 1 <- 1*8 ...
        __syncthreads();
        //printf("%d<-%d\n", v[gId], v[gId * threadsPerBlock]);
    }
    int new_step = numBlocks / 2;
    while (new_step) {
        if (gId < new_step) {
            v[gId] += v[gId + new_step];
        }
        new_step = new_step / 2;
    }
    __syncthreads();
    if (gId == 0) {
        *sum = v[gId];
    }
}
```

Código del kernel



```
Microsoft Visual Studio Debug Console

N: 8192
GPU result: 8192
CPU result: 8192
[SUCCESS] Kernel validation.

C:\Users\Administrator\Desktop\lab01\x64\Debug\lab01.exe (process 13516) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Imagen de la consola en la que se muestran los resultados y el mensaje de validación del kernel



UNIVERSIDAD
PANAMERICANA

Universidad Panamericana

Campus Guadalajara

Escuela de ingenierías

Fundamentos de programación en paralelo

CONCLUSIONES

Escribe tus observaciones y conclusiones.

En esta práctica es indispensable entender el proceso en paralelo, donde un ciclo puede tardar más de lo esperado y los resultados copiarse antes de haber terminado el proceso, ya que para tal proceso se utilizan ciclos. La sincronización por lo tanto se realiza en cada momento donde necesitemos que un proceso termine antes de seguir con las demás líneas del código, incluso aunque estén dentro del kernel.