CUDA Assignment Report

High Performance Computing Course – Riccardo Caprile (43707074)

CUDA Cluster Specifics

- **GPU :** GeForce GTX 1650
- **RAM :** 4GB
- **Warp : 32**
- **Maximum number of threads per block : 1024**

How to run the CUDA program

nvcc -o heat heat.cu -run

Code Analysis

The objective of this homework is to parallelize a program which simulates the 2D heat conduction. The program was , at the beginning , implemented using a for-loop on the CPU. We have to optimize the performances using a CUDA kernel.

The first step for resolving this problem is modifying the function step_kernel_mod(), firstly adding the keyword __global__ , in this way we guarantee that the function will be executed by the GPU. Then , we have to change the nested for loops. This is a bidimensional problem , so , the threads must be grouped in blocks to form a 2D grid. To guarantee that the N elements are computed in parallel , we have to compute the two indexes I and J in this way : I = blockIdx.x * blockDim.x + threadIdx.x , and j = blockIdx.y * blockDim.y + threadIdx.y, where threadIdx represents the thread coordinates inside the block, blockIdx the block coordinates inside the grid and block Dim the grid dimensions in thread units. The work inside this function is done and so we have to move inside the main() function.

Firstly we need 2 auxiliary dynamic arrays called aux_tmp1 and aux_tmp2. They are allocated in the GPU, using the CUDA function cudaMalloc() with the same size of their corresponding ones (temp1 and temp2) which have been allocated in the CPU with the system call malloc().

The next thing to do is set the number of threads per block (threadsPerBlock) and the number of blocks per grid (blocksPerGrid) for the parallel computation. Because the warp is 32 , the number of maximum threads per block is 1024 and the grid we are considering is bidimensional , the number of threads per block is 32 x 32 = 1024 , and the number of blocks per grid is (ni/32+1 , nj / 32 +1). Those two variables represent the dimension of the matrix which it is used for the computation of the algorithm.

In the next part of the program , the iteration for resolving the problem start. At each step , we copy the values inside the variable temp1 and temp2 to their corresponding variables in the GPU aux_tmp1 and aux_tmp2 using the CUDA function cudaMemcpy. After this the execution of the function step_kernel_mod is executed using the number of threads per block and the blocks per grid initialized before. When the function has finished its work , the results are copied back inside temp1 and temp2 using again cudaMemcpy(). After all of this , the time spent is computed and printed.

In the end , we have to deallocate the GPU memory space, used for initializing the auxiliary variables used before, with the CUDA function cudaFree().

In the next part , there are the performances reached with the serial version and the parallel version of the program , considering the execution time and max error.

Serial Version

- ni = 1000 , nj = 1000 : **Execution time** = 1.70 seconds , **Max Error** = 0.000
- ni = 10000 , nj = 10000 : **Execution time** = 170.28 seconds , **Max Error** = 0.000
- ni = 30000 , nj = 30000 : Invalid Memory Reference

Parallel Version

- ni = 1000 , nj = 1000 : **Execution time** = 0.38 seconds , **Max Error** = 0.00001
- ni = 10000 , nj = 10000 : **Execution time** = 32.82 seconds , **Max Error** = 0.00002
- ni = 30000 , nj = 30000 : Invalid Memory reference

Conclusions

Looking at the performances above , we got the best result with a 1000 x 1000 matrix and the error is very low. WIth a 10000 x 10000 matrix the error is still low but the execution time has increased. When we have to face the 30000 x 30000 matrix , the process requires to allocate two matrices with combined size that exceed the amount of memory we have available with the current machine , and so it results in a invalid memory reference.

The aim of this assignment was to parallelize a program using CUDA. The starting code was written for the CPU and executed sequentially , but we wanted to parallelize it using CUDA for exploiting the power of GPU acceleration.

The performances showed that the parallel version was extremely quicker compared to the sequential version.