CUDA

AUTHORS: Garbarino Giacomo, Parmiggiani Manuel

GPU CLUSTER SPECIFICS

- GPU: GeForce GTX 1650.
- Number of stream multiprocessor: 14.
- Warp: 32.
- Maximum number of threads per stream multiprocessor: 1024.
- Maximum number of blocks per stream multiprocessor: 16.
- Maximum number of threads per block: 1024.
- RAM: 4GB.

HOW TO EXECUTE A CUDA PROGRAM

1. nvcc -o heat_parallelized heat_parallelized.cu -run.

DISCUSSION

The program to parallelize implements the calculation of the 2D heat conduction through 200 iterations. The first step of the parallelization consists of modifying the function ***step_kernel_mod()***. At first, it's necessary to add the keyword ***__global__*** before the type of the function at the declaration level to guarantee that it's executed by the GPU. Then, it's needed to replace the nested for loops with nested if conditions. Since it's a bi-dimensional problem, the threads must be organized in blocks to form a 2D grid. To guarantee that N elements are computed in parallel, the two indexes i and j are respectively initialized with ***threadIdx.x + blockIdx.x * blockDim.x*** and ***threadIdx.y + blockIdx.y * blockDim.y***. Once the function has been appropriately modified, the next step consists of making some changes in the main. At first, two dynamic arrays, named ***dev_temp1*** and ***dev_temp2***, are allocated in the GPU with the same size as their corresponding ones in the CPU, which are respectively ***temp1*** and ***temp2***. Then, the number of threads per block and the number of blocks are specified to set the parallel computation. Since the warp is equal to 32, the maximum number of threads per block is equal to 1024 and the grid is bidimensional, the number of threads per block is set to ***(32,32)*** (32 multiplied by 32 is equal to 1024), while the number of blocks is set to ***(ni/32+1, nj/32+1)*** where ***ni*** and ***nj*** are the dimensions of the matrix used to compute the 2D heat conduction. After that, the iteration to resolve the problem with the ***step_kernel_mod()*** function starts. At each step, firstly each data contained in ***temp1*** and ***temp2*** are copied into their corresponding ones in the GPU with the function ***cudaMemcpy()***, then the function ***step_kernel_mod*** is executed with the number of threads per block and the number of blocks set before, and once it finished, the result is copied back to CPU and saved in ***temp1*** and ***temp2*** again using ***cudaMemcpy()***. In the end, the two GPU memory allocations are deallocated with the function ***cudaFree()***. Here are the performances according to the matrix dimensions:

- **ni=1.000 and nj=1.000:** execution time = 0.38s, error = 0.00001.
- **ni=10.000 and nj=10.000:** execution time = 32.65s, error = 0.00002
- **ni=30.000 and nj=30.000:** invalid memory reference.

The best result is achieved with a 1.000x1.000 matrix. With a 10.000x10.000 matrix, the error is still minimal, but it takes too much more time than in the previous case. The case with a 30.000x30.000 matrix returns an invalid memory reference. This happens because with 4GB GPU memory is not enough to contain 900 million elements (30.000 per 30.000). Since each element is a float, each element occupies 4 byte, so the matrix in the GPU occupies a total of 3.6 GB, but the GPU memory already contains other data and so this matrix is too big to be memorized.