Università
di Genova

MSc Computer Science
Data Science and Engineering Curriculum

CUDA assignment

High Performance Computing

Andrea Bricola, Roberto Di Via, Matteo Interlando

February, 2023

Contents

1 Introduction

The goal of this homework is to parallelize the following program, which simulates the diffusion of heat across a two-dimensional grid. It was initially implemented using a simple for-loop structure on the CPU. Our goal is to optimize the performance of the program by parallelizing the inner loop using a CUDA kernel, and evaluate the results using different problem sizes, and different configurations of blocks and threads.

1.1 Some background

Heat transfer by conduction occurs when thermal energy is transmitted from a hotter object to a colder one through direct physical contact. In a flat surface, heat conduction takes place from one point to another, driven by the temperature difference between them. The rate of heat conduction is proportional to the coefficient of thermal diffusivity (α), which depends on the partial derivatives of temperature at adjacent points.

To study heat conduction on a flat surface, we represent it as a matrix and assign a temperature (T) to each point. For the purpose of thermal diffusivity analysis, we consider a surface made of silver with a diffusivity coefficient of 8.42×10^{-5}.

$$\frac{d}{dt}T = \alpha \left(\frac{d^2}{dx^2}T + \frac{d^2}{dy^2}T \right)$$

We solve this problem with the following approximation:

$$\frac{\Delta T}{\Delta t} = \alpha [\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}]$$

1.2 How to compile and execute the program

First of all, I have to connect to the INFN Cluster. This cluster contains a machine equipped with a Nvidia Geforce GTX 1650. This graphic card is built on Turing (TU117) architecture and contains 896 CUDA cores building up 14 Streaming Multiprocessors. [ref]

To compile the program we used the Nvidia Cuda Compiler **nvcc** in this way:

```
nvcc progetto\_cuda.cu
```

While the command to run the executable is the following:

```
time ./a.out
```

2 Analysis of the program

The "heat.c" program uses a $n_i \times n_j$ matrix as a data structure to model the surface and assigns a randomly generated temperature to each cell. The program then calculates the changes in temperature from one time step to the next by computing the new temperature of each pixel.

The program consists of two main functions, the first one is the "step_kernel_mod" which is the function that we are going to accelerate with CUDA, and the second one is "step_kernel_ref" which is the original CPU implementation that we are going to use as a reference to compare the performance of the program with and without CUDA acceleration.

```
int main() {
    int nstep = 200; // number of time steps

    // Specify our 2D dimensions
    const int ni = 1000;
    const int nj = 1000;
    float tfac = 8.418e-5; // thermal diffusivity of silver
    ...

    // Execute the CPU-only reference version
    for (istep=0; istep < nstep; istep++) {
        step_kernel_ref(ni, nj, tfac, temp1_ref, temp2_ref);
        ... swap the temperature pointers ...
    }

    // Execute the modified version using same data
    for (istep=0; istep < nstep; istep++) {
        step_kernel_mod(ni, nj, tfac, temp1, temp2);
        ... swap the temperature pointers ...
    }
}
```

The "step_kernel_mod" function, that is the function that we are going to accelerate with CUDA, uses two nested for-loops to iterate over all the cells in the 2D grid, except for the boundary cells. Inside the loops, the program calculates the temperature derivatives using the values of the surrounding cells and updates the temperature value of the current cell using these derivatives.

```
void step_kernel_mod(int ni, int nj, float fact, float* temp_in, float* temp_out) {
    ...
    // loop over all points in domain (except boundary)
    for ( int j=1; j < nj-1; j++ ) {
        for ( int i=1; i < ni-1; i++ ) {
          // find indices into linear memory
          // for central point and neighbours
          i00 = I2D(ni, i, j);
          im10 = I2D(ni, i-1, j);
          ip10 = I2D(ni, i+1, j);
```

```
          i0m1 = I2D(ni, i, j-1);
          i0p1 = I2D(ni, i, j+1);

          // evaluate derivatives ...
          d2tdx2 = ...
          d2tdy2 = ...

          // update temperatures ...
          temp_out[i00] = ...
        }
    }
}
```

3 Parallelization with CUDA

CUDA (Compute Unified Device Architecture) is a technology developed by Nvidia to perform general-purpose computation on Nvidia GPUs. It is an extension of the C/C++ programming language and provides a programming model based on parallel threads. These threads are organized into grids and blocks, where each block contains multiple threads, and each grid contains multiple blocks.

In our CUDA implementation, we utilized a one-dimensional grid of blocks, each containing one-dimensional blocks of threads. The number of threads in a block is fixed and the number of blocks deployed is proportional to the size of the problem. To decide the number of blocks we used the following idiom.

```
dim3 threads(1024);
dim3 dimblock(((ni-ni/2) * (nj-nj/2) + threads.x - 1) / threads.x);
```

When the total number of threads is less than the size of the problem, the device function must include a stride so threads can hit all the cells of the problem matrix.

To parallelize the program with CUDA, we would first need to identify the computationally intensive parts of the program, which in this case the "step_kernel_mod" function that iterates over all the cells in the 2D grid, except for the boundary cells. We would then need to convert this function into a CUDA kernel using the "__global__" keyword. The kernel would have the same logic as the original loop, but it would be executed in parallel on the GPU.

To launch the kernel, we would need to define the grid and block dimensions, which determine how the GPU threads are organized and how the kernel is executed. We would need to choose the dimensions based on the problem size and the number of CUDA cores available on the GPU.

For a 1000x1000 matrix problem, there are 1 million cells in the matrix and 250,000 threads are initialized. We utilize the CudaMallocManaged function, part of the Unified Memory API, to allocate two matrices in the GPU's linear memory. This simplifies the process of allocating arrays on both the host and device without the need for manually transferring data between them [ref]. The GPU matrices, temp1 and temp2, are the same size as the original matrices from the serial code and store the temperatures from the previous and next time steps, respectively.

```
temp1_ref = (float*)malloc(size);
temp2_ref = (float*)malloc(size);
cudaMallocManaged(&temp1, size);
```

```
cudaMallocManaged(&temp2, size);
```

During the execution, each CUDA thread runs the Kernel function. In the kernel code, each thread is assigned to a specific cell within the matrix based on its thread and block ID. A common approach in CUDA programming is to launch one thread per data element, meaning that the loop is parallelized by writing a kernel that accommodates the number of threads to be greater than the size of the array. This is explained in a reference [article by Mark Harris]. Instead of using a monolithic kernel, which processes the entire array in one pass, we launched fewer threads than the array size and implemented a grid-stride loop. This way, the threads can cover the entire array in a few iterations.

```
int index = blockIdx.x * blockDim.x + threadIdx.x;
int stride = blockDim.x * gridDim.x;
for (int idx = index; idx < (nj-2)*(ni-2); idx += stride ) {
    int i = idx % (ni-2) + 1;
    int j = idx / (ni-2) + 1;
    ...
}
```

After conducting some experiments, we found that the performance of the grid-stride kernel was comparable to that of the monolithic kernel in terms of execution time. No changes were made to the core computation of the surface points, as the calculation of partial derivatives remained the same.

3.1 Experiments

The execution time of the program was measured by utilizing CUDA's timing events, which are triggered by calling cudaEventRecord(start_mod) at the start and cudaEventRecord(end_mod) at the end of the computation. The measurement captures the duration of heat conduction calculation after n_step iterations.

3.1.1 Different problem sizes

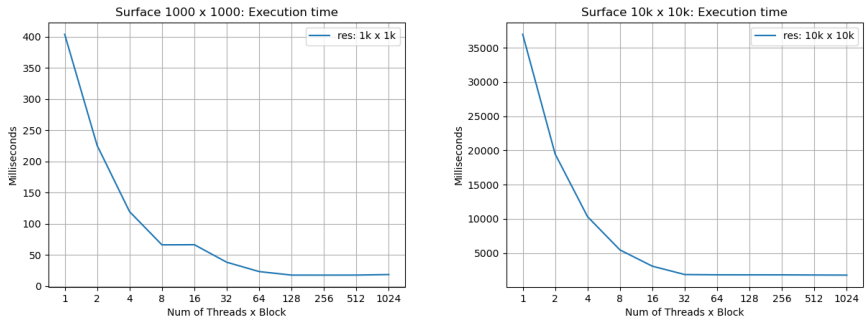
The purpose of this assignment is to evaluate the program's performance by conducting tests on surface sizes of 1000×1000, 10.000×10.000 and 30.000×30.000. However, when it comes to the surface size of 30.000×30.000, the process requires allocating two matrices with a combined size of 7.2 Gigabytes, which exceeds the computer's available memory, resulting in a runtime error. So, the 30.000 scenario is not feasible with the current machine.

	1000 x 1000	10.000 x 10.000	30.000 x 30.000
Serial exec time	1358 ms	137474 ms	NaN
Parallel exec time	17.5ms	1800 ms	NaN

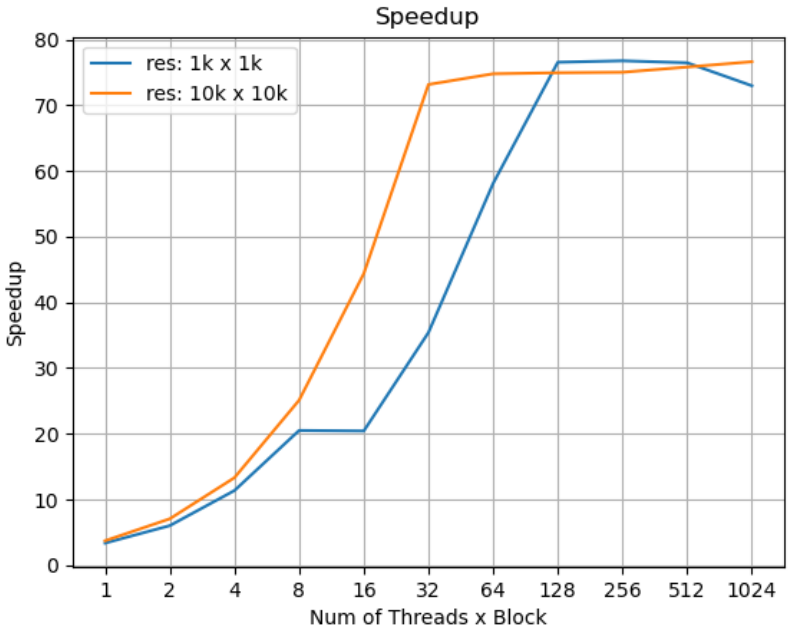
Parallelization of the problem of calculating heat transfer in a surface after n steps lead the program to a great reduction in execution time. For the problemn of 10k $\times$ 10k surface, the execution time went from 137 seconds to 1.8 seconds. This reduction corresponds to a -98.7% variation.

3.1.2 Different configurations of blocks & threads per block

The CUDA guidelines advise that the number of threads per block in a program should be a multiple of 32, as CUDA launches threads in warps of this size. In order to optimize the performance of the program, it is important to also consider the amount of shared memory available on the GPU when choosing the number of threads per block. In our experimentation with the CUDA program, we evaluated different configurations of threads per block by testing it with different powers of two.



The results align with our expectations: when a number less than 32 is selected as the number of threads per block, the performance is bad. On the other hand, when the number is a multiple of 32, the performance is comparable. Parallelization with CUDA, with a suitable configuration, resulted in a speed increase of 80%.


Speedup
res: 1k x 1k
res: 10k x 10k
Speedup
0
10
20
30
40
50
60
70
80
1
2
4
8
16
32
64
128
256
512
1024
Num of Threads x Block

4 Conclusions

This assignment aimed to parallelize a code for high-performance computing by utilizing the CUDA platform. The code performs numerical simulations of heat diffusion in a two-dimensional domain, with the goal of reducing the runtime and improving the computational efficiency. The original code was written for a CPU and executed sequentially, but the parallelized version was implemented using CUDA and was able to utilize the power of GPU acceleration to achieve significant performance improvements.

The results showed that the parallelized version was able to achieve a substantial speed-up compared to the sequential CPU version, demonstrating the effectiveness of the parallelization process. In addition, the use of managed memory in the CUDA implementation allowed for easier management of data between the CPU and GPU, further enhancing computational efficiency. This assignment highlights the benefits of parallelization in high-performance computing and the potential of the CUDA platform to deliver significant improvements in performance.