Università di Genova

Computer Engineerig

# Cuda assignment

High Performance Computing

Martin Martuccio , Lorenzo Mesi

April, 2024

## Contents

# 1 Introduction

The goal of this assignment was to parallelize a program that models the diffusion of heat on a two-dimensional grid. The original version of the program used a loop on a single CPU. The main goal was to increase performance by using CUDA to parallel computations using GPUs and evaluate results using different problem sizes, and different block and thread configurations.

## 1.1 Mathematical concepts

The heat diffuser in a flat surface is described by the formula below. The rate at which it spreads depends on a thermal diffusion coefficient and partial derivatives.

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

We solve this problem with the following approximation:

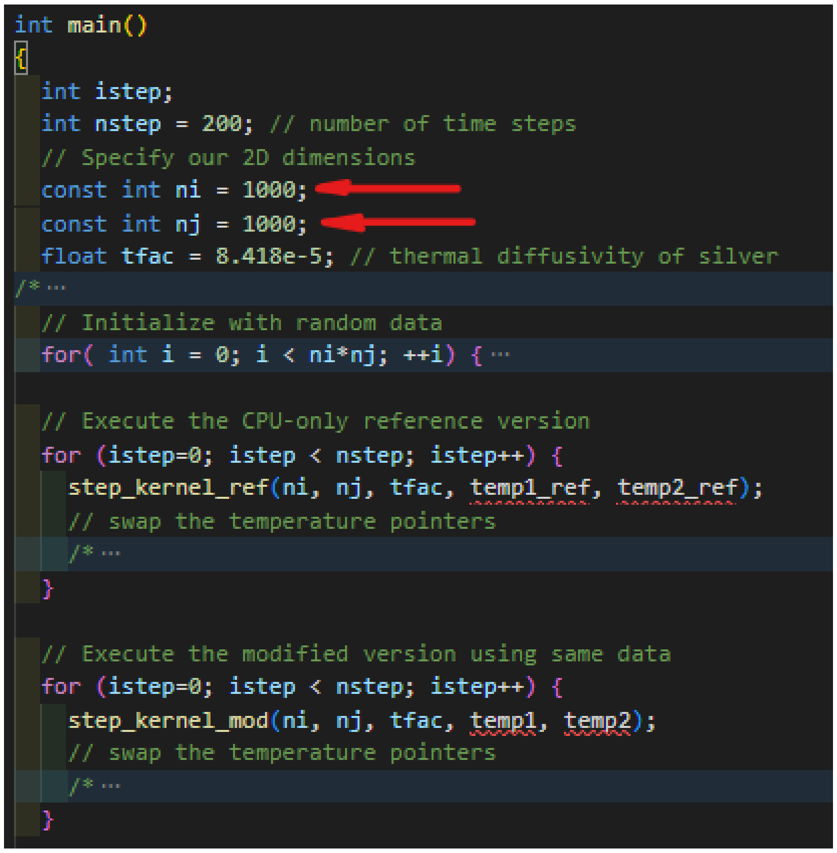
$$\frac{\Delta T}{\Delta t} = \alpha\left(\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}\right)$$

Our program computes the diffusion of heat and the variation of temperature in every single portion of the surface following a specific formula. We can vary the size of the problem by dividing the surface into smaller and smaller portions to make the computation more accurate. Larger is the problem size and more computational power is required. We would like to use threads to perform computations in order to reduce the execution time.

## 2 Analysis of the program

The program uses a $ni \times nj$ matrix where each element represents a cell initialized with a random temperature value (the total number of cells represent the size of our problem).

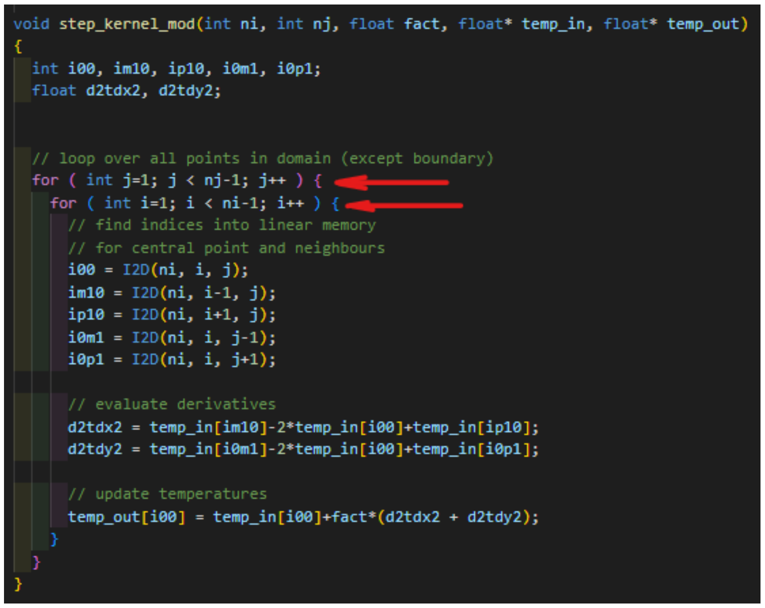
The program is built around two principal functions: "*step kernel mod*" and "*step kernel ref*". The first one is the "*step kernel mod*" which is the function that we are going to accelerate with **CUDA**. The second one is "*step kernel ref*" which is the original **CPU** implementation.



```
int main()
{
    int istep;
    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
/*...
    // Initialize with random data
    for( int i = 0; i < ni*nj; ++i) {...

    // 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
      /*...
    }
```

Figure 1: Main function

The focus of this task is based on the "*step kernel mod*" function, which will be speed up using CUDA. That function consists of two loops running on the whole surface, considering each cell of the grid, skipping the edge cells. The program, for each element of the cell, computes the new value, also using the temperatures of adjacent cells



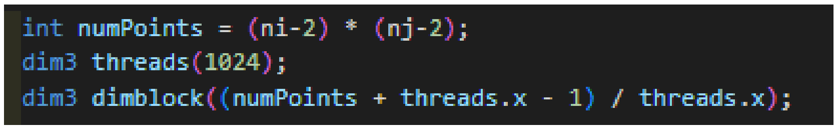
```
void step_kernel_mod(int ni, int nj, float fact, float* temp_in, float* temp_out)
{
  int i00, im10, ip10, i0m1, i0p1;
  float d2tdx2, d2tdy2;


  // 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 = temp_in[im10]-2*temp_in[i00]+temp_in[ip10];
      d2tdy2 = temp_in[i0m1]-2*temp_in[i00]+temp_in[i0p1];

      // update temperatures
      temp_out[i00] = temp_in[i00]+fact*(d2tdx2 + d2tdy2);
    }
  }
}
```

Figure 2: Step kernel mode function

## 3 Parallelization with CUDA

CUDA is an extension of the C/C++ programming language and provides a programming model based on parallel threads. It allows to use the GPU to run several threads on the same operation, but on different data. To do this, define the number of blocks and the number of threads for each block in according the problem size. The threads are grouped in batches called warp and are 32. The grid contains multiple blocks.

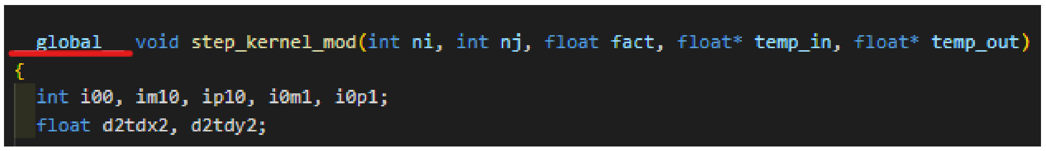
We decided to use a one-dimensional grid of blocks, each containing one-dimensional blocks of threads. The max number of threads in a block is fixed at 1024 and the number of blocks is based on the problem size. In our case we will execute the program with different number of threads and to consequence the number of blocks will be different.



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

Figure 3: Dimension of threads and blocks

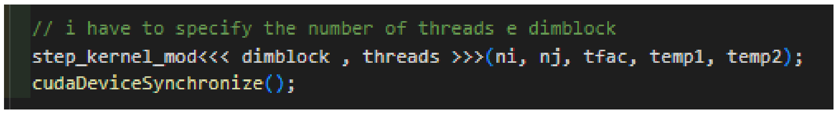
We convert the function into a CUDA kernel using the "*__global__*" keyword.



```
__global__ void step_kernel_mod(int ni, int nj, float fact, float* temp_in, float* temp_out)
{
    int i00, im10, ip10, i0m1, i0p1;
    float d2tdx2, d2tdy2;
```

Figure 4: Kernel Function

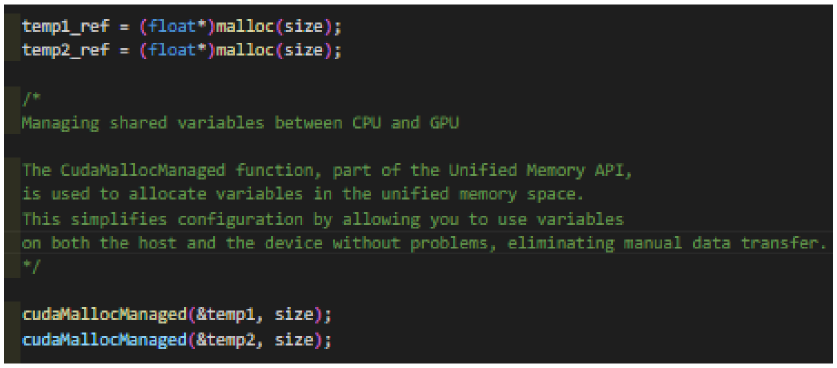
The kernel would have the same logic as the original loop, but it would be executed in parallel on the GPU. When we call the function we should now pass as parameters the size of threads and blocks used



```
// i have to specify the number of threads e dimblock
step_kernel_mod<<< dimblock , threads >>>(ni, nj, tfac, temp1, temp2);
cudaDeviceSynchronize();
```

Figure 5: Calling the Kernel Function

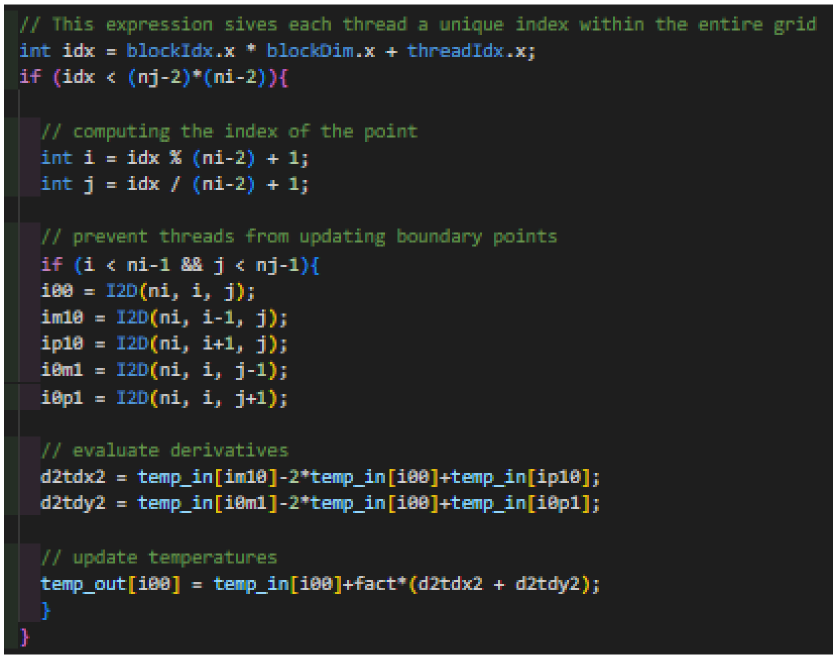
We used the CudaMallocManaged function to allocate space in Unified Memory. This hardware/software technology allows applications to allocate data that can be read or written from code running on either CPUs or GPUs. This simplifies the process of allocating arrays on both the host and device without the need for manually transferring data between them.



```
temp1_ref = (float*)malloc(size);
temp2_ref = (float*)malloc(size);

/*
Managing shared variables between CPU and GPU

The CudaMallocManaged function, part of the Unified Memory API,
is used to allocate variables in the unified memory space.
This simplifies configuration by allowing you to use variables
on both the host and the device without problems, eliminating manual data transfer.
*/

cudaMallocManaged(&temp1, size);
cudaMallocManaged(&temp2, size);
```

Figure 6: Allocation of memory in UM

Cuda allows us to use multiple threads to distribute the workload. Usually for **N** elements to compute, we need to launch **N** threads that are aware of each individual instance. In our case each cell is associated with a thread to which an identification ID is associated based on the thread and the corresponding block. As we increase the number of cells that make up our surface (increasing the size of the problem), we should launch more threads.



```
// This expression sives each thread a unique index within the entire grid
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx < (nj-2)*(ni-2)){

  // computing the index of the point
  int i = idx % (ni-2) + 1;
  int j = idx / (ni-2) + 1;

  // prevent threads from updating boundary points
  if (i < ni-1 && j < nj-1){
  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 = temp_in[im10]-2*temp_in[i00]+temp_in[ip10];
  d2tdy2 = temp_in[i0m1]-2*temp_in[i00]+temp_in[i0p1];

  // update temperatures
  temp_out[i00] = temp_in[i00]+fact*(d2tdx2 + d2tdy2);
  }
}
```

Figure 7: Modified Kernel Function

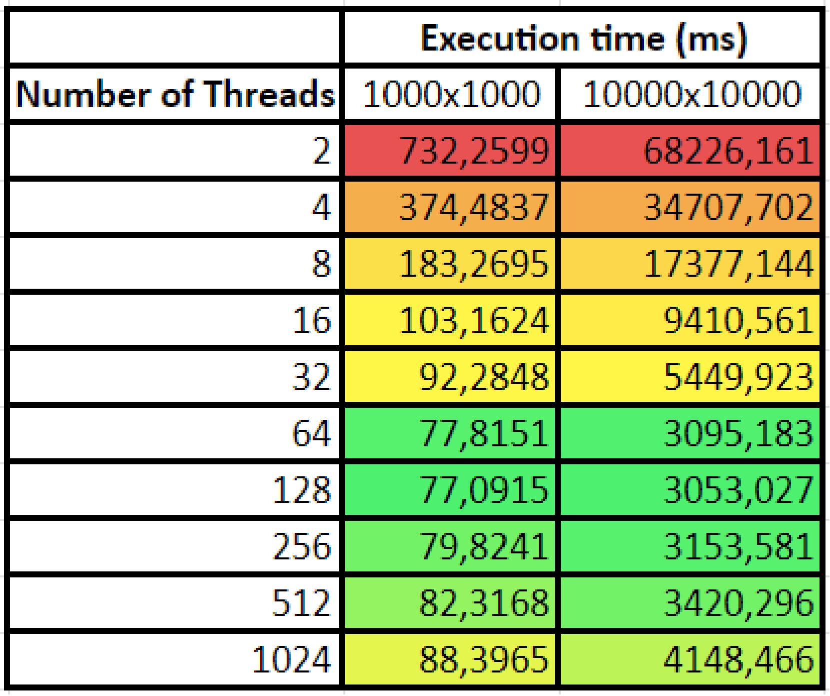
Following the changes, we carried out several repeated tests measuring the execution times thanks to the use of functions that are made available by CUDA such as *cudaEventCreate*, *cudaEventRecord*, *cudaEventSynchronize*, *cudaEventElapsedTime*. We have computed the average times (considering around 30 runs of the program) and reported in a table for our considerations.

## 4 Experiments

The delivery of the project is to evaluate the performance and the improvement in relation to the workload (*problem sizing*) and the different configurations of threads and blocks. The required tests were with a subdivision of the surface in cells with a total size of **1000x1000**, **10 000x10 000** and **30 000x30 000**.

Running the program with a sizing of the problem 30 000 x 30 000 **is not feasible** due to a poor availability of computer memory.

We tried to use different configurations during the various tests. Very common is to use a number of threads per block that follows the trend of the power of 2. In the following table we show the different execution times obtained with the different configurations and we have adopted a graduated color scale that goes from a high execution time (red) at an acceptable and good execution time (green).



| | Execution time (ms) | |
|---|---|---|
| Number of Threads | 1000x1000 | 10000x10000 |
| 2 | 732,2599 | 68226,161 |
| 4 | 374,4837 | 34707,702 |
| 8 | 183,2695 | 17377,144 |
| 16 | 103,1624 | 9410,561 |
| 32 | 92,2848 | 5449,923 |
| 64 | 77,8151 | 3095,183 |
| 128 | 77,0915 | 3053,027 |
| 256 | 79,8241 | 3153,581 |
| 512 | 82,3168 | 3420,296 |
| 1024 | 88,3965 | 4148,466 |

Figure 8: Execution time table

## 4.1 Consideration on execution time

The threads are grouped in batches called warp and are 32. We noticed how the execution times are significantly reduced with the use of a number of threads that are multiple of 32. Using a number of threads too high, performance decay and so should be better to use a configuration that finds a right balance between the size of the problem and the management of the resources used and that must be managed in run-time. In our case (1000x1000), **using 128 threads led to better performance**, with a decrease of an order of magnitude compared to the execution time with 2 threads. In the case

(10000x10000) the performance has improved even more.

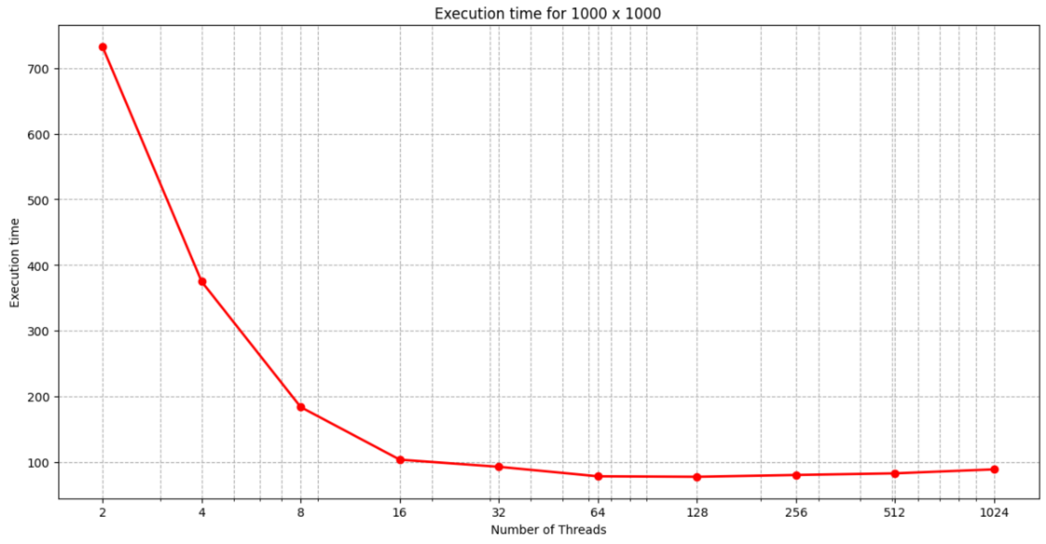


Figure 9: Execution time graph 1000x1000

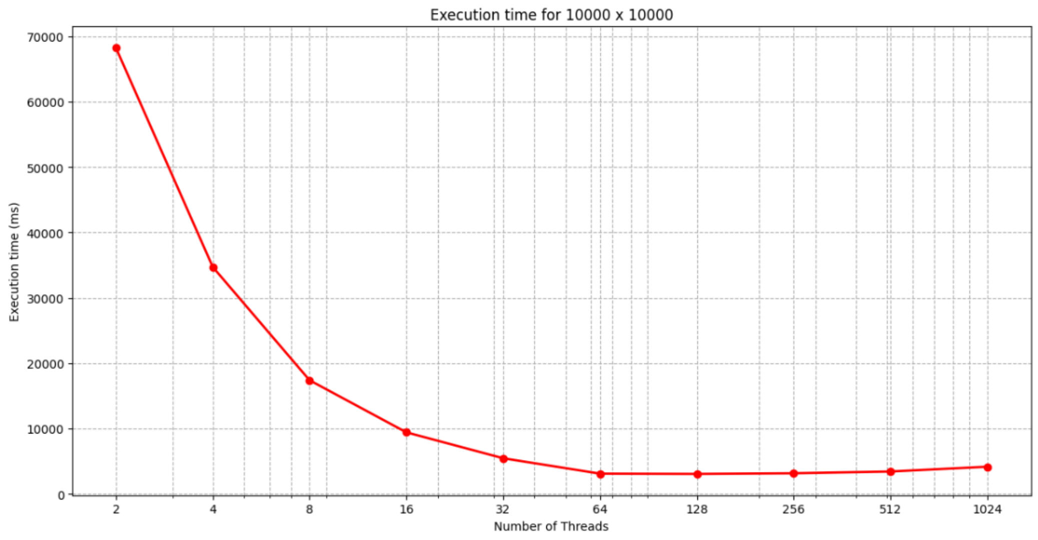


Figure 10: Execution time graph 10000x10000

## 4.2 Consideration speed up

We also computed the speed up based on the execution time of the program to analyze how it changed with the number of threads used.

*In the graphs below we used a logarithmic scale on the abscissa*

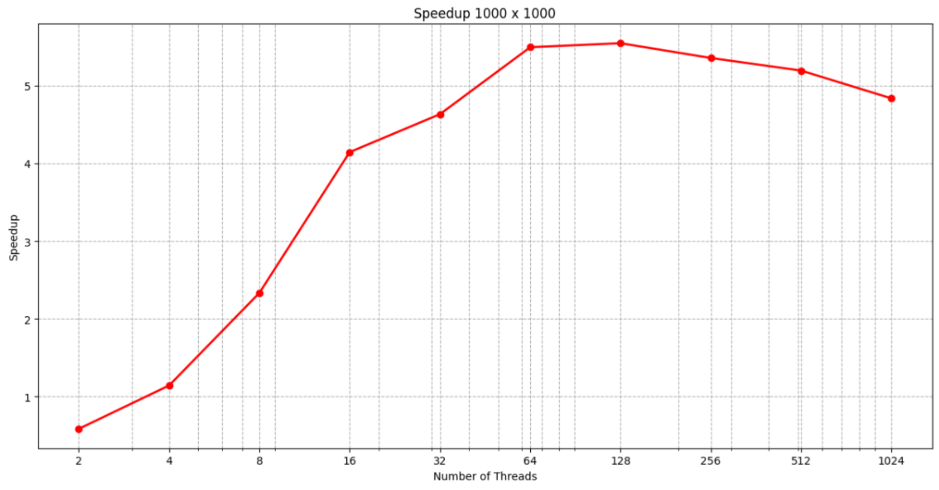


Figure 11: Speed up graph 1000x1000

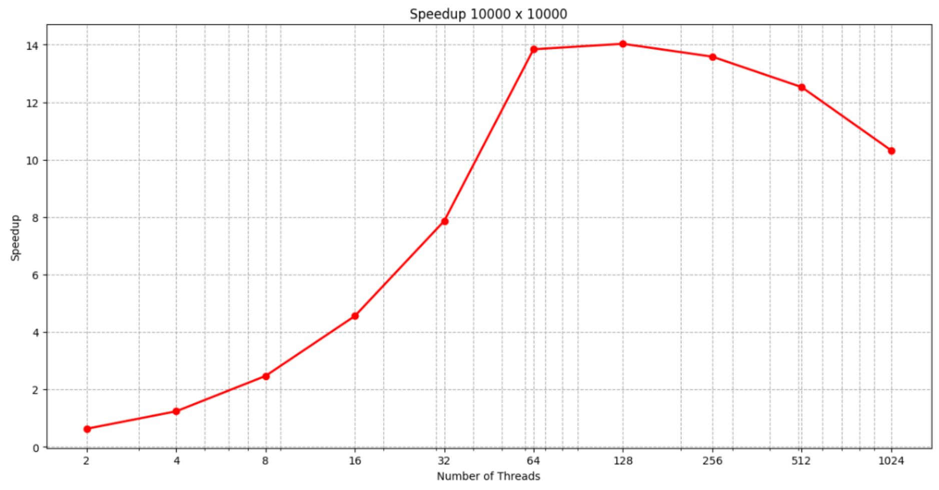


Figure 12: Speed up graph 10000x10000

Observing the speed up, it is easy to visualize that the maximum value we reach, as we said before, with a thread number equal to 128 (with which the execution time is the lowest)

## 5 Conclusions

This project had the goal of parallelizing a code for high-performance computing using the CUDA platform.The code performs heat diffusion simulations in a two-dimensional domain, with the aim of reducing execution time and improving computational efficiency.The parallel version has been implemented using CUDA and uses the GPU for better performance.

The results showed that the parallel version was able to achieve a remarkable speed-up compared to the sequential version.

In particular, observing the speed-up, it is evident that the best performance is obtained using 128 threads, regardless of the different dimensionality of the problem. This configuration has been shown to minimize execution time, finding an optimal balance between the number of threads and the size of the problem to maximize computational efficiency and without reaching threads overhead.