Solving the heat equation using CUDA$\backslash C++$

Alessia Arpaia, Gianluca Tutino

July 2023

Abstract

In this paper we employ parallel computing to solve the heat equation more efficiently by leveraging the power of multiple processors or computing resources.
The heat equation is a partial differential equation that describes how temperature or heat diffuses over time in a given region. It is widely used in various fields, including physics, engineering, and computer science, to model and analyze heat transfer phenomena.
We use $CUDA\backslash C++$ (Compute Unified Device Architecture) that is a parallel computing platform and programming model developed by NVIDIA.

Introduction

In this project, we propose a parallel algorithm to solve the heat equation in two dimensions on University of Naples Federico II's HPC cluster I.Bi.S.Co. (Infrastructure for Big data and Scientific Computing).
First of all we give a brief overview of the theory concerning the heat equation, in particular we explain the finite difference method that we use to solve the heat equation.
This method divide the spatial domain into a grid, and discretize the time domain into discrete time steps.
In order to do this, we use CUDA which involves parallelizing the computation on the GPU using CUDA programming techniques. We can implement the heat equation solver using CUDA in different ways, for example through the domain decomposition, data allocation, data initialization, synchronization, memory access and memory transfer etc. As we will see.
Next, we explore the design and implementation details of our parallel approach using CUDA.
Finally we show same experimental results and conclusions.

Studied Model

The heat equation in two dimensions describes the distribution of heat or temperature over time in a two-dimensional region. It is a partial differential equation that can be written as follows:

$$\partial u/\partial t = \alpha(\partial^2 u/\partial x^2 + \partial^2 u/\partial y^2)$$

where:

- $u(x, y, t)$ represents the temperature distribution at position (x, y) and time t.
- α is the thermal diffusivity, a constant that combines the material's thermal conductivity, density, and heat capacity.
- $\partial u/\partial t$ represents the rate of change of temperature with respect to time.
- $\partial^2 u/\partial x^2$ and $\partial^2 u/\partial y^2$ represent the second derivatives of temperature with respect to the spatial coordinates x and y, respectively.

The heat equation states that the change in temperature with respect to time at any point is proportional to the Laplacian of temperature at that point, which is the sum of the second derivatives of temperature with respect to x and y.

Numerical methods are commonly employed to solve the two-dimensional heat equation. We chose the finite difference methods.

To solve this equation using finite difference methods, we need to discretize the spatial domain into a grid or mesh with grid spacing Δx in the x-direction and Δy in the y-direction. We also discretize the time domain into discrete time steps with a step size Δt.
The finite difference approximations for the second derivatives in the heat equation are:

$$\partial^2 u/\partial x^2 \approx (u(i+1,j) - 2u(i,j) + u(i-1,j))/(\Delta x)^2$$

$$\partial^2 u/\partial y^2 \approx (u(i,j+1) - 2u(i,j) + u(i,j-1))/(\Delta y)^2$$

where $u(i,j)$ represents the temperature at the grid point (i,j) in the spatial domain.

Using these approximations, we can rewrite the heat equation as:

$$u(i,j,t+\Delta t) = u(i,j,t) + \alpha\Delta t * ((u(i+1,j,t) - 2u(i,j,t) + u(i-1,j,t))/(\Delta x)^2 +$$
$$+(u(i,j+1,t) - 2u(i,j,t) + u(i,j-1,t))/(\Delta y)^2)$$

This equation gives us the update rule to compute the temperature at each grid point (i,j) at the next time step $(t+\Delta t)$ based on the temperature values at the current time step (t).

Note that the values at the borders can not be computed from the equation above, since some of the required values will be out of the spatial region (e.g. $u(i-1,j,t)$ when $i=0$). This is where the boundary conditions are used.

To solve the heat equation numerically using finite difference methods, we follow these steps:

1. Discretize the spatial domain: Divide the x and y axes into grid points with spacing Δx and Δy, respectively. Create a 2D grid or mesh with (N_x+1) points in the x-direction and (N_y+1) points in the y-direction.
2. Discretize the time domain: Choose a time step size Δt. In order to be coherent with the model that we are using the largest value of time step is:

$$\Delta t \leq \frac{1}{2\alpha}\frac{1}{\frac{1}{\Delta x^2}+\frac{1}{\Delta y^2}} = \frac{\Delta x^2 \Delta y^2}{2\alpha(\Delta x^2+\Delta y^2)}$$

3. Initialize the temperature: Set the initial temperature values $u(i,j,0)$ for all grid points (i,j) at $t=0$.
4. Time-stepping loop: Iterate over time steps $(t = \Delta t, 2\Delta t, 3\Delta t, ...)$ until the desired simulation time is reached.
5. Update temperature: For each grid point (i,j) at each time step t, use the finite difference approximation mentioned earlier to compute the temperature at the next time step $(t+\Delta t)$.

6. Handle boundary conditions: Apply the appropriate boundary conditions to ensure that the temperature values at the boundaries of the spatial domain are correctly accounted for in the computations.
7. Repeat steps 5 and 6 for all interior grid points, excluding the boundaries.
8. Visualization and analysis: After the time integration, analyze and visualize the temperature distribution to study the heat transfer process. Plot temperature profiles, observe heat propagation, and extract relevant information from the numerical solution.

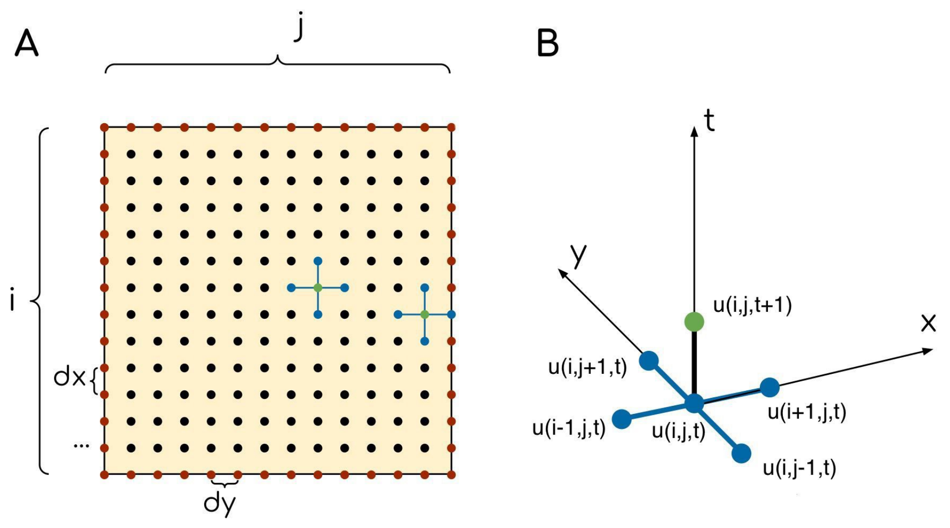


Figure 1: The grid (A) and the template (B) of the numerical scheme for the heat equation. Red dots indicate the grid nodes, where the values are taken from the boundary conditions. Black dots are internal grid nodes. The grid node in which the values is computes is shown in green. There are two examples of how the template (B) is applied to the grid (A).

C++ Code (HeatEquation.cu)

The C header files used in this code are the following:

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
```

At this point we define the parameters of our problem:

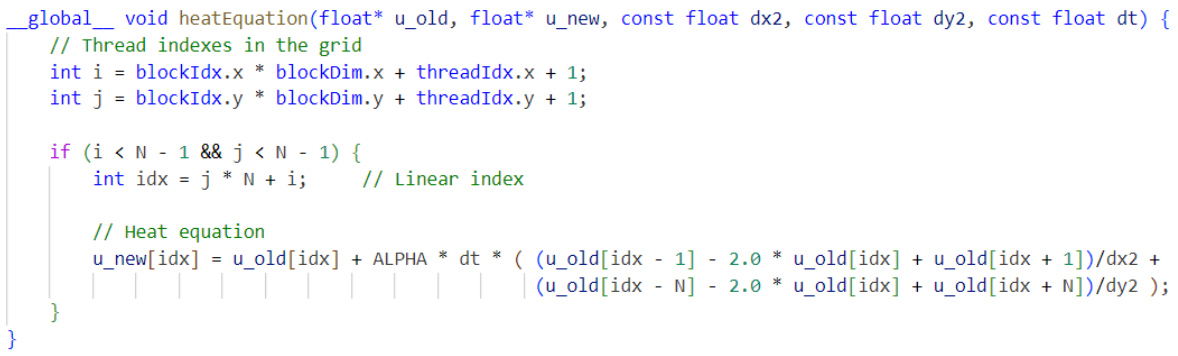
```
#define N 10000                         // Size of the grid
#define numSteps 1000                 // Number of iteration
#define ALPHA 0.1                     // Heat equation constant
#define SQUARE_SIZE 1000                // Dimension of the initial heated square
#define BLOCK_SIZE 32                 // Thread block size for kernel
#define OutputNum 200                 // Number of iterations to print grid
```

Where:

- N: the dimension of the square matrix $N \times N$ that represents the grid in which we study the heat behaviour;

- numSteps: the number of Δt's for which we study the heat equation;
- ALPHA: the conductivity of the material of which is composed our grid;
- $SQUARE\_SIZE$: the dimension of the submatrix with heat equal to 1.0;
- $BLOCK\_SIZE$: the parameter that link everything to the parallel computing, we'll use it to determine the thread block size for CUDA kernel.
- OutputNum: this variable indicates the number of iterations the code has to wait before printing the grid in a .txt file

Now we define the function that involves the system evolution given from the heat equation:

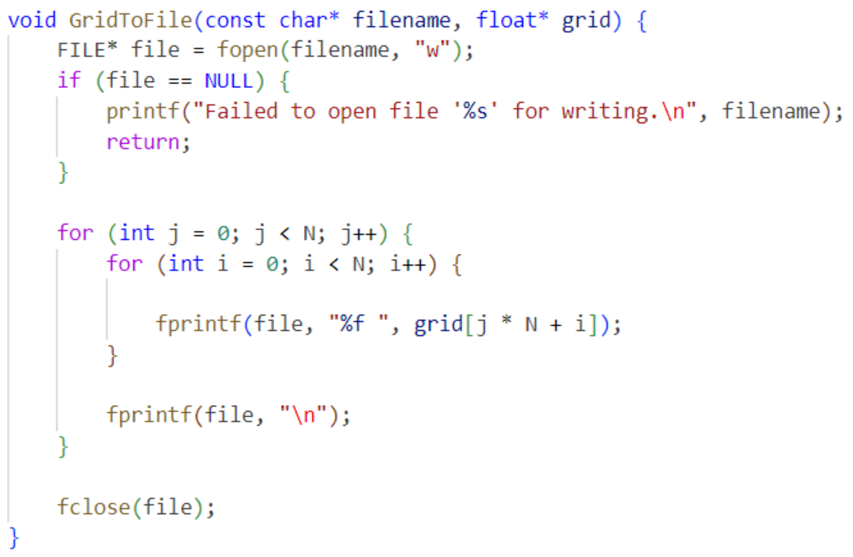


```
__global__ void heatEquation(float* u_old, float* u_new, const float dx2, const float dy2, const float dt) {
    // Thread indexes in the grid
    int i = blockIdx.x * blockDim.x + threadIdx.x + 1;
    int j = blockIdx.y * blockDim.y + threadIdx.y + 1;

    if (i < N - 1 && j < N - 1) {
        int idx = j * N + i;     // Linear index

        // Heat equation
        u_new[idx] = u_old[idx] + ALPHA * dt * ( (u_old[idx - 1] - 2.0 * u_old[idx] + u_old[idx + 1])/dx2 +
                                                 (u_old[idx - N] - 2.0 * u_old[idx] + u_old[idx + N])/dy2 );
    }
}
```

This is the CUDA kernel. We identify the thread with his position in x and y. At this point we define idx transposing the i and j (matrix position) to a vectorial index and the new value on the grid will be the updated value given by the heat equation. So each thread is responsible for computing the equation at a specific grid point.
We defined another function 'GridToFile'. For the last of output visualization we made the code print the grid every OutputNum iterations :



```
void GridToFile(const char* filename, float* grid) {
    FILE* file = fopen(filename, "w");
    if (file == NULL) {
        printf("Failed to open file '%s' for writing.\n", filename);
        return;
    }

    for (int j = 0; j < N; j++) {
        for (int i = 0; i < N; i++) {

            fprintf(file, "%f ", grid[j * N + i]);
        }

        fprintf(file, "\n");
    }

    fclose(file);
}
```

Then we build the main function. First things first we defined deltas in time and space. The timestep is derived by dx and dy. That's the largest time step possible to be coherent with the model that we consider:

```
int main() {

    // Deltas
    const float dx = 0.01;   // Horizontal grid spacing
    const float dy = 0.01;   // Vertical grid spacing

    const float dx2 = dx*dx;
    const float dy2 = dy*dy;

    const float dt = dx2 * dy2 / (2.0 * ALPHA * (dx2 + dy2)); // Largest stable time step
```

The grids' declaration follows. The grids are meant to be matrixes, but we use a linear representation (vectors) because the memory organization of CUDA is linear. In fact, in the CPU we define two vectors of dimension $N \times N$ with float values (u_old, u_new). The other two vectors (d_u_old, d_u_new) are declared here but they will be used on the GPU.

```
    // Declaring the inital and final grids
    float* u_old, *u_new;
    float* d_u_old, *d_u_new;

    u_old = (float*)malloc(N * N * sizeof(float));
    u_new = (float*)malloc(N * N * sizeof(float));
```

We can then initialize the grid in the following way: all points are cold (0.0), except a square of dimension $SQUARE\_SIZE$ placed at the grid's center. When we need the visualization we can use the function 'GridToFile' to have a text file with the initial grid (now commented for the reason explained in the Output part).

```
    // Initialize the grid
    for (int j = 0; j < N; j++) {
        for (int i = 0; i < N; i++) {
            int idx = j * N + i;
            if (i >= (N - SQUARE_SIZE) / 2 && i < (N + SQUARE_SIZE) / 2 &&
                j >= (N - SQUARE_SIZE) / 2 && j < (N + SQUARE_SIZE) / 2) {
                u_old[idx] = 1.0;
            } else {
                u_old[idx] = 0.0;
            }
        }
    }

    // // Writing the initial grid to a file
    // GridToFile("heat_0000.txt", u_old);
```

Now we allocate two vectors in the GPU via the CudaMalloc command. These vectors have the same dimension as the first two allocated in the CPU. In fact then we copy the initial grid to the gpu:

```
    // Allocating memory on the GPU
    cudaMalloc((void**)&d_u_old, N * N * sizeof(float));
    cudaMalloc((void**)&d_u_new, N * N * sizeof(float));

    // Copying the initial grid to the GPU
    cudaMemcpy(d_u_old, u_old, N * N * sizeof(float), cudaMemcpyHostToDevice);
```

At this point we implement the parallel computing. First we execute a domain decomposition declaring the number of thread blocks and the amount of thread in each block. In this way each block will take a part of the grid and each thread in the block will compute the heat equation for each grid value.

```
// Threaed block size and thread block's grid size
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid((N - 1) / (dimBlock.x + 1), (N - 1) / (dimBlock.y + 1));
```

In order to record the execution time of the parallel computing part, we define the variables in which we will allocate the time of starting and stopping of this chunk of code. We start the timer by the command cudaEventRecord(start). The kernel is executed numSteps times and each iteration we swap the old grid with the new grid. When we want to save the grids there is the chunk of code where every outputNum iterations the program saves the grid in a text file. Finally, we can stop the timer and save the timedelta between start and stop (given in milliseconds by default). The function CudaEventSynchronize commands the CPU to wait that every process in the GPU is completed. This is used to measure a more accurate execution time:

```
// Declaring the variables to measure execssution time
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

// Starting the timer
cudaEventRecord(start);

// Launching the kernel
for (int t = 0; t <= numSteps; t++) {
    heatEquation<<<dimGrid, dimBlock>>>(d_u_old, d_u_new, dx2, dy2, dt);

    // Swap the old and new grids
    float* temp = d_u_new;
    d_u_new = d_u_old;
    d_u_old = temp;

    // // Output visualization
    // if (t % OutputNum == 0 && t != 0){
    //     // Copying the final grid back to the CPU
    //     cudaMemcpy(u_new, d_u_old, N * N * sizeof(float), cudaMemcpyDeviceToHost);
    //     // Writing the final grid to a file
    //     char filename[64];
    //     sprintf(filename, "heat_%04d.txt", t);
    //     GridToFile(filename, u_new);
    // }
}
```

```
// Stopping the timer
cudaEventRecord(stop);

// Synchronizing CPU threads
cudaEventSynchronize(stop);

// Saving the execution time
float exec_time = 0;
cudaEventElapsedTime(&exec_time, start, stop);
exec_time /= 1000;  //Execution time in seconds
```

Finally, we print the execution time of the parallel computing's part and we clear the environment both in the CPU (u_old, u_new) and GPU (d_u_old, d_u_new):

```
    // Copying the final grid back to the CPU
    cudaMemcpy(u_new, d_u_old, N * N * sizeof(float), cudaMemcpyDeviceToHost);

    // Showing the parallel execution time
    printf("Parallel execution time: %3f s \n", exec_time);

    // Free memory
    free(u_old);
    free(u_new);
    cudaFree(d_u_old);
    cudaFree(d_u_new);

    return 0;
}
```

Bash Code (HeatEquation.sh)

To run the code we create a bash file.

```
#!/bin/bash

echo "Compile program"
nvcc HeatEquation.cu -o HeatEquation
echo


echo "Execute program"
srun -N 1 -n 1 --gpus-per-task=1 -p gpus --reservation=maintenance ./HeatEquation
echo

echo "Clear temporary file"
rm HeatEquation
echo
```

First we compile the program invoking the NVIDIA CUDA compiler (nvcc) to compile the CUDA source file HeatEquation.cu and generate an executable file named HeatEquation.

Then we execute the program, specifing the number of nodes, number of tasks, number of GPUs per task, partition, and reservation needed for running the program.

Finally we clean up any temporary files generated during the compilation or execution.

Experimental results and conclusion

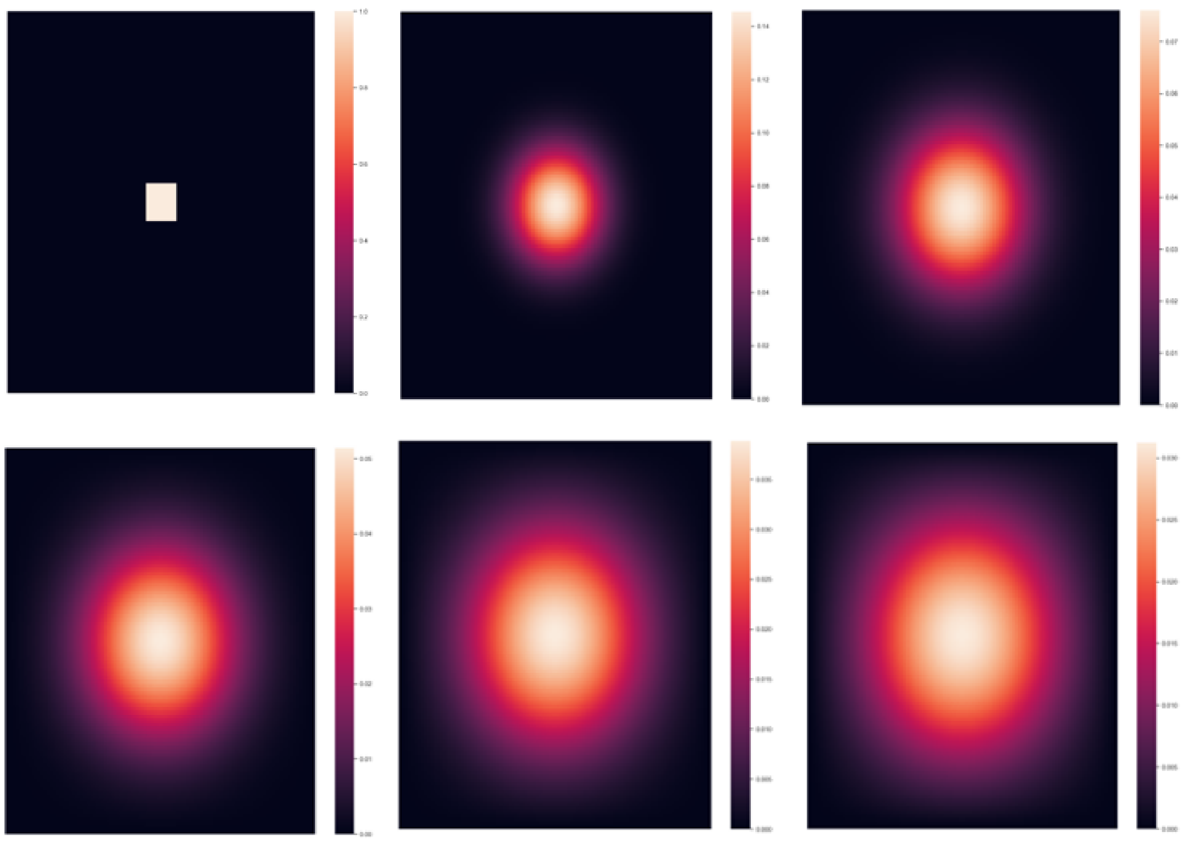
In this section, we present the results obtained from our project.

As we saw in the code, the sections that were about printing the grids in a text file were commented. That's because the parameter N was set to 10000. With a so big grid dimension, the text files given in output are close to 1GB each. Given that this grids' printing is inside the parallel computing part of the code, the execution time is largely increased. But we are interested only in the time it takes to perform the computation, not also the visualization.

That's why in order to have a visualization we used the following parameters:

- N = 100
- $SQUARE\_SIZE = 10$
- $BLOCK\_SIZE = 32$
- numSteps = 1000
- outputNum = 200
- ALPHA = 0.1

So every 200 iterations the code generated a text file to save the grid. This is the result:



We can see the heat expansion through the entire surface. It's also interesting that the heat maximum change from one heatmap to the other. This because the model obviously considers the energy conservation. So, if the sum of the heat values with the initial grid is 100, it will be 100 for every updated grid.

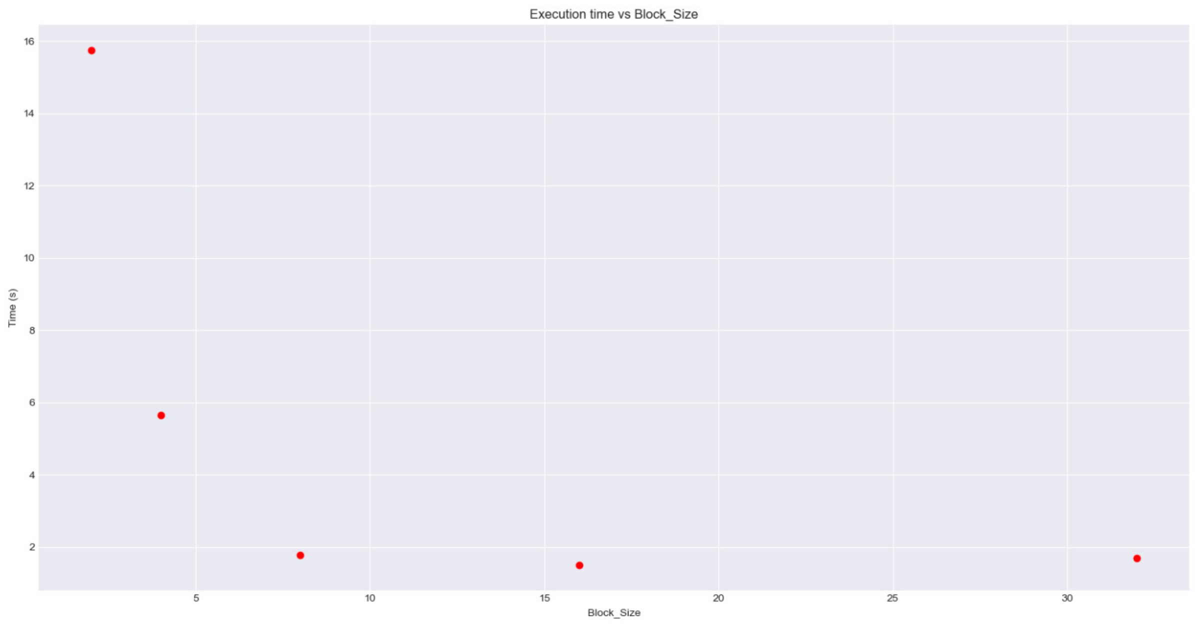
With these parameters and without visualization (so without printing the text file) the parallel execution time is: 2.92 milliseconds.

However N=100 does not justify the parallel computations. Moreover we are interested to see how the execution time changes related to the variable $BLOCK\_SIZE$ so we decide to repeat the experiment with these parameters:

- N = 10000
- $SQUARE\_SIZE = 1000$
- numSteps = 1000
- ALPHA = 0.1

As regards the variable $BLOCK\_SIZE$ we use the values: $[2, 4, 8, 16, 32]$.

Finally we create a graphic Execution time vs $BLOCK\_SIZE$ and it indicates that the execution time is decreasing as the number of threads per block increases. As a result we can observe a decreasing exponential trend.


Execution time vs Block_Size
Time (s)
Block_Size