Parallel Computing Project

Department of Computer Science and Media Tech

Andreas Christopoulos Charitos [ac222yf]

Linnéuniversitetet

Table of contents

Introduction

Introduction

Welcome to my presentation on parallel computing project.

What will be presented?

- hardware and software used during the simulations
- problem definition
- code used during the simulations
- results of the simulations
- conclusion and key points

Experimentation environment

Experimentation environment-Hardware

In this section we will introduce the hardware as well as the software used during the simulations.

Hardware

Below we can see an overview of the hardware used

Host (CPU)

- Intel Xeon CPU E5-2650 v4
- 2.2 - 2.9 GHz
- 12 CPU cores
- 24 Threads per CPU
- 30720 KB memory cache

Experimentation environment-Hardware

Hardware (cont.)

Device (GPU)

- 1 x GeForce GTX Titan X GPU
- 1 – 1.1 GHz
- 12 GB Memory
- 3072 cores
- Maximum threads per block 1024

Experimentation environment-Software

Software
The software used

OpenMP [1]

- an implementation of multi-threading
- it is a method of parallelizing a primary thread (a series of instructions executed consecutively), divide a task among a specified number of sub-threads
- after the execution of the parallelized code, the sub-threads are joined back into the primary thread, which then continues to the end of the program
- homogemous computing, 1. load data to RAM then process with CPU(single or multi-thread) 2. finally store the result
- figure.1 provides an illustration of multi-threading process

[1] Open Multi-Processing (OpenMP)

Experimentation environment-Software

Software(cont.)

CUDA [2]

- a parallel computing platform and application programming interface (API) model created by Nvidia using GPUs
- GPUs design is more effective than CPUs allowing very efficient malipulation of large blocks of data
- heterogenous / accelerated programming, a part of the code (the most intensive tasks) is executed on GPU and the rest on CPU
- 1. load data to RAM 2. pre-process on CPU 3. load data to GPU memory and process 4. finally copy the data back to RAM and store the results
- figure.2 provides an illustration of CUDA workflow

[2] Compute Unified Device Architecture (CUDA)

Experimentation environment-Software

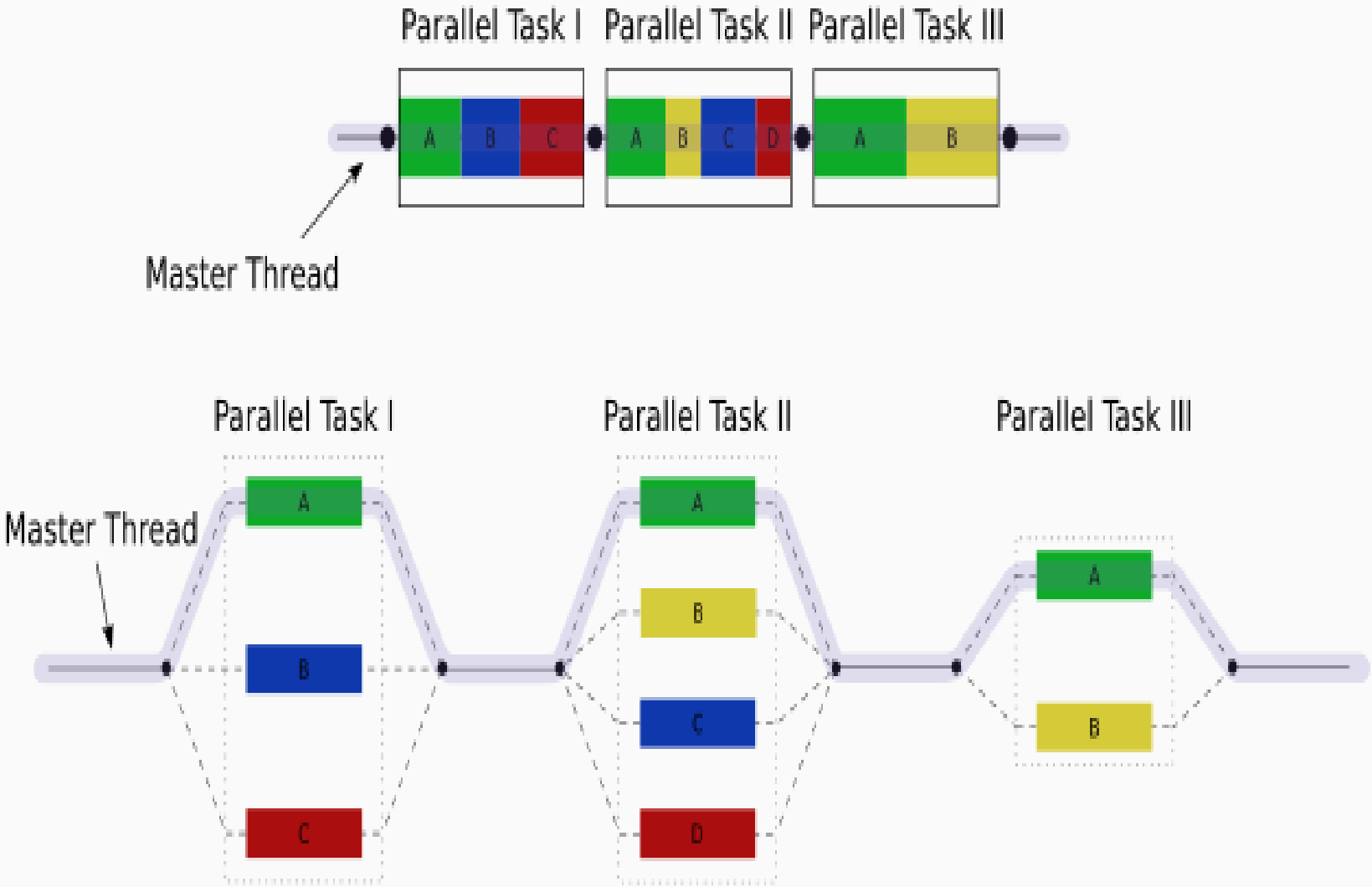

Figure 1: Multithreading illustration

Experimentation environment-Software

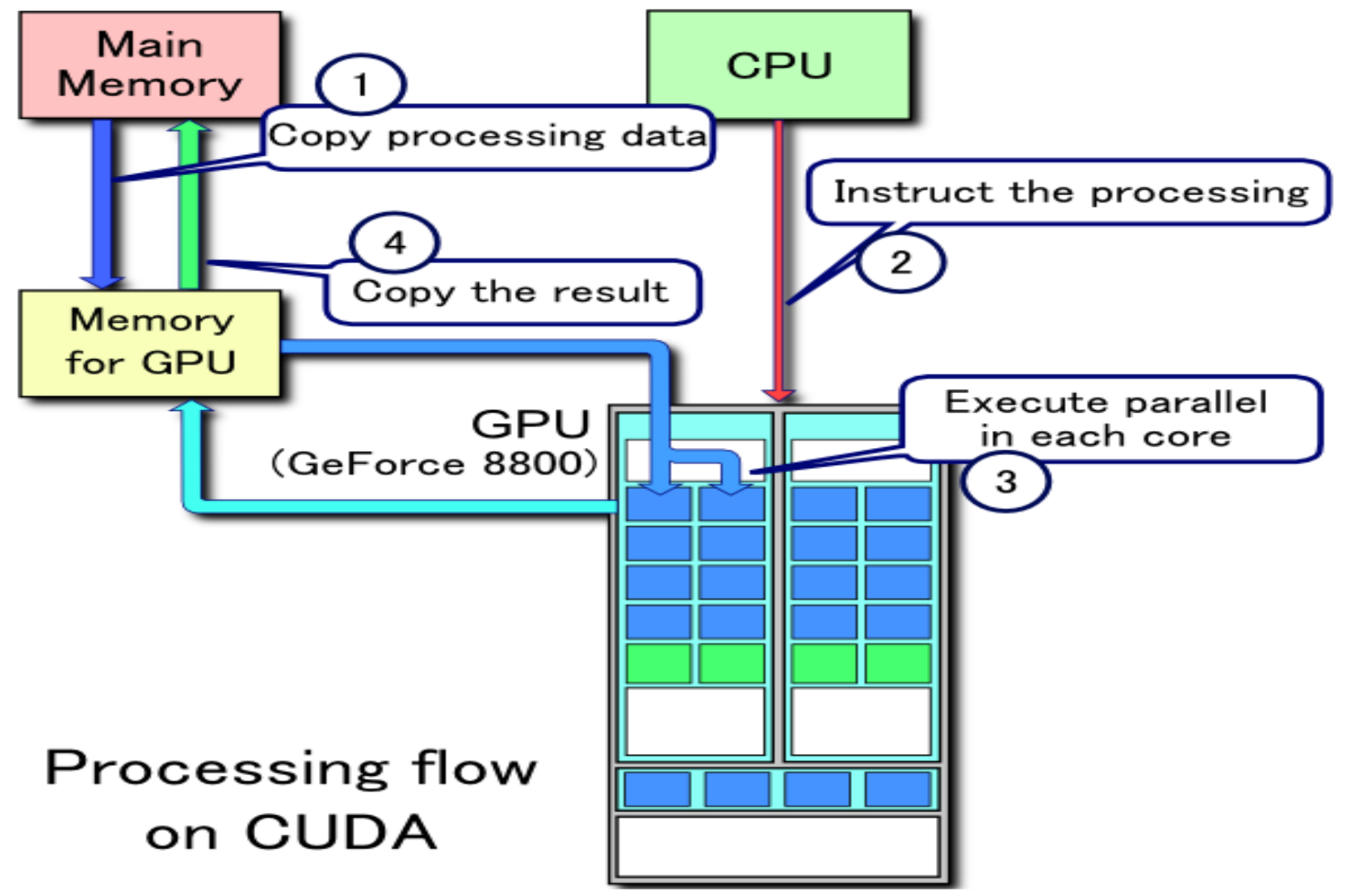

Figure 2: CUDA processing flow

Problem definition

Addressed problem

What is the addressed problem?

The π (ratio of the circumference of a circle to its diameter) can be approximated using the integral:

$$\int_0^1 \frac{4}{(1+n*n)} dn = \pi \tag{1}$$

or the following equation:

$$\sum_{i=0}^{iterations} \frac{4}{(1+n_i*n_i)} \Delta n \approx \pi \tag{2}$$

A sequential C-code to approximate π is given in slide 11.

Addressed problem

The problem consists of 2 tasks:

1. Task 1: Calculate π in parallel using two 12-core CPUs on Ida
   - Parallelize the code using OpenMP
   - Measure and visualize execution time and speedup for 1, 6, 12, 24 and 48 threads, for each of the following numbers of iterations: 24000000, 48000000, 96000000
2. Task 2: Calculate π in parallel using the GPU on Ida (optional)
   - Parallelize the code using CUDA
   - Measure and visualize (for instance, using Excel charts) execution time and speedup, for each of the following numbers of iterations: 24000000, 48000000, 96000000

Addressed problem

```c
#include <stdio.h>
#include <stdlib.h>
#define PI 3.14159265358979323846264

int main (int argc, char *argv[]){
double m, ni, mypi= 0.0;
int i, iterations;
iterations=24000000;
m= 1.0 / (double)iterations;

for (i = 0; i < iterations; i++){
        ni = ((double)i + 0.5) * m;
        mypi += 4.0 / (1.0 + ni * ni);
}
mypi *= m;
printf("MyPI = %.20lf\n", mypi);
printf("MyPI -PI = %.20lf\n", (mypi -PI));
}
```

Code

Code parallelization OpenMP

```
1  #include <stdio.h>
2  #include <omp.h>
3  #define PI 3.14159265358979323846264
4
5  void func(int iterations, int num_threads)
6  {
7      const double m = 1.0 / (double)iterations;
8      double sum = 0.0;
9      omp_set_num_threads(num_threads);
10     double start_time = omp_get_wtime();
11     #pragma omp parallel for reduction(+: sum)
12     for (int i = 0; i < iterations; i++)
13     {
14         double ni = (i + 0.5) * m;
15         sum = sum + 4.0 / (1.0 + ni * ni);
16     }
17     double MyPI = m * sum;
18     double run_time = omp_get_wtime() - start_time;
19     double error = MyPI -PI;
20     printf("%.23lf,%.23lf,%f", MyPI, error, run_time);
21 }
22 void main()
23 {
24     for (int i = 0; i < 5000; ++i)
25     {
26         func(#iterations, #threads);
27         printf("\n");
28     }
29 }
```

Code parallelization CUDA

```
1   #include "cuda_runtime.h"
2   #include <stdio.h>
3
4   #define ITERATIONS #iterations
5   const int threads = 1024; // nVidia's optimal block size should be muptiple of 32
6
7   // Synchronous error checking call. Enable with nvcc -DDEBUG
8   inline void checkCUDAError(const char *fileName, const int line)
9   {
10      #ifdef DEBUG
11          cudaThreadSynchronize();
12          cudaError_t error = cudaGetLastError();
13          if(error != cudaSuccess){
14          printf("Error at %s: line %i: %s\n", fileName, line, cudaGetErrorString(error));
15          exit(-1);
16          }
17      #endif
18  }
19  __global__ void integrateSimple(float *sum)
20  {
21      __shared__ float ssums[threads];
22      // Each thread computes its own sum.
23      int global_idx = threadIdx.x + blockIdx.x * blockDim.x;
24      if(global_idx < ITERATIONS){
25          float step = 1.0f / ITERATIONS;
26          float x = (global_idx + 0.5f) * step;
27          ssums[threadIdx.x] = 4.0f / (1.0f + x * x);
28      }
29      else{
30      ssums[threadIdx.x] = 0.0f;
31      }
32      // The 1st thread will gather all sums from all other threads of this block into one
```

Code parallelization CUDA (cont.)

```
  __syncthreads();
    if(threadIdx.x == 0)
    {
    float local_sum = 0.0f;
        for(int i = 0; i < threads; ++i)
        {
        local_sum += ssums[i];
        }
        sum[blockIdx.x] = local_sum;
        }
}
int main()
{
    const float PI = 3.14159265358979323846264;
    int deviceCount = 0;
    cudaError_t error = cudaGetDeviceCount(&deviceCount);
    if (error != cudaSuccess)
    {
        printf("cudaGetDeviceCount returned %d\n-> %s\n", (int)error, cudaGetErrorString(error));
        return 1;
    }
    deviceCount == 0 ? printf("There are no available CUDA device(s)\n") :
                                                printf("%d CUDA Capable device(s) detected\n" , deviceCount);
    /*--------- Simple Kernel ---------*/
    int blocks = (ITERATIONS + threads - 1) / threads;
    float *sum_d;
    float step = 1.0f / ITERATIONS;
```

Code parallelization CUDA (cont.)

```
    for (int i = 0; i < 5000; ++i)
    {   // Allocate device memory
        cudaMallocManaged((void **)&sum_d, blocks * sizeof(float));
        // CUDA events needed to measure execution time
        cudaEvent_t start, stop;
        float gpuTime;
        // Start timer
        cudaEventCreate(&start);
        cudaEventCreate(&stop);
        cudaEventRecord(start, 0);
        // calculate pi
        integrateSimple<<<blocks, threads>>>(sum_d);
        cudaDeviceSynchronize(); // wait until the kernel execution is completed
        checkCUDAError(__FILE__, __LINE__);
        // Stop timer
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&gpuTime, start, stop);
        // Sum result on host
        float MyPI = 0.0f;
        for (int i = 0; i < blocks; i++)
        {
            MyPI += sum_d[i];
        }
        MyPI *= step;
        cudaFree(sum_d);
        /*printf("\n=====================================\n\n");*/
        printf("%.23lf,%.23lf,%f", MyPI, fabs(MyPI - PI), gpuTime/1000);
        printf("\n");
    }
    // Reset Device
    cudaDeviceReset();
    return 0;}
```

Code compilation

The code used for the compilation of the scipts is shown below

```
gcc FILE_NAME.c -O2 -fopenmp -o FILE_NAME  //OpenMP

nvcc FILE_NAME.cu -o FILE_NAME --gpu-architecture=compute_35 //CUDA
```

Results

Experimental results OpenMP

As mentioned previously Task 1 consists of 2 tasks:

- Parallelize code using OpenMP
- Measure and visualize execution time and speedup for 1, 6, 12, 24 and 48 threads, for each of the following numbers of iterations:
  - 24000000
  - 48000000
  - 96000000

Experimental results OpenMP

In order to preform statistical analysis we follow the next steps:

1. run the code for 5000 times per iteration and thread
2. remove the outlier times from the data
3. in order to perform the step 2, observations with times that are lower than the 25% and greater than 75% (1st and 3rd quantile of the data) are removed from the dataset
4. perform statistical analysis with histograms for every combination (fig.3, fig.4, fig.5), a line plot (fig.6) of the mean times and a summary table 1

Experimental results OpenMP

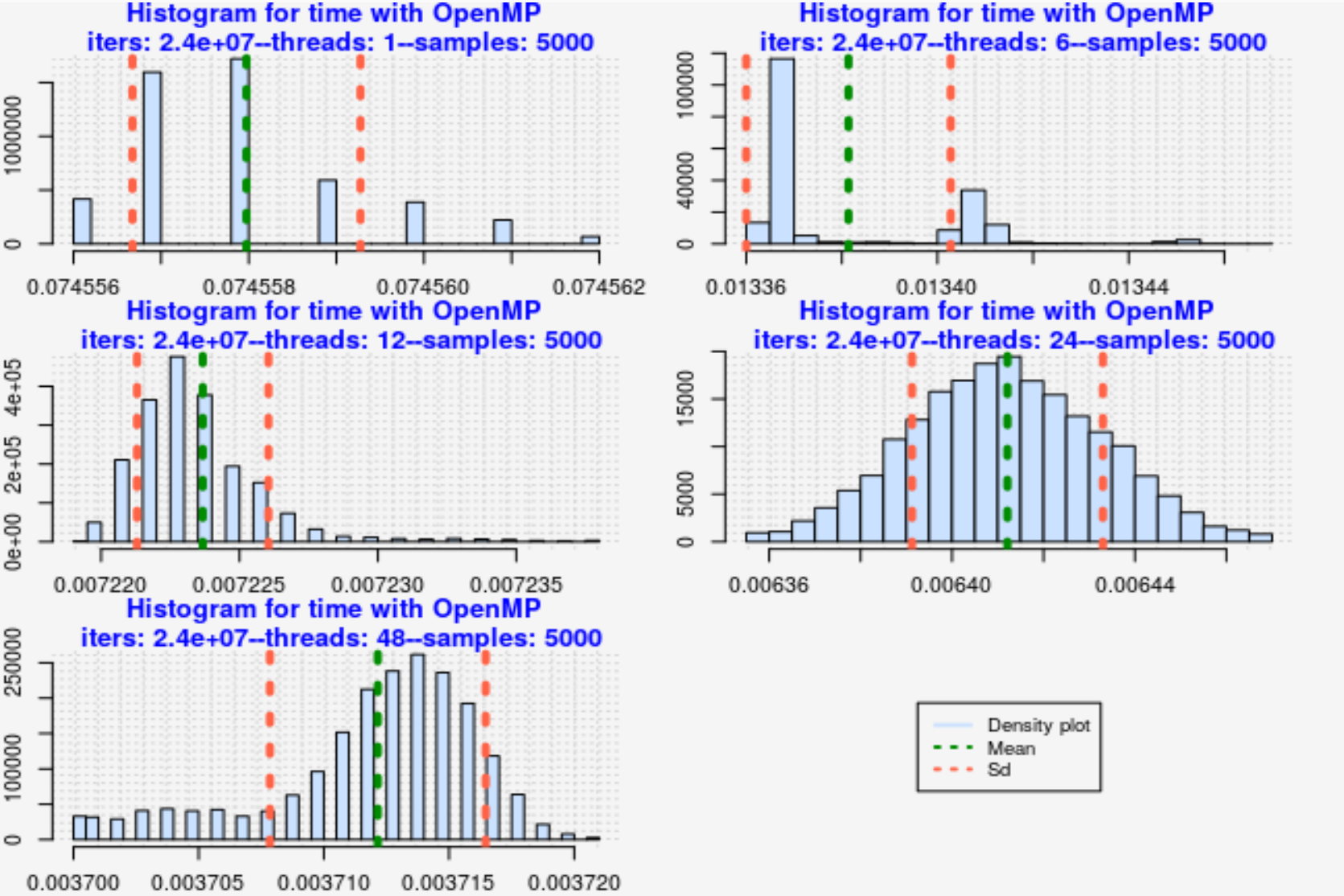

Figure 3: Histograms for 24000000 iterations with OpenMP

Experimental results OpenMP

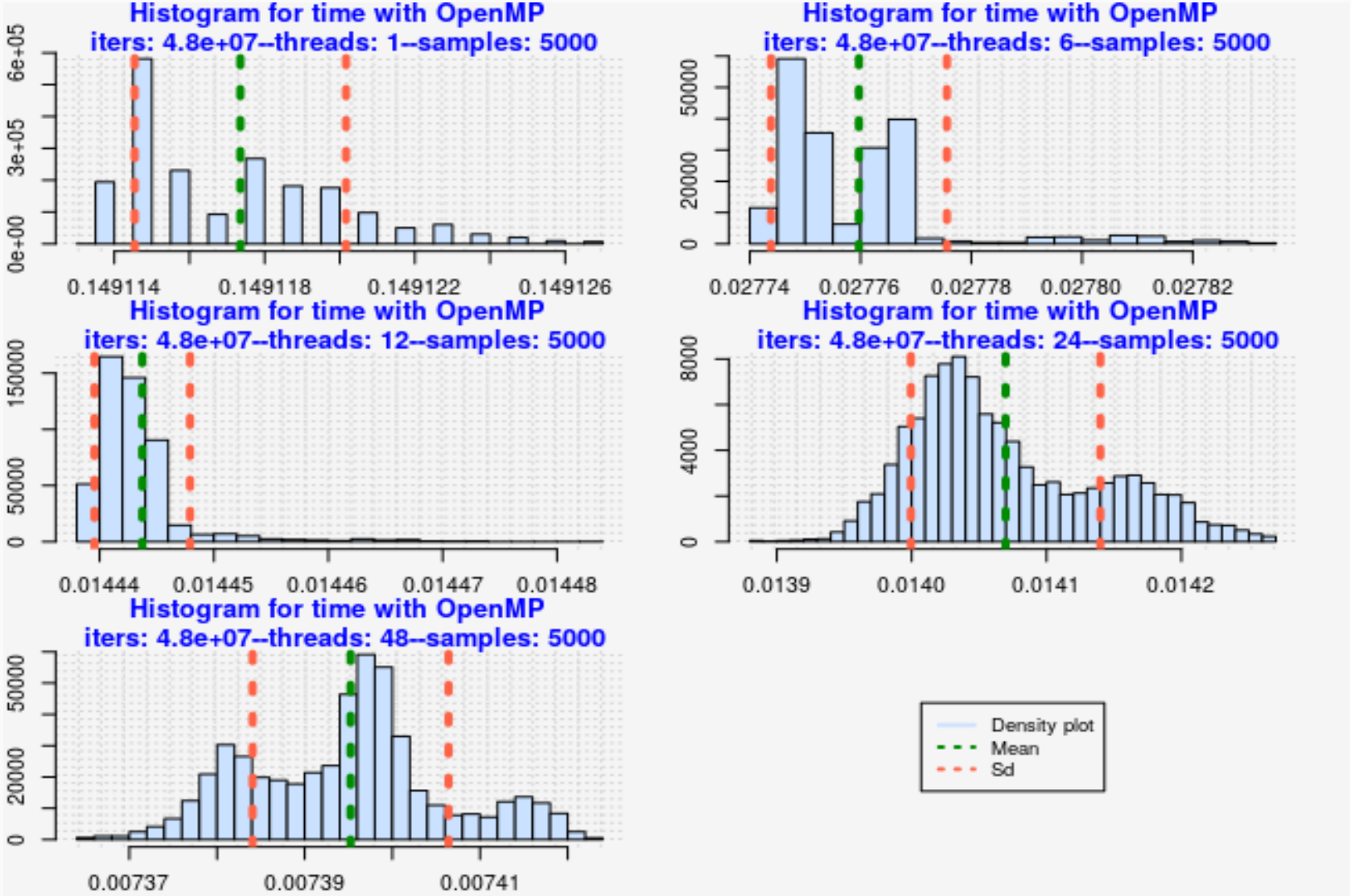

Figure 4: Histograms for 48000000 iterations with OpenMP

Experimental results OpenMP

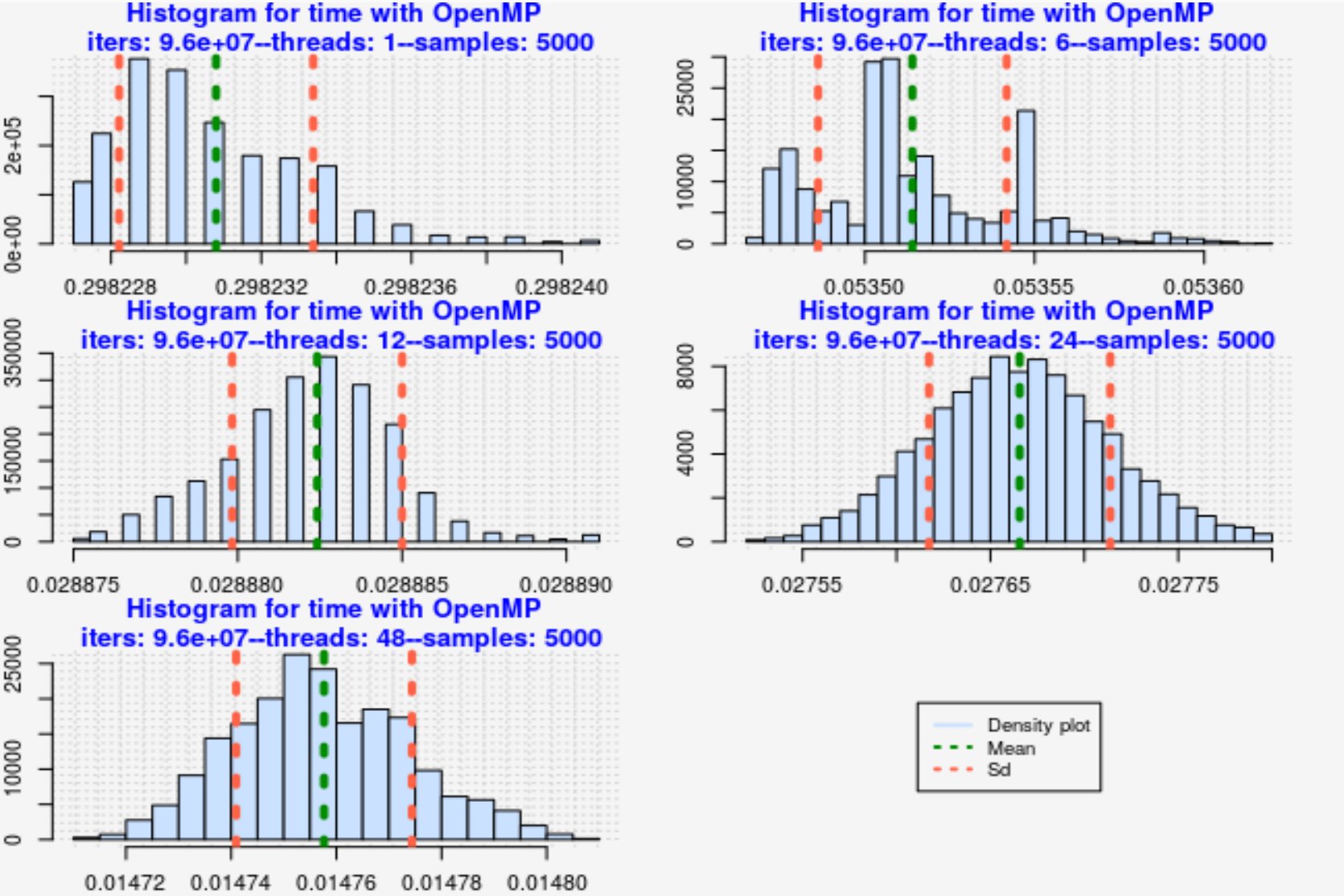

Figure 5: Histograms for 96000000 iterations with OpenMP

Experimental results OpenMP

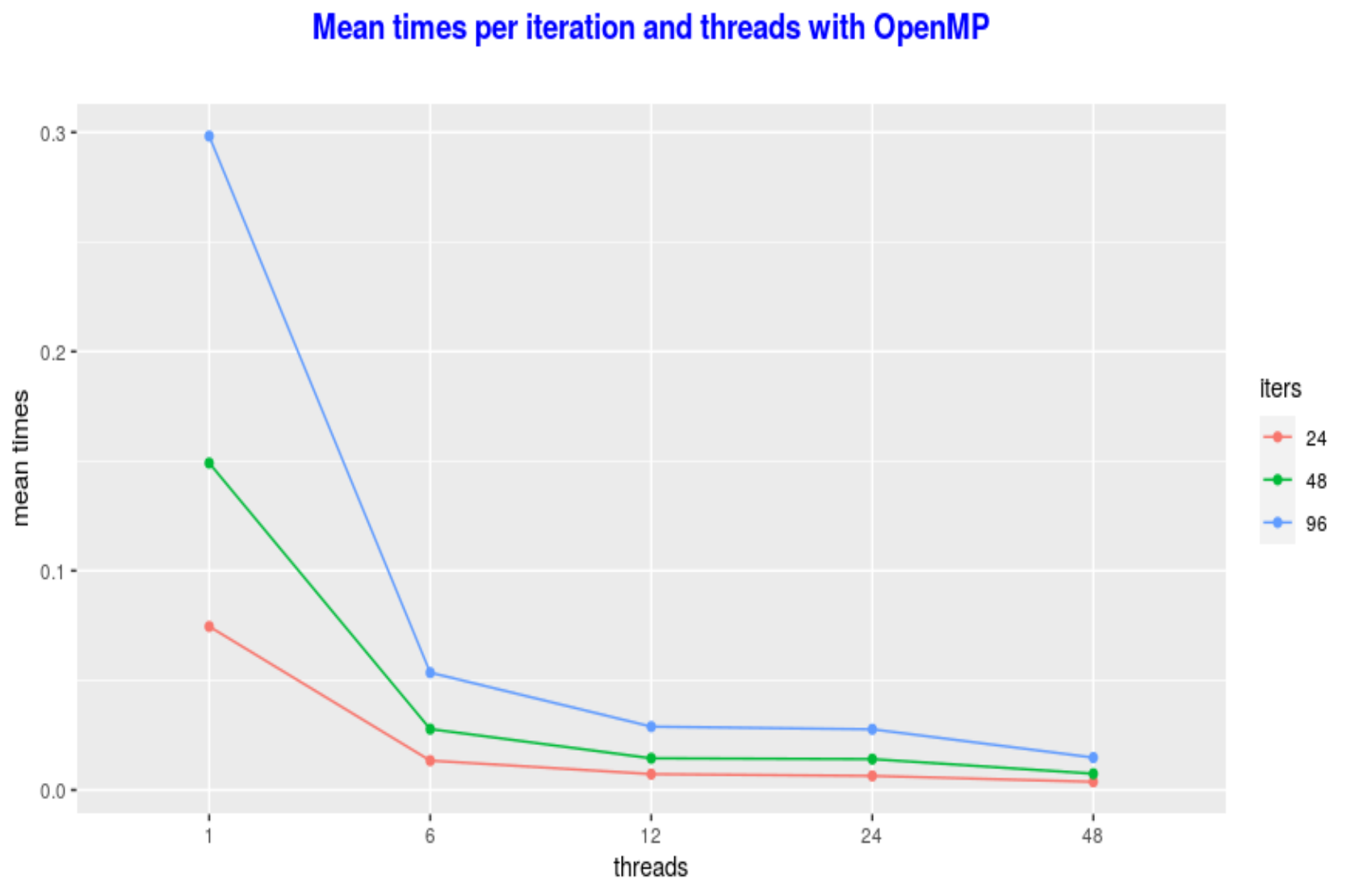

Figure 6: Line plot of mean times per iterations and threads

Experimental results OpenMP

iterations \ threads	1	6	12	24	48	serial
24000000	0.0746	0.0134	0.00722	0.00641	0.00371	0.0750
48000000	0.149	0.0278	0.0144	0.0141	0.00740	0.0150
96000000	0.298	0.0535	0.0289	0.0277	0.0148	0.30

Table 1: Mean times with OpenMP and serial code. The table shows the mean times for 5000 simulation steps for the serial code and the code parallelized with OpenMP.

Experimental results CUDA

As mentioned previously Task 2 consists of 2 tasks:

- Parallelize the code using CUDA
- Measure and visualize (for instance, using Excel charts) execution time and speedup, for each of the following numbers of iterations:
  - 24000000
  - 48000000
  - 96000000

Experimental results CUDA

we follow the same steps as with OpenMP:

1. run the code for 5000 times per iteration and thread
2. remove the outlier times from the data
3. in order to perform the step 2, observations with times that are lower than the 25% and greater than 75% (1st and 3rd quantile of the data) are removed from the dataset
4. perform statistical analysis with histograms for every iteration and a bar plot of the mean times (fig.7), bar plot (fig.8)

Experimental results CUDA

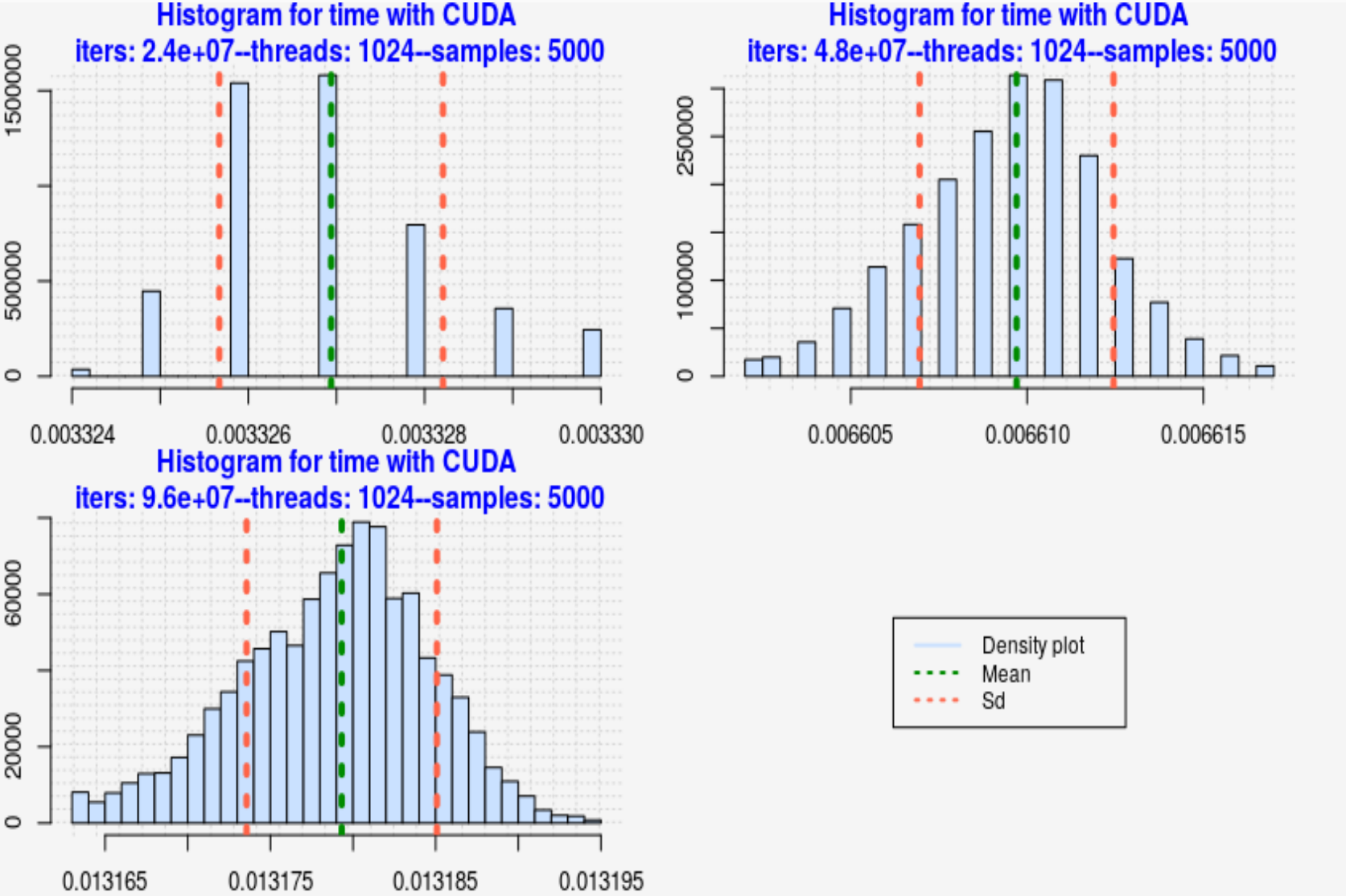

Figure 7: Histograms for different iterations with CUDA using 1024 threads

Experimental results CUDA

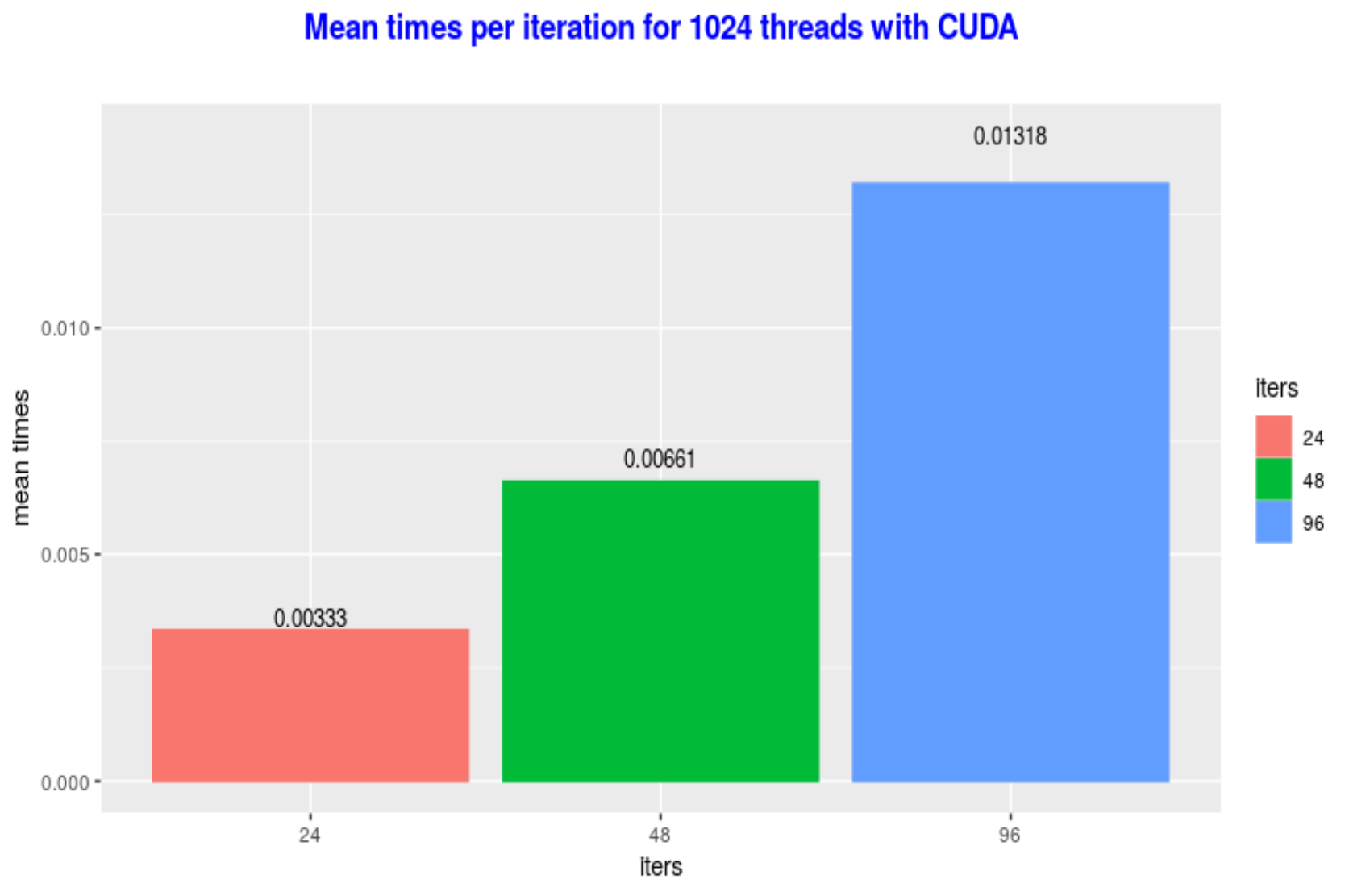

Figure 8: Bar plot of mean times per iterations with CUDA

Experimental results speedup

Speedup Results

- figure 9 illustrates the speedup with OpenMP compared with serial code for the different number of iterations and threads
- figure 10 illustrates the speedup with CUDA compared with serial code for each iteration combination
- table 2 illustrates the speedup with CUDA compared with 48 threads in OpenMP implementation

Experimental results speedup OpenMP

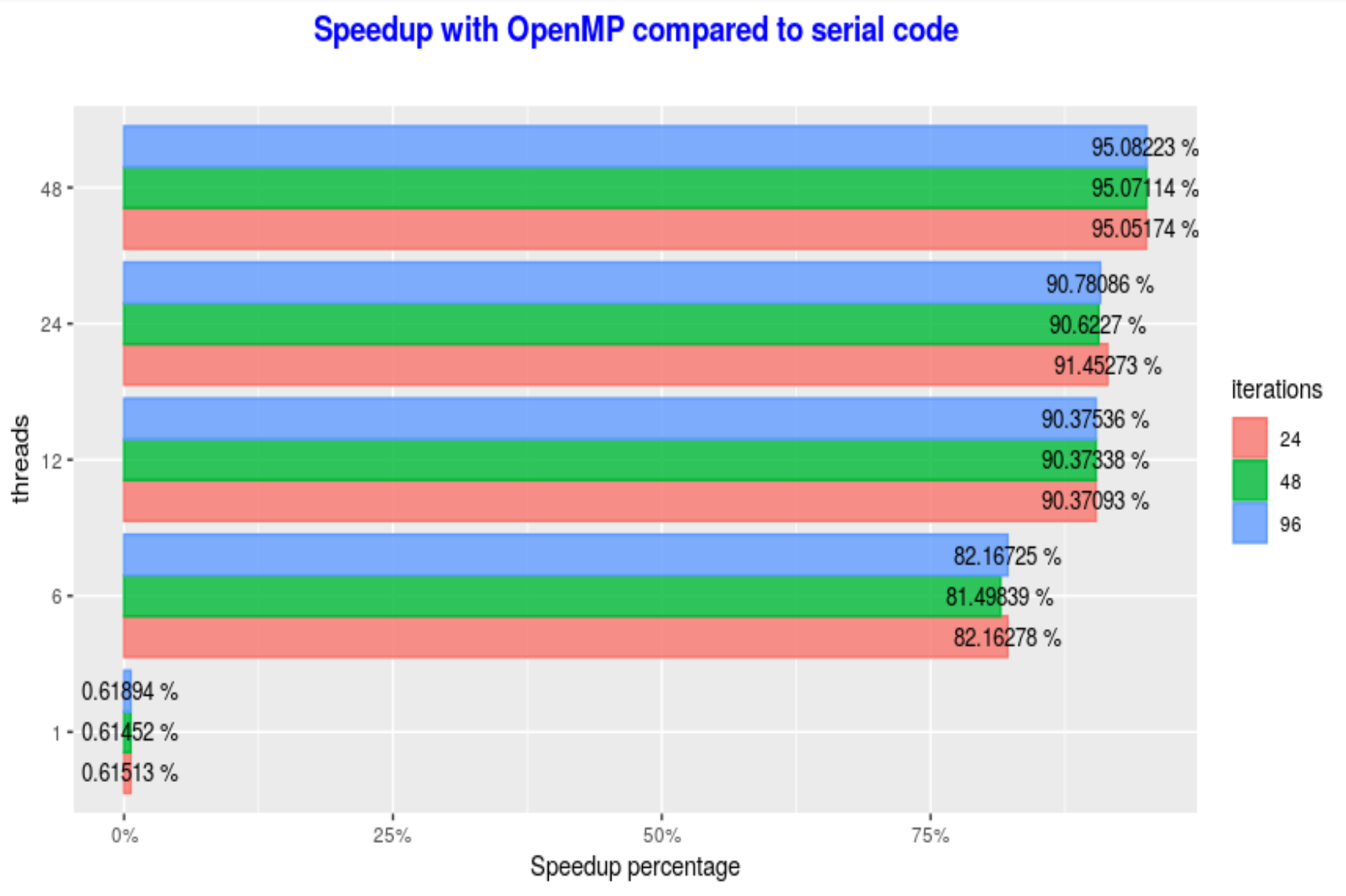

Figure 9: Speedup of time with OpenMP compared to serial code

Experimental results speedup CUDA

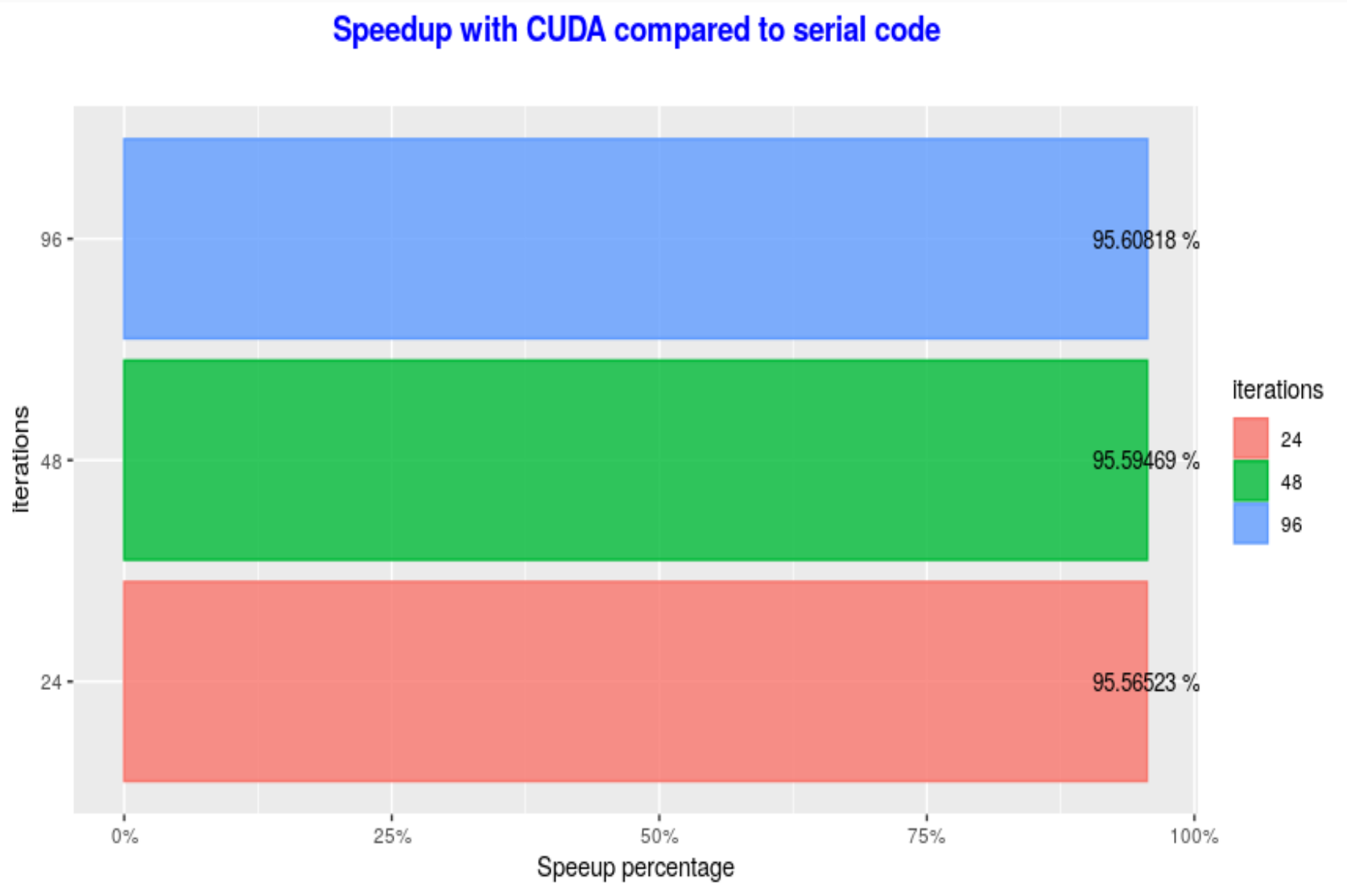

Figure 10: Speedup of time with CUDA compared to serial code

Experimental results speed up OpenMP and CUDA

iterations	OpenMP	CUDA	speedup
24000000	0.00371	0.00333	10.2426%
48000000	0.00740	0.00661	10.6757%
96000000	0.0148	0.0132	10.8108%

Table 2: CUDA speedup results table for mean times. The table illustrates the speed up from OpenMP (48 threads) and CUDA (1024 threads-max)

Conclusion

Conclusion and key points

- it is normal that with more iterations the measured times are increasing
- from the histograms with OpenMp we can see that with 24 threads the distribution of time tends to follow the normal distribution
- from the histograms with CUDA we can also see that with more iterations the distribution of times tend to follow the normal distribution
- moreover, from the line plot of measured times with OpenMP we can see that after the 6 threads the increase on times is very small for the same number of threads as iterations increase

Conclusion and key points

- parallelizing the code using 48 threads in OpenMP the speedup is roughly 95% for each iteration combination and the speedup with CUDA is 95.6%
- in addition, parallelizing the code on GPU the execution times are smaller, and the speedup with CUDA compared to OpenMP is between 10-11%
- the code used for parallelizing the code with CUDA can be further optimized using sum reduction (not implemented)

Thank you!
Any questions?