Linnæus University
Sweden

Report

Parallel Programming

Supercomputing programming with openmp and CUDA

Author: Helena Tevar
Supervisor: Sabri Pllama
Semester: HT19
Course code: 2DV605

Linnæus University
Sweden

Table of contents

Methodology

This assignment was made using Windows 10 as OS and tools as WinSCP for file transferring and PuTTy to execute the code in IDA and run the two tasks in two different files. The files can be found on IDA Helena/2DV605 folder, files `task1.c` and `taks2.cu`, in the repository https://github.com/LenaTevar/2DV605, and in the annex of this report.

This report shows the resulted execution time of the calculation of PI using different machines. In order to understand the reality of the results returned by IDA's supercomputer, I also executed the program created for this assignment in a standard computer and compared the results of all the machines.

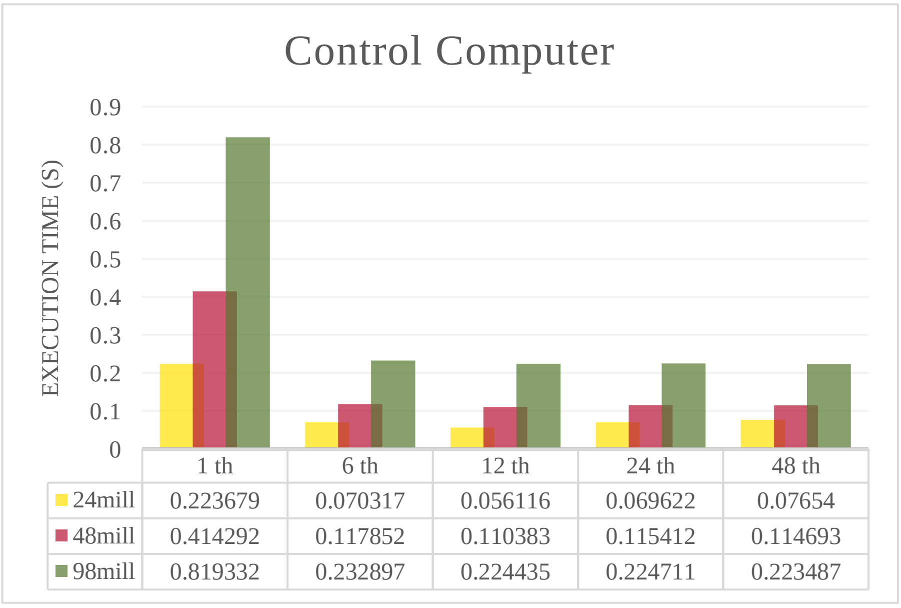
Using OpenMP

The program created for the assignment has a function called calcPI that was created by extracting the PI calculation provided to ease the manipulation of the threads and iterations. Inside of it is located the execution time evaluation by using *openMP* functionality `omp_get_wtime()`. The piece of code running in parallel is the for-loop that runs the PI algorithm. It works in a number of threads provided by the arguments of the method and it protects the shared variables `mypi` and `ni` by using *openMP*'s function `reduction()`.

The main method runs two nested for loops that manage the two variables that we had to modify for this assignment, being the number of threads and the iterations.

Standard Computer

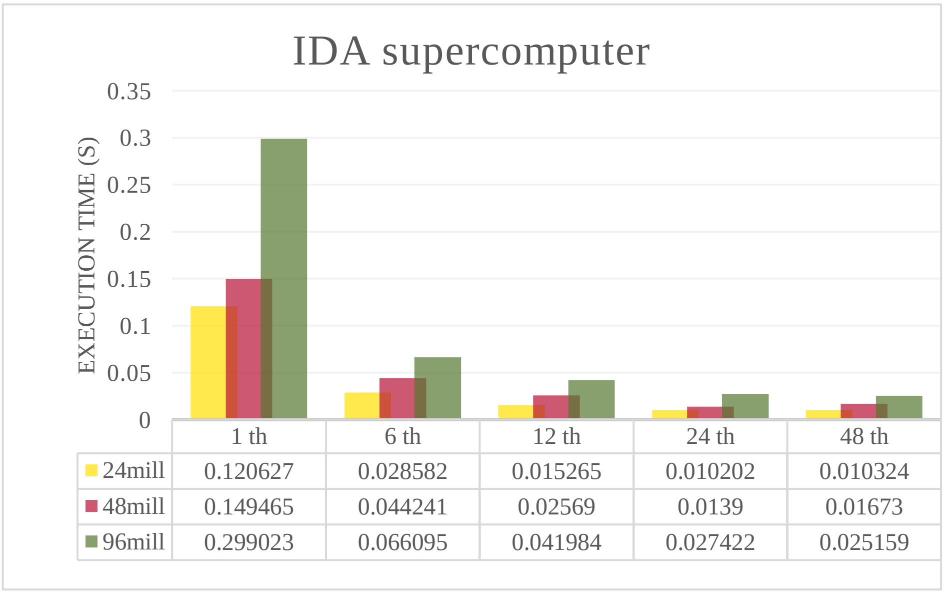
The computer used as control was a hp laptop with a processor Intel Core i7-6500U, 2 cores with 4 logic processors. The results with this computer were as follows.



	1 th	6 th	12 th	24 th	48 th
🟨 24mill	0.223679	0.070317	0.056116	0.069622	0.07654
🟥 48mill	0.414292	0.117852	0.110383	0.115412	0.114693
🟩 98mill	0.819332	0.232897	0.224435	0.224711	0.223487

IDA supercomputer

After executing the code in IDA, the results showed a big improvement in execution time in comparison with an ordinary computer, except the iteration that executed 48 threads, that was slightly worse than the 24-thread execution.



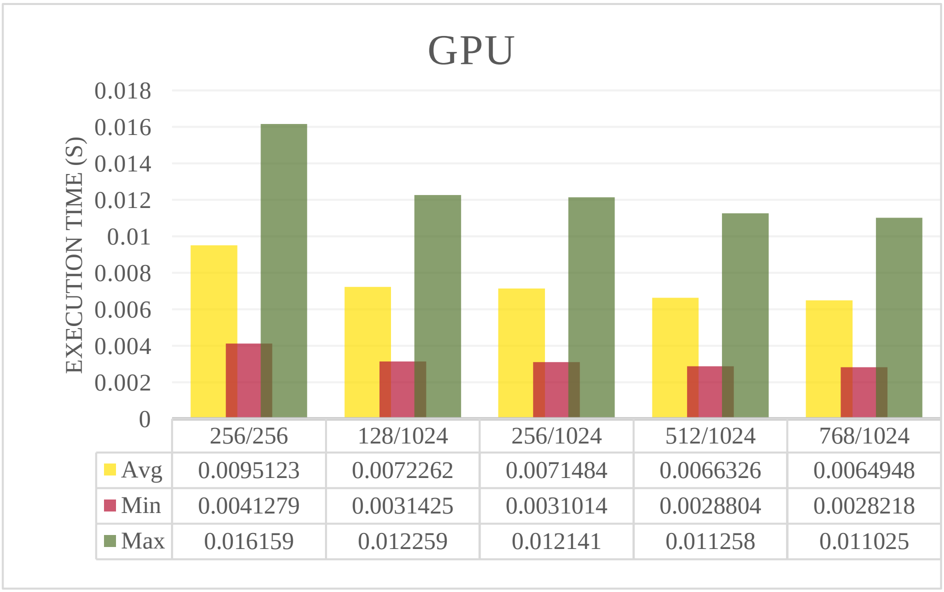
	1 th	6 th	12 th	24 th	48 th
🟨 24mill	0.120627	0.028582	0.015265	0.010202	0.010324
🟥 48mill	0.149465	0.044241	0.02569	0.0139	0.01673
🟩 96mill	0.299023	0.066095	0.041984	0.027422	0.025159

CUDA programming

In this occasion I modified the code from openMP so it will work using CUDA and IDA GPU. The algorithm remains as in the previous task with the only difference being that the only part that has been programmed to run in parallel has been the for-loop instead the whole algorithm, so CPU was the one that finish the calculations. For this part of the assignment I tried different number of grids and threads to test the GPU execution times and used `nvprofile` to get the execution time of the loop.

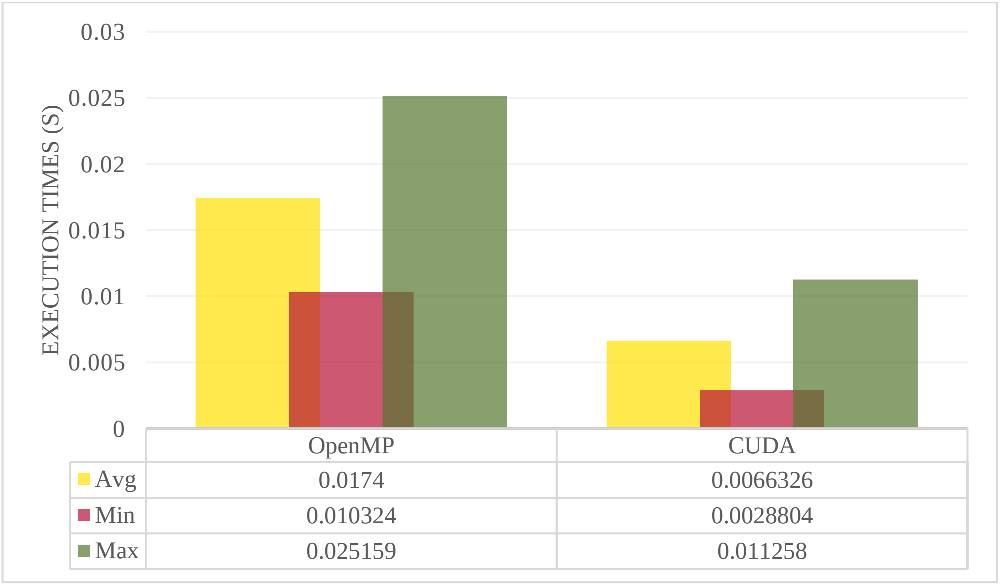
Results

The profile resulted in IDA showed the average execution time of the three number of iterations, the longest execution time and the smallest execution time of the three runs. The improvement showed that the slowest time in GPU, with 256 blocks and 256 threads in 0.016159 seconds, is close to the fastest run in OpenMP.



	256/256	128/1024	256/1024	512/1024	768/1024
🟨 Avg	0.0095123	0.0072262	0.0071484	0.0066326	0.0064948
🟥 Min	0.0041279	0.0031425	0.0031014	0.0028804	0.0028218
🟩 Max	0.016159	0.012259	0.012141	0.011258	0.011025

OpenMP vs CUDA



	OpenMP	CUDA
🟨 Avg	0.0174	0.0066326
🟥 Min	0.010324	0.0028804
🟩 Max	0.025159	0.011258

As the data shows, the slowest run in CUDA is closer to the smallest execution time in OpenMP with an average improvement of 37% while using GPU accelerators instead of CPU parallelization.

Appendix I

OpenMP Code

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define PI 3.1415926535897932384626‌4



void calcPI(int iterations, int threads) {
    double m, mypi = 0.0;
    double ni = 0.0;
    double difTime;
    int i;
    double start = omp_get_wtime();

    m = 1.0 / (double)iterations;
    //OBS!!! catch problems with shared variables!!!
#pragma omp parallel for num_threads(threads), reduction(+: mypi
), reduction(+:ni)
    for (i = 0; i < iterations; i++)
    {
        ni = ((double)i + 0.5) * m;
        mypi += 4.0 / (1.0 + ni * ni);
    }

    mypi *= m;
    double end = omp_get_wtime();
    difTime = end - start;

    printf("Iterations %d - threads %d\tExecution time %f\n", it
erations, threads, difTime);
    printf("\tMyPI = %.70f\n", mypi);
    printf("\tMyPI - PI = %.70f\n\n", (mypi - PI));


}



int main(int argc, char* argv[])
{
    int iteLength = 3;
    int iteArr[3] = { 24000000, 48000000, 96000000};
    int thLength = 5;
    int threads[5] = {1,6,12,24,48};

    for (int thIndex = 0; thIndex < thLength; thIndex++)
    {
        for (int iteIndex = 0; iteIndex < 3; iteIndex++) {
            calcPI(iteArr[iteIndex], threads[thIndex]);
        }
    }

}
```

Appendix II

Cuda Code

```
#include <stdio.h>
#include <cuda.h>
/*
Tested blocks/threads:
256/256
256/1024
128/1024
512/1024
768/1024
*/
#define NUM_BLOCK  768
#define NUM_THREAD  1024
#define PI  3.14159265358979323846

/* Kernel function */
__global__ void cal_pi(double *mypi, int iter,
    double m, int nthreads, int nblocks) {
    int i;
    double ni;
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    // Sequential thread index across the blocks
    for (i = idx; i< iter; i += nthreads * nblocks) {
        ni = (i + 0.5) * m;
        mypi[idx] += 4.0/(1.0 + ni * ni);
    }
}


int main(void) {
    double pi = 0;

    int iteArr[3] = { 24000000, 48000000, 94000000 };
    //double iteArr[3] = { 24000000000, 48000000000, 94000000000
 };
    //int iteArr[3] = { 1, 2, 4 };

    /* Setting upt grid and block dimesions */
    dim3 dimGrid(NUM_BLOCK,1,1);
    dim3 dimBlock(NUM_THREAD,1,1);
    printf("REPORT # of blocks = %d, # of threads/block = %d\n", 
NUM_BLOCK, NUM_THREAD);

    /* Host and Device variables (arrays) */
    double *h_pi, *d_pi;
    for (int i = 0; i < 3; i++){
        int currentIter = iteArr[i];

        double step = 1.0 / currentIter;
        size_t size = NUM_BLOCK*NUM_THREAD*sizeof(double);

        /* Allocate on host*/
        h_pi = (double *)malloc(size);

        /*Allocate on device*/
        cudaMalloc((void **) &d_pi, size);
        /* Set d_pi to zero */
        cudaMemset(d_pi, 0, size);
        /* Run Kernel */
        cal_pi <<<dimGrid, dimBlock>>> (d_pi, currentIter, 
            step, NUM_THREAD, NUM_BLOCK);

        /* Copy results from device to the host*/
        cudaMemcpy(h_pi, d_pi, size, cudaMemcpyDeviceToHost);

        /* Finish pi in host */
        for( int j = 0; j < NUM_THREAD*NUM_BLOCK; j++)
            pi += h_pi[j];
           pi *= step;
           printf("\tMyPI = %20.18f \n",pi);
           printf("\tMyPI - PI = %20.18f \n",pi-PI);
       }

    printf("\tCheck nvprof for more time estimation.\n\n");

    /* Clean host and device var*/
    free(h_pi);
    cudaFree(d_pi);

    return 0;
}
```