HPC Lab5 Report

Konstantinos Konsoulas
Ioannis Roumpos

January 2023

Contents

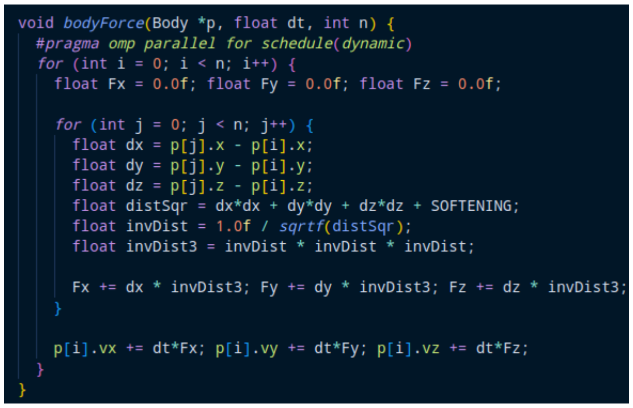
1 Introduction

In this lab we were assigned to optimise a sequential N-body simulator C code using both the OpenMP API and the Nvidia Cuda API.

An N-body simulator finds the position of particles in sequential steps of the algorithm and then calculates the accumulated force for the particle. Based on that we are able to calculate the current velocity of the particle

2 CPU OpenMP

The iterative nature of the code enables us to parallelise the workflow using the available CPU threads which is achieved by the use of OpenMP. In particular the main sections that allowed this optimisation were the bodyForce() function and the part that calculated the new particle positions as shown below:



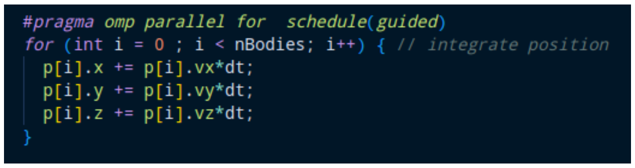
```
void bodyForce(Body *p, float dt, int n) {
  #pragma omp parallel for schedule(dynamic)
  for (int i = 0; i < n; i++) {
    float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;

    for (int j = 0; j < n; j++) {
      float dx = p[j].x - p[i].x;
      float dy = p[j].y - p[i].y;
      float dz = p[j].z - p[i].z;
      float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
      float invDist = 1.0f / sqrtf(distSqr);
      float invDist3 = invDist * invDist * invDist;

      Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
    }

    p[i].vx += dt*Fx; p[i].vy += dt*Fy; p[i].vz += dt*Fz;
  }
}
```

Figure 1: bodyForce Function Parallelism



```
#pragma omp parallel for  schedule(guided)
for (int i = 0 ; i < nBodies; i++) { // integrate position
  p[i].x += p[i].vx*dt;
  p[i].y += p[i].vy*dt;
  p[i].z += p[i].vz*dt;
}
```

Figure 2: Body Position Parallelism

The code required in order to achieve this performance increase was minimal and proposes little room for error. Last but not least the graphs highlighting the performance difference for different number of threads are given below :

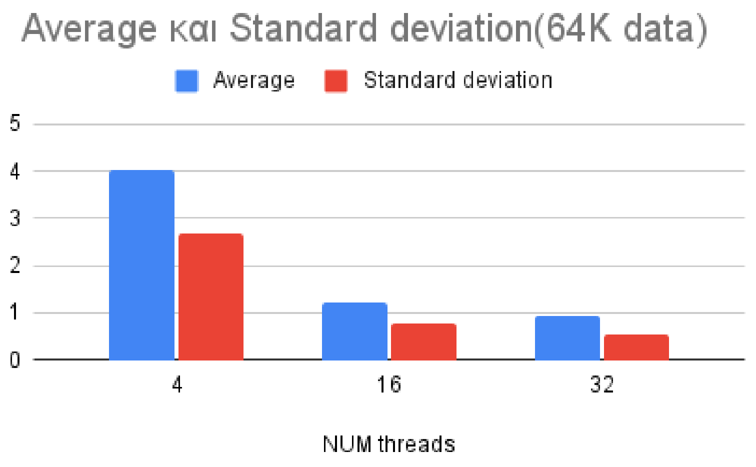


Figure 3: Average & Standard deviation 64K

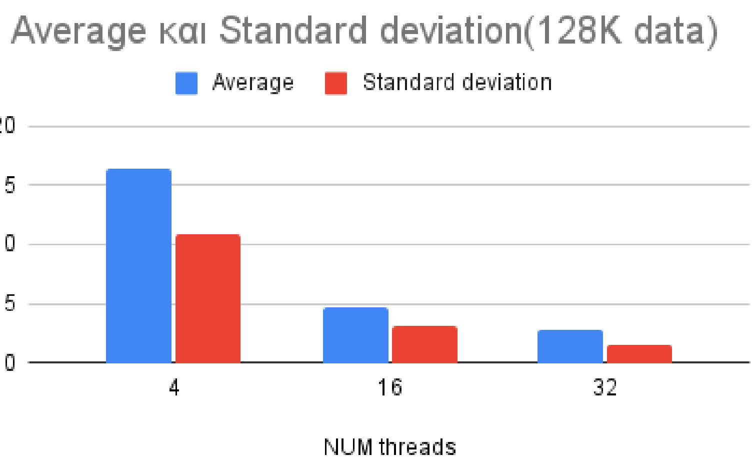


Figure 4: Average & Standard deviation 128K

3 GPU Cuda

In this part of the lab it is required to transfer the calculations from the CPU to the GPU of the system. This enables us to make use of the parallel nature of the GPU in order to increase the calculation's throughput and achieve a way faster computation.

The new code was tested and verified that performs the calculations correctly (minimum of 2 digit tolerance) and is guaranteed to function for any size of input.

3.1 Optimisations

After the calculations transfer from the CPU to the GPU we started to experiment and applying different optimisation techniques.

The first technique which improved execution time was **Data Prefetching**. Because of their design, accessing cache memories is typically much faster than accessing main memory, so prefetching data and then accessing it from caches is usually many orders of magnitude faster than accessing it directly from main memory. We managed to take advantage of this technique by applying it into the two available kernels of code that we constructed. The result was a successful decrease in execution time which proved the technique's effectiveness for our current implementation.

The second technique that was applied is called **Structure of Arrays (SoA)**. The SoA is a layout separating the elements of a struct into one array per element.

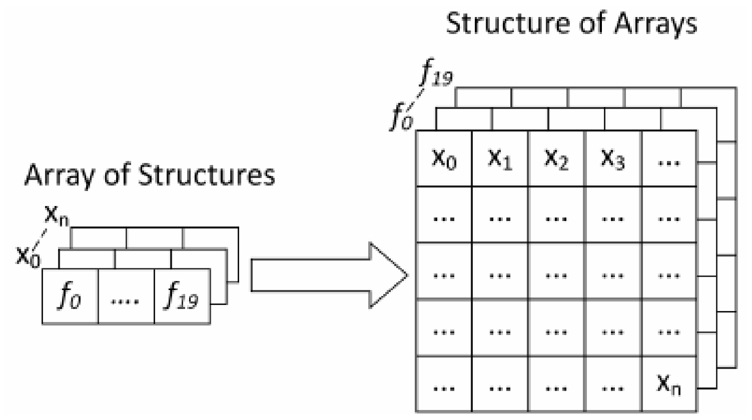


Figure 5: Arrays of Structure vs Structure of Arrays

This data structure is proposed when only a specific part of the struct is needed. Then only those parts need to be iterated over, allowing more data to fit onto a single cache line. This technique was implemented into a separate file in order to demonstrate its minimum decrease in execution time.

3.2 Failed Optimisations

Unfortunately not all of the optimisations turned out to be successful. One of those was the **Loop Unrolling** technique. After modifying the code in order to make sure that the number of bodies is a multiple of 8 we launched a separate kernel that implemented a loop unroll of 8 times. Executing this new version highlighted that this technique was a catastrophe having a drastic increase in execution time. Lowering the size of the unroll seemed to decrease the damage but the fact seemed to be that we were experiencing a negative correlation between unrolling and execution time decrease.

3.3 Measurements

The code itself iterates 10 times through the calculations and takes the average of them in order to calculate a special metric that is called interactions per second. Thenceforth the less the execution time is, the greater the interactions per second are. So to compare different implementation we can use this metric.

The graph below shows the comparison of the GPU optimised vs GPU unoptimised implementation of N-body simulation:

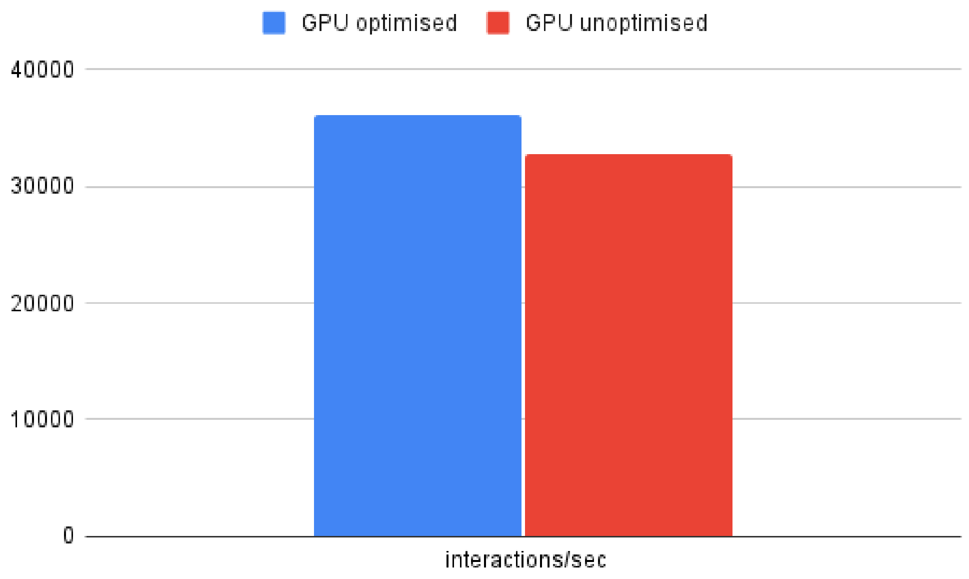


Figure 6: GPU Optimised vs Unoptimised Millions Interactions per Second

For more detailed measurements you can consult the measurements.xls file.

4 Contact

Konstantinos Konsoulas (2975): kkonsoulas@uth.gr
Ioannis Roumpos (2980) : iroumpos@uth.gr