;

Department of Electrical and Computer Engineering

University of Thessaly

ECE415: High Performance Computing (HPC)

Lab 5: Parallel Implementation and Optimization of nbody simulation on GPU using CUDA

Ioanna-Maria Panagou, 2962

Nikolaos Chatzivangelis, 2881

Fall Semester 2022-2023

Contents

1 Summary

In this lab, we transformed a sequential CPU implementation of a NBody simulation algorithm that models the behavior of a system of particles to parallel CPU implementation and then a GPU implementation using CUDA and performed optimizations to increase GPU utilization and memory bandwidth.

2 Introduction

An N-body simulation is a type of simulation that models the behavior of a system of particles that interact with one another through gravitational forces. The "N" in N-body refers to the number of particles in the system, which can be any positive integer. The goal of an N-body simulation is to predict the motion of all the particles over time, based on the initial positions and velocities of the particles, and the forces acting on them. This can be used to study a wide range of phenomena, including the formation and evolution of galaxies, the dynamics of planetary systems, and the behavior of star clusters.

The most basic form of an N-body simulation is known as a direct N-body simulation. In this type of simulation, the force acting on each particle is calculated by summing the forces due to all the other particles in the system. This can be computationally expensive, because the number of calculations required grows rapidly with the number of particles in the system. So our task was to optimize the provided code in order to speed up the simulation.

N-body simulators have been key tools in many scientific fields, particularly in the study of celestial mechanics, the large-scale structure of the universe, and the formation and evolution of stars and galaxies. It is also used to simulating systems like Crowd simulation, particle based simulations and molecular dynamics which can help to predict the behavior of real-world systems in these fields.

3 Evaluation

For each implementation we ran experiments multiple times and for different number of simulated particles *(25000, 65536, 131072, 200000)* in order to decide whether we had speedup or not. Every successful optimization is incorporated to the previous version of the code, i.e. each version contains all the previous successful optimizations unless noted otherwise.

3.1 Device Query

The output of the device query is contained in the deliverables. The most useful features of the Tesla K80 device of the CSL-artemis that assisted us during development are:

```
Total amount of global memory:                 11441MBytes (11997020160 bytes)
Total amount of constant memory:               65536 bytes
Total amount of shared memory per block:       49152 bytes
Maximum number of threads per multiprocessor:  2048
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z):  (1024, 1024, 64)
Max dimension size of a grid size (x,y,z):     (2147483647, 65535, 65535)
```

4 Optimization Steps

4.1 CPU Baseline

The provided sequential CPU implementation did not execute in a reasonable time frame which was rational because this is the simplest, naive NBody simulation implementation. Figure 1 presents the CPU execution time for a variety of particles number. The code complexity is $O(N^2)$ because for each particle the distance with the all the other particles is calculated in order to measure the resultant force applied to this particle. Based on the computed force, the velocity and the new position of each particle is determined in each time step.

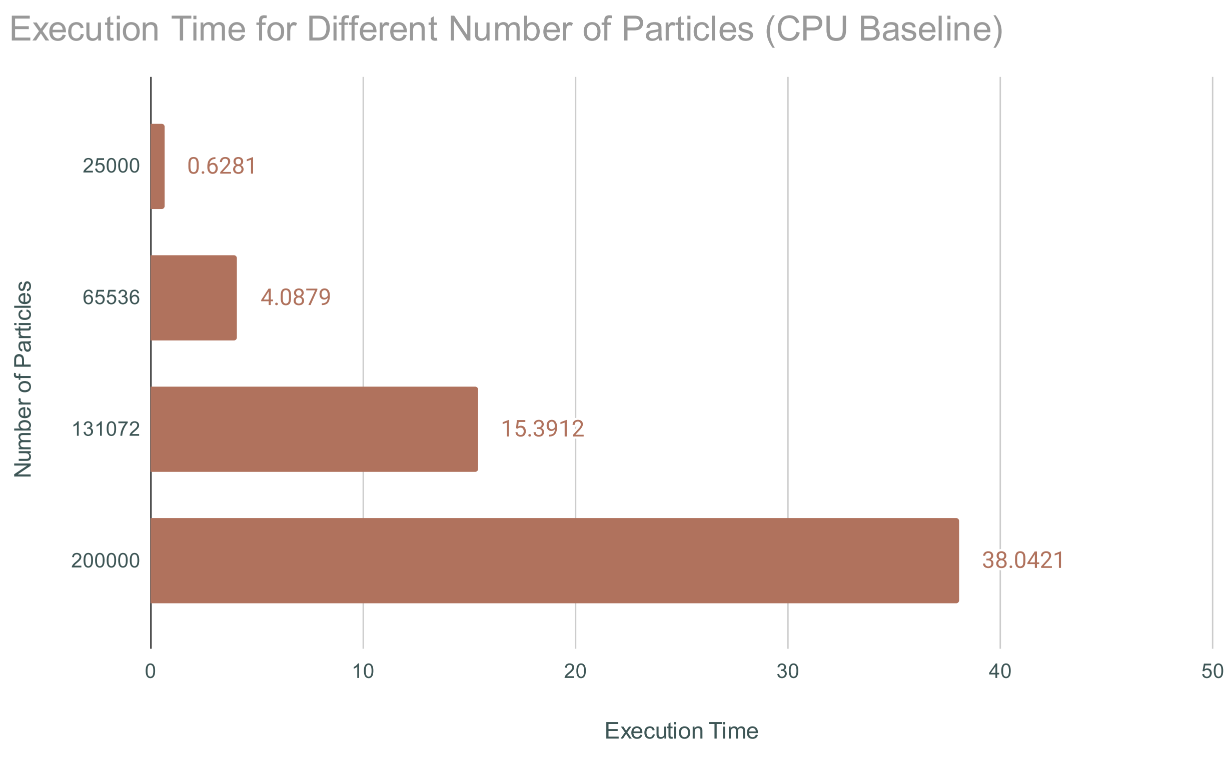

Figure 1: CPU Baseline implementation execution time (sec) for different number of particles

4.2 OpenMP

The first part of this assignment was to parallelize the baseline code using the OpenMP directives. So we implemented three parallel versions, presented below.

4.2.1 OpenMP 1st Version (Omp1)

Function `bodyForce()` calculates the force applied to each particle and then its new position. In this function, we parallelized the internal for loop which calculates the three total forces (Fx, Fy, Fz) applied by all the particles in the simulation. The modified code is presented in the following listing 1. This implementation had a speedup around **9.5X** compared to the baseline version.

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

        #pragma omp parallel for reduction(+: Fx, Fy, Fz)
        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 = Fx + dx * invDist3;
            Fy = Fy + dy * invDist3;
            Fz = Fz + dz * invDist3;
        }
        p[i].vx += dt*Fx; p[i].vy += dt*Fy; p[i].vz += dt*Fz;
    }
}
```

Listing 1: 1st parallel version of `bodyForce()`

4.2.2 OpenMP 2nd Version (Omp2)

In this version, to avoid the overhead of creating and destroying threads in each loop iteration, we moved the parallel region outside the outer loop, which also resulted in a simpler implementation, as shown in listing 2.

```
void bodyForce(Body *p, float dt, int n) {
    #pragma omp parallel for
    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;
    }
}
```

Listing 2: 2nd parallel version of `bodyForce()`

4.2.3 OpenMP 3rd Version (Omp3)

In this version, we moved the position updates inside the parallel region and parallelized both loops (listing 3). The two loops could not be merged, since the positions should not be updated before the velocities have been calculated for the current step.

```
void bodyForce(Body *p, float dt, int n) {

  #pragma omp parallel
  {
  #pragma omp for
  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;
  }

  #pragma omp for
  for (int i = 0 ; i < n; i++) { // integrate position
```

```
        p[i].x += p[i].vx*dt;
        p[i].y += p[i].vy*dt;
        p[i].z += p[i].vz*dt;
      }
    }
}
```

Listing 3: 3rd parallel version of `bodyForce()`

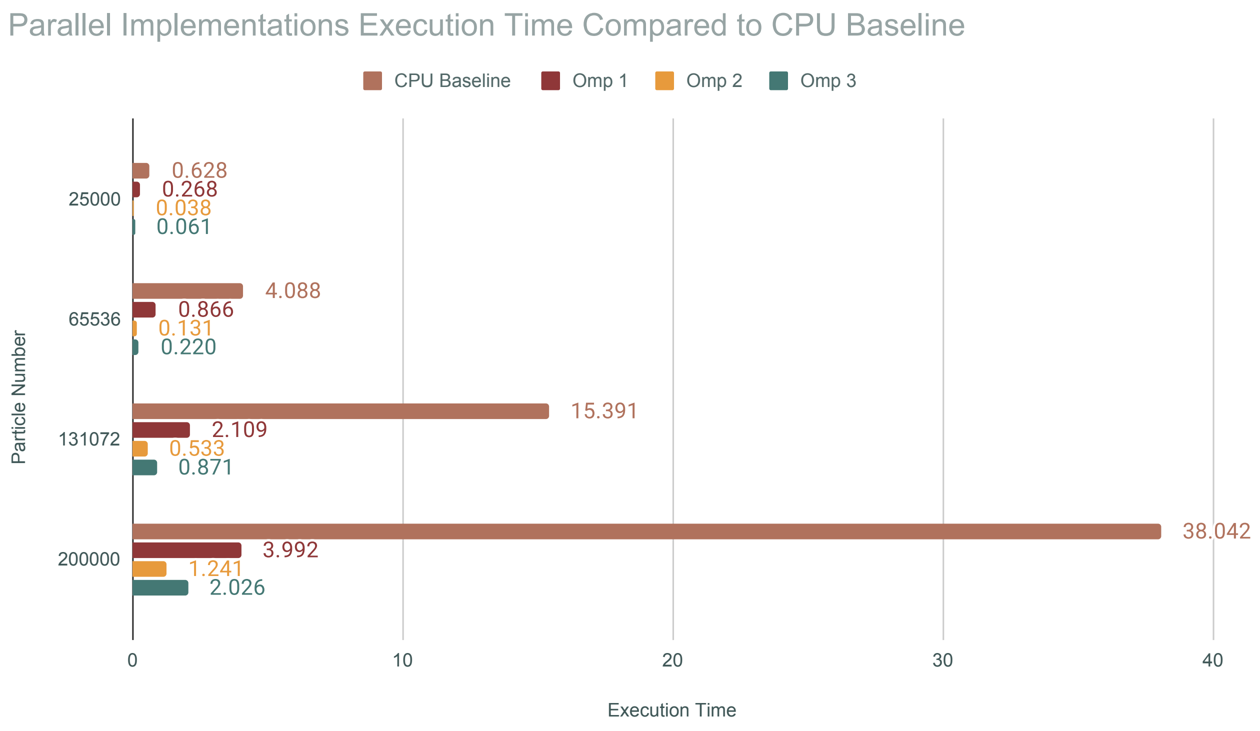

Figure 2: Parallel Implementations Execution Time (sec) Compared to CPU Baseline

4.2.4 Evaluation

We tested the three different parallel versions for different number of threads and 64 threads proved the optimal number of threads across different implementations (figure 3). We also observed that the Omp_2 version beat the other versions for all particle numbers we tested (figure 2).

4.3 GPU Baseline

For the baseline GPU implementation (listing 4) , we chose a 1D topology and assigned each particle to a thread. Using the CUDA occupancy calculator, we found out that using 512 threads in each block results in 100% streaming multiprocessor utilization. For each iteration:

1. We transfer the buffer that contains the body data to the GPU allocated memory.

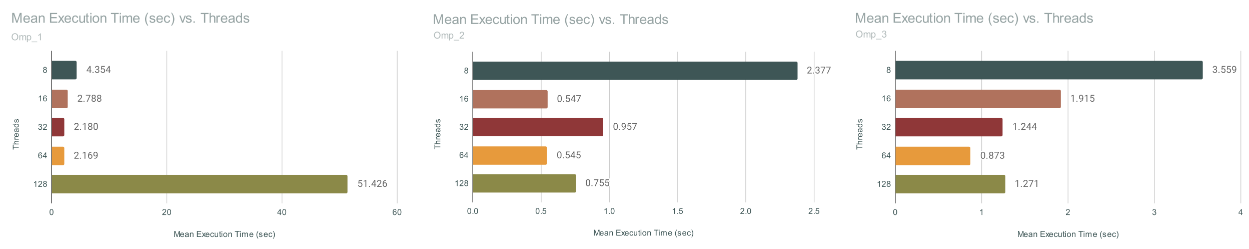

Figure 3: Mean Execution Time (sec) for different number of threads for number of particles equal to 131072

2. We execute the `bodyForce()` kernel.

3. We copy the results from the device back to the host.

4. We perform the position update on the host data.

The above steps are repeated for all implementations except noted otherwise.

```
__global__ void bodyForce(Body *p, float dt, int n) {

  # we assign each particle to a thread
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  float Fx = 0.0f, Fy = 0.0f, 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 = 1.0 / 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;

}
```

Listing 4: GPU Baseline

From figure 4, we can see that the performance the GPU baseline implementation achieves is slightly slower than the most optimized OMP implementation.

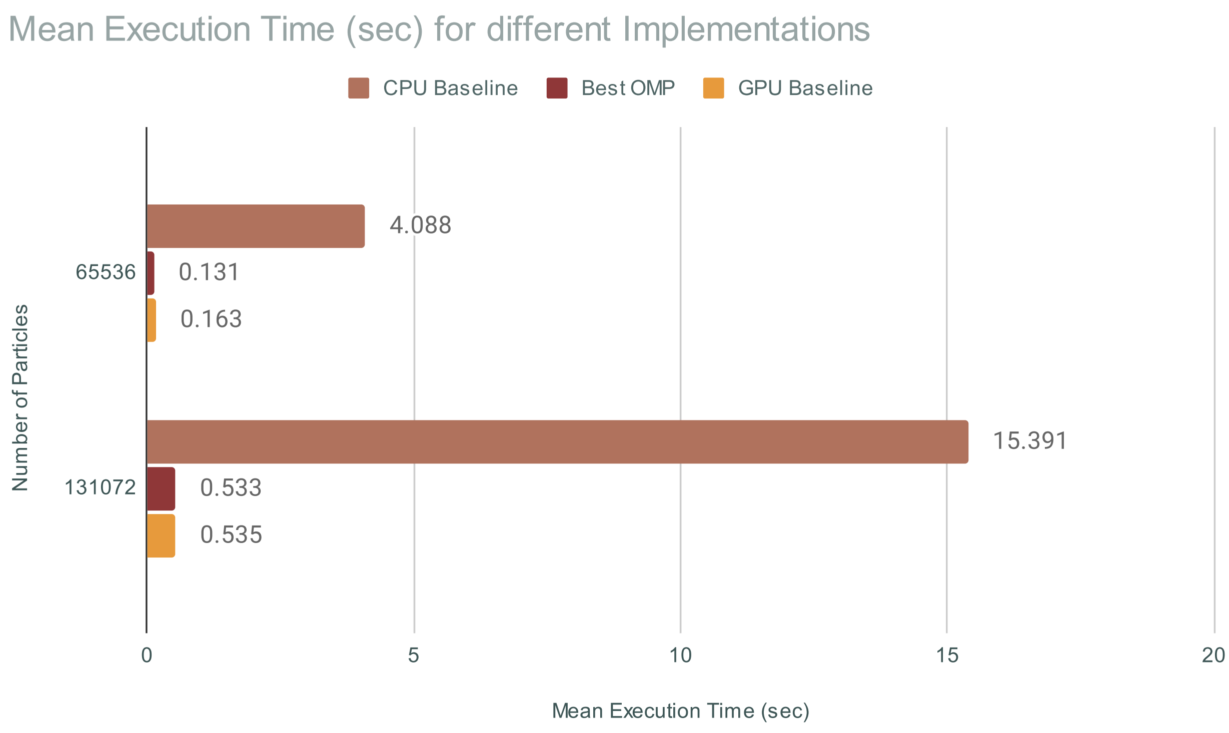

Figure 4: Mean execution time (sec) for CPU Baseline, Best OMP and GPU Baseline implementations

4.4 Tiling

The code of 4 suffered from a lot of global memory accesses. To mitigate this problem, we decided to store the necessary body data in the shared memory of each block. However, since shared memory is limited, we have to further split the bodies into `ceil(nBodies/THREADS_PER_BLOCK)` tiles. Before the start of the loop that iterates over the tiles, each body reads its data struct from the global memory (line 8 of 5). Then, a number of bodies equal to the number of threads in the block is brought into the shared memory. In each iteration of the loops in lines 12-29, each thread computes its interactions with bodies 0-THREADS_PER_BLOCK-1, THREADS_PER_BLOCK-2*THREADS_PER_BLOCK-1, ... The variables F_x, F_y, F_z accumulate the normalized displacement across all tiles. We perform the calculations for the last tile separately, as it may have less bodies than the number of threads per block, if the number of particles is not exactly divisible by the number of threads. The last step is to update the velocities (line 48) and write the data struct register *myBody* back to the global memory. Note that it is imperative to call the `__syncthreads()` function in lines 13 and 31, so that all threads have finished the previous iteration, before changing the data in the shared memory, because some threads could still be reading from the shared memory.

```
1 __global__ void bodyForce(Body *p, float dt, int nBodies, int tiles) {
2   extern __shared__ Body privBodies[];
3
```

```
 Body myBody;
 int id = threadIdx.x + blockIdx.x * blockDim.x;
 int i, tile;
 float Fx = 0.0f, Fy=0.0f, Fz=0.0f;

 // read your data from global memory
 myBody = p[id];

 float dx, dy, dz, distSqr, invDist, invDist3;

// for the first tiles-1 tiles
 for (tile=0; tile<tiles-1; tile++) {
   __syncthreads();
   int idx = threadIdx.x + tile * blockDim.x;
   privBodies[threadIdx.x] = p[idx];
  __syncthreads();

   for (int j=0; j<blockDim.x; j++) {
       dx = privBodies[j].x - myBody.x;
       dy = privBodies[j].y - myBody.y;
       dz = privBodies[j].z - myBody.z;

       distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
       invDist = 1.0f / sqrtf(distSqr);
       invDist3 = invDist * invDist * invDist;

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

 __syncthreads();
 int idx = threadIdx.x + (tiles-1) * blockDim.x;
 privBodies[threadIdx.x] = p[idx];
 __syncthreads();

// for the last tile
 for (int k=(tiles-1)*blockDim.x, j=0; k<nBodies; k++, j++) {
   dx = privBodies[j].x - myBody.x;
   dy = privBodies[j].y - myBody.y;
   dz = privBodies[j].z - myBody.z;

   distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
   invDist = 1.0f / sqrtf(distSqr);
   invDist3 = invDist * invDist * invDist;

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

```

// update register
  myBody.vx += dt*Fx; myBody.vy += dt*Fy; myBody.vz += dt*Fz;

//copy back to global memory
  p[id] = myBody;

}
```

Listing 5: Tiling

Figure 5 presents the execution time for the GPU Baseline and Tiling Implementation. Using tiling, we achieved a mean speedup of x1.2 (mean speedup is the average of the speedups achieved across the 4 different number of particles we tested our code for) from the GPU Baseline.

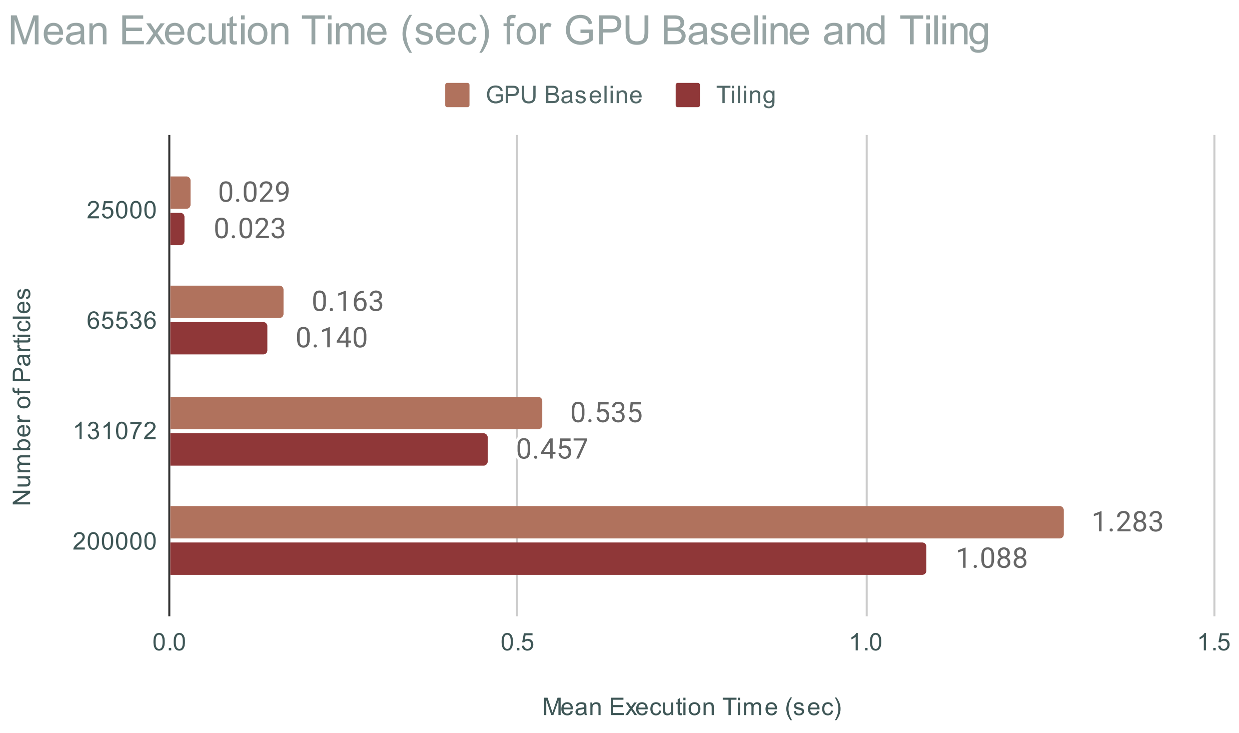

Figure 5: Mean execution time (sec) for GPU Baseline and Tiling

We noticed, however, that, we do not need to copy and store both the positions and the velocities in the shared memory, since the velocity of each body is written only once at the end of each iteration and is not read by the other bodies. So, we split the body data to two different arrays consisting of $float3$ type elements, one for the positions and one for the velocities. Only the positions need to be stored in shared memory. In addition, both the positions and the velocities are copied to the device only on the first iteration. For the subsequent iterations, we only copy the host positions to the device, since those are the only ones altered in the host. The next optimizations (until the Structure of Arrays optimization) will follow the same pattern. The updated `bodyForce()` kernel is presented in listing 6.

```
__global__ void bodyForce(float3 *pos, float3 *vel, float dt, int nBodies,
   int tiles) {
  extern __shared__ float3 privBodies[];

  float dx, dy, dz, distSqr, invDist, invDist3;
  float Fx = 0.0f, Fy=0.0f, Fz=0.0f;

  float3 myPos;
  float3 myVel;
  int i, tile;
  int id = threadIdx.x + blockIdx.x * blockDim.x;

  // read position from global memory
  myPos = pos[id];
  // read velocity from global memory
  myVel = vel[id];

  for (tile=0; tile<tiles-1; tile++) {
    __syncthreads();
    int idx = threadIdx.x + tile * blockDim.x;
    // copy only the positions
    privBodies[threadIdx.x] = pos[idx];
    __syncthreads();

    for (int j=0; (j<blockDim.x); j++) {
      dx = privBodies[j].x - myPos.x;
      dy = privBodies[j].y - myPos.y;
      dz = privBodies[j].z - myPos.z;

      distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
      invDist = 1.0f / sqrtf(distSqr);
      invDist3 = invDist * invDist * invDist;

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

  __syncthreads();
  int idx = threadIdx.x + (tiles-1) * blockDim.x;
  // copy only the positions
  privBodies[threadIdx.x] = pos[idx];
  __syncthreads();

  for (int k=(tiles-1)*blockDim.x, j=0; k<nBodies; k++, j++) {
    dx = privBodies[j].x - myPos.x;
    dy = privBodies[j].y - myPos.y;
    dz = privBodies[j].z - myPos.z;

    distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
    invDist = 1.0f / sqrtf(distSqr);
    invDist3 = invDist * invDist * invDist;

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

  myVel.x += dt*Fx; myVel.y+= dt*Fy; myVel.z += dt*Fz;
  vel[id] = myVel;
}
```

Listing 6: Tiling v2

We tried performing the position update on the host side, but it introduced error that was not within acceptable limits. We achieved an average speedup of x1.04 (figure 6) from the original tiling implementation for the 3 greatest number of particles: 65536, 131072 and 200000.

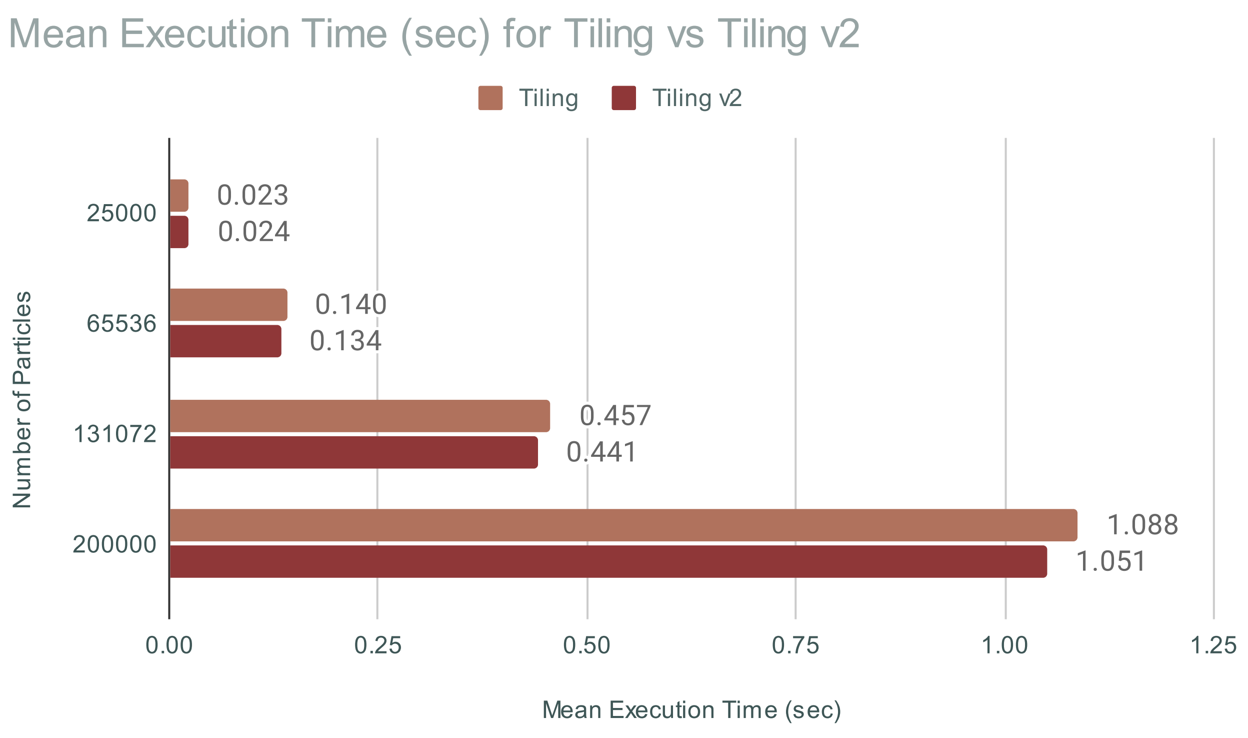

Figure 6: Mean execution time (sec) for Tiling and Tiling v2

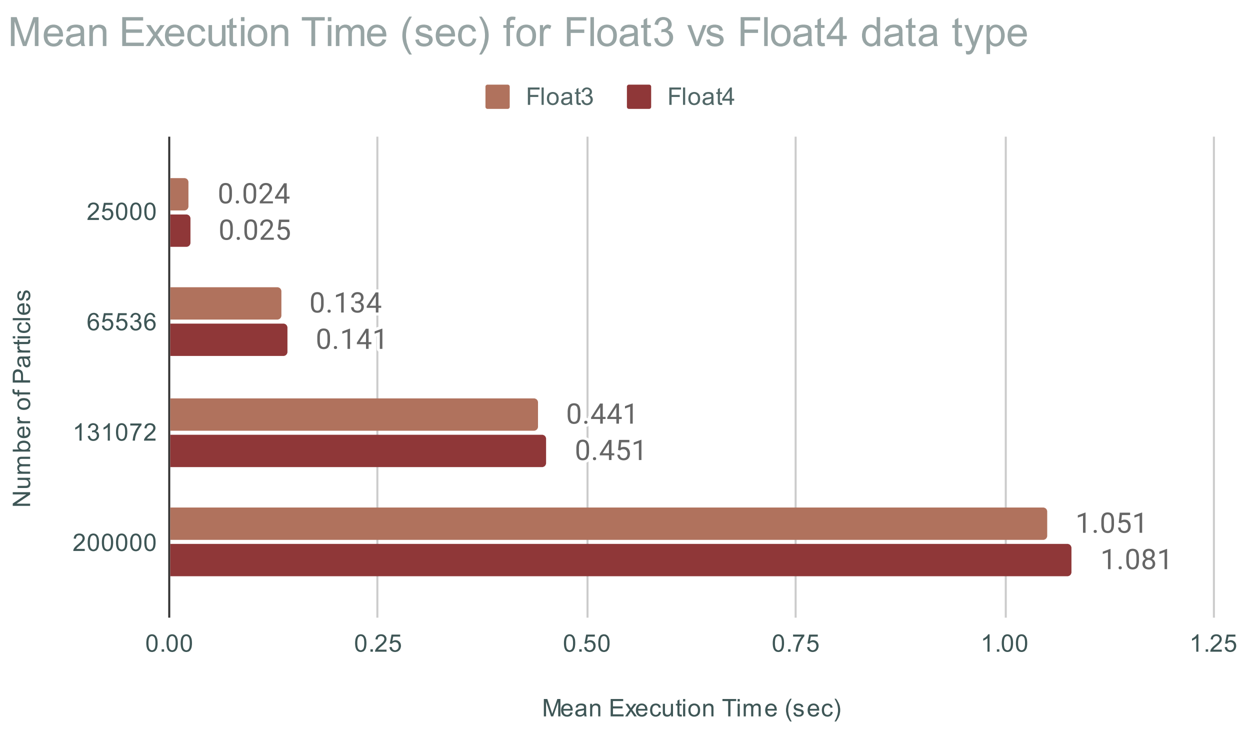

Figure 7: Mean execution time (sec) for float3 vs float4 data type

4.5 Float4 vs Float3 data type

To allow for coalesced global memory accesses, we replaced the float3 data type with the float4 data type. In this way, the compiler aligns each element to a 16 byte boundary, thus enabling coalesced memory accesses. However, this optimization did not have the expected outcome, with the execution times being slightly higher as shown in 7. Using float4 data type demands higher memory bandwidth, which probably outweighs the benefit of coalesced memory accesses. The NVIDIA documentation also states that the float3 built-in vector type is aligned to a 4-byte boundary, which means that the requests are probably already coalesced.

4.6 Loop Unrolling

The inner loop iterates over all bodies that are currently in the shared memory. We can unroll the loop to perform less branch condition checking and less loop variable increments. We experimented with different unrolling factors: 4, 8, 16 and 32. An unrolling factor of 8 yields the best performance as shown in 8. Therefore, all subsequent optimizations use the `#pragma unroll 8` directive to instruct the compiler to unroll the inner loop (the one with the j loop variable).

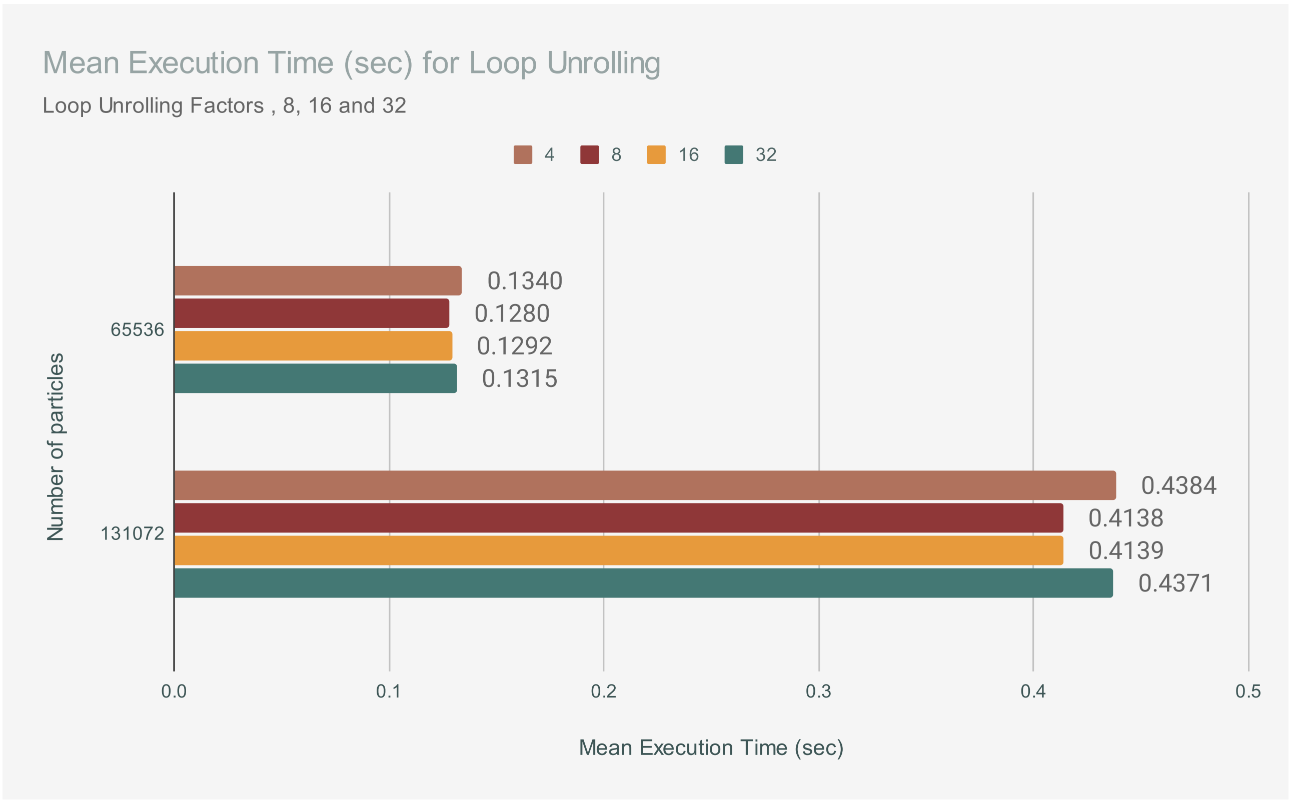

Figure 8: Mean Execution Time (sec) for unrolling factors 4, 8, 16 and 32 for number of particles equal to 64K and 128K

4.7 CUDA Intrinsics

Using CUDA Intrinsics, we can write code that can be directly mapped to the GPU instruction set and use approximate floating-point units that the compiler would be hesitant to use on its own. We replaced the 1/sqrtf function, with the intrinsic `__frsqrt_rn` that uses a fast approximate algorithm to find the inverse of the square root. The use of this approximate intrinsic did not introduce significant error and aided performance.

We also noticed that we can reduce the number of operations performed by merging the inverse square root and the 2 multiplications, since:

$$\left(\frac{1}{\sqrt{x}}\right)^3 = \frac{1}{(\sqrt{x})^3} = \frac{1}{x^{\frac{3}{2}}} = x^{-\frac{3}{2}}$$

We replaced the 3 operations with a single `__powf` intrinsic that raises the *distSqr* variable to the power of $\frac{-3.0}{2.0}$ and we observed a major leap in performance of about x1.6 from the previous successful optimization as shown in 9.

4.8 Structure of Arrays

We know that a structure of arrays layout can be more effective for GPUs than an array of structures layout, which is our current implementation. The reason for this is that consecutive

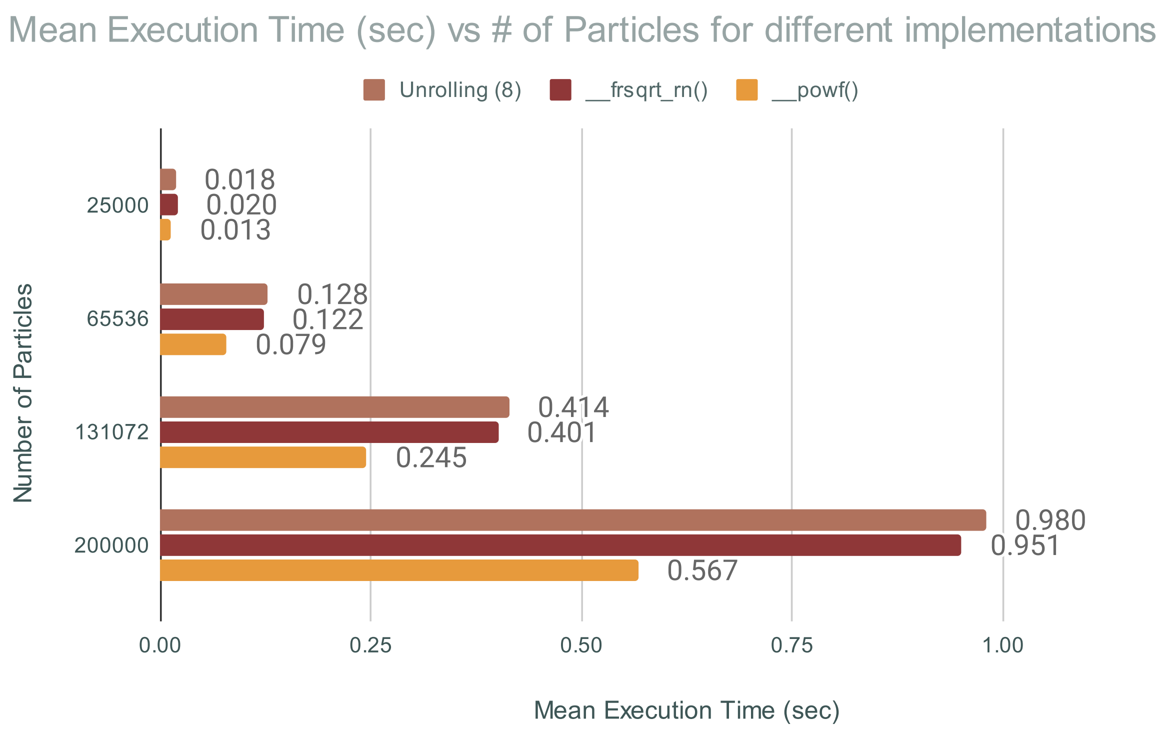

Figure 9: Mean Execution Time (sec) vs Number of Particles for different Implementations

accesses to fields of the body struct, such as x, y, or z, for consecutive bodies leads to scattered and non-consecutive memory accesses. However, if the fields are stored in using a structure of arrays layout, it can lead to more efficient and coalesced memory accesses. To address this issue, we have transformed our array of bodies into a single body element that contains arrays of floating-point values for `x, y, z, vx, vy` and `vz`. The new listing of the `bodyForce()` kernel is provided in 7.

```
__global__ void bodyForce(Body p, float dt, int nBodies, int tiles) {
  __shared__ float priv_x[THREADS_PER_BLOCK];
  __shared__ float priv_y[THREADS_PER_BLOCK];
  __shared__ float priv_z[THREADS_PER_BLOCK];

  float dx, dy, dz, distSqr, invDist, invDist3;
  float Fx = 0.0f, Fy=0.0f, Fz=0.0f;

  int i, tile;
  int id = threadIdx.x + blockIdx.x * blockDim.x;

  float x, y, z, vx, vy, vz;

  // fetch your data from the global memory
  x = p.x[id];
  y = p.y[id];
```

```
z = p.z[id];
vx = p.vx[id];
vy = p.vy[id];
vz = p.vz[id];

for (tile=0; tile < tiles-1; tile++) {
  __syncthreads();
  int idx = threadIdx.x + tile * blockDim.x;
  priv_x[threadIdx.x] = p.x[idx];
  priv_y[threadIdx.x] = p.y[idx];
  priv_z[threadIdx.x] = p.z[idx];
  __syncthreads();

  #pragma unroll 8
  for (int j=0; j<blockDim.x; j++) {
    dx = priv_x[j] - x;
    dy = priv_y[j] - y;
    dz = priv_z[j] - z;

    distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
    invDist3 = __powf(distSqr, -3.0/2.0);

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

__syncthreads();
int idx = threadIdx.x + (tiles-1) * blockDim.x;
priv_x[threadIdx.x] = p.x[idx];
priv_y[threadIdx.x] = p.y[idx];
priv_z[threadIdx.x] = p.z[idx];
__syncthreads();

#pragma unroll 8
for (int k=(tiles-1)*blockDim.x, j=0; k<nBodies; k++, j++) {

  dx = priv_x[j] - x;
  dy = priv_y[j] - y;
  dz = priv_z[j] - z;

  distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
  invDist3 = __powf(distSqr, -3.0/2.0);

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

}

```

```
  vx += dt*Fx; vy += dt*Fy; vz += dt*Fz;

  // write back the data
  p.vx[id] = vx;
  p.vy[id] = vy;
  p.vz[id] = vz;

}
```

Listing 7: Structure of Arrays Layout

The operations performed in main also needed to be modified:

1. At the beginning of the first iteration, the 6 arrays (x, y, z, vx, vy, vz) are copied to the device. For all other iterations, only the position arrays (x, y and z) are copied.
2. The bodyForce() kernel is executed
3. The velocities (vx, vy, vz) are copied back to the host side
4. The new positions (x, y, z) are calculated.

The Structure of Arrays layout was a valuable optimization that had a noticeable improvement in the execution time (figure 10).

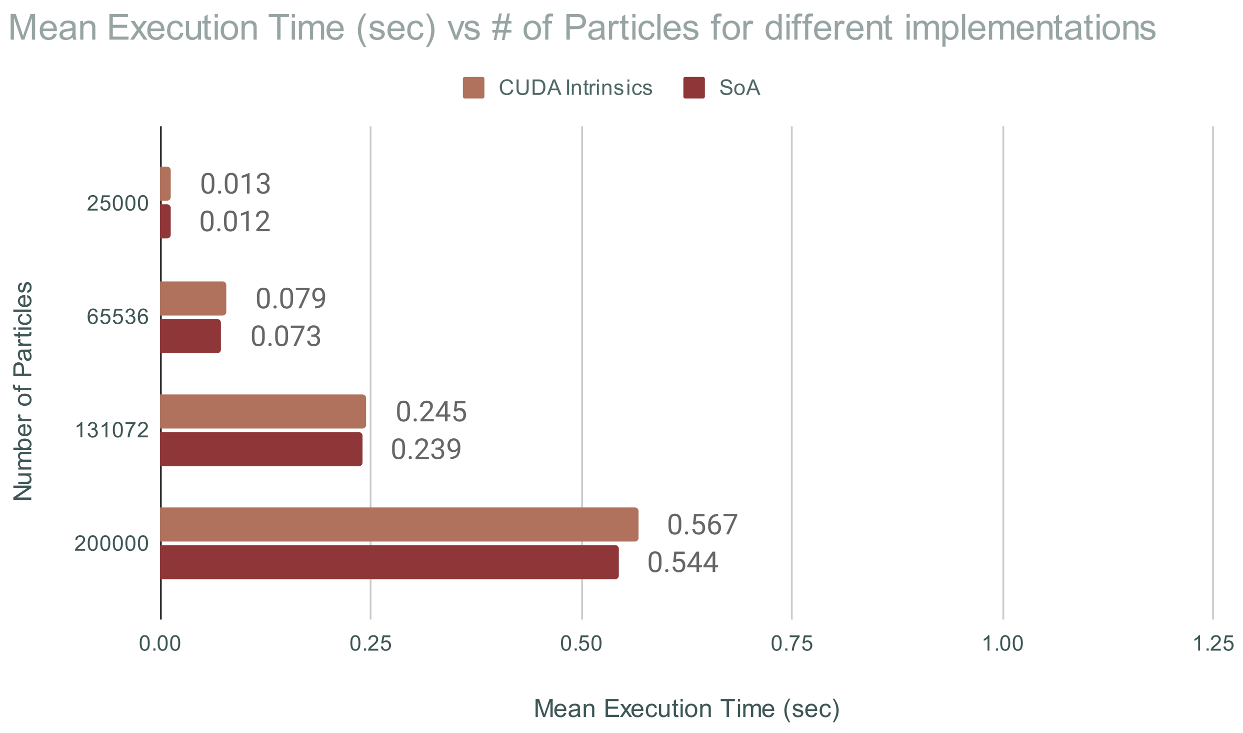

Figure 10: Mean Execution Time (sec) vs Number of Particles for different Implementations

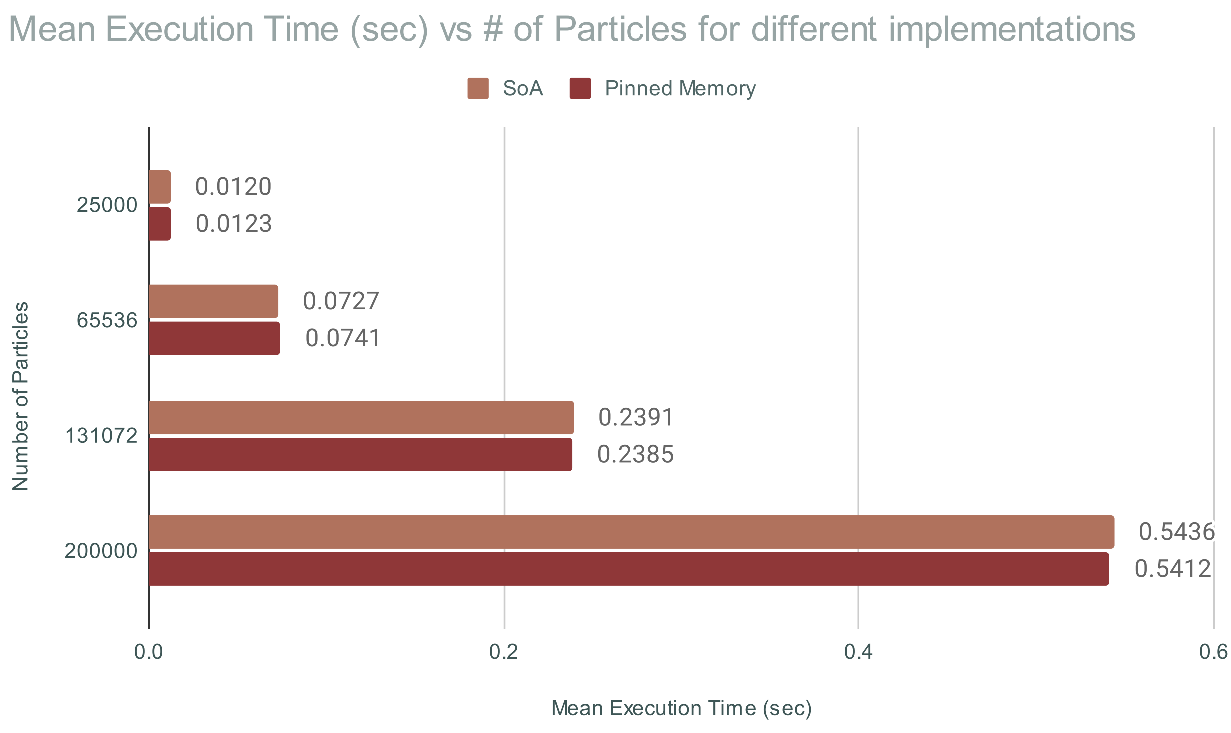

Figure 11: Mean Execution Time (sec) for different implementations

4.9 Pinned Memory

To speedup the memory transfers to and from the device, we allocated the host arrays using the cudaHostAlloc() function instead of a plain malloc(), but did not notice a significant improvement in performance, but rather a decline in performance for some number of particles (figure 11). So, our final code does not include this optimization.

4.10 -ftz=true flag

The inclusion of a softening factor of 1e-9 ensures that the code will not encounter denormal numbers, which are extremely small numbers that require special handling. By avoiding denormal numbers, it is possible to use the -ftz=true flag with the nvcc compiler, which will enable flush-to-zero mode and drop support for the denormal inputs. This eliminates the overhead of processing denormal numbers, avoids unnecessary calculations and improves performance by x1.3 on average as shown in figure 12, without sacrificing accuracy.

5 Conclusions

In this assignment, we had to optimize a sequential CPU implementation of a NBody simulation algorithm using OpenMP and then CUDA. We managed to gain a moderate speedup of X80.64.

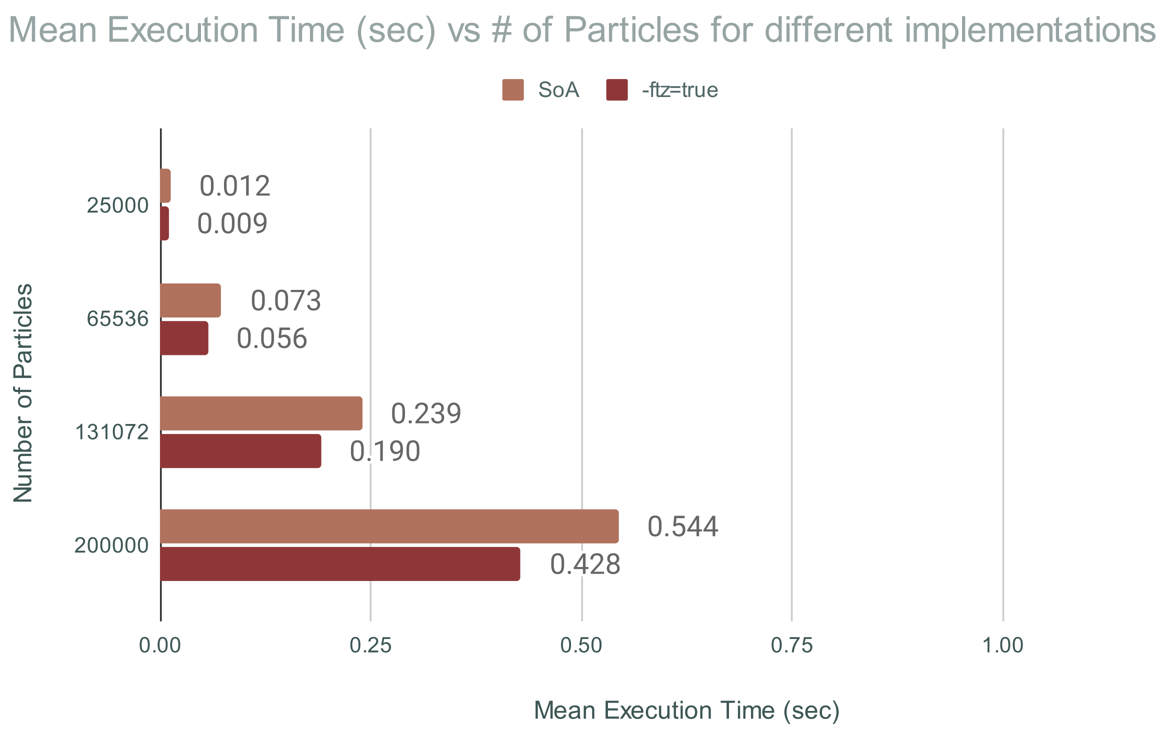

Figure 12: Mean Execution Time (sec) for different implementations

From our attempts to parallelize the provided CPU implementation with OpenMP we managed to gain an overall speedup of X29. We also experimented with the number of threads to use and we found out that the best option was to set as a max thread number the number of the available hardware threads in the system (64).

In terms of the GPU implementation, CUDA Occupancy Calculator helped us to determine the optimal geometry based on the gear we had in our disposal. The given NBody simulation algorithm had plenty of room for improvements and allowed us to exploit various CUDA mechanisms in order to accomplish a better speedup. This argument is enhanced by the final achieved speedup, which was tripled, compared to our baseline implementation of the GPU code (figure 14).

Figure 15 presents the speedup of each optimization step from the previous one. As we can observe, some attempts nearly doubled the gained speedup. For example, CUDA Intrinsic `__powf()` nearly reduced the execution time in half (figure 13) which is reasonable because these functions allow for more fine-grained control over the GPU's execution and can lead to improved performance for certain types of calculations, similar to these of this assignment. Finally, our findings demonstrate that it is always important to explore new compiler flags for our programs. As we can comprehend from figure 15 `-ftz=true` improved our speedup 24% compared to the previous optimization step.

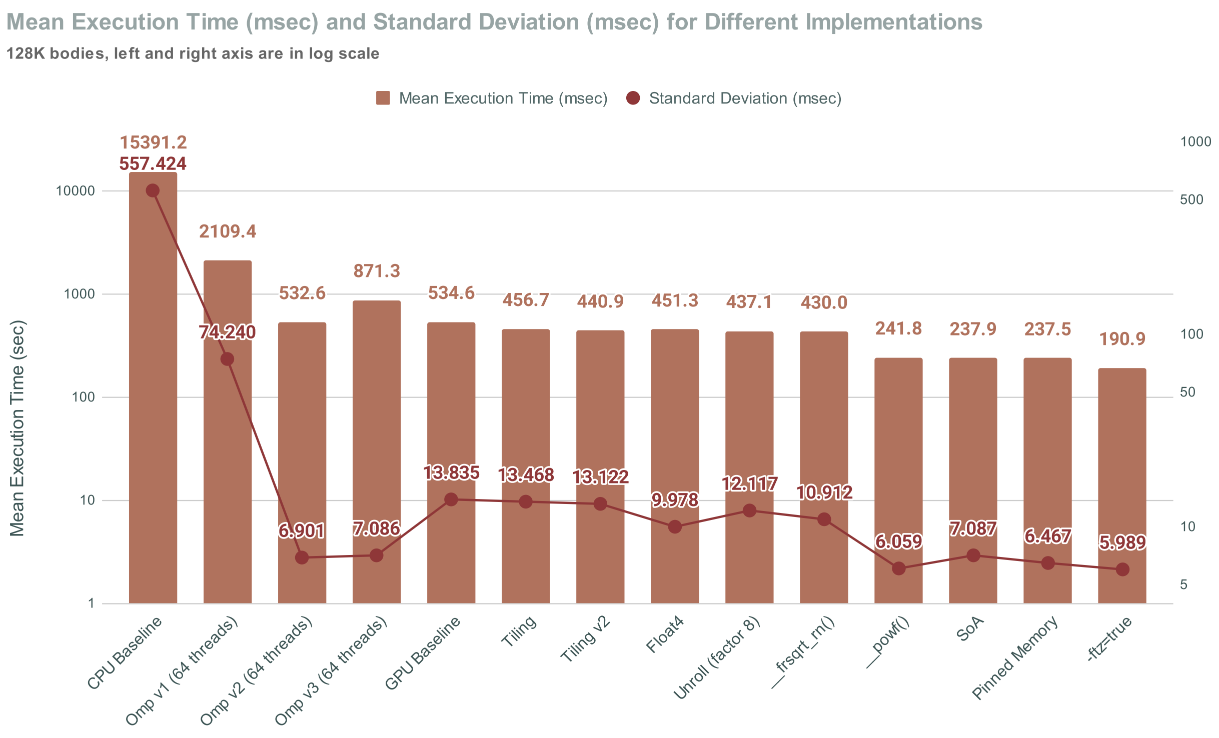

Figure 13: Mean Execution Time (msec) and Standard Deviation (msec) for Different Implementations

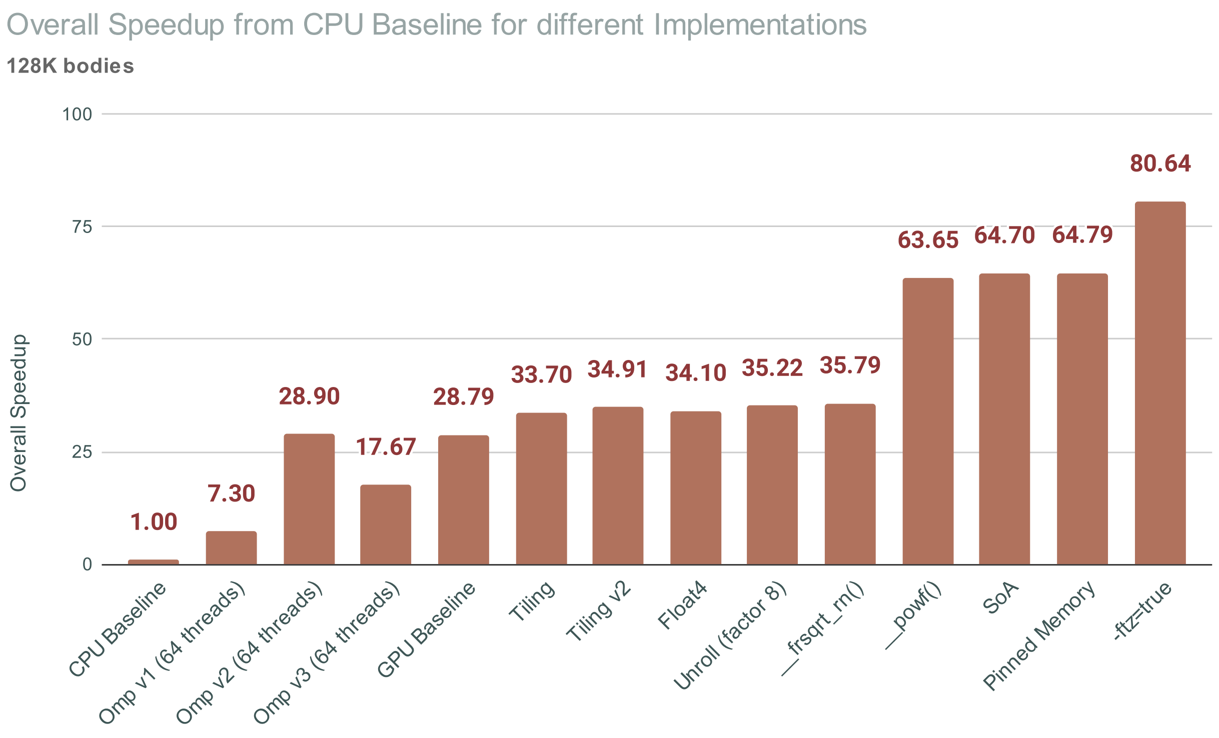

Figure 14: Overall Speedup from CPU Baseline for different Implementations

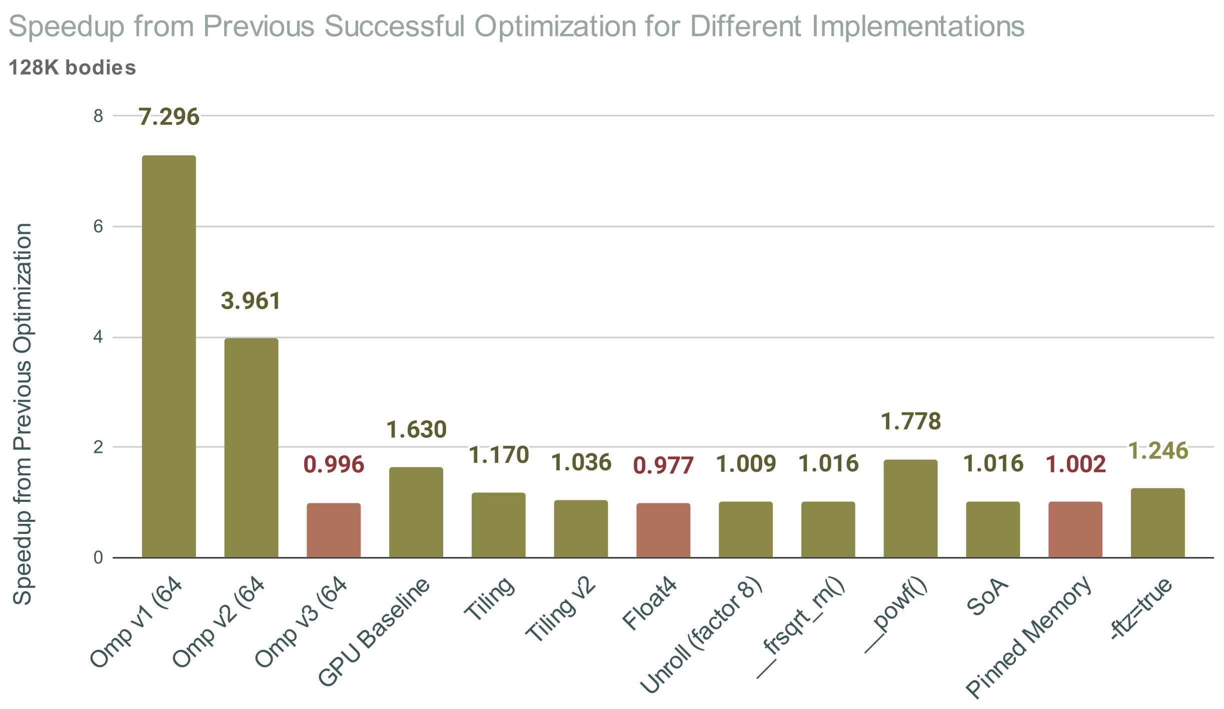

Figure 15: Speedup from Previous Successful Optimization for Different Implementations. The green columns were deemed successful optimizations.