ECE415 - High Performance Computing

– Homework 5 –

N-body Simulation Using CUDA

submitted by

Aggeliki-Ostralis Vliora, Spyridon Liaskonis

January 19, 2025

Aggeliki-Ostralis Vliora, Spyridon Liaskonis
`avliora@uth.gr, sliaskonis@uth.gr`
Student ID: 03140,03381

Contents

1 Introduction

In this project we explore the parallelization and optimization of an N-body simulation, harnessing the power of both CPUs and GPUs to achieve significant performance gains. N-body simulations are computationally demanding, requiring the calculation of gravitational forces between every pair of particles in a system. This inherent complexity motivates the exploration of parallel computing techniques to accelerate these simulations, especially for large-scale problems.

Our investigation will be conducted in two primary stages:

1. CPU Parallelization with OpenMP: We begin by parallelizing the provided sequential N-body code using OpenMP. This will allow us to distribute the workload across multiple CPU cores and exploit the inherent parallelism in the force and position calculations.

2. GPU Acceleration with CUDA: Next, we transition the simulation to a GPU using CUDA. This stage will involve not only porting the code but also optimizing it to effectively utilize the massive parallelism offered by GPUs. Our goal is to achieve optimal performance for varying dataset sizes, scaling the simulation to leverage the available GPU memory.

By combining the strengths of OpenMP and CUDA, we aim to demonstrate the substantial performance improvements achievable through parallel computing techniques. This will enable more efficient and scalable N-body simulations.

2 Evaluation

2.1 Hardware

The code was evaluated on a multicore system powered by Intel Xeon E5-2695 processor. This machine features two processors, each with 14 cores, resulting in a 28-core configuration. With support for simultaneous multithreading (SMT), or hyperthreading, each core can manage two threads concurrently, allowing us to parallelize the algorithm across 56 threads. The system's CPU topology is presented in Figure 1.

As for the CUDA algorithms, the evaluation was done using a Tesla K80 GPU. Information about our system's GPUs can be found in Appendix A.

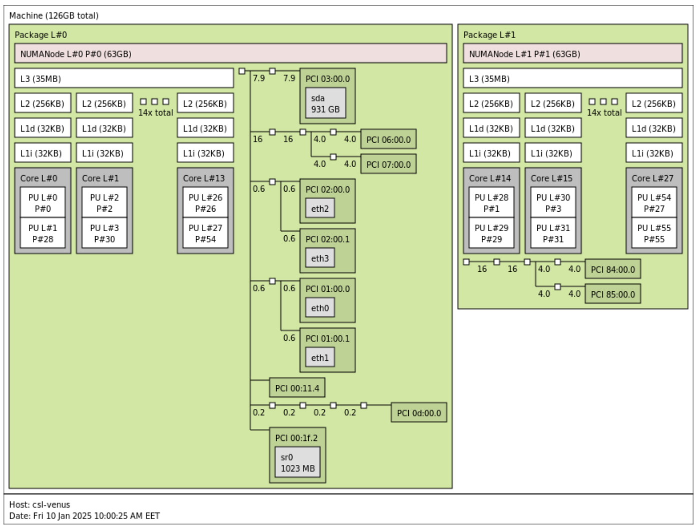

Figure 1: CPU Topology of system used for evaluation

2.2 Software

For all the CPU implementations (serial, parallelized), we used Intel's ICX compiler (version 2025.0.0, Build 20241008). Each code implementation was compiled with the -fast flag, enabling aggressive optimizations to maximize performance.

As for the GPU implementation, the compiler's version is NVCC 11.5.50.

2.3 Algorithm Execution

To evaluate each algorithm implementation we perform the following steps:

1. Compile the program with the appropriate compiler/flags (`icx` for serial/OpenMP, `nvcc` for CUDA).

2. Repeatedly execute the program for N number of iterations (in our evaluation we set N=10).

3. For each iteration, the program performs P iterations of the n-body simulation algorithm. In our case, P is also set to 10. For each simulation iteration, the iteration execution time is saved. At the end of the simulation, the total execution time and the number of iterations per second are calculated and saved.

4. Using all the N total execution times, we calculate the mean execution time of our algorithm. We use this time as a metric for performance comparison between all implementations.
5. We perform these steps for three different numbers of particles: 30k, 64k, and 128k.

To ensure the accuracy of the parallel implementations, we compare the calculated coordinates after the second iteration with those obtained from the serial implementation. Optimizations that introduce errors beyond the second decimal place are rejected.

3 Serial Algorithm

3.1 Implementation

The base implementation is a serial implementation consisting of two main areas of computation.

First, the forces acting on body i due to body j are computed for all bodies as:

$$\vec{F}_{ij} = \frac{\vec{r}_{ij}}{\|\vec{r}_{ij}\|^3}$$

where,

- $\vec{r}_{ij} = \vec{p}_j - \vec{p}_i = (x_j - x_i, y_j - y_i, z_j - z_i)$
- $\|\vec{r}_{ij}\|^2 = (x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2 + \text{SOFTENING}$
- $\|\vec{r}_{ij}\| = \sqrt{\|\vec{r}_{ij}\|^2}$

The total force acting on body i due to all other bodies is then:

$$\vec{F}_i = \sum_{j \neq i} \vec{F}_{ij} = \sum_{j \neq i} \frac{\vec{r}_{ij}}{\|\vec{r}_{ij}\|^3}$$

After computing the total force, the velocity of body i is updated using the formula:

$$\vec{v}_i \leftarrow \vec{v}_i + \Delta t \cdot \vec{F}_i$$

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

		for (int j = 0; j < numBodies; 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 1: Velocity Calculation

The code implementing the velocity update based on the acting forces is shown in Listing 1.

After obtaining the new velocities, the second part of the computations is computing the new coordinates of each body for the next time step. This is done as:

$$x_i \leftarrow x_i + v_{x_i} \cdot \Delta t$$

$$y_i \leftarrow y_i + v_{y_i} \cdot \Delta t$$

$$z_i \leftarrow z_i + v_{z_i} \cdot \Delta t$$

The code implementing the new position computation is shown in Listing 2.

```c
for (int i = 0 ; i < numBodies; 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 2: Position Calculation

3.2 Results

Figure 2 shows the execution times (in seconds) of the baseline algorithm for varying problem sizes. It is evident that the execution time increases significantly as the number of particles increases. These results demonstrate the algorithm's time complexity, which could pose challenges for large-scale simulations. Therefore, further optimization is crucial to improve the algorithm's efficiency and scalability.

In the next sections, we experiment with different optimization techniques, utilizing OpenMP for CPU parallelism and CUDA for GPU acceleration.

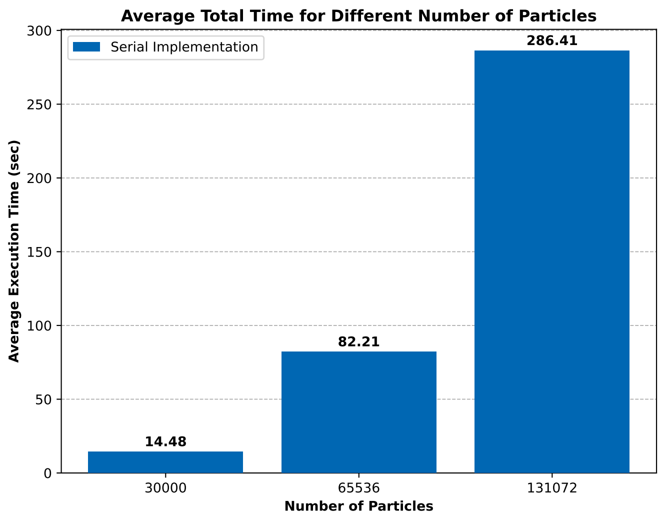

Figure 2: Execution Times of the Serial Implementation

4 OpenMP Algorithm

As shown in Section 3, the algorithm is separated into two distinct computationally intensive components. These components, although computationally intensive, also contain a lot of parallelism. In this section, we use OpenMP to explore and exploit this parallelism effectively.

4.1 Implementation

As discussed above, the parallelism of the algorithm is hidden in the two main loops that perform the computations used for the velocity and position calculation.

Here, using OpenMP we choose to parallelize these two loops. We create a parallel region outside the two loops and then we parallelize each for loop. As for the scheduling, after conducting several experiments, comparing static, dynamic, and guided scheduling, we chose dynamic scheduling with the default chunk size since it performed the best out of all other options. The code for the OpenMP implementation is shown in Listing 3 and 4.

```
    for (int iter = 1; iter <= nIters; iter++) {
    StartTimer();

        #pragma omp parallel
        {
                bodyForce(p, dt, nBodies); // compute interbody forces

                #pragma omp for schedule(dynamic)
                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;
                }
        }
  }
```

Listing 3: Position Calculation - OpenMP

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

Listing 4: Velocity Calculation - OpenMP

4.2 Results

Figure 3 shows the results of the OpenMP implementation. As expected, parallelizing the computationally intensive loops yielded a significant performance improvement. By distributing the workload across multiple CPU cores, we achieved a remarkable 29x speedup compared to the serial implementation.

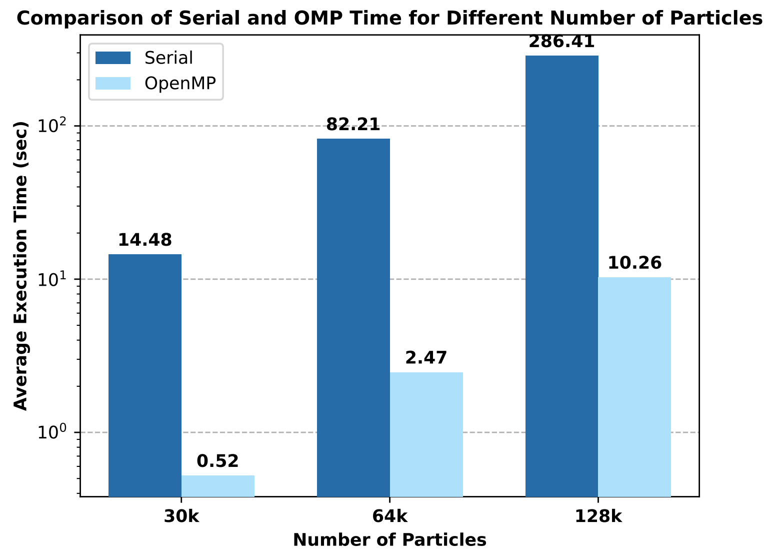

Figure 3: Serial vs OpenMP time comparison

5 GPU Implementation

5.1 Naive Implementation

For the first GPU implementation, we choose to port the velocity calculation component to the GPU since it is the most computationally intensive part of the algorithm. The position calculation is still performed on the host (CPU).

5.1.1 Implementation

When porting the code to the GPU, we create `numBodies` threads. Each thread acts as a body; thus, we completely parallelize the velocity computation for each body. When the velocities are computed, we transfer the new data back to the CPU where the new positions are determined from the new velocities. The CPU then sends the data back to the GPU to calculate the next new data. This process repeats as many times as the iterations of the simulation.

As for the geometry of the grid and blocks, we chose a 1D grid of 1D blocks where each block has 1024 threads.

```
__global__ void bodyForce(Body *p, float dt, int n) {
        int tid = threadIdx.x + blockIdx.x*blockDim.x;

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

        for (int i = 0; i < n; i++) {
                dx = p[i].x - p[tid].x;
                dy = p[i].y - p[tid].y;
                dz = p[i].z - p[tid].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;
        }

        if (tid < n) {
            p[tid].vx += dt*Fx;
            p[tid].vy += dt*Fy;
            p[tid].vz += dt*Fz;
        }
}
```

Listing 5: Single Kernel - Naive implementation

5.1.2 Results

We observe that the execution times in our first GPU implementation show noticeable improvement. Specifically, as the number of particles increases, the speedup also increases. This is because the algorithm benefits from the parallel processing capabilities of the GPU, which allow it to handle larger workloads more efficiently.

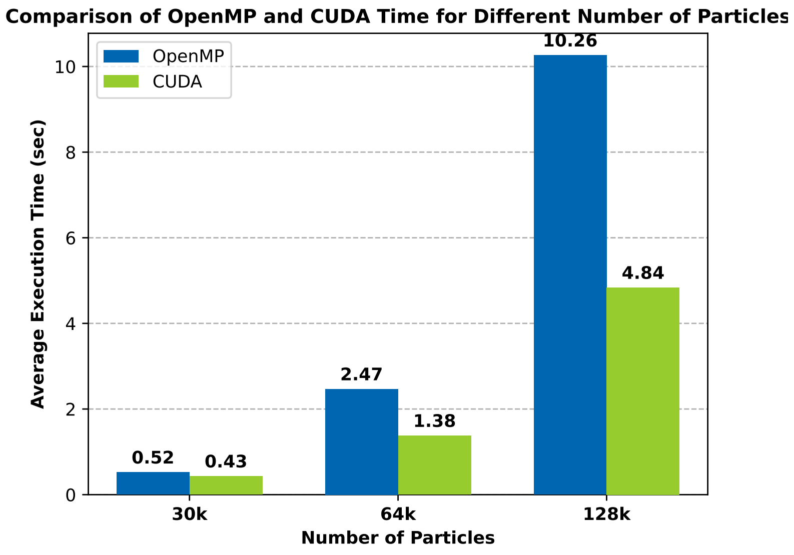

Figure 4: OpenMP vs CUDA Baseline Time Comparison

5.2 Data Distribution

5.2.1 Implementation

For the first optimization, we choose to change the distribution of the data. In the initial implementation, the data for each body reside in an array of structs. Each struct contains 6 elements, x,y, and z, representing the position of the body in a 3D coordinate system, and vx, vy, vz representing the velocity components of the body in the 3D space along the x, y, and z axes, respectively. In this implementation, we change the data distribution to a structure of arrays. One struct is created and inside of it, we store six matrices of size `numBodies`, one for each element mentioned above. This way, when subsequent threads access subsequent data (ie. thread 0 accesses x[0], thread one accesses x[1], and so on), they can benefit from memory coalescing. The code below shows the implementation of the data distibution optimization.

```
__global__ void bodyForce(Body p, float dt, int n) {
        int tid = threadIdx.x + blockIdx.x*blockDim.x;
        int tile;
        float dx, dy, dz;
        float distSqr, invDist, invDist3;
        float Fx = 0.0f;
        float Fy = 0.0f;
        float Fz = 0.0f;
        float3 curr_pos;

        curr_pos.x = p.position[tid].x;
        curr_pos.y = p.position[tid].y;
        curr_pos.z = p.position[tid].z;

        for (int i = 0; i < n; i++) {
                dx = p.position[i].x - curr_pos.x;
                dy = p.position[i].y - curr_pos.y;
                dz = p.position[i].z - curr_pos.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;
        }

        if (tid < n) {
            p[tid].vx += dt*Fx;
            p[tid].vy += dt*Fy;
            p[tid].vz += dt*Fz;
        }
}
```

5.2.2 Results

Figure 5 shows the results of the current optimization. We observe that no performance was gained by the new data distribution which in the current state of the code is expected since each thread, accesses the same elements from the global memory at each iteration.

5.3 Tiling

Tiling is a technique used to break down large data sets into smaller, more manageable blocks or tiles to optimize memory access patterns and reduce memory latency during computation. In this section, we use tiling to minimize the latency of the memory accesses on the global memory.

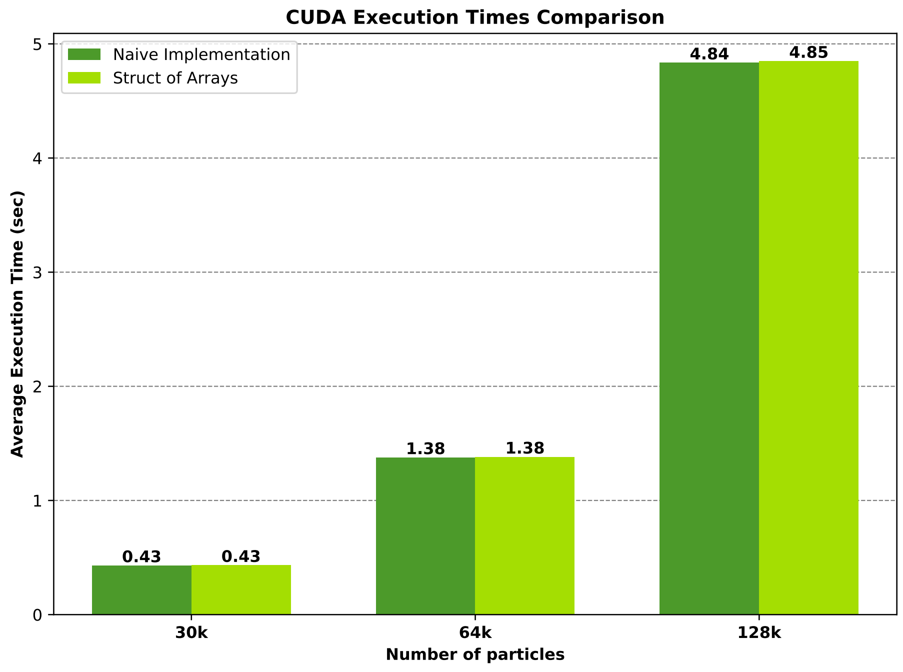

Figure 5: Struct of Arrays Optimization

5.3.1 Implementation

From Listing 5 we observe that in each iteration of the for loop, each thread performs six memory accesses on the global memory. Using tiling we attempt to eliminate these memory accesses.

First, we break the input data into $\lceil \frac{\texttt{numBodies}}{\texttt{TOTAL\_THREADS\_PER\_BLOCK}} \rceil$ tiles. To minimize global memory accesses we create an array of bodies (of size total threads_per_block) in shared memory. We bring each tile individually, and we load the corresponding bodies into the shared memory array for each tile. Then, each thread in the block performs its computations using the data stored in shared memory instead of accessing global memory. This reduces the number of global memory accesses significantly, as each thread now only accesses the shared memory, which is faster than global memory. Once all computations for the current tile are completed, the next tile is loaded into shared memory, and the process is repeated until all tiles are processed. The code below shows the implementation of the tiling optimization.

```
__global__ void bodyForce(Body *p, float dt, int tiles, int n) {
        int tid = threadIdx.x + blockIdx.x*blockDim.x;
        ...
        __shared__ Body private_bodies[THREADS_PER_BLOCK];
        Body curr_body = p[tid];

        for (tile = 0; tile < tiles-1; tile++) {
                private_bodies[threadIdx.x] = p[threadIdx.x + tile * blockDim.x];
                __syncthreads();
                for (int i = 0; i < THREADS_PER_BLOCK; i++) {
                        dx = private_bodies[i].x - curr_body.x;
                        dy = private_bodies[i].y - curr_body.y;
                        dz = private_bodies[i].z - curr_body.z;
                                        .
                                        .
                                        .
                }
                __syncthreads();
        }

        // Load last tile into shared memory
        private_bodies[threadIdx.x] = p[threadIdx.x + (tiles-1) * blockDim.x];
        __syncthreads();

        int last_bodies = (n%THREADS_PER_BLOCK == 0) ? THREADS_PER_BLOCK :
        ↪  n%THREADS_PER_BLOCK;

        for (int j = 0; j < last_bodies; j++) {
                                .
                                .
                                .
        }

        curr_body.vx += dt*Fx;
        curr_body.vy += dt*Fy;
        curr_body.vz += dt*Fz;

        // Update global memory
        p[tid] = curr_body;
}
```

The above code shows that the main loop has been separated into two loops. This is done to avoid redundant calculations when the total number of bodies is not divisible by the total number of threads per block, which leads to the creation of idle threads. Data loaded into shared memory by those threads should not be taken into consideration.

In this section, we tested two different implementations of the tiling algorithm, one with the initial data distribution of an array of structures and another one with the optimization of the previous section, a structure of arrays.

5.3.2 Results

Figure 6 shows the comparison between the tiling implementation with and without the struct of arrays optimization. Here, we observe that the struct of arrays outperforms the array of structs implementation. This small performance boost of the first optimization is attributed to the use of shared memory. When using shared memory, the first step is to populate it by loading all the data required for each tile. This process can benefit significantly from coalesced memory accesses. Without the struct of arrays, memory coalescing is not feasible because the data layout does not allow threads to access consecutive memory addresses efficiently.

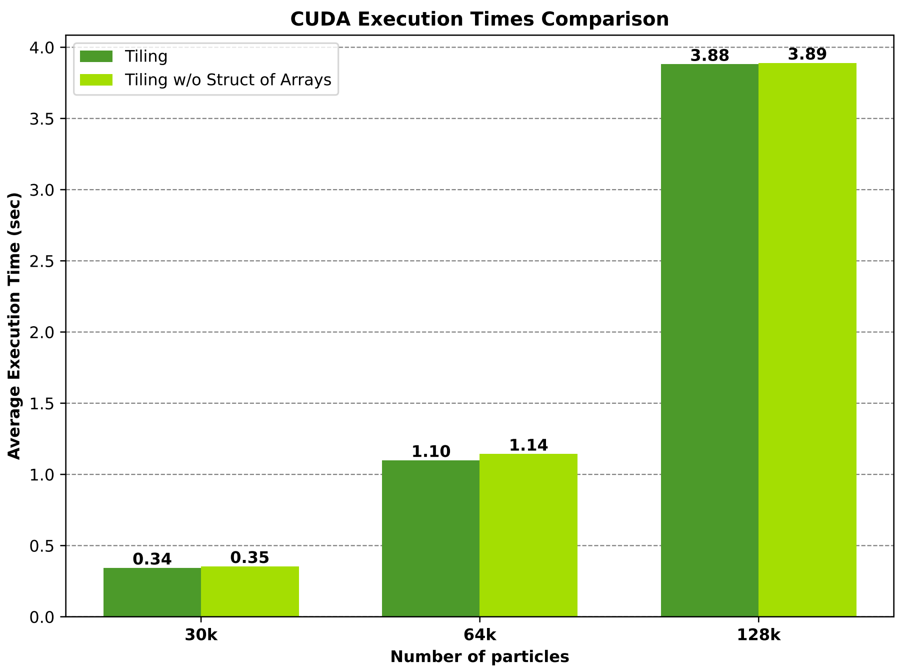

Figure 6: Comparison of Different Tiling Implementations

As for the performance improvements achieved by the tiling implementation compared to the previous approach, we achieve a speedup of 1.25x. Figure 7 illustrates the results.

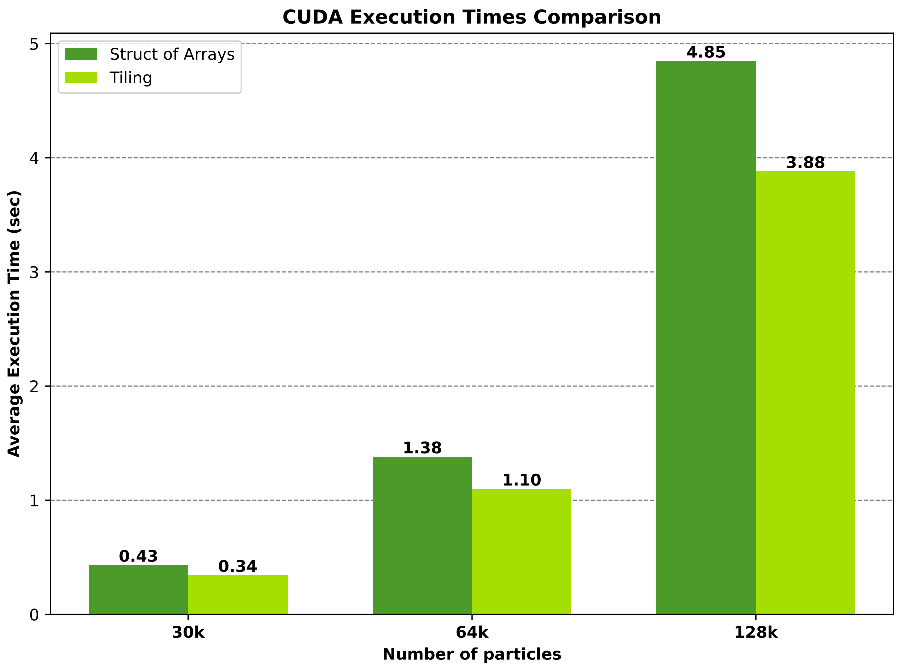

Figure 7: Tiling Implementation

5.4 Loop Unroll

In this section, we apply loop unrolling on the loops inside the `bodyForce()` kernel.

5.4.1 Implementation

We use the compiler directive `#pragma unroll <factor>`, which instructs the compiler to unroll a loop explicitly by the specified factor. The directive is used for both loops in the `bodyForce` kernel.

5.4.2 Results

For this optimization, we experimented with different unroll factors. Figure 8, shows the execution time of the simulation for unroll factors of 2, 4, 8, 16, and 24. We observe that for a factor of 16, our program achieves its highest performance so far.

Figure 9, shows the overall results in execution times that loop unrolling achieved compared to the previous implementation.

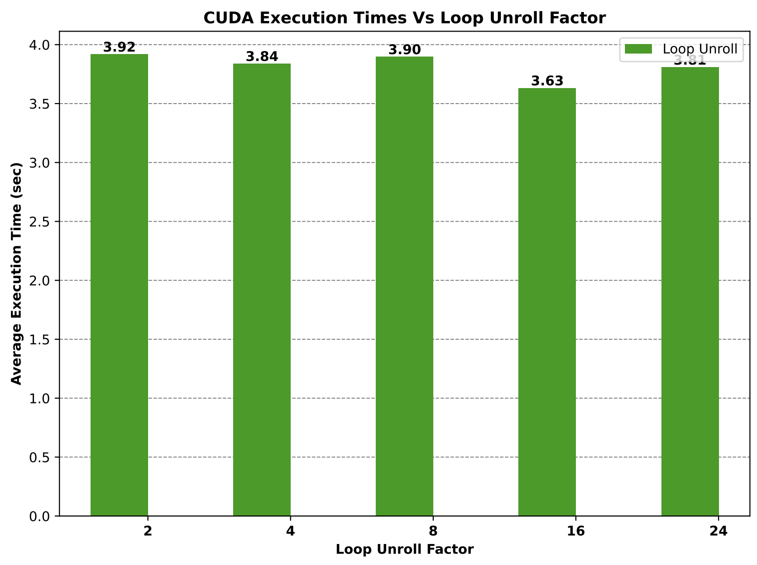

Figure 8: Loop Unroll Factor Comparison (N = 128k)

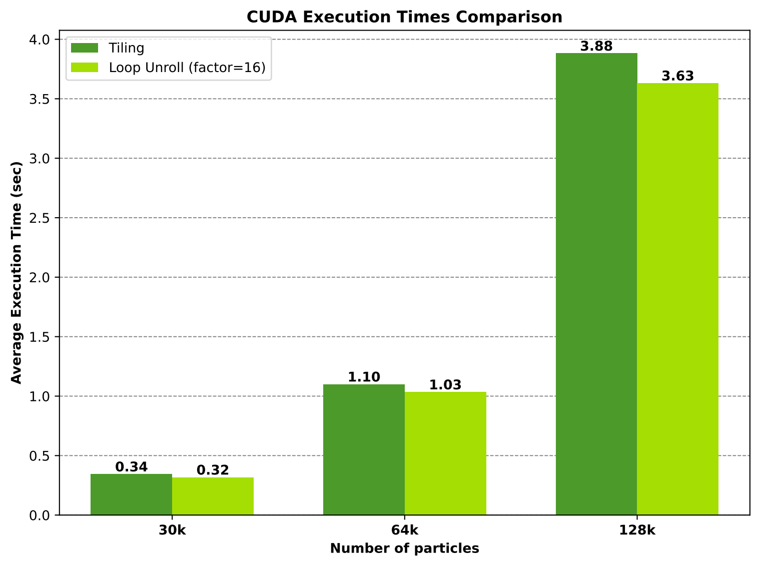

Figure 9: Loop Unroll Optimization Results

5.5 Position Calculation Kernel

Next, we try to parallelize the position calculation component.

5.5.1 Implementation

Here, we create a separate kernel, responsible for calculating the new position of each body. Again, we use the same geometry as with the previous optimizations. Each thread represents a body and is responsible for calculating its new position. The code for the new kernel is shown in Listing 6.

```
__global__ void calculatePositions(Body p, float dt, int n) {
        int tid = threadIdx.x + blockIdx.x*blockDim.x;
        if (tid >= n) {
                return;
        }

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

Listing 6: Position Calculation Kernel

5.5.2 Results

Figure 10, shows the performance gain of the current optimization. As expected, the new kernel had a small performance boost for the experiments with the largest number of particles since now there is no need for transferring the new coordinates back and forth between the host and the device. More specifically, the data transfers are now limited to only the first transfers of the initial coordinates and velocities of the bodies and one last transfer of the final coordinates.

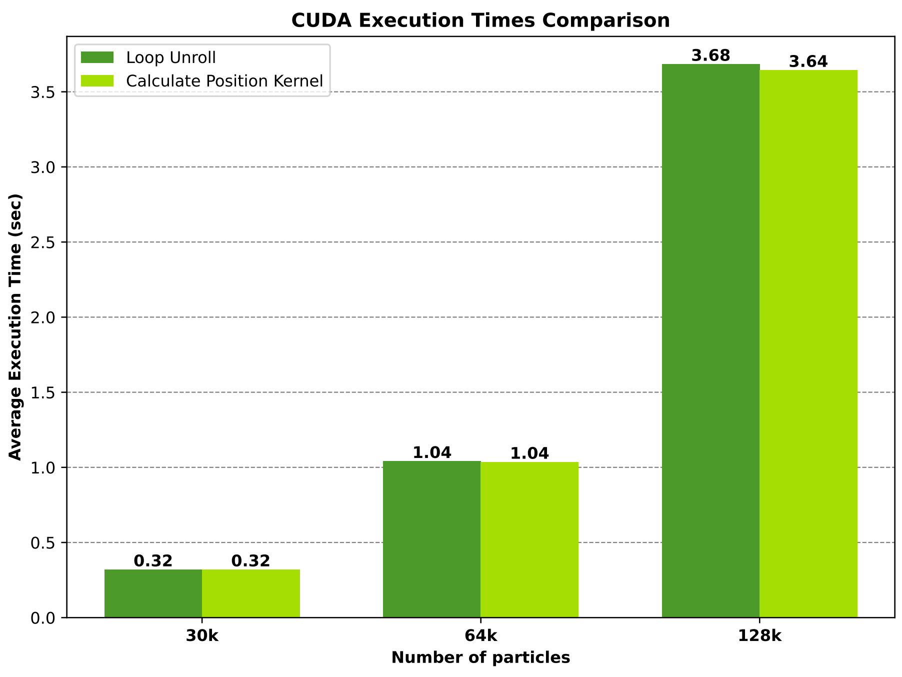

Figure 10: Calculate Position Kernel

5.6 FP16 - Half Precision Arithmetic

Modern hardware accelerators, such as GPUs, often feature specialized units for mixed-precision, like NVIDIA's Tensor Cores. Leveraging mixed precision arithmetic can offer advantages such as increased speed and reduced memory usage due to its smaller data size. However, it's crucial to consider the potential trade-offs with reduced precision and conversion overhead. Careful evaluation is necessary to determine if the performance benefits outweigh any potential accuracy issues for a given application.

5.6.1 Implementation

In this section, we choose to store all of the bodies's data in FP16 format. The code below shows the changes made to the `bodyForce` kernel.

```
__global__ void bodyForce(Body p, float dt, int tiles, int n) {
        int tid = threadIdx.x + blockIdx.x*blockDim.x;
        int tile;

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

        __shared__ __half body_coordinates_x[THREADS_PER_BLOCK];
        __shared__ __half body_coordinates_y[THREADS_PER_BLOCK];
        __shared__ __half body_coordinates_z[THREADS_PER_BLOCK];
        __half curr_x = p.x[tid];
        __half curr_y = p.y[tid];
        __half curr_z = p.z[tid];

        for (tile = 0; tile < tiles-1; tile++) {
                body_coordinates_x[threadIdx.x] = p.x[threadIdx.x +
                ↪  tile*blockDim.x];
                body_coordinates_y[threadIdx.x] = p.y[threadIdx.x +
                ↪  tile*blockDim.x];
                body_coordinates_z[threadIdx.x] = p.z[threadIdx.x +
                ↪  tile*blockDim.x];

                __syncthreads();
                #pragma unroll 16
                for (int i = 0; i < THREADS_PER_BLOCK; i++) {
                        dx = __half2float(__hsub(body_coordinates_x[i], curr_x));
                        dy = __half2float(__hsub(body_coordinates_y[i], curr_y));
                        dz = __half2float(__hsub(body_coordinates_z[i], curr_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();
        }

        // Bring last tile into shared memory;
        body_coordinates_x[threadIdx.x] = p.x[threadIdx.x +
        ↪  (tiles-1)*blockDim.x];
        body_coordinates_y[threadIdx.x] = p.y[threadIdx.x +
        ↪  (tiles-1)*blockDim.x];
        body_coordinates_z[threadIdx.x] = p.z[threadIdx.x +
        ↪  (tiles-1)*blockDim.x];
        __syncthreads();

        int last_bodies = (n%THREADS_PER_BLOCK == 0) ? THREADS_PER_BLOCK :
        ↪ n%THREADS_PER_BLOCK;

        #pragma unroll 16
        for (int i = 0; i < last_bodies; i++) {
                dx = __half2float(__hsub(body_coordinates_x[i], curr_x));
                dy = __half2float(__hsub(body_coordinates_y[i], curr_y));
                dz = __half2float(__hsub(body_coordinates_z[i], curr_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;
        }

        if (tid < n) {
            p.vx[tid] = __hadd(p.vx[tid], __float2half(dt*Fx));
            p.vy[tid] = __hadd(p.vy[tid], __float2half(dt*Fy));
            p.vz[tid] = __hadd(p.vz[tid], __float2half(dt*Fz));
        }
}
```

As shown in the code above, we store all the body's data in FP16 format. All data kept in shared memory are also in FP16 format.

For all FP16 operations, CUDA intrinsic functions are used such as `__hsub`, `__hadd` and `__hmul`. All operations involving half-precision arithmetic, such as distance calculations, are done using the intrinsic functions mentioned above. Then, the results are stored back in FP32 format to avoid any potential overflow/underflow. After the distance and force calculations are done, results are converted back to FP16 and the new velocities are to be calculated using the CUDA intrinsics.

5.6.2 Results

Figure 11 shows the execution times[1] of the current implementation. Here, we observe that the execution times increased slightly compared to the previous implementation.

[1]This optimization was evaluated on a different GPU due to the lack of support of mixed precision arithmetic in the Tesla K80.

A reason for this is the frequent use of the float to half and half to float conversions. Any decrease in execution time caused by the smaller memory transfers and faster computations of the half-precision arithmetic numbers is offset by the overhead introduced by additional conversion steps.

Removing any conversion steps between half and floats leads to better execution times, but with the reduced precision a lot of operations result in overflow/underflow which then leads to substantial errors in the final accuracy.

Regarding the accuracy of the current implementation, as anticipated, it experienced a significant drop due to the reduced precision. As evident from Figure 12, errors occurred even in the integer components (although not as frequently). Notably, the majority of errors (95%) occurred in the second, third, and fourth decimal places, with percentages of 20%, 60%, and 15%, respectively.

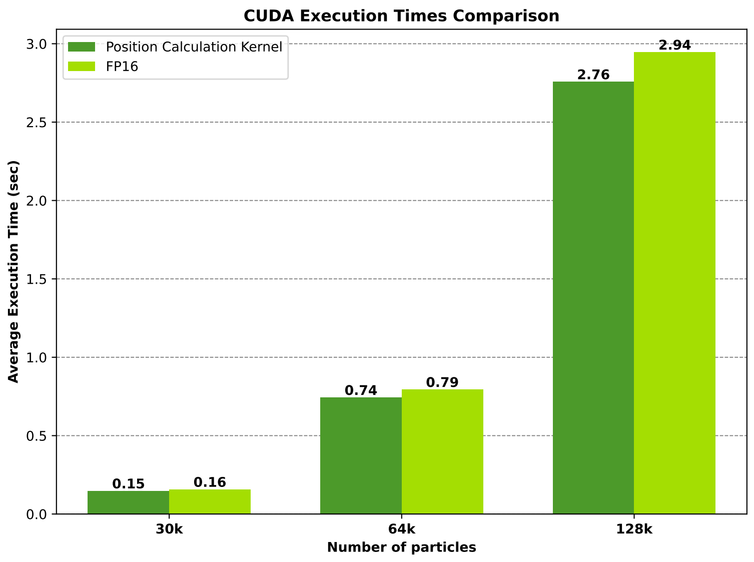

Figure 11: FP16 Optimization Results

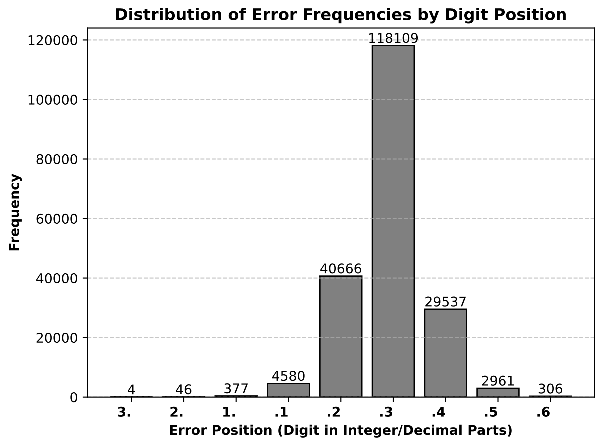

Figure 12: Error distribution for FP16 arithmetic

5.7 Pinned Memory

In our next attempt, we use the pinned memory to allocate space for the structure of arrays on the host. Pinned memory is a region of host memory that cannot be paged out to disk by the OS, allowing faster data transfers to and from the GPU. In this section, we replace malloc calls for the array memory allocations with the `cudaAllocHost()` API function. In doing so, the array is now allocated to pinned memory.

5.7.1 Results

Figure 13 compares the average execution times between the implementation using pinned memory and the previous approach. As observed, the performance remains the same, which is expected since there aren't enough memory transfers to benefit from the advantages of pinned memory.

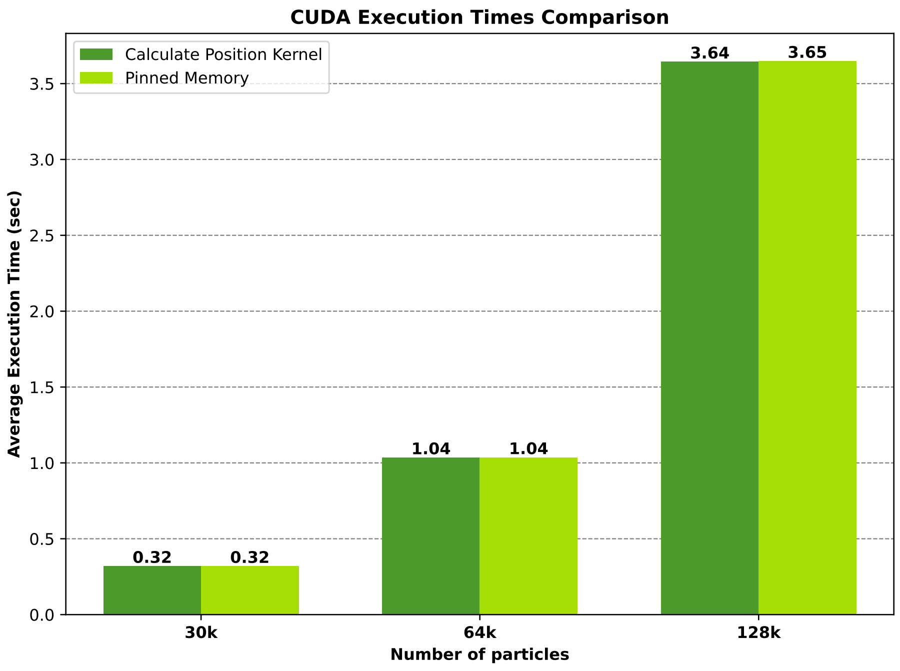

Figure 13: Pinned Memory

5.8 Fast inverse square root - rsqrtf

CUDA provides an approximate reciprocal square root function called rsqrtf() for single-precision floating-point numbers. This function calculates an approximation of $\frac{1}{\sqrt{x}}$, and it is designed to be fast while sacrificing some accuracy compared to the standard 1.0f / sqrtf(x).

5.8.1 Results

From Figure 14, we observe that, as expected, the execution times of this implementation are twice as fast as the previous one, with a measured speedup of 2.12x. As for accuracy, we still manage to maintain an acceptable level of performance that meets the requirements of the application. Figure 15 shows the distribution of error frequencies by digit position. We observe that the errors[2] start to appear after the third decimal point with only 16 errors, with most of the errors appearing at the sixth decimal point.

[2]The error is calculated as the difference between the coordinates obtained from the serial implementation and those obtained from the parallel implementation after the **second** iteration of the simulation.

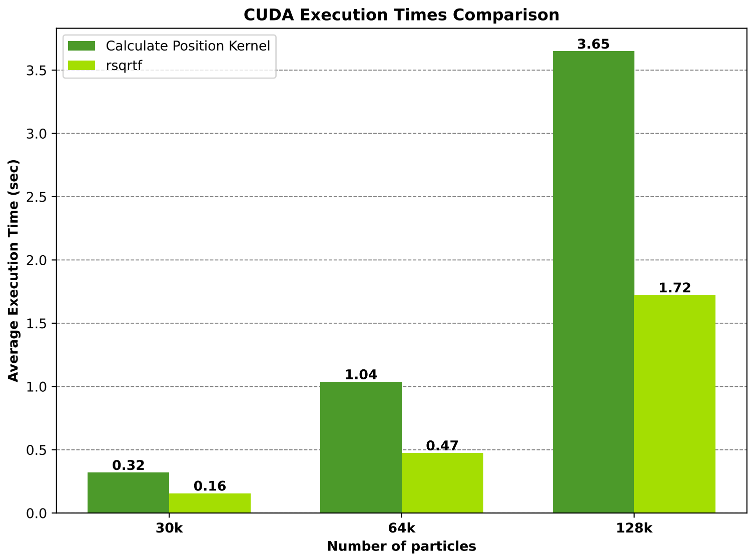

Figure 14: rsqrtf Implementation Results

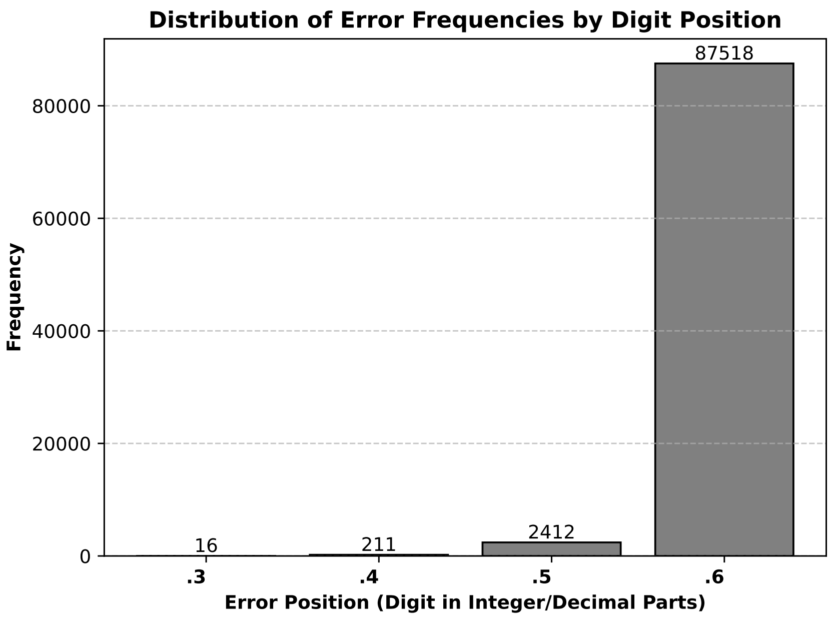

Figure 15: Error Distribution for rsqrtf Implementation

5.9 Floating-Point Denormals Support

Denormal (or subnormal) numbers are floating-point numbers that are very close to zero but not exactly zero. They are represented with less precision compared to normalized floating-point numbers.

Multi-instruction sequences such as the reciprocal squared root used in the n-body simulation must do extra work and take a slower path for denormal values. Nvcc provides the `-ftz=true` flag which causes all denormalized numbers to be flushed to zero. Here, we use this flag in an attempt to speed up any operations evolving denormal (or subnormal) values.

5.9.1 Results

Using the flush-to-zero flag we observe a significant performance boost on our code, as shown in Figure 16, achieving a speedup of 1.15x . As for the accuracy, no effect is observed compared to the previous implementations due to the use of a softening factor (Figure 17).

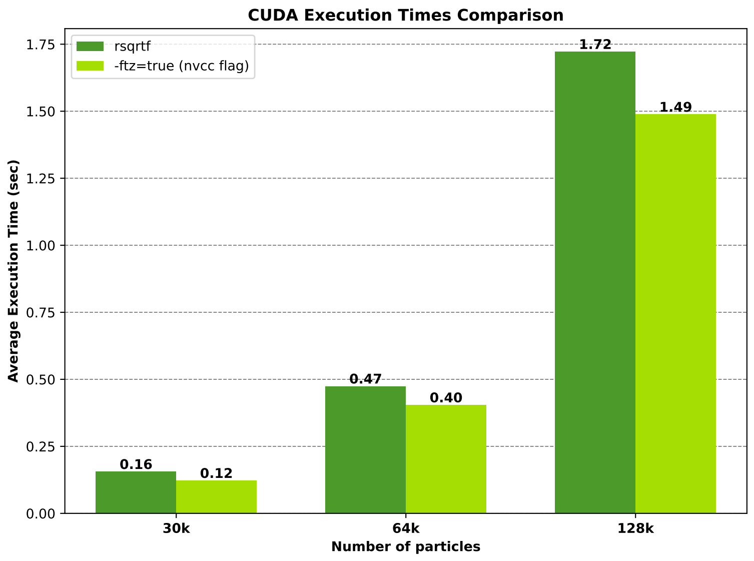

Figure 16: Flush-to-zero Optimization Results

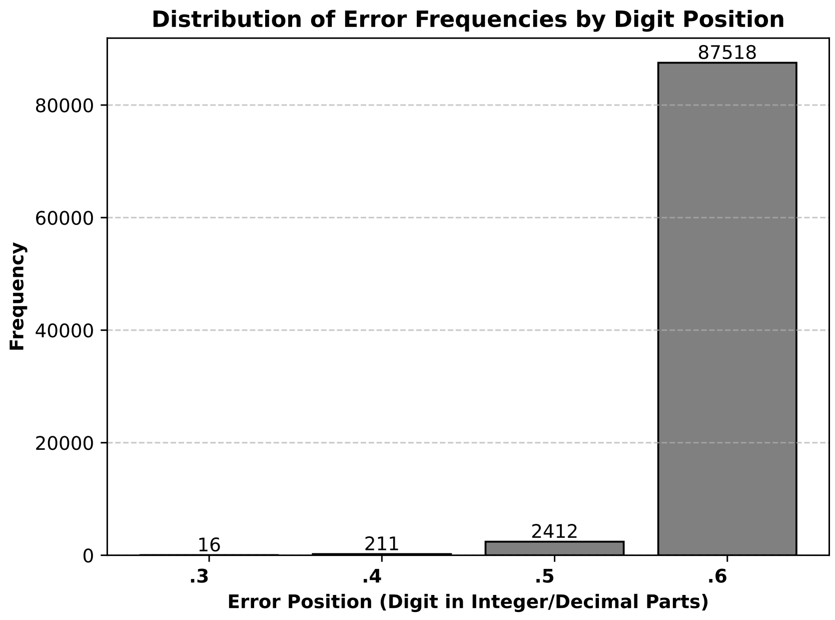

Figure 17: Error Distribution for ftz Implementation

5.10 Vectorized Memory Access: float3 vs float4

Next, we explored the use of float3 and float4 vector datatypes to represent groups of three or four floating-point values, respectively. This approach was applied to the structure of arrays, where we combined the six arrays into two distinct ones, a position and a velocity array of type float3/float4.
Although our implementation required only three variables to represent positions or velocities, we opted to use the float4 datatype instead of float3 due to performance considerations. GPUs are optimized for memory access aligned to 16-byte boundaries, which float4 inherently satisfies. Also, float4 takes full advantage of hardware optimizations designed for vectorized operations. In contrast, float3 often requires additional padding to maintain alignment, which can introduce inefficiencies in memory access and processing.

5.10.1 Results

Figure 18, shows the results from the current optimization. Here, we observe that neither float3 or float4 (figure 19) improved the overall execution time.

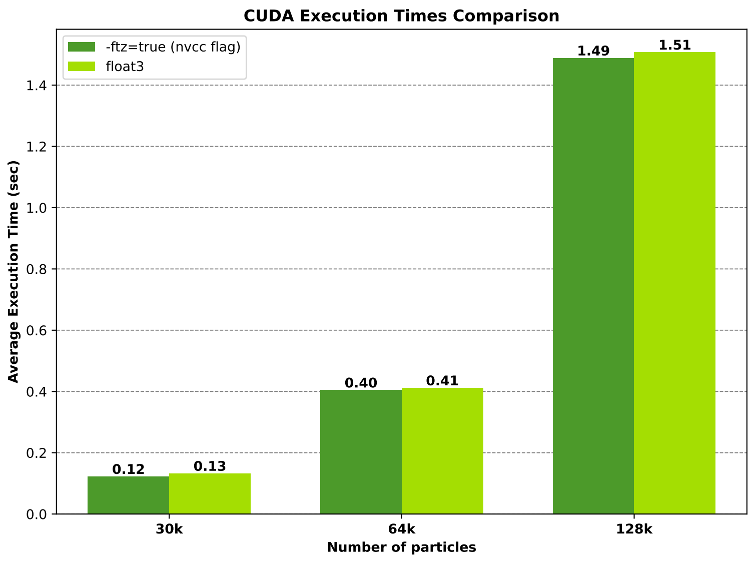

Figure 18: Float3 Results

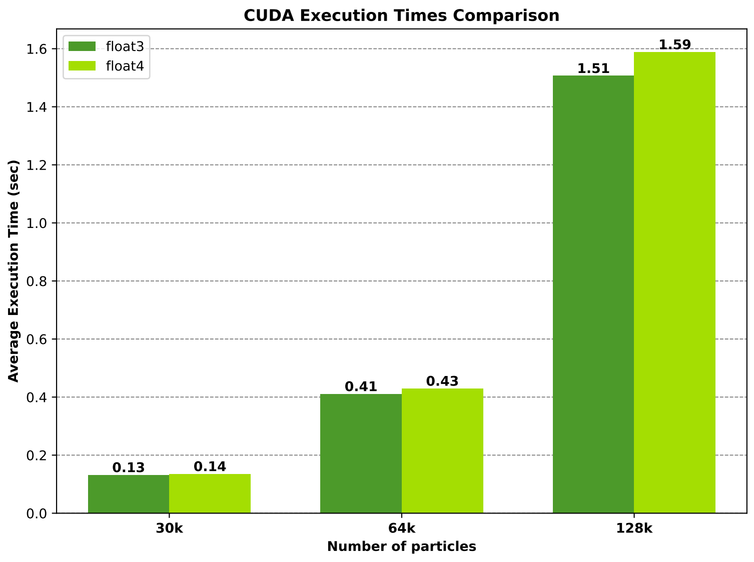

Figure 19: Float3 vs Float4

6 Conclusions

In this project, we successfully parallelized and optimized an N-body simulation using both OpenMP for CPU parallelization and CUDA for GPU acceleration. We began by analyzing the serial implementation of the algorithm, identifying its computationally intensive components and potential for parallelization.

Our OpenMP implementation focused on parallelizing the two main loops responsible for velocity and position calculations. By employing dynamic scheduling, we achieved efficient workload distribution across the available CPU cores, resulting in a significant performance improvement compared to the serial version.

For GPU acceleration, we initially ported the velocity calculation component to CUDA, utilizing a naive implementation with a one-dimensional grid of one-dimensional blocks. Subsequent optimizations included data distribution changes to enhance memory coalescing, tiling to reduce global memory accesses, loop unrolling to minimize loop overhead, and the introduction of a separate kernel for position calculation to further reduce data transfers between the host and device.

We also explored the use of half-precision arithmetic to leverage the specialized units on modern GPUs, but found that the overhead of frequent conversions between half and single-precision negated any potential benefits. Pinned memory allocation was also investigated but did not yield noticeable improvements due to the limited number of memory transfers in our application.

Significant performance gains were achieved by utilizing the rsqrtf() function for fast inverse square root calculations and enabling flush-to-zero optimization for handling denormalized numbers. These optimizations resulted in substantial speedup without compromising the accuracy of the simulation.

Finally, we experimented with vectorized memory access using float3 and float4 datatypes, but found no significant performance improvements.

Overall, this project demonstrated the effectiveness of parallelization and optimization techniques in accelerating N-body simulations. By combining OpenMP and CUDA, we achieved substantial performance gains compared to the serial implementation, enabling efficient simulations of larger particle systems. As shown in Figure 20, by strategically applying these optimizations, the CUDA implementation achieved a remarkable 192x speedup compared to the initial serial implementation and a 7x speedup compared to the OpenMP parallelized version.

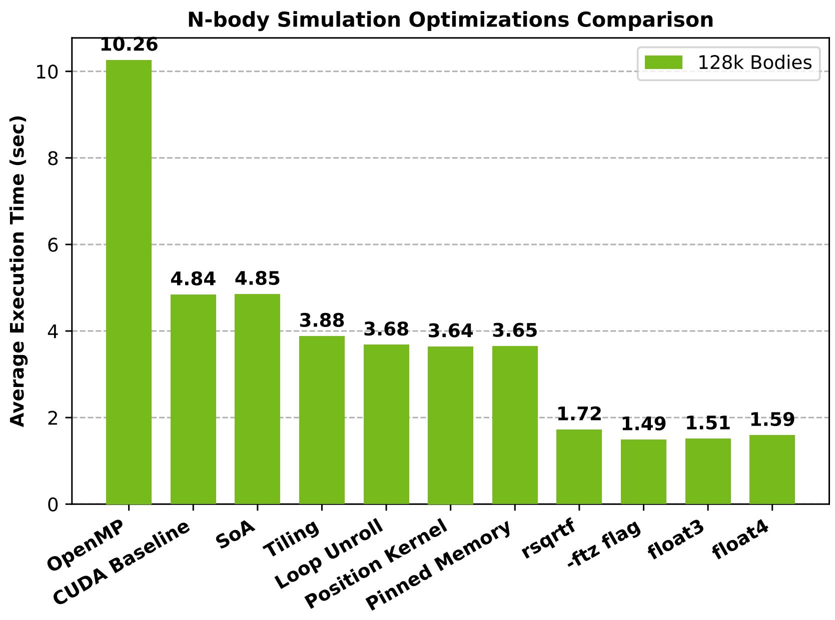

Figure 20: Timing Results Across All Optimizations

A System configuration: GPU Device Query

In the following sections, we provide the output of deviceQuery; NVIDIA's sample utility that provides detailed information about the GPU(s) on a system. The GPUs that we used for the evaluation of all optimizations are shown in the following Listing.

A.0.1 Venus

```
  ./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 4 CUDA Capable device(s)

Device 0: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.4 / 11.5
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes
     (11997020160 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
     =(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
      2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  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)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy
     engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 6 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice()
        with device simultaneously) >

Device 1: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.4 / 11.5
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes
     (11997020160 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
     =(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
      2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  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)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy
     engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 7 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice()
        with device simultaneously) >

Device 2: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.4 / 11.5
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes
     (11997020160 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
     =(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
      2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  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)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy
     engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 132 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice()
        with device simultaneously) >

Device 3: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.4 / 11.5
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes
     (11997020160 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
     =(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
      2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  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)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy
     engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 133 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice()
        with device simultaneously) >
> Peer access from Tesla K80 (GPU0) -> Tesla K80 (GPU1) : Yes
> Peer access from Tesla K80 (GPU0) -> Tesla K80 (GPU2) : No
> Peer access from Tesla K80 (GPU0) -> Tesla K80 (GPU3) : No
> Peer access from Tesla K80 (GPU1) -> Tesla K80 (GPU0) : Yes
> Peer access from Tesla K80 (GPU1) -> Tesla K80 (GPU2) : No
> Peer access from Tesla K80 (GPU1) -> Tesla K80 (GPU3) : No
> Peer access from Tesla K80 (GPU2) -> Tesla K80 (GPU0) : No
> Peer access from Tesla K80 (GPU2) -> Tesla K80 (GPU1) : No
> Peer access from Tesla K80 (GPU2) -> Tesla K80 (GPU3) : Yes
> Peer access from Tesla K80 (GPU3) -> Tesla K80 (GPU0) : No
> Peer access from Tesla K80 (GPU3) -> Tesla K80 (GPU1) : No
> Peer access from Tesla K80 (GPU3) -> Tesla K80 (GPU2) : Yes

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.4, CUDA
    Runtime Version = 11.5, NumDevs = 4, Device0 = Tesla K80,
   Device1 = Tesla K80, Device2 = Tesla K80, Device3 = Tesla K80
Result = PASS
```

Listing 1: Output of deviceQuery on Venus