Electrical and Computer Engineering
University of Thessaly (UTH)

ECE415 - High Performance Computing (HPC)

Fall Semester — Educational year 2024-2025

Lab05

N-Body Parallel implementation using OMP & CUDA

Dimitris Tsalapatas - AEM: 03246

Georgios Kapakos - AEM: 03165

Contents

1 Introduction

In physics and astronomy, an N-Body simulation models the movement of multiple particles influenced by forces like gravity. N-body simulations are implemented in astrophysics tools to investigate the dynamics of few-body systems like the Earth-Moon-Sun system to understanding the evolution of the large-scale structure of the universe. In cosmology, N-body simulations help explore how structures like galaxy clusters and halos form due to dark matter. They're also used to understand the evolution of star clusters over time.

The "N" in N-Body represents the number of particles, which can be any positive integer. These simulations aim to predict particle motion over time using their initial positions, velocities, and the forces influencing them.

N-body simulations are computationally intensive because they involve calculating pairwise interactions between all particles in the system, resulting in O(N^2) complexity as the number of particles increases. This is a bottleneck for large systems, as the computations grow exponentially.

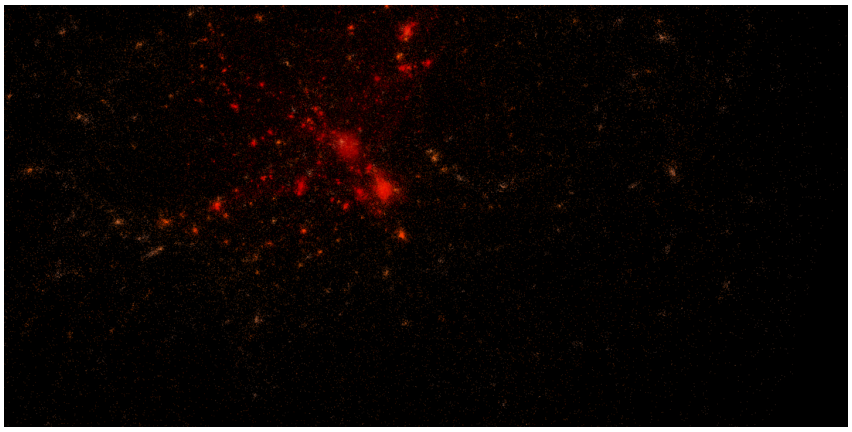


Figure 1: Simulation of a galaxy cluster using N-Body simulation.

2 Device Query

We used the DeviceQuery command and found out the Tesla K80 GPU that we use has the following specifications:

- **Device Name**: Tesla K80
- **CUDA Capability Version**: 3.7
- **Total Global Memory**: 11441 MBytes
- **Total CUDA Cores**: 2496
- **GPU Max Clock Rate**: 824 MHz (0.82 GHz)
- **Memory Clock Rate**: 2505 MHz
- **Memory Bus Width**: 384-bit
- **L2 Cache Size**: 1572864 bytes

- **Total Shared Memory per Block**: 49152 bytes
- **Total Constant Memory**: 65536 bytes
- **Total Number of Registers per Block**: 65536
- **Maximum 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)

3 Error Checking

For the checking of our code we have constructed a python script that checks the first iteration's position values (x, y, z), of the particles in the CPU with the position values in the GPU after each optimization step. The values which are to be checked are printed with an accuracy of 6 decimal points, and a threshold of 0.005 is implemented for checking the accuracy of the results.

We did not observe any errors on our results for 131072 body size we observed that only two differences were present for a threshold of 0.005.

4 Evaluation

For the Original code we used the -gcc compiler. On the CPU accelerated code we switched to the -icx compiler and finally for the GPU implementation we used the -nvcc compiler.

We evaluate the performance of each run by taking the average execution time and the total run time along with the Billion interactions per second for the N-Body simulation system. We demonstrate the improvement for each optimization as a speedup with regard to the previous execution. We experiment different optimizations, where there is an optimization that results in an execution time improvement, we keep it and continue with optimizing this code.

For each implementation we ran the experiments for multiple times and for a different number of particles with the following values: 32.768, 65.536, 131.072, 262.144, to test different sizes and see how the optimization may be influences by the size.

The speedup is calculated by the old value divided by the new value. We only calculate the speedup for the 131072 bodies.

For the plots we plot the average execution time for at each iteration. After, running the iterations excluding the first one always, as well as the standard deviation at each iteration.

The CUDA code is measured by cuda events where the cpu code is measured by functions provided to us by the timer.h library. We measure the iterations adding the cpu and gpu time.

5 Optimizations

The provided code is highly optimizable for both CUDA and OpenMP due to the inherent parallel nature of the computations. Each particle in the system interacts with every other particle independently, meaning the force calculations for each particle can be executed concurrently without dependencies.

5.1 CPU Baseline

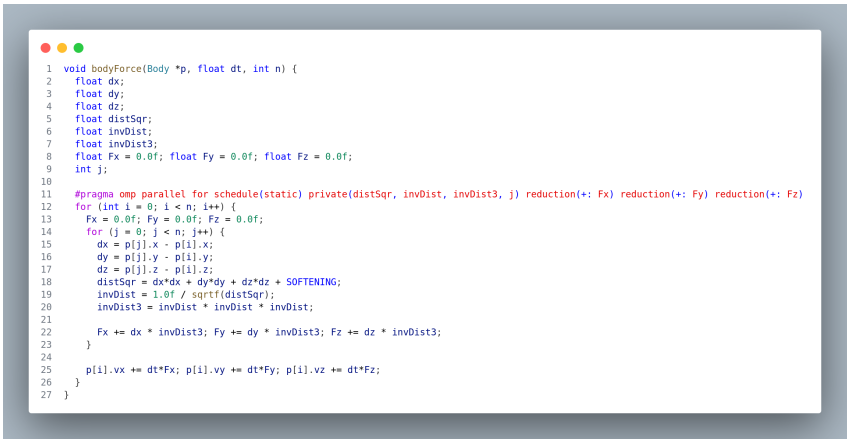
The sequential CPU implementation resulted in a very poor performance, this comes to no surprise as it is the simplest implementation of the N-Body simulation. The code has a complexity of O(N^2), this scales exponentially as the sizes increase.

5.2 OpenMP

This piece of code is ideal for OpenMP, where parallel threads on a CPU can share the workload efficiently using loop parallelization.

5.2.1 1st OMP Approach

We place the *#pragma omp parallel for* directive to parallelize the outer loop. We use the private directive for the variables distSqr, invDist, invDist3 and j to ensure that each thread will have its own private copy of these variables, avoiding data races. We also use the reduction directive to aggregate contributions to the Fx, Fy and Fz variables across threads safely. In the end we apply static scheduling, assigning equal sized chunks of iterations to threads in a static manner.



```
void bodyForce(Body *p, float dt, int n) {
  float dx;
  float dy;
  float dz;
  float distSqr;
  float invDist;
  float invDist3;
  float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;
  int j;

  #pragma omp parallel for schedule(static) private(distSqr, invDist, invDist3, j) reduction(+: Fx) reduction(+: Fy) reduction(+: Fz)
  for (int i = 0; i < n; i++) {
    Fx = 0.0f; Fy = 0.0f; Fz = 0.0f;
    for (j = 0; j < n; j++) {
      dx = p[j].x - p[i].x;
      dy = p[j].y - p[i].y;
      dz = p[j].z - p[i].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;
    }

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

Figure 2: OpenMP code optimization, with directives for CPU optimization

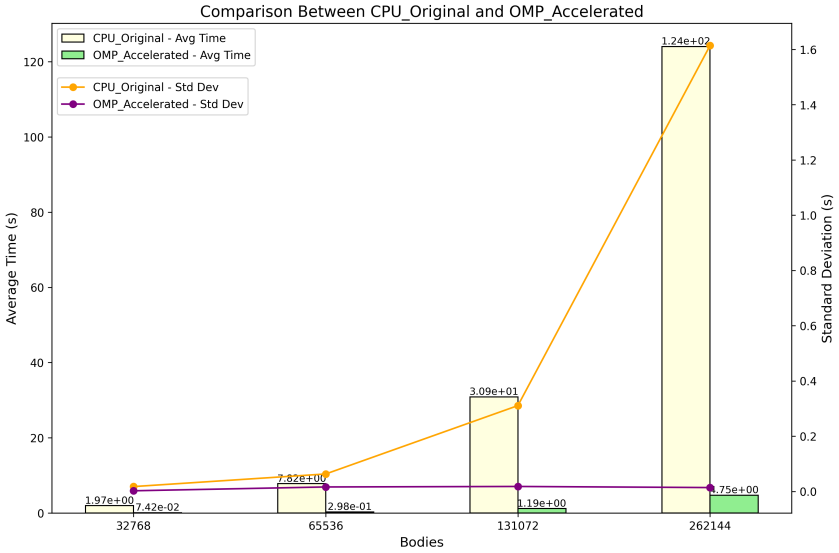


Figure 3: Comparison between baseline cpu and 1st omp approach

The OpenMP code provided a speedup of x25.96 times, as the computation of forces is split in the CPU cores.

5.2.2 2nd OMP Approach

We figured out that there was a simpler implementation for the openMP code as private is not needed for the variables: distSqr, invDist, invDist3 as they are declared and used within the inner loop. Where j is local to each iteration of the outer loop and doesn't persist across threads. The variables: Fx, Fy, and Fz are specific to each body and computed independently within each thread, so there's no shared accumulation requiring a reduction directive. So we removed all the unnecessary directives.

```c
void bodyForce(Body *p, float dt, int n) {
  float dx;
  float dy;
  float dz;
  float distSqr;
  float invDist;
  float invDist3;
  float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;
  int j;

  #pragma omp parallel for schedule(static)
  for (int i = 0; i < n; i++) {
    Fx = 0.0f; Fy = 0.0f; Fz = 0.0f;
    for (j = 0; j < n; j++) {
      dx = p[j].x - p[i].x;
      dy = p[j].y - p[i].y;
      dz = p[j].z - p[i].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;
    }

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

Figure 4: Reducing the omp directives on our code, keeping only the essential for optimization on the CPU.

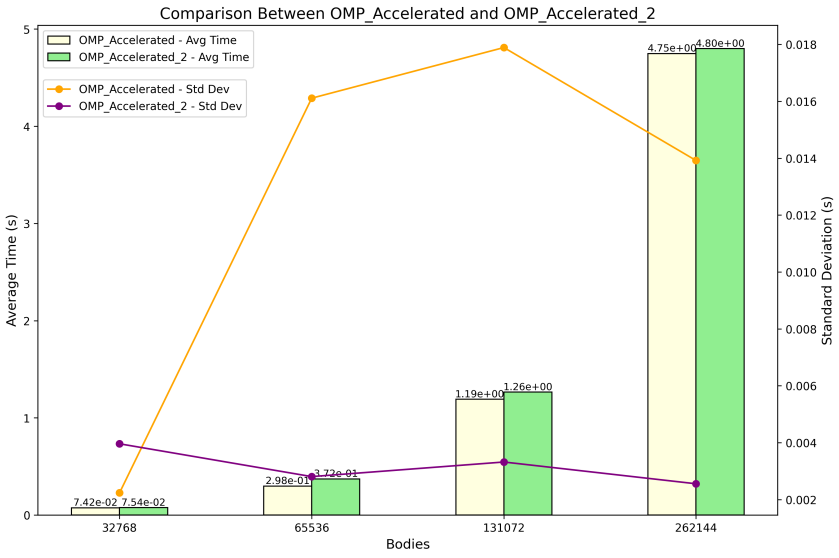


Figure 5: Comparison between 1st omp and 2nd omp approach.

The implementation of the second code brought an optimization of x0.944 times, proven to be worse than the previous OpenMP code. This has basically no difference at all, for a CPU code.

5.2.3 Comparison of omp methods

Optimisation Method	Threads	Bodies	Average Time (s)	Std Dev Time (s)	Average Interaction (Billion/s)
OMP_Accelerated	4	131072	1.045e+01	2.310e+00	1.645
	8		6.275e+00	2.074e+00	2.738
	14		2.861e+00	7.806e-01	6.005
	28		2.104e+00	3.019e-02	8.165
	56		1.156e+00	3.365e-02	14.863
OMP_Accelerated_2	4	131072	1.058e+01	2.816e+00	1.624
	8		4.497e+00	2.550e-03	3.82
	14		2.649e+00	5.450e-02	6.484
	28		1.941e+00	8.931e-02	8.852
	56		1.155e+00	3.654e-02	14.869

Table 1: Execution Time for OMP with different OMP_NUM_THREADS and Bodies equal to 131072

This table showcases the difference in iteration per step for every implementation ranging from 4 to 56 threads. We note that the 56 threads implementation is the best for both omp codes. Lastly in this run it seems that the second omp code is better in comparison to the first code. This is done for 131072 bodies.

So in the end, we continue with the second omp code for the next comparisons. As it seems to have a slight speedup x1.0008 times faster.

5.3 GPU Implementation

CUDA leverages the independence of these calculations by assigning a GPU thread to handle computations for each particle or interaction. CUDA's shared memory can further optimize performance by reducing redundant memory accesses during repeated interactions within a block. Additionally, the structure of the code allows for memory coalescing and the use of fast math functions to speed up calculations.

5.3.1 GPU Baseline

At our first implementation of the code in the GPU, we transfer the function bodyForce to the GPU. Each thread now computes the force acting on a single body (index), leveraging the GPU's parallel power. For each iteration of the code we follow the procedure below:

- Transfer the buffer, who contains the data of body from the CPU to the GPU.
- Call the kernel bodyForce().
- Transfer the data back from the GPU to the CPU.
- Calculate the new position values based on the values calculated on the GPU.

This procedure on the main function will be applied for each of the GPU optimizations below.

```cpp
__global__ void bodyForce(Body *p, float dt, int n) {
  int index = blockDim.x * blockIdx.x + threadIdx.x;
  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[index].x;
    float dy = p[j].y - p[index].y;
    float dz = p[j].z - p[index].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[index].vx += dt*Fx; p[index].vy += dt*Fy; p[index].vz += dt*Fz;
}
```

Figure 6: Naive GPU implementation.

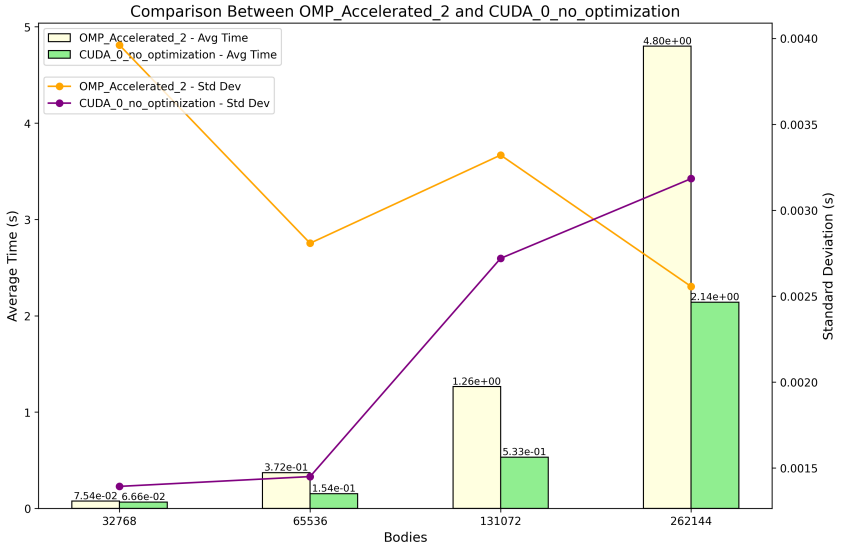


Figure 7: Comparison between 2nd omp approach and gpu naive implementation.

We observe that the GPU implementation brought a huge speedup when compared with the OpenMP accelerated code. The speedup is x2.36 times faster. This is the case, due to the massive parallelization capabilities of the GPU. Unlike the CPU, which has fewer but more powerful cores, the GPU is designed to handle thousands of smaller, lightweight threads simultaneously, making it highly efficient for compute-intensive tasks such as the N-Body simulation.

5.3.2 Registers

Registers are used to store the current position of the body, instead of accessing them from the global memory. Registers are the fastest memory on a GPU, providing low-latency, high-bandwidth access compared to global memory. By storing frequently used data in registers, we significantly reduce the number of global memory reads, improving computational efficiency and performance.

```cuda
__global__ void bodyForce(Body *p, float dt, int n) {
  int index = blockDim.x * blockIdx.x + threadIdx.x;
  float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;
  float px_reg = p[index].x, py_reg = p[index].y, pz_reg = p[index].z;

  for (int j = 0; j < n; j++) {
    float dx = p[j].x - px_reg;
    float dy = p[j].y - py_reg;
    float dz = p[j].z - pz_reg;
    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[index].vx += dt*Fx; p[index].vy += dt*Fy; p[index].vz += dt*Fz;
}
```

Figure 8: Registers implementation on the CUDA code.

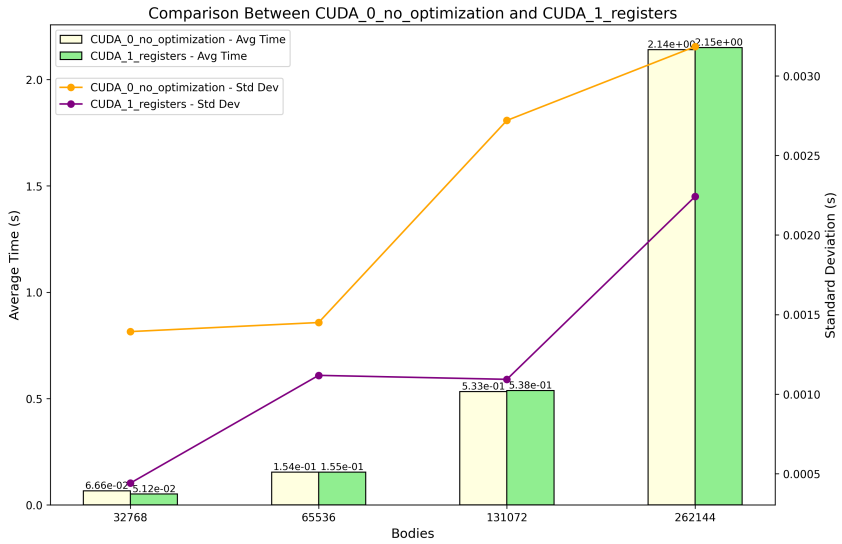


Figure 9: Comparison between the naive cuda code and the registers implementation.

The speedup from registers is x0.99 times the first cuda implementation. The registers made the code slightly slower, even though we do not use enough registers to exceed the number of registers used per thread. This speedup difference is irrelevant, but we expected the registers to deliver a speedup, but such case was not made.

5.3.3 Struct of Arrays(SoA)

SoA format is generally better-suited for CUDA due to its memory access and performance advantages.In an AoS format, each data element contains multiple fields, and these structures are stored consecutively in memory, where in SoA, each field of the structure is stored in a separate array. SoA ensures coalesced memory access, improves cache utilization, and reduces alignment overheads.

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

  float Fx = 0.0f, Fy = 0.0f, Fz = 0.0f;
  float px = p.x[index];
  float py = p.y[index];
  float pz = p.z[index];

  // Direct pairwise force calculation
  for (int j = 0; j < n; j++) {
      float dx = p.x[j] - px;
      float dy = p.y[j] - py;
      float dz = p.z[j] - pz;
      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;
  }

  // Update velocities
  p.vx[index] += dt * Fx;
  p.vy[index] += dt * Fy;
  p.vz[index] += dt * Fz;
}
```

Figure 10: Structure of Arrays implementation(SoA).

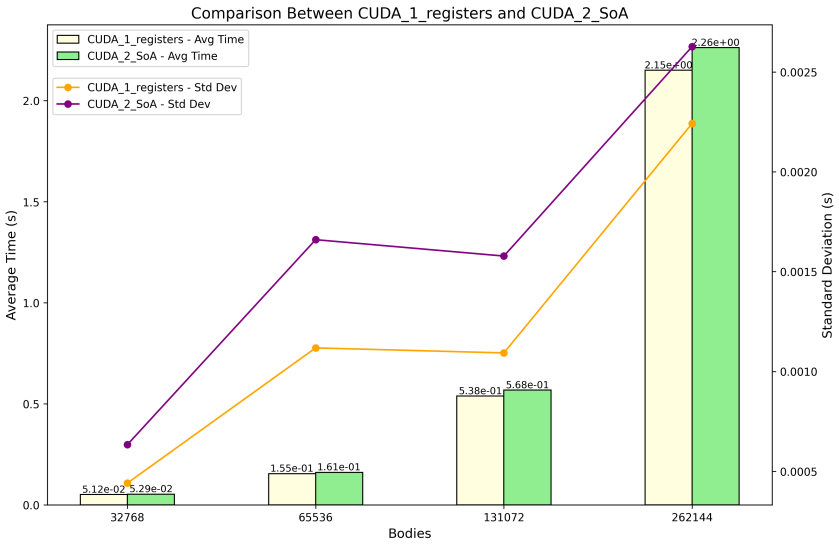


Figure 11: Comparison between registers implementation and SoA.

The Struct of Arrays resulted in a speedup of x0.947 times. This can be explained by reason that the memory accesses are non-coalesced memory accesses, which can significantly impact GPU performance. In a SoA layout, data for a single entity is stored across separate arrays, leading to scattered memory access patterns.

5.3.4 Float3 type

We use the float3 type to describe the attributes of position and velocity of the Body struct. Now the x, y, z variables are transformed into 3-D vectors. Float3 optimizes the memory ac-

cesses enabling for coalesced memory accesses on the GPU.

```cpp
__global__ void bodyForce(Body p, float dt, int n) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    float Fx = 0.0f, Fy = 0.0f, Fz = 0.0f;
    float px = p.position[index].x;
    float py = p.position[index].y;
    float pz = p.position[index].z;

    for (int j = 0; j < n; j++) {
        float dx = p.position[j].x - px;
        float dy = p.position[j].y - py;
        float dz = p.position[j].z - pz;
        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.velocity[index].x += dt * Fx;
    p.velocity[index].y += dt * Fy;
    p.velocity[index].z += dt * Fz;
}
```

Figure 12: Float3 with SoA implementation.

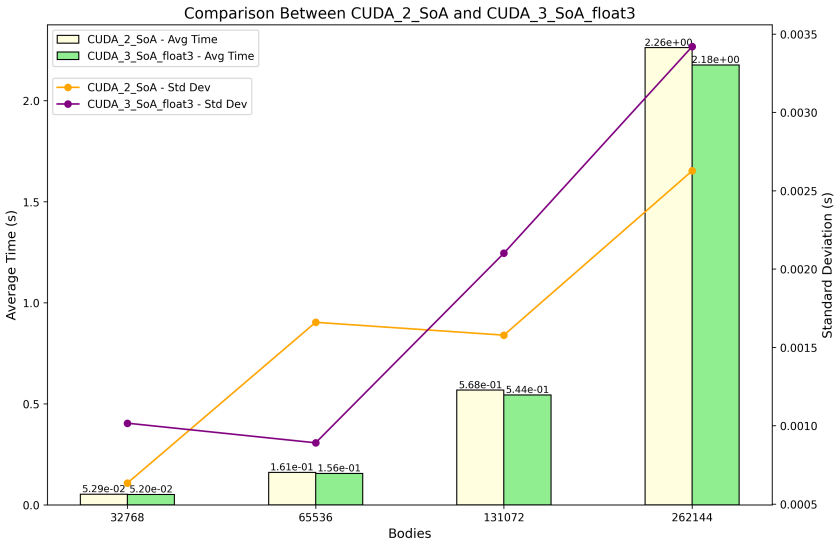


Figure 13: Comparison between float and float3 implementation on SoA.

The introduction of float3 types has achieved a speedup of x1.044 times, this is done a comparison to the SoA float implementation. This is due to the improved memory alignment and reduced overhead associated with vectorized data types. Using float3 allows related data (e.g., x, y, z components) to be grouped together in memory, which minimizes the overhead of managing separate variables and improves data locality.

But in comparison to the registers implementation which performed better, it has a x0.989 times speedup, which is still worse.

5.3.5 Float4 type

Using float4 has more advantages even when only using three components (x, y, z), because of its optimal 16-byte alignment, which matches the GPU's preferred memory transaction size. This ensures efficient coalesced memory access, maximizing memory bandwidth and reducing latency.

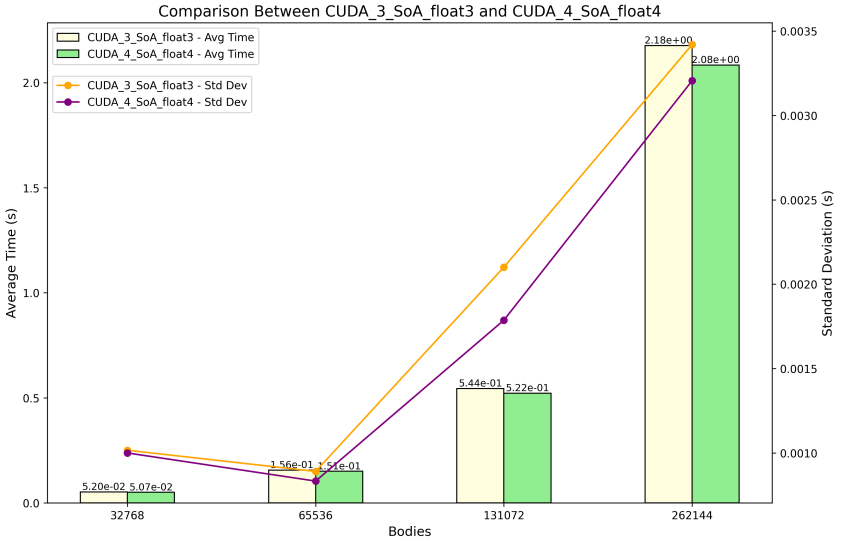


Figure 14: Comparison between float3 and float4 implementation on SoA.

The float4 type also made an improvement on the execution time, achieving a speedup of x1.042 times, compared to the float3 implementation. This is due to the fact that float4 ensures memory alignment to 16 bytes, which matches the preferred memory access granularity of many GPU architectures. By aligning data to 16-byte boundaries, float4 reduces memory access overhead and improves coalesced memory transactions, allowing the GPU to fetch data more efficiently.

It also provides a x1.03 speedup, when compared to the registers implementation.

5.3.6 Tiling

A common strategy to reduce memory traffic is to partition the data into subsets called tiles so that each tile fits into the shared memory. Shared memory access reduces expensive global memory accesses. An important criterion is that kernel computation on these tiles can be performed independently of each other. Tiling is a program transformation technique that localizes the memory locations accessed among threads and the timing of their accesses. In our case the code has a lot of global memory accesses which are considered expensive. We face this problem with tiling as we store the data of body in the shared memory of each block. Because of the size limitations of shared memory, we set a size of NUM_THREADS, for the shared memory at each block.

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

    float Fx = 0.0f, Fy = 0.0f, Fz = 0.0f;
    float px = p.position[index].x;
    float py = p.position[index].y;
    float pz = p.position[index].z;

    __shared__ float4 position_s[MAX_THREADS_PER_BLOCK];

    for (int tile = 0; tile < gridDim.x; tile++) {
        __syncthreads();
        int tiledIndex = tile * blockDim.x + tx;
        position_s[tx] = p.position[tiledIndex];
        __syncthreads();

        for (int j = 0; j < blockDim.x; j++) {
            float dx = position_s[j].x - px;
            float dy = position_s[j].y - py;
            float dz = position_s[j].z - pz;
            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;
        }
        __syncthreads();
    }

    p.velocity[index].x += dt * Fx;
    p.velocity[index].y += dt * Fy;
    p.velocity[index].z += dt * Fz;
}
```

Figure 15: Tiling implementation with shared memeory and SoA float4.

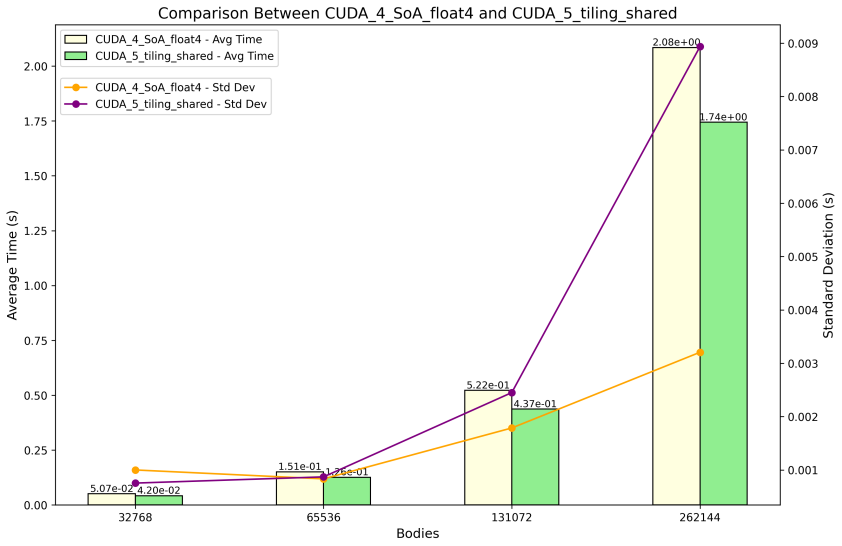


Figure 16: Comparison between tiling and no tiling implementation of float4 SoA.

The shared memory tiling implementation helped us improve our code by about x1.195 times, because it significantly reduced global memory accesses by leveraging the much faster shared memory within the GPU. By dividing the computation into smaller tiles, the data needed for each tile is loaded into shared memory, where threads within the same block can access it at high speed.

5.3.7 Loop Unrolling

Loop unrolling is beneficial for GPUs because it minimizes control flow overhead and maximizes parallel execution. By unrolling a loop, multiple iterations are executed in parallel, reducing the need for frequent loop condition checks and variable increments. This enhances the throughput of the GPU's execution units, allowing them to perform more calculations per clock cycle. We unroll the inner loop, minimizing the times the code is executed.

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


    float Fx = 0.0f, Fy = 0.0f, Fz = 0.0f;
    float px = p.position[index].x;
    float py = p.position[index].y;
    float pz = p.position[index].z;

    __shared__ float4 position_s[MAX_THREADS_PER_BLOCK];

    for (int tile = 0; tile < gridDim.x; tile++) {
        __syncthreads();
        int tiledIndex = tile * blockDim.x + tx;
        position_s[tx] = p.position[tiledIndex];
        __syncthreads();

        #pragma unroll
        for (int j = 0; j < blockDim.x; j++) {
            float dx = position_s[j].x - px;
            float dy = position_s[j].y - py;
            float dz = position_s[j].z - pz;
            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;
        }
        __syncthreads();
    }

    p.velocity[index].x += dt * Fx;
    p.velocity[index].y += dt * Fy;
    p.velocity[index].z += dt * Fz;
}
```

Figure 17: Loop unrolling implementation.

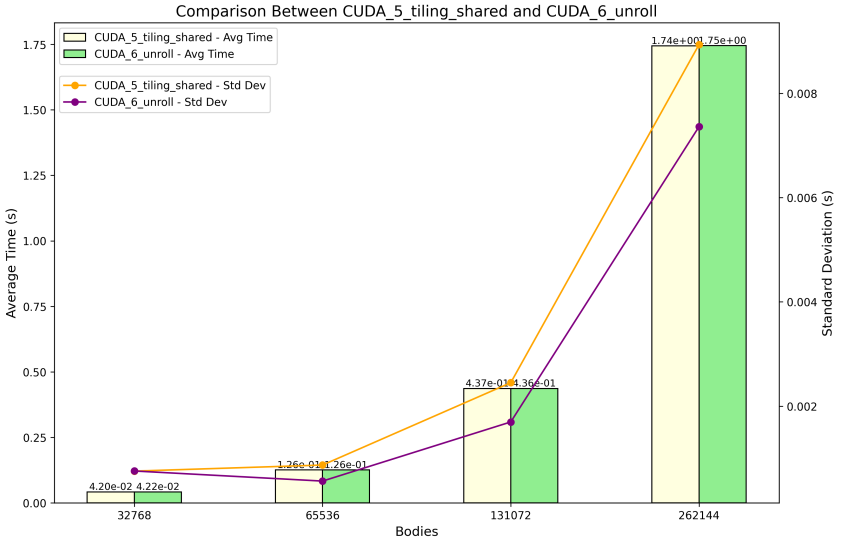


Figure 18: Comparison of the loop unrolling addition.

The unroll method brought an optimization of x1.002 times faster. This is because of the reduction in loop overhead and the improved instruction-level parallelism (ILP). By unrolling the loop, multiple iterations of the loop are executed in parallel within a single iteration, reducing the overhead of loop control (such as incrementing counters and checking conditions). This allows the processor to perform more computations with fewer instructions and control flow operations.

5.3.8 Pinned Memory

We decided not to use pinned memory to speedup the memory transfers from the CPU to the GPU and vice versa as not much time is spent on memcpys from and to the host. This is portrayed in the results we got from the nvidia profiler (-nvprof). Pinned memory will likely add an overhead in our program and slow down the execution.

```
==240090== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.58%  2.04064s        10  204.06ms  194.18ms  243.23ms  bodyForce(Body, float, int)
                    0.22%  4.4073ms        20  220.37us  200.64us  356.19us  [CUDA memcpy HtoD]
                    0.21%  4.2634ms        20  213.17us  199.62us  250.27us  [CUDA memcpy DtoH]
      API calls:   92.66%  2.05722s        40  51.430ms  172.63us  243.82ms  cudaMemcpy
                    7.10%  157.54ms         2  78.772ms  204.73us  157.34ms  cudaMalloc
                    0.10%  2.2380ms       404  5.5390us     122ns  510.53us  cuDeviceGetAttribute
                    0.09%  1.9149ms         4  478.72us  429.43us  595.32us  cuDeviceTotalMem
                    0.03%  591.50us         2  295.75us  280.46us  311.04us  cudaFree
                    0.01%  177.14us        10  17.713us  11.002us  39.556us  cudaLaunchKernel
                    0.01%  166.66us         4  41.666us  38.812us  44.286us  cuDeviceGetName
                    0.00%  101.07us        20  5.0530us  3.0070us  13.791us  cudaEventRecord
                    0.00%  73.092us        10  7.3090us  7.1430us  7.5600us  cudaEventSynchronize
                    0.00%  30.931us        10  3.0930us  2.6140us  5.2690us  cudaEventElapsedTime
                    0.00%  12.683us         4  3.1700us  1.4330us  7.4350us  cuDeviceGetPCIBusId
                    0.00%  3.9090us         2  1.9540us     699ns  3.2100us  cudaEventCreate
                    0.00%  2.8680us         3     956ns     404ns  1.9070us  cuDeviceGetCount
                    0.00%  2.3460us         8     293ns     163ns     651ns  cuDeviceGet
                    0.00%     959ns         4     239ns     190ns     320ns  cuDeviceGetUuid
```

Figure 19: -nvprof results for 131072 bodies with the unrolling code.

Pinned memory, while faster for data transfers, requires system resources to lock pages in physical memory, which can increase memory allocation time and potentially reduce the available memory for other processes. Because, our application is compute-bound, with a bottleneck in kernel execution rather than memory transfers, the use of pinned memory would not provide a significant performance benefit and probably lead to a slight slowdown due to, the additional management overhead. That is the reason why we decide to avoid pinned memory, for the particular occasion.

5.3.9 Flags & fast math

The -ftz=true flag in CUDA is used to flush denormalized numbers to zero, improving computational efficiency by eliminating the performance penalty associated with denormal (subnormal) numbers. Denormal numbers are very small floating-point values that require special handling, often involving slower arithmetic operations because they cannot be represented in the standard normalized form. When the -ftz flag is enabled, CUDA treats any denormal numbers as zero during computation, avoiding the overhead of these extra operations and maintaining higher performance.

The rsqrtf() function in CUDA, is a fast approximation of the reciprocal square root, , and is part of the CUDA fast math library. Unlike the standard sqrtf() function, rsqrtf() trades some precision for increased speed, making it ideal for applications that require numerous square root calculations but can tolerate minor inaccuracies.

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

    float Fx = 0.0f, Fy = 0.0f, Fz = 0.0f;
    float px = p.position[index].x;
    float py = p.position[index].y;
    float pz = p.position[index].z;

    __shared__ float4 position_s[MAX_THREADS_PER_BLOCK];

    for (int tile = 0; tile < gridDim.x; tile++) {
        __syncthreads();
        int tiledIndex = tile * blockDim.x + tx;
        position_s[tx] = p.position[tiledIndex];
        __syncthreads();

        #pragma unroll
        for (int j = 0; j < blockDim.x; j++) {
            float dx = position_s[j].x - px;
            float dy = position_s[j].y - py;
            float dz = position_s[j].z - pz;
            float distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
            float invDist = rsqrtf(distSqr);
            float invDist3 = invDist * invDist * invDist;
            Fx += dx * invDist3;
            Fy += dy * invDist3;
            Fz += dz * invDist3;
        }
        __syncthreads(); // Synchronize before next tile
    }

    p.velocity[index].x += dt * Fx;
    p.velocity[index].y += dt * Fy;
    p.velocity[index].z += dt * Fz;
}
```

Figure 20: Fast math sqrt function implementation.

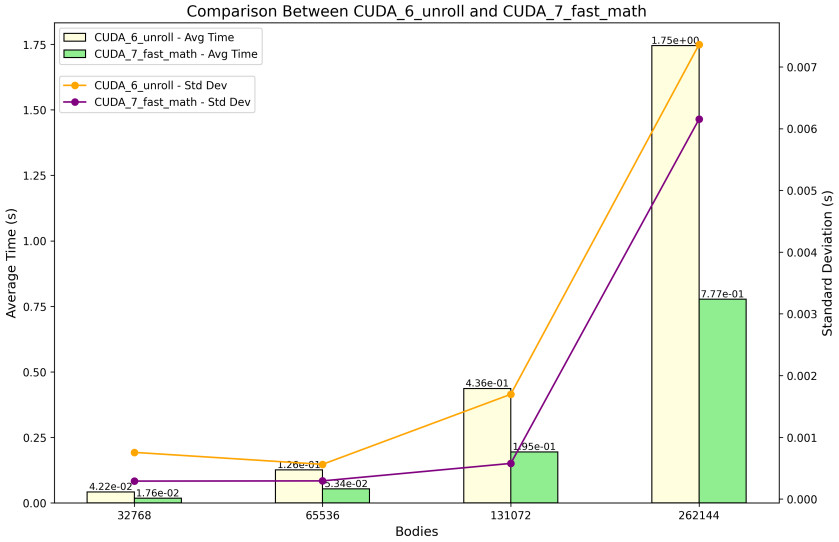


Figure 21: Comparison between unroll and optimized code with the flags for denormals and fast math

The usage of the flag plus the fast math calculations have a major speedup on our code by x2.236 times faster. This is because fast math optimizations enable the compiler to use less precise but much faster mathematical operations, for the square root. Also flushing the denormals to zero can shows on the speedup of our code with the flag (ftz=true).

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

  if (index >= n) return; // Guard against out-of-bounds

  float Fx = 0.0f, Fy = 0.0f, Fz = 0.0f;
  float px = p.x[index];
  float py = p.y[index];
  float pz = p.z[index];

  __shared__ float position_s[MAX_THREADS_PER_BLOCK][3];

  for (int tile = 0; tile < gridDim.x; tile++) {
      __syncthreads();
      int tiledIndex = tile * blockDim.x + tx;
      position_s[tx][0] = p.x[tiledIndex];
      position_s[tx][1] = p.y[tiledIndex];
      position_s[tx][2] = p.z[tiledIndex];
      __syncthreads();

      #pragma unroll
      for (int j = 0; j < blockDim.x; j++) {
          float dx = position_s[j][0] - px;
          float dy = position_s[j][1] - py;
          float dz = position_s[j][2] - pz;
          float distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
          float invDist = rsqrtf(distSqr);
          float invDist3 = invDist * invDist * invDist;
          Fx += dx * invDist3;
          Fy += dy * invDist3;
          Fz += dz * invDist3;
      }
      __syncthreads(); // Synchronize before next tile
  }

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

Figure 22: Final implementation of the code without float4.

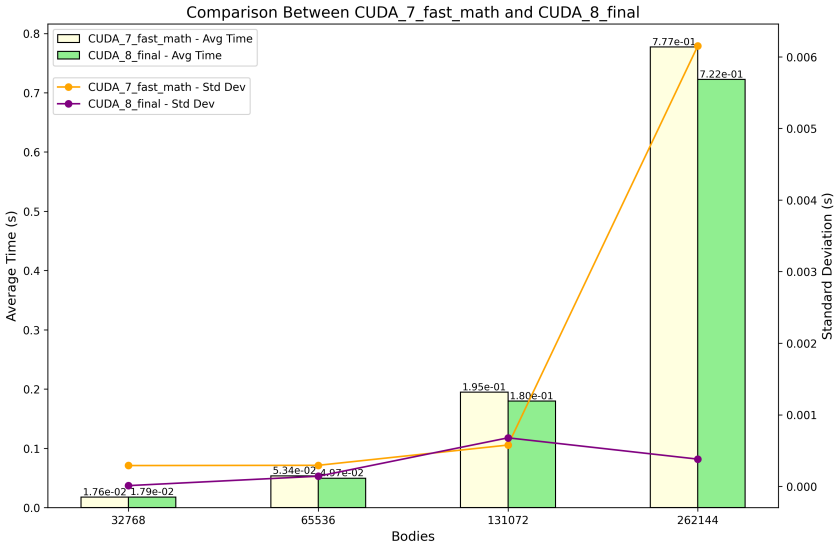


Figure 23: Comparison between float4 and float implementation.

Finally, we decided to try to switch back using floats instead of float4 types and we observed

that floats result in a speedup of x1.08 times. This can be attributed to the reduced memory usage and improved memory bandwidth utilization when using float

6 Conclusion

We observe that the optimizations that made the bigger difference where the omp and cuda baseline implementations which had a huge speedup. Where the best optimization on the cuda code was the -ftz=true flag to flush all denormals to zero as well as the fast math function for the square root.

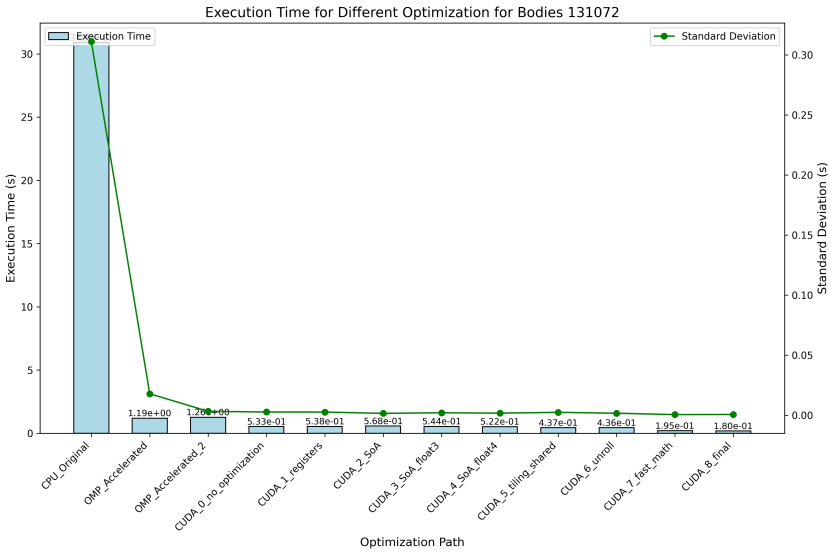


Figure 24: Comparison for 131072 bodies for all the different implementations

The total speedup for 131072 bodies is x171 times from the initial sequential code to the fully optimized code.