ECE1782 Project Report

GPU-Accelerated Simulation of Gravitational N-body Problem

Ray Huang (huangr19, 1001781565)
Lichen Liu (liuli15, 1001721498)
Sining Qin (qinsinin; 1001181106)
Haiqi Xu (xuhaiqi, 1001095076)

Table of Content

1. Problem Statement

1.1 Introduction

The N-body problem is about predicting the positions of n number of celestial bodies that are affected by the gravitational force from each other. Solving the N-body problem when n = 2 is straightforward, which is stated in the classical theory by Johann Bernoulli [1], with an assumption that the mass of all participating bodies is not subject to change. However, solving the N-body problem with n >= 3 is proven to be tricky. A quantitative approach to the generalized 3-body problem has not been found yet, as this problem can only be solved quantitatively under certain restricted conditions. Thus, predicting body states at a certain point of time remains a computationally heavy task, which requires the exact simulation of all states prior to this point of time.

As a simple visualization, Figure 1 shows two examples generated by our simulator when the number of bodies is small. The left picture is simulated on a solar system setup and the right picture simulates the moon-earth system.

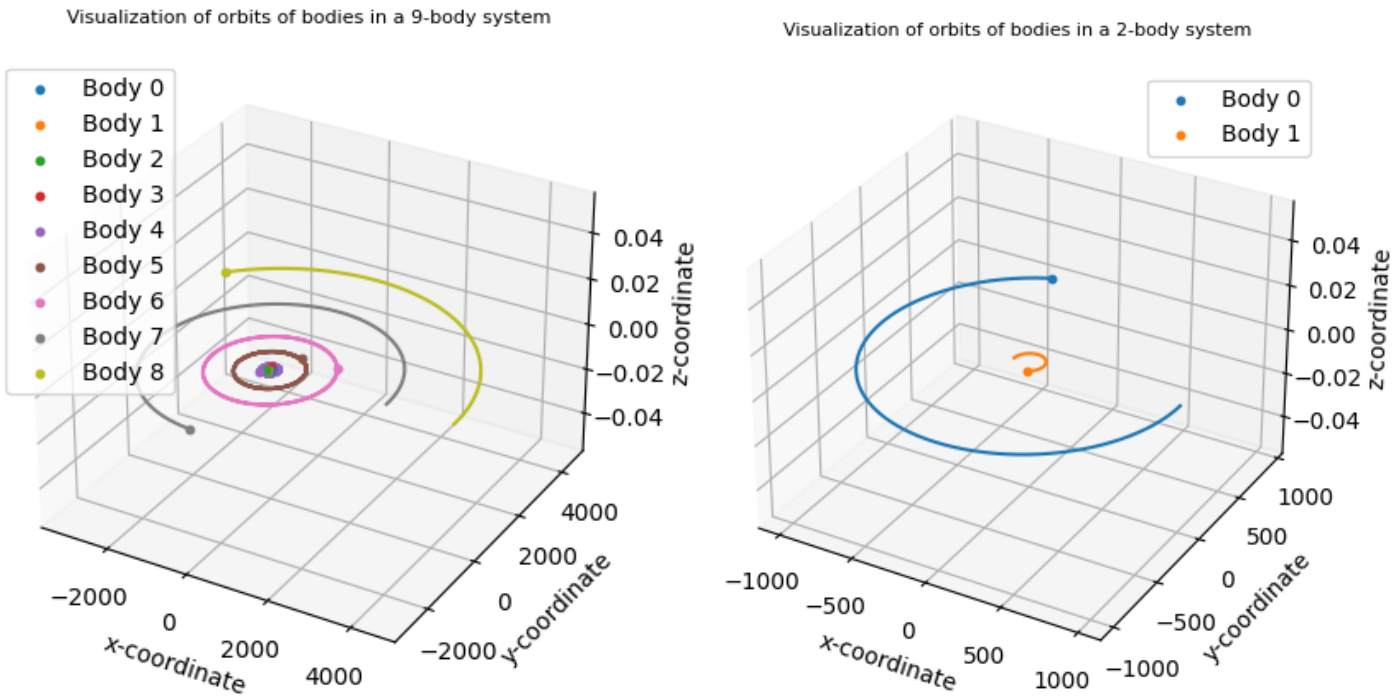


Figure 1. Visualization of a few simple N-body problem orbits generated by our simulator program. Left: orbits of 9 bodies in the solar system. Right: orbits of a double star system

However, the problem becomes more complicated and computational intensive as the number of bodies increases. To illustrate this, we used the sample data of a spiral galaxy for simulation purposes from[16] to simulate a galaxy with 134810 stars with our simulator and the

result is in Figure 2. When it comes to the scale of the universe, the number of bodies could be an order of magnitude larger than a simple solar system, and using CPU for simulation hence become inadequate. This motivates us to create a GPU implementation to speed up the simulation.

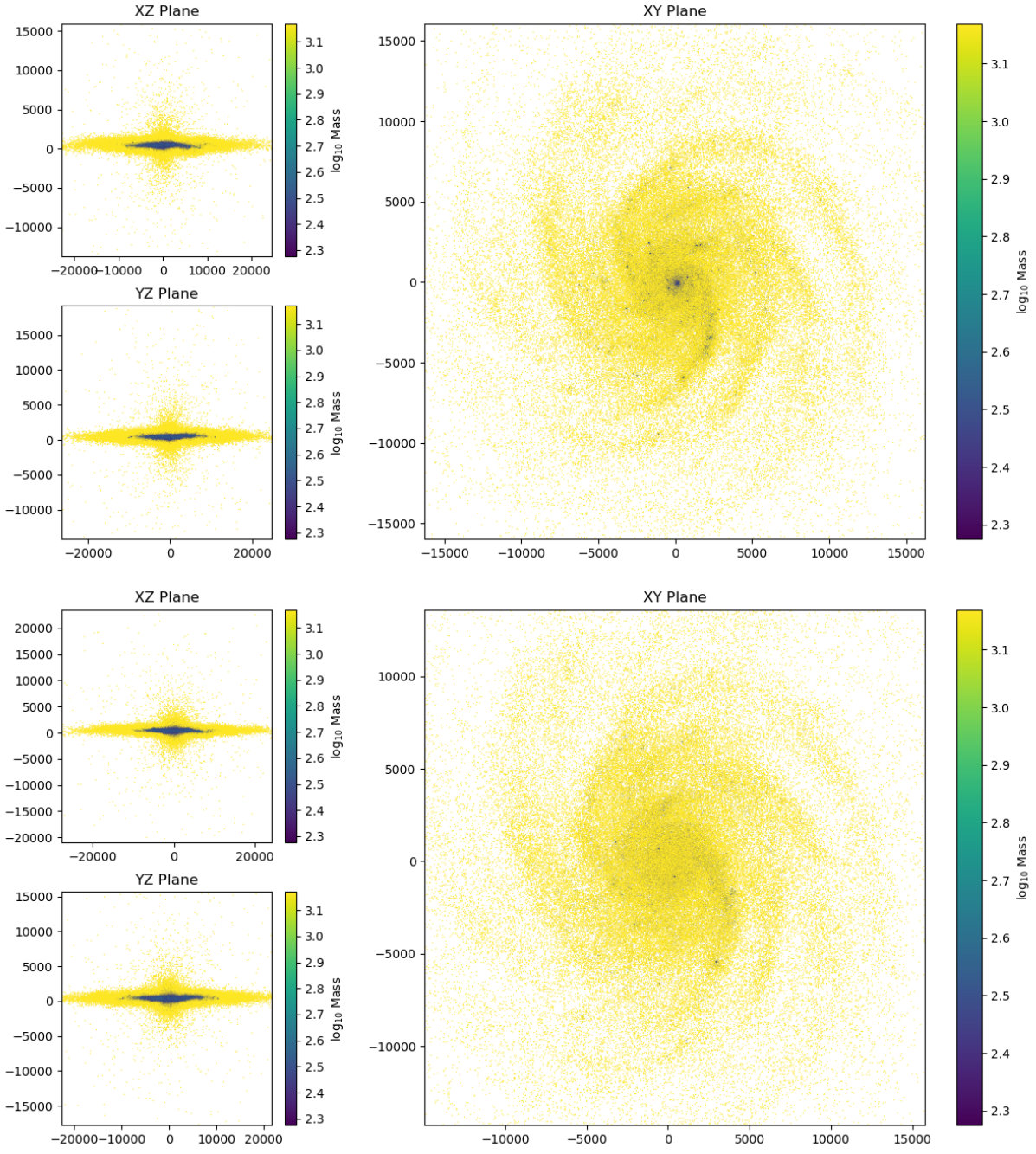


Figure 2. An example spiral galaxy with 138410 stars before (top) and after (bottom) 10000 iterations of (timestamps)

1.2 Problem

The purpose of this project is to design a GPU algorithm that accelerates the simulation of an N-body system where bodies are subject to gravitational effects from each other. We build on an existing solution [2] from Nvidia, aiming to further improve its scalability and performance. Our algorithm is evaluated with experiments performed on GeForce GTX980, an Nvidia GPU of the Maxwell architecture.

Given a system with N bodies, let two bodies from the system be represented by indices i and j, where $1 \le i \le N$, $1 \le j \le N$, and $i \ne j$. For a body i, let $\mathbf{x}_i$ denote its position, m_i denote its mass, $\mathbf{f}_{ij}$ denote the gravitational force applied on body i by body j, $\mathbf{F}_i$ denote the total gravitational force applied on i by all other bodies in the system, and $\mathbf{a}_i$ denote the acceleration on body i as an effect of $\mathbf{F}_i$.

To formulate the N-body system problem, the following equations are required:

$$F_i = \sum_{j=1, j\neq i}^{N} f_{ij} = \sum_{j=1, j\neq i}^{N} G\frac{m_i m_j}{|x_i - x_j|^2} \cdot \frac{(x_j - x_i)}{|(x_i - x_j)|} = Gm_i \sum_{j=1, j\neq i}^{N} \frac{m_j(x_j - x_i)}{|x_i - x_j|^3} m_i$$

$$\approx Gm_i \sum_{j=1}^{N} \frac{m_j(x_j - x_i)}{(|x_i - x_j|^2 + \epsilon^2)^{3/2}} \quad (1)$$

$$a_i = \frac{F_i}{m_i} \approx G \sum_{j=1}^{N} \frac{m_j(x_j - x_i)}{(|x_i - x_j|^2 + \epsilon^2)^{3/2}} \quad (2)$$

Here, ϵ^2 is the softening factor that precludes division by zero in computation (i.e., collision in physics) and limits the magnitude of the force between the bodies, both of which are desired in numerical astrophysical simulations.

Knowing the acceleration of a specific body at a time step t_k, we can calculate its velocity and position at the next time step t_{k+1} as below:

$$v_{k+\frac{1}{2}} = v_k + \frac{1}{2}a_k dt \quad (3)$$

$$x_{k+1} = x_k + v_k dt + \frac{1}{2}a_k dt^2 \quad (4)$$

$$v_{k+1} = v_{k+\frac{1}{2}} + \frac{1}{2}a_{k+1} dt \quad (5)$$

Here, $dt = t_{k+1} - t_k$, accelerations a_k and a_{k+1} are calculated using *Equation 2*.

An algorithm solving the N-body system includes the following steps:

a. Given the initial conditions of the N-body system, consisting of positions, velocity, and mass for each of the N bodies;
b. Compute the acceleration a_k for each of the N bodies using *Equation 2*;
c. Update the velocity of each of the N bodies at $t_{k+1/2}$ using *Equation 3*;
d. Update the position of each of the N bodies at t_{k+1} using *Equation 4*;
e. Compute the acceleration a_{k+1} for each of the N bodies using *Equation 2*;
f. Update the velocity of each of the N bodies at t_{k+1} using *Equation 5*;
g. Repeat *Step c* to *Step f* as k increments.

1.3 Motivation

The simulation of the N-body system when N is large is extremely computationally heavy – pairwise gravitational forces among all bodies need to be calculated, making the overall computational complexity $O(N^2)$.

At the same time, the N-body problem is highly parallelizable – states of all bodies can be updated in lockstep, and within each step, the gravitational force between a pair of two bodies is independent of those between all other pairs of bodies in the system.

A simple multithreaded reference algorithm can be used here to prove the concept. Let each thread be responsible for calculating the gravitational forces applied on a body by all other bodies during lockstep. This results in a global memory access pattern also in the order of $O(N^2)$, as each body needs to access the masses and positions of all other N-1 bodies.

Thus, it is clear that the N-body problem can be optimized by:

a. Exploiting the intrinsic parallelism of the problem by distributing the $O(N^2)$ calculations among the many cores of a GPU hardware; and
b. Improving data sharing among GPU threads to reduce time spent on data transfer.

Each optimization that we implemented will be discussed in detail in *Section 5. Evaluation*.

2. Related Work

We came across several previous papers that achieved acceleration of the N-body astrophysical simulation by taking advantage of its parallelizable nature. [3] exploited the power of loop unrolling, which is a common technique that reduces the loop processing time by reducing loop frequency, branch instructions, and conditional instructions (see details in *Section 4.2*). [2] implemented a novel approach termed tiling to achieve optimal data reuse among different GPU threads working on the same computational tile (see details in *Section 3.2*).

The limitation of [3] is that it only considered the loop unrolling technique for optimization and nothing beyond, but it served as a great learning material for us on the mathematics involved in the N-body astrophysical simulations. [2] presented more diverse optimization opportunities, but only performed experiments for the performance of a system with 16,384 bodies. This didn't seem to have fully demonstrated the capabilities of either the modern GPU hardware or the proposed optimization techniques.

After implementing and closely evaluating the existing algorithms, we believe there is still room for further performance improvements.

3. Solution Overview

Regardless of the various optimization techniques that we adopted in different solutions, the high-level host execution flow remains unchanged. The high-level host side pseudocode is as shown below:

(**Note**: On lines of kernel definition and invocation, the **bolded** parameters are read by the kernel, whereas the *italic* parameters are written by the kernel)

```
global void update_velocity(masses, input_intermedia_v, input_acceleration,
                            output_velocities) {...}
```

Code 1. Function declaration of the kernel that calculates and stores updated velocities

```
global void update_position(input_positions, input_velocities, input_accelerations,
                            output_positions, output_intermedia_v) {...}
```

Code 2. Function declaration of the khat calculates and stores updated positions

```
global void calculate_acceleration(masses, input_positions,
                                   output_accelerations) {...}
```

Code 3. Function declaration of the kernel that calculates and stores updated accelerations

```
positions_1[n];
velocities_1[n];
accelerations_1[n];

positions_2[n];
velocities_2[n];
accelerations_2[n];

masses[n];
intermedia_v[n];
```

Code 4. Variable declarations

```
calculate_forces(masses, positions_1, accelerations_1);
loop {
        update_position(positions_1, velocities_1, accelerations_1, positions_2, intermedia_v);
        cudaDeviceSynchronize();
        calculate_acceleration(masses, positions_1, accelerations_2);
        cudaDeviceSynchronize();
        update_velocity(masses, intermedia_v, accelerations_1, velocities_2);
        cudaDeviceSynchronize();

        swap(positions_1, positions_2);
        swap(velocities_1, velocities_2);
        swap(accelerations_1, accelerations_2);
}
```

Code 5. Main body that is executed repeatedly at every time step

Note that acceleration calculation is the major bottleneck of the total computation time, since it requires O(N) to compute the accumulated acceleration for one body, and the total computation required is $O(N^2)$. Hence, The focus of our optimization algorithm is to speed up the acceleration calculations.

3.1 Simple Pairwise Calculation

A simple approach for N-body simulation is to assign the acceleration calculation of each body to a single thread. Each thread sequentially reads the positions and masses of all other bodies to calculate the net acceleration resulting from the net gravitational force exerted on the body the thread is responsible for. Therefore with N threads taking care of N bodies, the total global memory access and computation during each time step is $O(N^2)$. For each time step t_k, the accelerations of all bodies are updated in lockstep by a single execution of the kernel.

To visualize the approach, imagine an N×N grid that represents the pairwise interaction between every pair of two bodies, where each element stores the gravitational force applied on the body represented by the row number by the body represented by the column number. For example, the element at (0, 2) of the matrix represents the force applied on body 0 by body 2. Each thread takes care of exactly one row, by sequentially reading in the data of all other bodies, as shown in Figure 3:

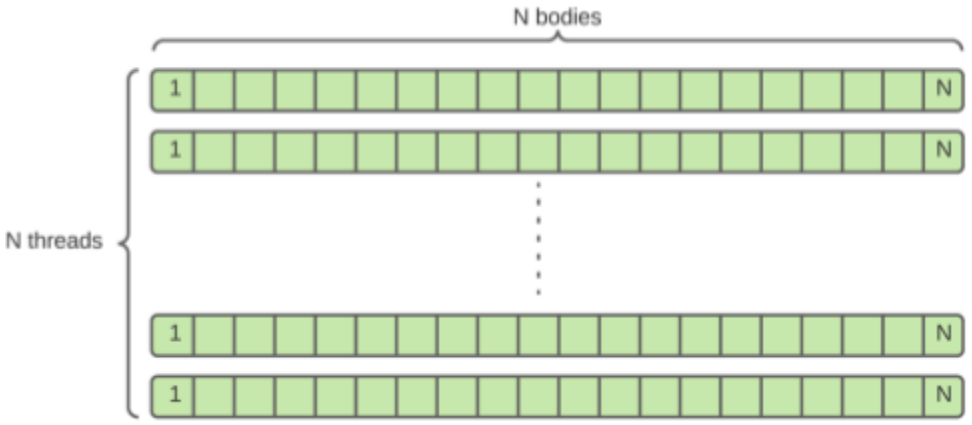


Figure 3. Illustration for the **Simple Pairwise Calculation** approach

The drawback of this **Simple Pairwise Calculation** approach is the lack of memory reuse – each thread has to separately read the positions and masses of all other bodies from the global memory. However, as shown in *Equation 2*, for any body i in a time step, the positions and masses of all bodies are read. Since in this **Simple Pairwise Calculation** approach each thread is in charge of calculations for body i, all positions and masses are read separately from global memory by individual threads without careful reuse among them. Therefore, this opens up opportunities for performance improvements through higher memory reuse, which can greatly reduce runtime due to reduced total global memory access and is discussed in *Section 3.2 and Section 3.3*.

3.2 Nvidia Tiling Approach

To increase data reuse, Nvidia introduced a novel approach of tiling in [2]. Since our final solution leverages this idea and its performance is eventually compared with this, we present a summary of the **Nvidia Tiling** approach below.

To avoid having individual threads access the data of all bodies from the global memory, Nvidia introduced the concept of computational tiles, as shown in blue blocks in Figure 4 and 5. Going back to the aforementioned N×N grid representing pairwise forces between all pairs of

bodies, let a tile be of size p×p, representing a computation sub-region of pairwise forces between p×p pairs of bodies.

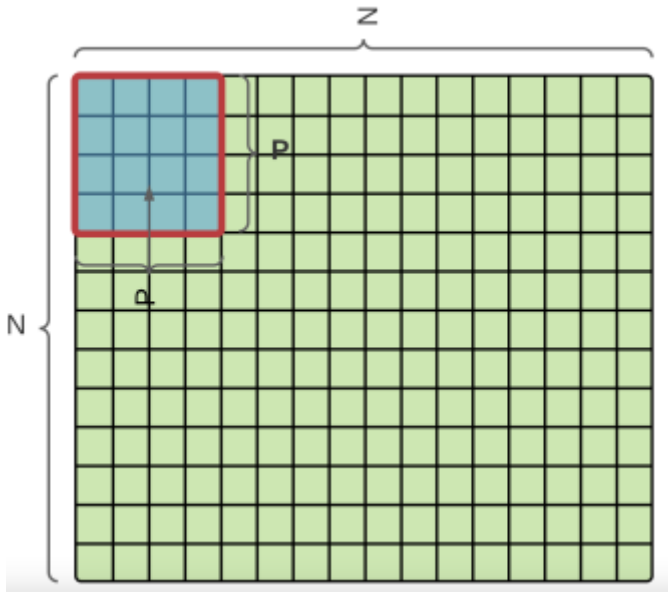


Figure 4. N×N pairwise force grid and a p×p tile (in blue)

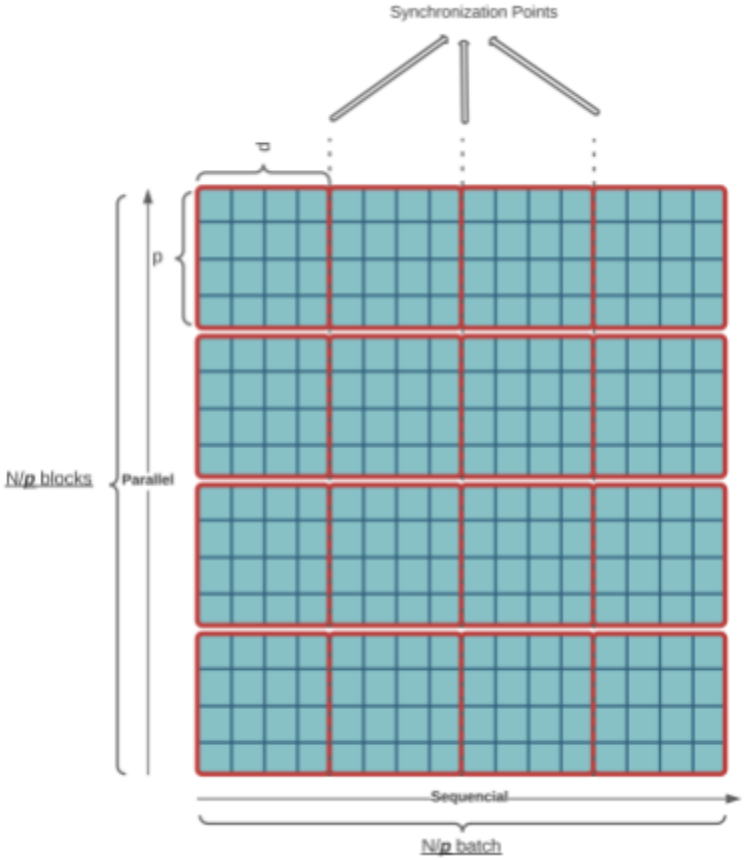


Figure 5. Tile scheduling where the horizontal direction represents sequential synchronization steps and the vertical direction represents parallelism

Same as the **Simple Pairwise Calculation** approach in *Section 3.1*, a separate thread is responsible for calculating the net acceleration for each body, requiring a total of N threads for the kernel. These N threads are divided into batches of size p, with each batch processed by a separate GPU block, which results in a total of N/p blocks required to be allocated for the kernel. Instead of processing other bodies all at once, the acceleration calculations for each thread are broken down into sequential steps, each also handling p bodies, which results in a total of N/p sequential steps in each time step.

A tile corresponds to a block in a sequential step. Tiles in the same row are handled by the same GPU block sequentially. In each sequential step, threads within the same block load the positions and masses of the same p bodies from the global memory into the shared memory of their block. These threads essentially reused data of the same p bodies when calculating accelerations, so that those data only need to be accessed in global memory and transferred to the local memory once per tile. This reduces the total global memory access from $O(N^2)$ to $O(N^2/p)$.

Note that synchronization is always required after the completion of one tile, as the states of the next p bodies have to be loaded into the shared memory altogether.

3.3 Our Approach: Improved 2D Tiling

After implementing and experimenting with the **Nvidia Tiling** approach, we confirmed that it greatly reduces the total memory access and in turn, increases the memory throughput. However, we also noticed new aspects of the original **Nvidia Tiling** approach where we can improve.

Firstly, the Nvidia benchmark only tested up to 16,384 bodies. On the GPU model (Nvidia GeForce GTX980) we use for experiments, this number is not significantly greater than the total number of cores available, which is 2048. Because each body is handled by a single thread, when the number of bodies is small, there is a chance where all threads are waiting on memory operations and cores become idle.

Secondly, the hardware (Nvidia G80 architecture) used for the Nvidia paper supports concurrent reads from multiple threads to a single shared memory location, which means there was no need to consider bank conflicts in shared memory. However, bank conflicts do happen for the hardware (Nvidia GeForce GTX980) we use and thus should be taken into consideration in our algorithm.

Thirdly, due to the necessary synchronization after each tile, it is possible that a straggler thread may slow down the overall kernel execution.

Lastly, the size of shared memory is restricted by the block size and it is not uncommon for shared memory to be not fully used. For example, let a block size be 512, then only states of 512 bodies should be stored in the shared memory. Each state consists of mass and position (3D), both in float, taking 4×4=16 bytes; only 16×512=8192 bytes of shared memory are required per block. The total amount of shared memory required by all blocks could be lower than the total amount of shared memory available on the hardware.

As a result, we came up with a **2D Tiling** solution aiming to address the aspects mentioned above, where “2D” is referring to the usage of 2D thread blocks in a 2D grid, in comparison with the 1D thread blocks in a 1D grid adopted by Nvidia.

At the core of this 2D solution, with 2D thread blocks in a 2D grid, the acceleration calculations for each body are now distributed among multiple threads rather than handled by a single thread. These multiple threads produce intermediate results that need to be summed up to

make the final acceleration results. The 2D thread blocks in a 2D grid split computation work and reuse data better among the parallel resources available on GPU.

This **2D Tiling** approach consists of a series of optimizations and improved algorithm designs, whose details are covered in *Section 4. GPU Optimizations* and performance results are summarized in *Section 5. Evaluation*.

4. GPU Optimizations

In this section, we are covering the following optimization ideas in order:

1. **2D Tiling** and intermediate result accumulation;
2. Loop unrolling;
3. Varying block size;
4. Register usage reduction;
5. Use rsqrt;
6. Pack 3D position and mass into float4;
7. Avoidance of bank conflicts.

4.1 2D Tiling and Intermediate Result Accumulation

4.1.1 2D Tiling

A thread block has p threads that execute tiles in sequence. Tiles are of size p × q, where p is the number of rows, which must be sufficiently large so that multiple warps can be interleaved to hide latencies in accumulation calculations, and q is the number of columns, which reflects the amount of data reuse and governs the size of the transfer of bodies from device memory into shared memory. This size also determines the register space and shared memory required.

The **Nvidia Tiling** approach minimizes the number of global memory read access, however, each batch processing is still sequential, as shown in Figure 5. For the kernel to start executing the next batch, the acceleration for a body induced from all other bodies in the current batch has to be accumulated to the updated acceleration variable first, then the kernel can safely proceed to the next batch. If we have a lot more bodies than the cores on the GPU, this sequential approach would not be a bad idea. However, when the number of bodies is only greater than the number of cores by a small factor, for example, when running 16,384 bodies in Nvidia's benchmark, we believe that the GPU resources could be underutilized. Underutilization can also be a problem for the **Simple Pairwise Calculation** approach (in *Section 3.1*).

In the **Nvidia Tiling** solution, for example, a thread would need to iterate through a total of 10000 (N) bodies by going through N/p batches sequentially. These N/p batches could potentially be processed concurrently as well. To do that, we add one more dimension to the thread block, making it a 2D thread block. Therefore now there are in total x_total × y_total threads distributed into block each with a size of blockDim.x × blockDim.y. Then we introduce a new variable named loop unrolling factor (luf), where luf represents how many bodies a thread needs to perform summation on and luf × x_total = N**.**

Each block will initialize a shared memory of size luf × blockDim.x, which comprises states of all bodies that will be read by threads in this block, as shown in Figure 6:

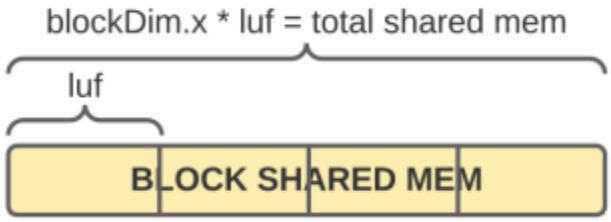


Figure 6. Shared memory of a single block

Within each block, threads that have the same threadIdx.y will work together to calculate the updated acceleration of the same body, in a way that each thread calculates a partial acceleration result. Each of these threads produces partial results based on subsets of other bodies, and the size of a subset is determined by luf. Later those partial results will be aggregated by a separate kernel. As a result, for a single body, there will be blockDim.x number of partial acceleration results generated, as shown in Figure 7:

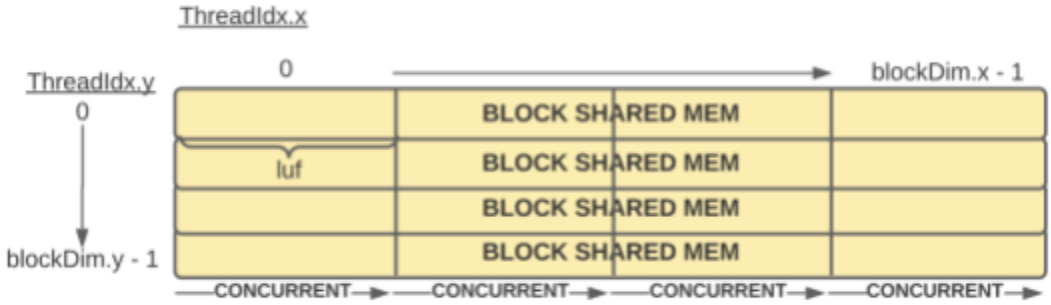


Figure 7. Calculation splitting and use of shared mem within a block

We want to find the updated acceleration of all N bodies, so we also have to make sure y_total is equal to N. As mentioned before, luf × x_total = N, so the overall kernel execution flow can be visualized as in Figure 8:

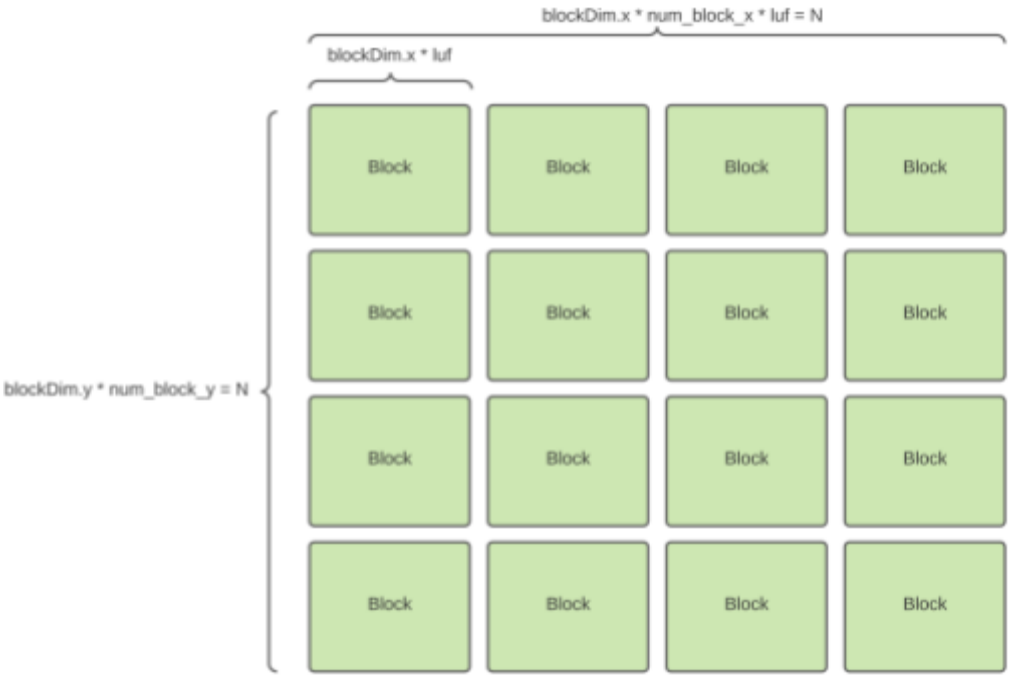


Figure 8. N to N computation distribution among all thread blocks

Note when N is small, all the blocks can be executed at the same time. In contrast, the **Nvidia Tiling** approach will have to accumulate the acceleration of each body sequentially, since the updated acceleration of each body is strictly calculated by a single thread.

4.1.2 Result Accumulation

A trade-off of our approach is that each block will generate an intermediate result, and at the end, for each body, there will be N/luf partial results generated. Hence in total, there will be a total of N×N/luf intermediate results. We have to introduce an extra kernel that sums all the partial results for each body. The extra computation time resulting from this extra kernel is small but not completely negligible, and we have experimented with different accumulation algorithms as elaborated below.

4.1.2.1 Simple Accumulation Approach

We first implemented a **Simple Accumulation** kernel. The kernel creates N threads and each thread sums across the N/luf results. The total runtime takes around 4.2% of the total kernel runtime, confirmed using nvprof (see *Appendix 1 - Figure 1*).

4.1.2.2. cuBLAS Matrix Multiplication Approach

To optimize the accumulation flow, we investigated and found a faster approach making use of cuBLAS matrix multiplication for row reduction [4][5], where cuBLAS stands for the CUDA Basic Linear Algebra Subroutine library. The idea is that summing up rows in the intermediate results is the same as doing a row reduction for a matrix. To row-reduce a matrix of size N×M, we simply need to multiply it with a matrix of M×1 with all 1's.

However, this **cuBLAS matrix multiplication** approach is not directly applicable to our code due to the memory layout. Recall that the problem is in 3D, so the partial results are in the form of vectors. The most intuitive way to store them in the memory will result in the memory layout shown in Figure 9:

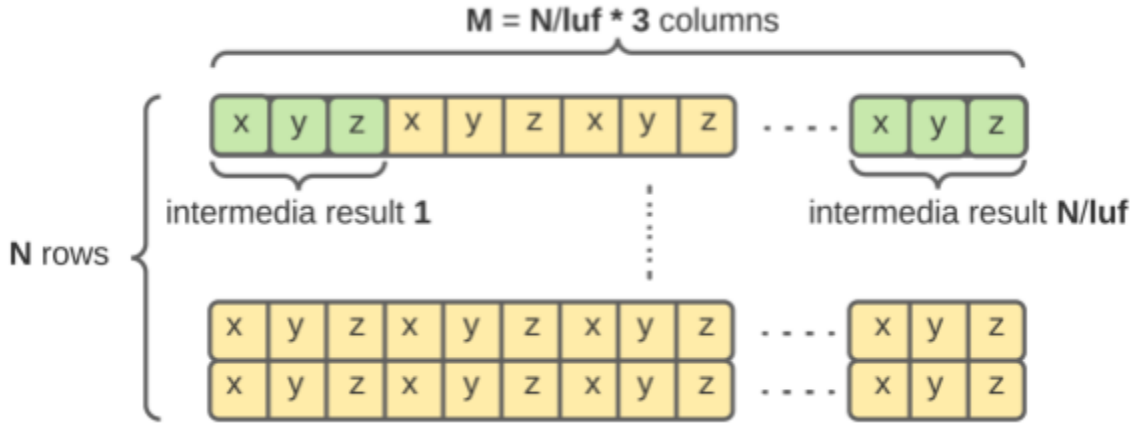


Figure 9. Original memory layout of intermediate results for all N bodies

Since the acceleration needs to be accumulated on x, y, and z dimensions respectively. We can't apply the row reduction directly. To solve this problem, we tweak the way we store them in the memory as shown in the picture below. The intermediate results of the same dimension will be stored in one row so we can use matrix multiplication to do the row reduction. The result will be a $3N \times 1$ matrix but with float3 it can be represented by a matrix of size N. This manipulation is very similar to matrix transpose, as shown in Figure 10.

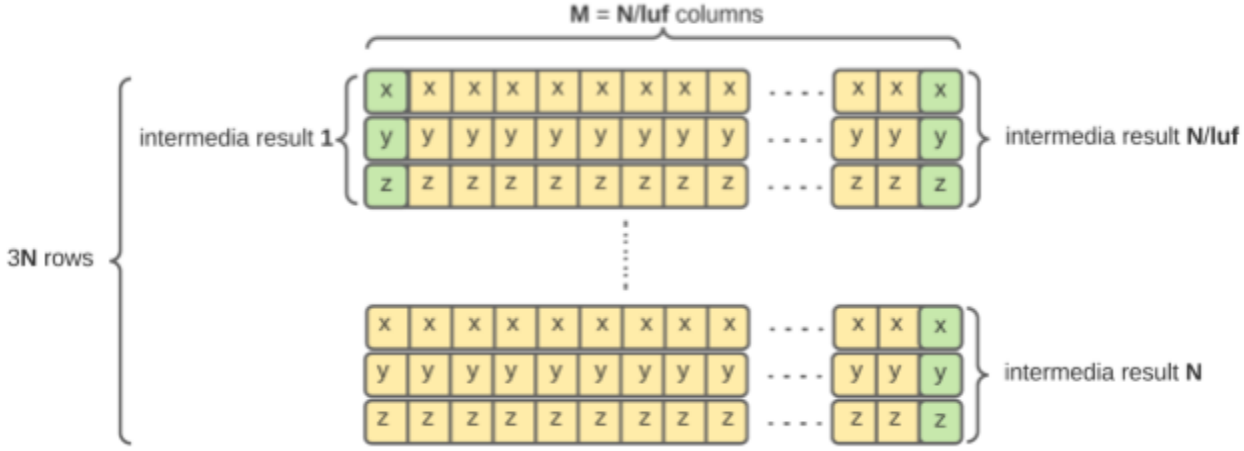


Figure 10. Transposed memory layout of intermediate results for all N bodies

With the approach above, we were able to further cut down the runtime of accumulation to 1.25% of the total kernel runtime (see *Appendix 1 - Figure 1*).

In summary, the **2D Tiling** approach has several advantages over the **Nvidia Tiling** approach. First, it achieves better core utilization when the number of bodies is not significantly greater than the total cores available on the GPU. Secondly, a straggler thread is less likely to slow down a lockstep update, because the total workload of any thread is reduced to calculate only partial, but not the entire accumulated acceleration of a body. Lastly, our approach requires only 50% of the_sync_thread() call in the acceleration calculation kernel. The partial

acceleration of a body is calculated in sequence, which is formulated as a for loop in the kernel code. After each iteration, a _sync_thread() call is required to make sure shared memory is only updated after accumulated acceleration from bodies that are currently stored is calculated. This is not a problem in the **2D Tiling** approach since shared memory is not repopulated in the kernel.

4.2 Loop Unrolling

Loop unrolling is a common approach suggested by different papers [2][3]. It still turned out to be very useful for our **2D Tiling**. Unrolling is tightly coupled with our **2D Tiling** approach. Apart from all the benefits already mentioned in *Section 4.1 2D Tiling and Result Accumulation*, it also allows us to flexibly decide on the size of shared memory size. In Nvidia's tiling approach, the size of shared memory is strictly proportional to p, which is also the block size. This can lead to shared memory underutilization. In our approach, shared memory size is calculated as blockDim.x × luf, which is controllable by luf.

For our implementation, we first implemented 2D loop unrolling without shared memory, then added shared memory on top of it to see how performance differs. Compared against the **Simple Pairwise Calculation** approach, 2D loop unrolling achieved more than **2.51x** speedup, and considerations in shared memory in tiling further increased the speedup by **1.67x** (see *Section 5.4*).

4.3 Varying Block Size

The **2D Tiling** approach has 3 parameters. Blockdim.x, blockdim.y and luf (loop unroll factor). While we tried varying the parameters to see how performance changes, there are several things we had to be aware of. First, larger blockdim.y is preferred since it allows more bodies in a tile to reuse shared memory. Second, we found out that the **2D Tiling** could overuse shared memory which results in a silent crash in the GPU. Hence we always limit each block to use only 16,384 bytes of the shared memory, by ensuring that Blockdim.x × luf ≤ 1024 (assuming each thread reads 16 bytes for 3D position and mass). Lastly, for simplicity in the implementation, we always require those numbers to be a power of 2, and that luf > blockdim.y.

With these things in mind, we sweep through different configurations. Note that, for simplicity, we only report the data such that:

1. Number of bodies = 100000, number of iterations = 20; and
2. Have a larger blockdim.x × blockdim.y × luf value which guarantees a larger tile size.

With the data we collected in *Appendix 2 - Table 1*, it turned out that when blockdim.x = 1, blockdim.y=512, luf = 1024, the kernel has the best performance.

4.4 Register Usage Reduction

In the initial implementation of the tiling approach, each thread needs 36 registers. With the help of CUDA Occupancy Calculator [6], it turned out that this limits the maximum core utilization to be only 75% percent of total cores on GPU. With various experiments, we successfully reduce the occupied registers per thread to 32, which is expected to give us 100% core utilization. This was achieved by using the *volatile* keyword for the loop counter variable when populating the shared memory.

However, as shown in Table 2 in *Section 5.4*, this didn't have any visible impact on performance. By the time of this report, we still couldn't figure out the reason behind this. One possible reason is that using the volatile keyword forces the CUDA program to read from the memory, whose latency cancels out the benefit of better core utilization.

4.5 Use rsqrt

As part of the acceleration calculation, the program needs to calculate the reverse sqrt. At the beginning, we implemented it as **1.0f / sqrt(x)**. However, later investigation (the Arithmetic Instructions section in [7]) shows that **sqrt(x)** in CUDA is actually implemented as **1.0f / rsqrt(x)**. Hence the same calculation can be achieved by simply using **rsqrt(x)** and it saves two reciprocal (i.e. division) operations.

With a rough estimation that one reciprocal operation takes 4 flops [7], using rsqrt reduces the required flops from 27 to 19 for calculating pairwise interaction, which is on the critical path of our code. This significantly improved our kernel's performance by more than **1.28x** (see *Section 5.4* for details).

4.6 Packing Position and Mass as float4

As the N-body simulation is a 3D problem, we represent the accelerations and positions of bodies using a 3D vector, which is implemented by CUDA's float3 type.

The idea is leveraged from Nvidia's reference paper [2]. Instead of storing position and mass as separate arrays in the memory, we store them as a single array of float4: {pos.x, pos.y, pos.z, mass}. With this method, each body's state requires a 4×4 bytes per read or store on a continuous memory region, instead of two separate reads with 12 bytes for position and 4 bytes for mass. This not only makes implementation much easier with shared memory but also makes it much easier to reason about access patterns and bank conflict. We don't have a comparison of how this approach speeds up our **2D Tiling** version, simply because the slower version (separating mass and positions) was too hard to implement for shared memory and we decided it isn't worth the time to just implement it for comparison purposes.

4.7 Avoidance of Bank Conflict

4.7.1 Shared Memory Writing

Recall that in each block, there are **blockdim.x** $\times$ **blockdim.y** threads in total, but a total number of **blockdim.x** $\times$ **luf** body states need to be read into the shared memory. Hence each thread will need to read in a number of **luf / blockdim.y** body states. In our initial **2D Tiling** implementation, we used a simple method where each thread writes to a continuous region of the memory, but this causes banking conflict in a warp, which is illustrated in the following picture. For simplicity, we use **luf = 1024, blockdim.y = 256**, so each thread writes the statuses of 4 bodies into the memory, where each status consumes 4 banks (3 for 3D position, 1 for mass).

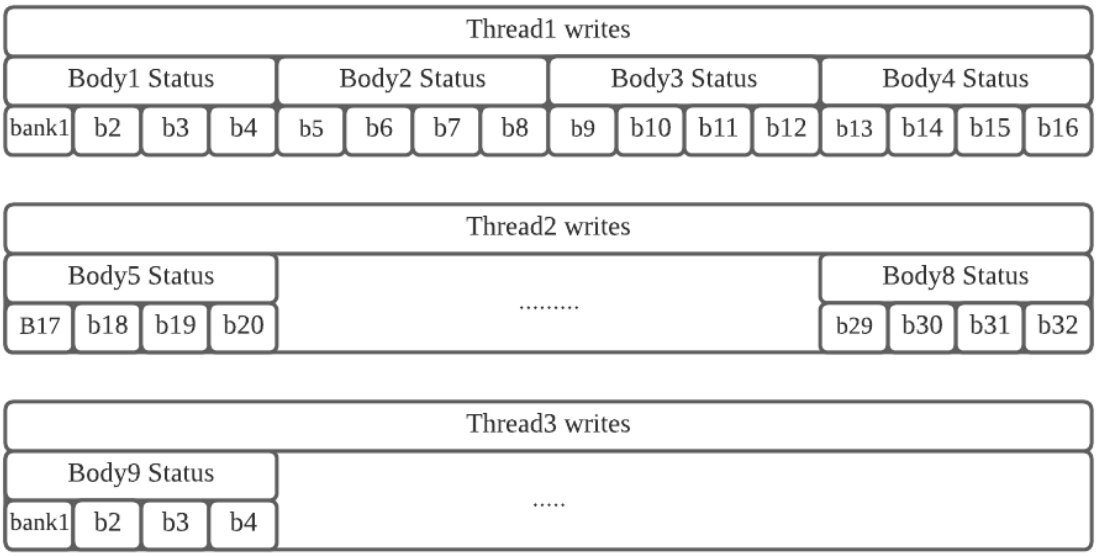


Figure 11. Bank access pattern when luf = 1024 and blockdim.y = 256

In Figure 11, it can be seen that body thread 1 and thread 3 will access bank 1 at the same time, causing a bank conflict. To fix this issue. We adjusted the write access pattern so the thread writes to the shared memory in a stride of 32 as illustrated in Figure 12. Although there is still bank conflict between thread 1 and thread 9, we do reduce the banking conflict on each bank from 15 to 3 in a warp.

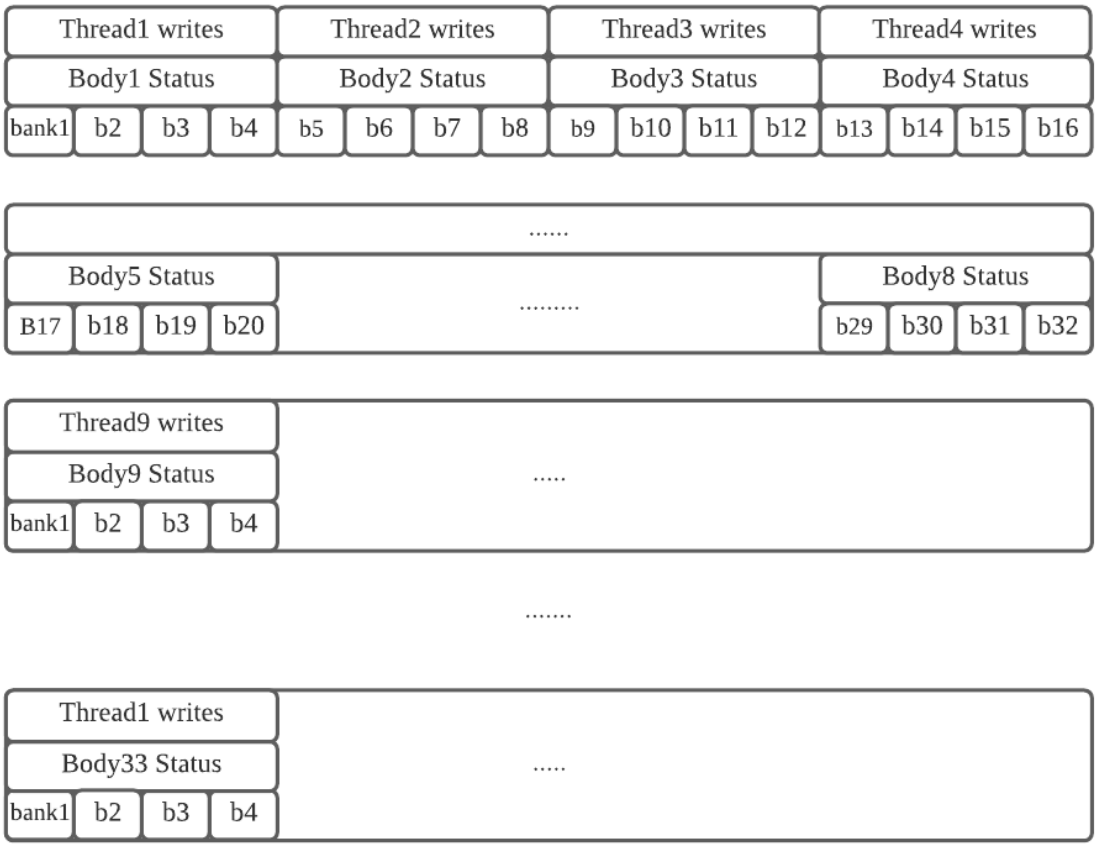


Figure 12. Adjusted bank access pattern when luf = 1024 and blockdim.y = 256

4.7.2 Shared Memory Reading

When Blockdim.x > 1, more than one thread with the same y dimension ID will be reading from the shared memory to calculate different parts of acceleration for one body. In the initial implementation. Each thread will be reading a continuous memory region starting from their offset. Figure 13 below illustrates an example when Blockdim.x = 4.

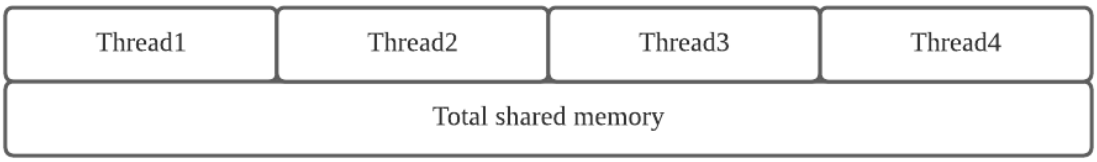


Figure 13. Shared memory access pattern when Blockdim.x = 4

When each thread needs to handle more than 8 bodies (which is $8 \times 4 = 32$ banks), all the threads will have a bank conflict when they read from the first body status from the offset.

The solution to this problem is to rearrange the read pattern so that each thread reads in a striding pattern, as shown in Figure 14.

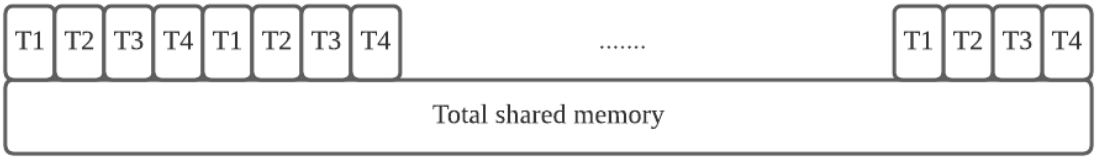


Figure 14. Shared memory access pattern with striding

This guarantees that memory access is free from bank conflict as long as the number of threads on the x dimension < 8.

We evaluated the impact of the above two changes together on blockdim.x = 4 and blockdim.x = 1 respectively and the result can be seen in *Section 5.4* and *Appendix 2 - Table 2*. When blockdim.x = 4, the optimization has a significant improvement of **2.7x**. However, when the blockdim.x=1, the change in the performance is negligible. We believe that this is because the bottleneck of the kernel is shared memory reading, but blockdim.x = 1 configuration only gets an improvement in the shared memory writing which is only a small portion of the workload. Because our best configuration uses blockdim.x = 1, the optimization doesn't have an impact on our final runtime. Nevertheless, the optimization itself was in the correct direction.

5. Evaluation

We report the following performance comparisons, using raw runtime data summarized in *Appendix 2*.

1. Performance improvements between different stages of our **2D Tiling** implementation for GPU;
2. Performance improvements between the **Simple Pairwise Calculation** approach and different stages of our **2D Tiling** implementation for GPU;

3. Performance improvements between the **Nvidia Tiling** approach and different stages of our **2D Tiling** implementation for GPU;
4. Performance improvements between our multithreaded CPU implementation and the final version of our **2D Tiling** implementation on GPU.

5.1 Experiment setup

5.1.1 GPU Hardware

The GPU hardware we have for experiments is from Nvidia, of model GeForce GTX980 of the Maxwell architecture. According to its specifications [8] from the Nvidia website, this model has 16 streaming multiprocessors (SMs), a total of 2048 CUDA cores and a frequency of at least 1064 MHz. In terms of memory, it has a memory speed of 7.0 Gbps, a GDDR5 memory interface, and a memory bandwidth of 224GB/sec; it has up to 96KB shared memory and up to 256KB register file per SM [9]. The maximum number of active thread blocks per SM is 32, the maximum number of concurrent warps per SM is 64, the maximum number of registers per thread is 255, and the maximum number of threads per thread block is 1024 [10].

5.1.2 Initial Conditions

Each timestamp in the N-body problem can be described by a collection of constant mass, position, and velocity of each body. We refer to the tuple of mass, position, and velocity of each body at a particular timestamp (snapshot) as body state, and a collection of these body states across all bodies as system state. Therefore, initial conditions are simply the system state at timestamp 0 and are the input into the N-body problem or our program. We have implemented our serialization formats for system states, CSV for readability and BIN for scalability of file size and deserialization speed over a large number of bodies. Our programs support both formats, and scripts are provided to convert between them.

For functional verification purposes, we have handwritten some simple systems involving 2-10 bodies, for example, various kinds of 2-body systems (such as the Sun-Earth system and a system with stars orbiting a common barycenter), Sun-Earth-Moon system, and the solar system.

For performance benchmarking purposes (and for real astrophysics simulation), we use online datasets of celestial bodies [11]. There are various kinds of models, with varying sizes of the system, and different numbers of galaxies. Online models are usually in Tipsy format, which is a popular format for astrophysics simulation, which encapsulates a lot of data such as body type (gas, star, and dark matter), body states, and units. We provide a script to convert from Tipsy format to our serialization format, which will also convert the units to fit our simulation (forcing gravitational constant G to 1), an example schema is to let velocity be in km/s, distance be in ps, mass be in Msun and multiplied by 4.3009e-3 (G in solar mass parsec kmps).

For our benchmarking, we are using the LOW and MED initial conditions from AGORA [12], which models a disk-like galaxy. LOW contains 112,500 stars (and also 100,000 dark matter and 100,000 gas, which are filtered out); while MED contains 1,125,000 stars and 1,000,000 dark matter (and 1,000,000 gas which are unused).

5.1.3 Accuracy and Correctness Check

One of the biggest challenges of an N-body problem is that small turbulence in the initial condition can easily develop to entirely different simulation results since the system is considered chaotic for most initial conditions [13]. A chaotic system is a complex dynamic system that is super sensitive to any state, in a sense that even tiny changes to it can result in huge differences in a later state [14]. Thus a tiny difference in the body state (velocity, positions) due to floating-point precision eventually will make the CPU and GPU simulation diverge. The discrepancy in floating-point precision between the two implementations can derive from multiple sources such as the order of internal algorithmic operation and hardware. Some of our optimization including fast square root also inevitably produce less accurate results. Initially, this effect seems neglectable, however as we increase the simulation steps and the number of participating bodies, this impact becomes much more severe.

To make sure the output produced by our system makes sense, we introduce an error measurement based on the aggregated square distance difference between the reference CPU solution and the GPU solution. The simulation result is also validated visually by plotting the system states at different timestamps so that we can further make sure the system is evolving correctly. Due to the complexity of the problem when there is a large number of bodies, we rely on handcrafted benchmarks such as the solar system and two-body system shown in figure1 for our validation. Since those smaller systems are easier to reason about and manually calculate the expected results.

5.2 CPU Reference Implementation

The performance of GPU implementation is measured against multithreaded CPU implementation in terms of percentage decrease of total processing time. We designed metrics to cross-compare performance boost/decrease among all our implemented optimization. Two CPU reference versions are implemented, while the **Basic Implementation** provides a functional reference, the **Edge-Sharing Acceleration Calculation Implementation** gives the best performance in conjunction with multithreading.

5.2.1 CPU Basic Implementation

This version is the most basic and trivial of the algorithm, i.e., it simply follows the pseudo code provided in [2]. For the acceleration part, it is essentially a double for-loop doing n^2 pairwise gravity acceleration (i.e., n^2 edges). The outer for loop visits each **target** body, and the inner for loop accumulates the gravity acceleration vector resulting from each **source** body.

As an optional and trivial optimization, a simple std::thread multithreading version is implemented on top of this implementation, where the parallelism comes from distributing the outer **target** body for loop to each thread. The threads are single-use and are joined and destructed later, meaning each simulation iteration (timestep) will allocate these threads. Quite obviously, the performance suffers from the thread creation overheads when the size of the input is small and the number of simulation iterations (timesteps) is large. To address this, a simple thread pool is implemented, where the worker threads wait for tasks from its task queue.

5.2.2 CPU Edge-Sharing Acceleration Calculation Implementation

It is worth noting that gravitational force between two bodies is exerted on both bodies, same in magnitude but opposite in direction. This gives an opportunity for optimization by reducing the number of acceleration vector calculations between two bodies (a pair) by half. In other words, after calculating body A's gravitational acceleration due to B, B's gravitational acceleration due to A can also be derived by simply reversing the direction! The inner for loop only needs to traverse and accumulate the acceleration of each **source** body that has an index larger than the current **target** body from the outer for loop, because the other acceleration has already been calculated and accumulated by previous **target** bodies (i.e., previous outer iteration). This reduces n^2 acceleration calculation pairwise edges down to $n^2/2$, which is demonstrated in Figure 15.

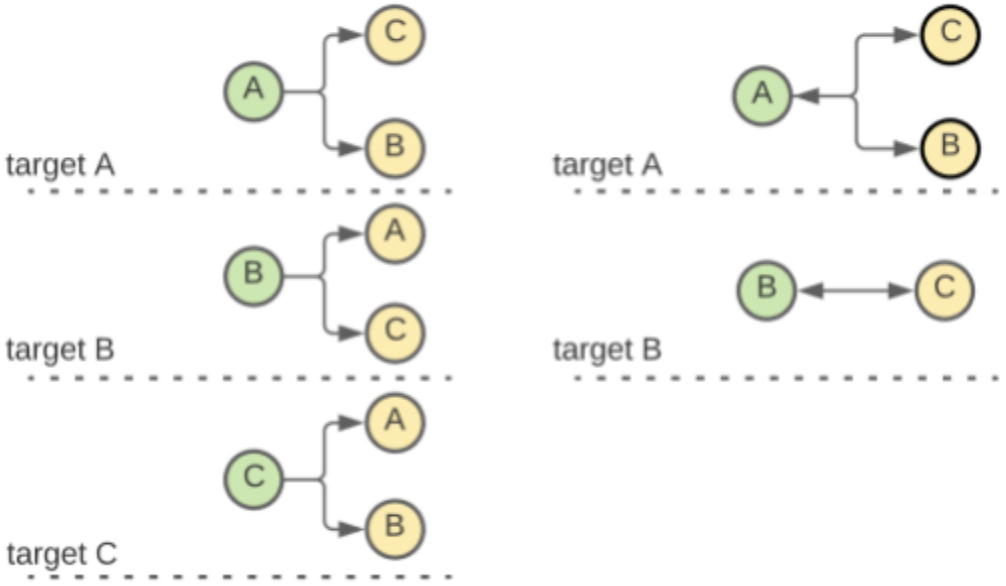


Figure 15. Left: calculating forces between two bodies every time. Right: calculating force between two bodies only once and store it for later calculations

However, this complicates parallelization. If the outer loop is still divided into continuous-range blocks, the first block (with more inner for loops, thus with the heaviest workload) and the last block (with less inner for loops, thus with the lightest workload) will have very imbalanced workloads, causing the thread that executes the first block to be much slower than others. The solution is also simple: for each block, we alternate between the **target** body of lowest (with heavier workload) and highest index (with lighter workload), so that the total workload of every thread can be balanced out, as shown in Figure 16. For each thread, we also allocate an individual buffer for accumulating accelerations, and merging them after all threads are finished, this avoids locking mechanisms that prevent race conditions.

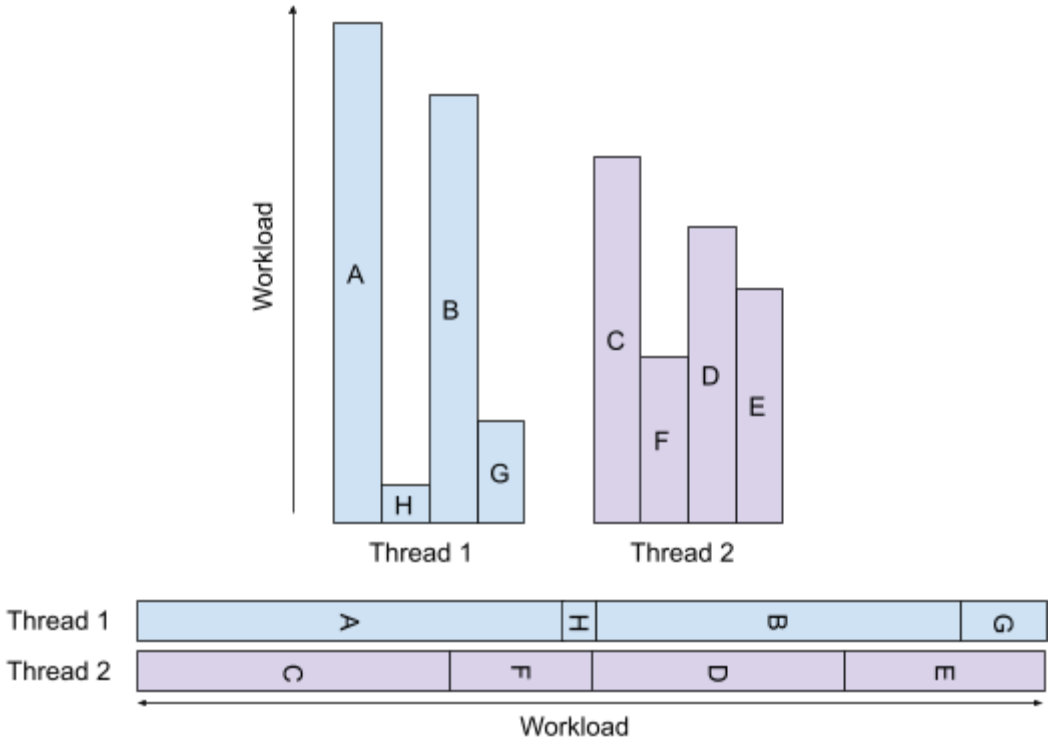


Figure 16. Acceleration calculation workload distribution between threads. Block with letter denotes all computation workload for calculating the acceleration for that target body. With edge-sharing, the acceleration computation of every target body has a different workload now, thus it is crucial to properly distribute them among threads for best performance

Another way of distributing workloads is to use a linearized index, or triu_indices (indices for the upper-triangle of a 2d array), to represent the acceleration calculation pairwise edges, and then convert this index back to **target** and **source** body id. However, experimentation shows that the index conversion is very expensive, and completely counterweights the functionalities it brings since the conversion requires many operations including multiplication, division, floating-point operations, and even sqrt operations.

5.3 Performance Measurements

5.3.1 Performance Metrics

We have decided to measure only the runtime per iteration (i.e., per time step) and use it for performance comparisons between different solutions and optimizations. Although ideally, we would also want to calculate GFLOPs and memory bandwidth and consider those for evaluation, there were two concerns:

1. The acceleration calculations involve square root and floating-point division. It is hard to quantify what is the correct number of flops for them. Nvidia's paper [2] treated those operations as 1 flop but we believe this is inaccurate.

2. In our optimizations, the majority of memory access happens on the shared memory, so it is expected that the global memory bandwidth is underutilized, and knowing the actual number doesn't seem to be very helpful.

Due to the reasoning above, runtime per iteration is the metric we use to evaluate optimization performance.

To compute runtime per iteration, we first time the execution of several iterations, then divide the total time by the number of iterations to obtain the average runtime per iteration. This average runtime per iteration is what we rely on for performance evaluation.

While varying parameters, only the runs with parameters that produce the best performance for a given body size have their speedups recorded.

5.3.2 Relevant Parameters in Experiments

5.3.2.1 General Parameters

Every experiment runs 20 iterations. While varying other parameters, each configuration is run over 20k, 100k, and 200k bodies, respectively.

5.3.2.2 Parameters for CPU Implementations

We have a total of two different CPU implementations, corresponding to *Section 5.2.1* (ID: CPU0 - **Basic Implementation**) and *Section 5.2.2* (ID: CPU1 - **Shared Edge**), respectively. For each CPU ID, there are two options for the number of threads – 1 and 4. The reason for capping the number of threads to 4 is that the CPU we use has 4 cores with hyperthreading turned off, so the most concurrency we can exploit is 4x.

Since when the number of bodies is 200k, the 1-thread run takes too long (over 30min), it is skipped for both CPU0 and CPU1.

5.3.2.3 Parameters for GPU Implementations

We have many versions of GPU solutions, with GPU0 and GPU1 being the reference implementations, representing the **Simple Pairwise Calculation** approach and the **Nvidia Tiling** approach respectively. GPU2-8 are staged optimizations with each building on its immediate predecessor.

For each GPU version, its additional optimizations, the parameters to vary and constraints on parameters are summarized in *Table 1* below:

ID	Solution	Optimizations	Parameters
GPU0	Simple Pairwise	Basic implementation	Block size = [16, 64, 128, 256]
GPU1	Nvidia Tiling	Tiling	
GPU2	2D Tiling	2D block loop unrolling	Block length (len) = [1, 2, 4]

GPU3		Use rsqrt	Block width (wid) = [64, 128, 256, 512] Loop unrolling factor (luf) = [256, 512, 1024] Constraints: $len \times luf \leq 1024$ $luf \geq wid$
GPU4		Basic 2D Tiling	
GPU5		Register reduction	
GPU6		Avoidance of bank conflicts	
GPU7		cuBLAS accumulation	
GPU8		Block parameterization	

Table 1. Experiment configurations swept for staged implementation of GPU optimizations

5.4 Performance Comparisons

As mentioned previously, only runs with the best results for a given body size are considered and raw runtime data are available in *Appendix 2*. Here, we only present the speedup results in tables, calculating from the raw data in *Appendix 2*. These tables are colour-coded – the darker the color, the better the performance.

First, we look at Table 2 calculated from *Appendix 2 - Table 2*. When 2D thread block is first introduced, there is a performance drop up to 22% compared to the **Nvidia Tiling** approach. However, with optimizations in **GPU3 - Use rsqrt** and **GPU4 - Basic 2D tiling**, **2.19-2.33x** performance improvement is achieved compared to **Nvidia Tiling**, whereas **6.61-7.10x** is achieved compared to **Simple Pairwise Calculation**. Optimizations in **GPU5 - Register reduction** and **GPU6 - Avoidance of bank conflicts** are not able to give speedups when looking at only the runs with the best performance. The advantage of GPU7 - cuBLAS accumulation becomes more significant as the number of bodies increases, but overall doesn't provide too much performance gain (**1.04-1.07x**). GPU8 - Block parameterization shows performance gain in the same range (**1.05x**) as GPU7.

ID	Speedup								
	vs. previous ID			vs. 0 - Simple Pairwise			vs. 1 - Nvidia Tiling		
Body#	20k	100k	200k	20k	100k	200k	20k	100k	200k
GPU0									
GPU1				2.92x	3.05x	3.23x			
GPU2				2.51x	2.73x	2.64x	0.86x	0.89x	0.82x
GPU3	1.58x	1.58x	1.56x	3.97x	4.32x	4.13x	1.36x	1.41x	1.28x
GPU4	1.67x	1.64x	1.71x	6.61x	7.10x	7.07x	2.26x	2.33x	2.19x

GPU5	1.00x	1.00x	0.98x	6.59x	7.09x	6.93x	2.25x	2.32x	2.15x
GPU6	1.00x	1.00x	0.98x	6.59x	7.06x	6.77x	2.26x	2.31x	2.10x
GPU7	0.99x	1.04x	1.07x	6.54x	7.36x	7.24x	2.24x	2.41x	2.25x
GPU8	1.05x	1.05x	1.05x	6.84x	7.71x	7.61x	2.34x	2.53x	2.36x

Table 2. Speedup results comparing staged implementations for GPU optimizations

Note that for GPU2-8, all runs deliver their best performance when len = 1. However, the benefits of **GPU6 - Avoidance of bank conflicts** only apply when len is bigger. With len = 4, configurations selected in *Appendix 2 - Table 3* produces **2.71-2.81x** speedup as shown in Table 3 below:

Comparison	Speedup		
Body#	20k	100k	200k
GPU6 vs. GPU5	2.81x	2.71x	2.72x

Table 3. Speedup results comparing GPU6 vs. GPU5

Using data in *Appendix 2 - Table 4*, we explore performance comparisons among CPU runs and cross-comparisons between GPU and CPU, whose results are presented in Table 4 and 5.

Between CPU1 and CPU0, with 1 thread, the advantage of CPU1 is greater at the low body count, whereas with 4 threads, it is greater at the high body count. With 4 threads instead of 1, the performance of CPU0 improved **3.39-3.65x** consistently, whereas CPU improved only **2.45x** at the low body count but reached **4.01x** at the high body count.

In GPU-CPU cross-comparisons, with 1 thread, the final GPU version exhibits higher performance improvements at the high body count; with 4 threads, performance gain stays about the same at higher body counts. The best performance occurs at the high body count for CPU0 but at the low body count for CPU1.

Comparison	Speedup	
Body#	20k	100k
CPU0	3.39x	3.65x
CPU1	2.45x	4.01x

Table 4. Speedup results comparing 1-threaded vs. 4-threaded CPU runs

Comparison	Speedup				
Body#, thread #	20k, 1	100k, 1	20k, 4	100k, 4	200k, 4

CPU1 vs. CPU0	1.50x	1.08x	1.67x	1.84x	1.78x
GPU8 vs. CPU0	879.14x	1031.18x	259.58x	282.26x	284.02x
GPU8 vs. CPU1	586.64x	614.23x	239.35x	153.14x	159.97x

Table 5. Speedup results comparing final GPU (GPU8) vs. CPUs and CPU1 vs. CPU0

6. Discussion

6.1 Overall Assessment

Our overall assessment of this project is that intrinsically, the computation involved in the N-body problem is easily parallelizable, but its memory access pattern is too simple, leaving limited room for performance improvements.

We have tried out various optimizations as detailed in *Section 4. GPU Optimizations*. We have achieved **259.58-284.02x** performance improvement compared to 4-thread CPU implementation, and **2.34-2.53x** compared to our implementation of the **Nvidia Tiling** solution. It is also **6.84x-7.71x** faster than our initial GPU implementation. Certain assumptions of ours have been proved wrong by experiment results. For example, our main optimization idea, 2D Tiling, reaches its best performance when the 2D thread block has 1 column. In other words, the best performance happens when vertical 1D thread blocks are launched in a 2D grid, rather than 2D thread blocks in a 2D grid. This implies that parallelism in acceleration calculation is beneficial when it is not too excessive.

6.2 Learning Experience

Nevertheless, it has been a wonderful learning experience to brainstorm, implement, debug and experiment with those optimizations. We have become more familiar with common optimization considerations for GPU such as loop unrolling, coalesce access to global memory, bank conflict avoidance in shared memory, register usage reduction, and parameter tuning. The performance gap between CPU and GPU kept reminding us of the power and importance of exploiting parallelism in the right problem on the right hardware.

6.3 Other Attempted GPU Optimizations

We have tried to extend the **CPU Edge-Sharing Acceleration Calculation** approach in *Section 5.2.2* to GPU, but failed as we couldn't find a way to resolve race conditions during the reads from and writes to memory.

For intermediate result accumulation in **2D Tiling**, other than the **Simple Accumulation** approach and the **cuBLAS Matrix Multiplication** approach detailed in *Section 4.1.2*, we have

also tried to implement an **Optimized Tree-Based** approach as available in [15]. In our test runs comparing the **Optimized Tree-Based** approach against cuBLAS on reduction of a 1D array, 27% performance is achieved with 100k array elements. However, due to the limited resources we have, we weren't able to finish integrating this **Optimized Tree-Based** approach into our GPU kernel.

Reference

[1] R. R. Bate, D. D. Mueller, and J. E. White, *Fundamentals of astrodynamics*. New York: Dover Publications, 2015. Page 1-49.

[2] L. Nyland, M. Harris, and J. Prins, "Chapter 31. Fast N-body simulation with CUDA," in *GPU Gems 3*, 2009. [Online]. Available: https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda

[3] K. Fujiwara and N.Nakasato, "Fast Simulations of Gravitational Many-body Problem on RV770 GPU," Extended Undergraduate Thesis, Dept. of Comp. Sci. and Eng., Uni. of Aizu, Aizu-Wakamatsu, Fukushima, Japan, 2008. [Online]. Available: https://arxiv.org/pdf/0904.3659.pdf

[4] Stackoverflow, "Reduce matrix rows with CUDA," December 28, 2021. [Online]. Available: https://stackoverflow.com/questions/17862078/reduce-matrix-rows-with-cuda

[5] NVIDIA Corporation, "cuBLAS," December 15, 2021. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html

[6] NVIDIA Corporation, "CUDA Occupancy Calculator," November 13, 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html

[7] NVIDIA Corporation, "CUDA C++ Programming Guide," November 13, 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[8] NVIDIA Corporation, "GeForce GTX 980 Specifications," November 13, 2021. [Online]. Available: https://www.nvidia.com/en-us/geforce/gaming-laptops/geforce-gtx-980/specifications/

[9] NVIDIA Corporation, "Maxwell: The Most Advanced CUDA GPU Ever Made," November 23, 2021. [Online]. Available: https://developer.nvidia.com/blog/maxwell-most-advanced-cuda-gpu-ever-made/

[10] NVIDIA Corporation, "Maxwell Architecture," November 13, 2021. [Online]. Available: https://developer.nvidia.com/maxwell-compute-architecture

[11] The N-Body Shop, "Initial Conditions," November 25, 2021. [Online]. Available: https://nbody.shop/data.html

[12] The N-Body Shop, "AGORA Initial Conditions," November 25, 2021. [Online]. Available: https://b2share.eudat.eu/records/b81608fbc7644e76ac327bfdff768aa8

[13] Wikipedia. "Three-body problem," December 26, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Three-body_problem

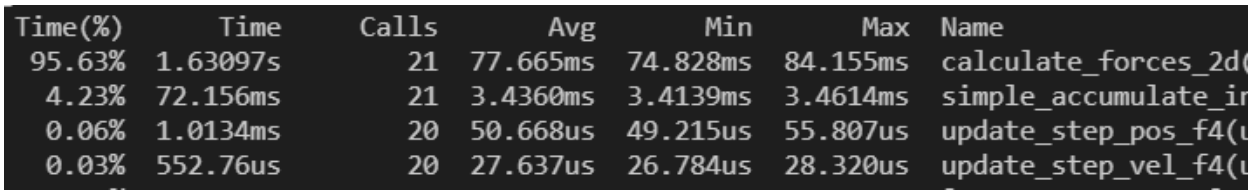
[14] Wikipedia. "Chaos theory," December 26, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Chaos_theory

[15] M. Harris, “Optimizing Parallel Reduction in CUDA,” November 23, 2021. [Online]. Available: https://developer.download.nvidia.cn/assets/cuda/files/reduction.pdf

[16] The YT project. “Using yt to view and analyze Tipsy outputs from Gasoline”, December 26, 2021. [Online]. Available: https://yt-project.org/doc/cookbook/tipsy_notebook.html#using-yt-to-view-and-analyze-tipsy-outputs-from-gasoline

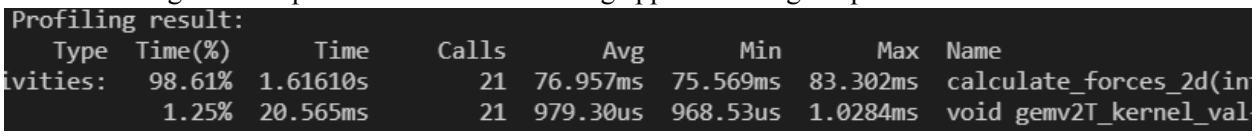
Appendix

1. Screenshots of Nvprof Results



Time(%)	Time	Calls	Avg	Min	Max	Name
95.63%	1.63097s	21	77.665ms	74.828ms	84.155ms	calculate_forces_2d(
4.23%	72.156ms	21	3.4360ms	3.4139ms	3.4614ms	simple_accumulate_ir
0.06%	1.0134ms	20	50.668us	49.215us	55.807us	update_step_pos_f4(u
0.03%	552.76us	20	27.637us	26.784us	28.320us	update_step_vel_f4(u

Figure 1. Nvprof results of our 2D tiling approach using simple summation kernel



Profiling result:

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
ivities:	98.61%	1.61610s	21	76.957ms	75.569ms	83.302ms	calculate_forces_2d(in
	1.25%	20.565ms	21	979.30us	968.53us	1.0284ms	void gemv2T_kernel_val

Figure 2. Nvprof results of our 2D tiling approach using cuBLAS summation kernel

2. Raw Runtime Data per iteration

Parameters			Time (per iteration)
blockdim.x	blockdim.y	luf	
1	512	1024	0.078378s
1	256	1024	0.078663s
2	512	512	0.085095s
2	256	512	0.085278s
4	256	256	0.090384s
4	128	256	0.096541s
8	128	128	0.098089s
9	64	128	0.098628s

Table 1. Raw runtime data per iteration for the final GPU version (GPU 8), with runs configured to produce the best performance when body count is 100k

ID	Parameter			Time
	Body #	Block size	luf	
GPU0	20k 100k 200k	128 64 64		0.025041s 0.604406s 2.407589s

GPU1	20k 100k 200k	128 256 256		0.008573s 0.198012s 0.746396s
GPU2	20k 100k 200k	1×128=128* 1×64=64 1×64=64	512 512 1024	0.009988s 0.221566s 0.910918s
GPU3	20k 100k 200k	1×256=256 1×128=128 1×128=128	1024 1024 512	0.006311s 0.140021s 0.582640s
GPU4	20k 100k 200k	1×512=512 1×512=512 1×256=256	1024 1024 1024	0.003790s 0.085134s 0.340424s
GPU5	20k 100k 200k	1×256=256 1×512=512 1×256=256	1024 1024 1024	0.003802s 0.085289s 0.347615s
GPU6	20k 100k 200k	1×256=256 1×512=512 1×512=512	1024 1024 512	0.003801s 0.085570s 0.355724s
GPU7	20k 100k 200k	1×256=256 1×512=512 1×512=512	512 1024 512	0.003830s 0.082145s 0.332343s
GPU8	20k 100k 200k	1×256=256 1×512=512 1×512=512	256 1024 512	0.003661s 0.078378s 0.316398s

Table 2. Raw runtime data per iteration for staged GPU implementations, with runs configured to produce the best performance (example: 1x128 = 128 means Blockdim.x = 1, Blockdim.y = 128. In total there are 128 threads in each block.)

ID	Parameters		Time	vs . 5
	Body #	Blockdim.x, Blockdim.y, luf		
GPU5	20k 100k 200k	1, 256, 1024	0.004149s 0.094586s 0.400883s	
	20k 100k 200k	4, 256, 256	0.012223s 0.273334s 1.084545s	

GPU6	20k 100k 200k	1, 256, 1024	0.004116s 0.089462s 0.377877s	1.01x 1.06x 1.06x
	20k 100k 200k	4, 256, 256	0.004346s 0.100756s 0.396748s	2.81x 2.71x2.7 2x

Table 3. Raw runtime data per iteration for GPU implementations of ID 5 and 6, with runs reflecting the benefits of bank conflict avoidance

ID	Name	Parameters		Time	vs. Our 2D Tiling
		Body #	Thread #		
CPU0	Basic CPU implementation	20k 20k 100k 100k 200k	1 4 1 4 4	3.218549s 0.950333s 80.822142s 22.123250s 89.863759s	879.14x 259.58x 1031.18x 282.26x 284.02x
CPU1	Shared Edge Acceleration Calculation	20k 20k 100k 100k 200k	1 4 1 4 4	2.147683s 0.876244s 48.142396s 12.002437s 50.614695s	586.64x 239.35x 614.23x 153.14x 159.97x

Table 4. Raw runtime data per iteration for CPU implementations