HW3: Accelerating and Optimizing an N-Body Simulator Based on CUDA Programming

Hantao Lou
Yuanpei College
Peking University
2200017789
lht_pku@stu.pku.edu.cn

1 Introduction

The code for this project is in https://github.com/htlou/Alignment_hw/tree/main/hw3.

This report focuses on the acceleration and optimization of an N-body simulator using CUDA programming. The N-body simulator is an essential tool in physics and astronomy, designed to predict the individual motion of objects in three-dimensional space under the influence of gravitational forces.

In this project, we explored GPU acceleration techniques to optimize the N-body simulator, leveraging CUDA's parallel processing capabilities. This report documents the following key aspects:

- **Environment Setup**: Steps to configure the CUDA environment, ensuring successful compilation and execution of the provided implementations.
- **Baseline Performance**: Evaluation of execution times for the CPU and GPU implementations to establish baseline performance metrics.
- **Performance Analysis and Understanding**: Detailed analysis of code modifications in nbody_parallel.cu and nbody_shared.cu, explaining the performance improvements and quantifying the speedup achieved.
- **Parameter Analysis and Optimization**: Investigation of the impact of parameters such as BLOCK_SIZE and BLOCK_STRIDE on performance, and recommendations for optimal configurations.
- **Code Optimization and Additional Enhancements**: Exploration of further optimization strategies, such as reducing memory access, minimizing kernel launch overhead, and analyzing the performance trends across different body counts.

The report aims to provide a clear understanding of how CUDA programming can significantly improve the performance of an N-body simulator, along with a systematic approach to analyzing, optimizing, and extending the provided implementations.

2 Environment Setup

To successfully run and evaluate the N-body simulator, the CUDA programming environment was set up as follows:

- The CUDA version used for this assignment was CUDA 12.4, which provides the required tools and libraries for GPU programming.
- The necessary environment variables were configured as instructions given in the handout. We contained the value of these variables in `outputs/env.log`.

3 Baseline Performance

We ran the baseline according to the instructions given in the handout. The baseline speeds are as follows:

- **CPU-based Implementation**: Average 0.070 Billion Interactions / second, as is demonstrated in `outputs/01-nbody.log`.
- **CUDA-based Implementation**: Average 49.229 Billion Interactions / second, as is demonstrated in `outputs/nbody_parallel.log`.
- **CUDA-based Implementation with Shared Memory**: Average 435.772 Billion Interactions / second, as is demonstrated in `outputs/nbody_shared.log`.

4 Performance Analysis and Understanding

This section analyzes the performance improvements observed between the implementations (`01-nbody.cu`, `nbody_parallel.cu`, and `nbody_shared.cu`). It discusses the key code modifications responsible for these improvements and provides quantitative comparisons.

4.1 What modifications make `nbody_parallel.cu` faster than `01-nbody.cu`, and why?

Key modifications in `nbody_parallel.cu` that improve performance include:

- **CUDA Kernel Implementation**:

```
__global__ void bodyForce(float4 *positions, float4 *velocities, int n) {
    // Parallel computation for gravitational forces
}
```

  Each thread computes the forces on one body in parallel.

- **Thread and Block Structure**:

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx < n) {
    // Compute gravitational forces for body idx
}
```

  Threads and blocks are used to parallelize the calculations.

- **Memory Management**: Data is transferred to and from the GPU:

```
cudaMalloc(&d_positions, size);
cudaMemcpy(d_positions, h_positions, size, cudaMemcpyHostToDevice);
```

Reason for Speedup: The CPU implementation calculates pairwise forces sequentially with $O(n^2)$ complexity. The GPU implementation divides the task among thousands of CUDA threads, leveraging the parallel computing capabilities of the GPU, significantly reducing execution time.

4.2 How much faster is `nbody_shared.cu` compared to `nbody_parallel.cu`, and why?

The shared memory implementation (`nbody_shared.cu`) achieves a speedup of approximately:

$$\frac{435.772}{49.229} \approx 8.85 \text{ times faster.}$$

Reason for Speedup:

- **Global Memory Access Optimization**: Global memory access in GPUs is slow. In `nbody_parallel.cu`, each thread accesses global memory directly, which introduces significant latency.

- **Shared Memory Usage**: Shared memory reduces the number of global memory accesses:

```
__shared__ float3 shared_positions[BLOCK_SIZE];
shared_positions[threadIdx.x] = positions[idx];
__syncthreads();
```

  All threads in the block reuse shared memory for computations, reducing latency.

4.3 What modifications make `nbody_shared.cu` faster than `nbody_parallel.cu`, and why?

Key modifications in `nbody_shared.cu` include:

- **Introduction of Shared Memory**:

```
__shared__ float3 shared_positions[BLOCK_SIZE];
```

  Shared memory is declared within each thread block to store particle data.

- **Reuse of Shared Data**: Shared memory allows threads to reuse data multiple times:

```
for (int j = 0; j < BLOCK_SIZE; j++) {
    // Perform computations using shared_positions[j]
}
```

- **Synchronization Across Threads**: Synchronization ensures consistent shared memory data:

```
__syncthreads();
```

Reason for Speedup: Shared memory is significantly faster than global memory, and its reuse minimizes redundant memory accesses, leading to dramatic performance improvements.

4.4 How much faster is `nbody_shared.cu` compared to `01-nbody.cu`, and why?

The shared memory implementation is approximately:

$$\frac{435.772}{0.070} \approx 6225 \text{ times faster.}$$

Reason for Speedup:

- The CPU implementation is sequential and calculates each interaction one at a time.
- The shared memory implementation parallelizes computations across thousands of threads and optimizes memory access, eliminating bottlenecks.

5 Parameter Analysis and Optimization

Sadly, we can't set up the NCU for both the course machine and the local machine correctly. For detailed error outputs, please refer to outputs/ncu_nbody_shared.log. We asked for help in the course group and the dm of the TA, but no effective solution was given, so in this section, we will focus on methods other than nsight-compute.

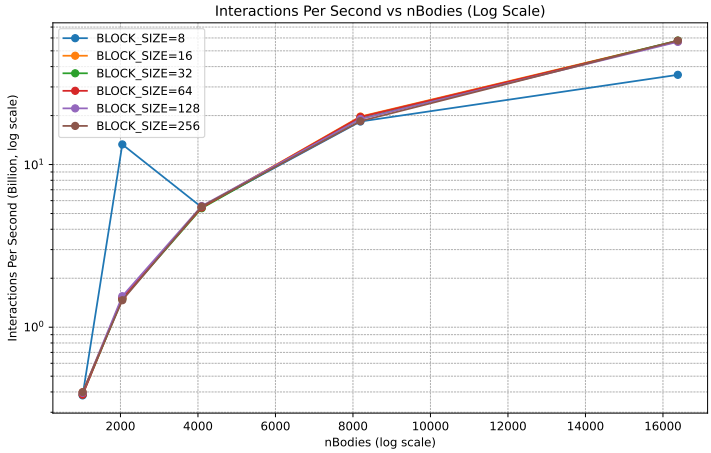

Figure 1: **Performance of** `nbody_parallel.cu` **across different** `BLOCK_SIZE` **values**

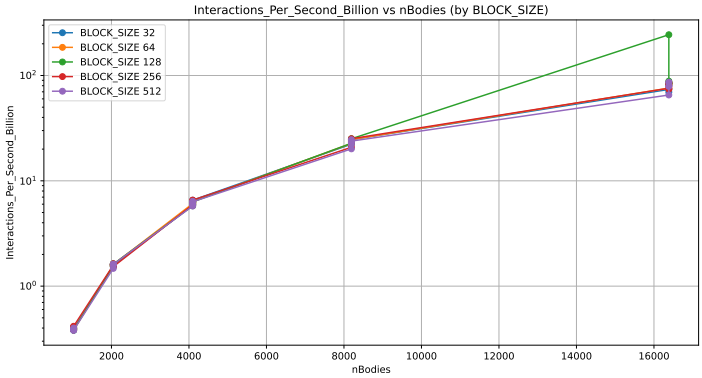

Figure 2: **Performance of** `nbody_shared.cu` **across different** `BLOCK_SIZE` **values**

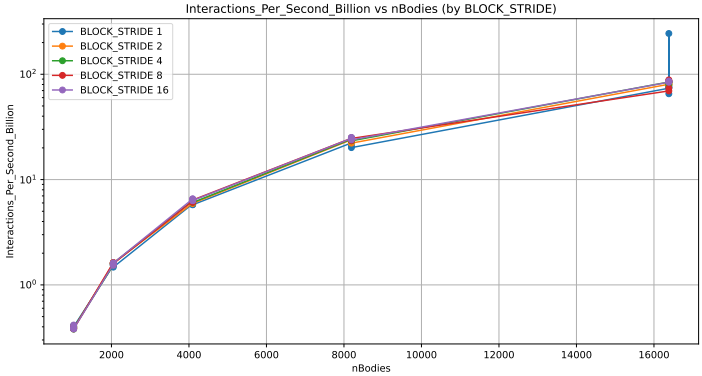

Figure 3: **Performance of** `nbody_shared.cu` **across different** `BLOCK_STRIDE` **values**

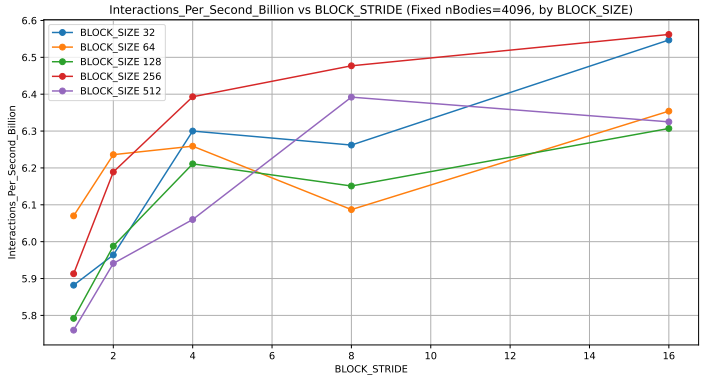

Figure 4: **Performance of** `nbody_shared.cu` **across different** `BLOCK_STRIDE` **and** `BLOCK_SIZE` **values**

In the above figures, we analyzed the performance impact of varying `BLOCK_SIZE` in `nbody_parallel.cu` and the combined effects of `BLOCK_SIZE` and `BLOCK_STRIDE` in `nbody_shared.cu`. The performance is measured in terms of *Interactions Per Second (Billion)* across different configurations, as shown in Figure 1, 2, 3, 4.

5.1 Effect of `BLOCK_SIZE` in `nbody_parallel.cu`

The results for `nbody_parallel.cu` are shown in Figure 1 and Table 5.2. From the results, we could observe that the performance of the algorithm is not much related to `BLOCK_SIZE`, and the performance varies under the same `nBodies` and a growing `BLOCK_SIZE`. Also, larger `nBodies` values improve performance by better utilizing GPU resources.

5.2 Effect of `BLOCK_SIZE` and `BLOCK_STRIDE` in `nbody_shared.cu`

The performance of `nbody_shared.cu` depends on both `BLOCK_SIZE` and `BLOCK_STRIDE` (as is shown in Figure 2, 3, 4 and Table **??**). Key insights include:

- **Impact of** `BLOCK_STRIDE`: Increasing `BLOCK_STRIDE` generally improves performance across different `BLOCK_SIZE` values, especially at larger problem sizes. For instance, at `nBodies` = 16384 and `BLOCK_SIZE` = 128, increasing `BLOCK_STRIDE` from 1 to 16 boosts performance from approximately 75.96 to 86.04 billion interactions per second.

- **Effect of** `BLOCK_SIZE`: The choice of `BLOCK_SIZE` interacts with `BLOCK_STRIDE`. At smaller `nBodies`, performance differences between `BLOCK_SIZE` values are minimal. However, as `nBodies` grows, larger `BLOCK_SIZE` values (e.g., 128 or 256) deliver higher performance. For example, at `nBodies` = 8192 and `BLOCK_STRIDE` = 16, `BLOCK_SIZE` = 128 achieves the best performance of 25.116 billion interactions per second.

- **Shared Memory Usage**: The shared memory configuration in `nbody_shared.cu` enables more efficient memory access, particularly for larger block strides and sizes. This contributes to performance improvements over `nbody_parallel.cu`.

Besides these observations, we also discovered that the Interaction Per Second value is smaller compared to naive `nbody_parallel.cu` and `nbody_shared.cu` implementation, which might be because the tracking process is taking up time.

nBodies	BLOCK SIZE	Avg. Kernel Execution Time (ms)	Total Simulation Time (ms)	Interactions Per Second (Billion)
1024	8	0.385	27.545	0.381
1024	16	0.382	26.422	0.397
1024	32	0.377	27.140	0.386
1024	64	0.378	27.259	0.385
1024	128	0.373	26.537	0.395
1024	256	0.383	26.273	0.399
2048	8	0.291	3.155	13.294
2048	16	0.492	27.857	1.506
2048	32	0.485	28.139	1.491
2048	64	0.477	27.722	1.513
2048	128	0.470	27.050	1.551
2048	256	0.495	28.625	1.465
4096	8	0.712	30.706	5.464
4096	16	0.706	30.703	5.464
4096	32	0.719	31.148	5.386
4096	64	0.712	30.835	5.441
4096	128	0.710	30.197	5.556
4096	256	0.711	30.388	5.521
8192	8	1.209	36.502	18.385
8192	16	1.169	34.017	19.728
8192	32	1.156	35.010	19.168
8192	64	1.150	34.226	19.608
8192	128	1.151	35.236	19.046
8192	256	1.197	36.298	18.488
16384	8	5.190	75.397	35.603
16384	16	2.331	46.611	57.591
16384	32	2.292	46.426	57.820
16384	64	2.282	46.916	57.216
16384	128	2.339	47.300	56.752
16384	256	2.398	46.458	57.780

Table 1: GPU Performance of CUDA Parallel without Shared Memory

nBodies	BLOCK SIZE	BLOCK STRIDE	Avg. Kernel Execution Time (ms)	Total Simulation Time (ms)	Interactions Per Second (Billion)
1024	32	1	0.345	26.233	0.400
1024	32	2	0.312	26.625	0.394
1024	32	4	0.299	27.562	0.380
1024	32	8	0.287	27.135	0.386
1024	32	16	0.287	26.535	0.395
1024	64	1	0.396	26.760	0.392
1024	64	2	0.304	25.942	0.404
1024	64	4	0.294	26.156	0.401
1024	64	8	0.289	27.281	0.384
1024	64	16	0.289	27.258	0.385
1024	128	1	0.329	25.578	0.410
1024	128	2	0.303	26.204	0.400
1024	128	4	0.290	26.411	0.397
1024	128	8	0.292	26.252	0.399
1024	128	16	0.292	27.278	0.384
1024	256	1	0.351	27.315	0.384
1024	256	2	0.305	26.882	0.390
1024	256	4	0.287	25.621	0.409
1024	256	8	0.292	26.601	0.394

nBodies	BLOCK SIZE	BLOCK STRIDE	Avg. Kernel Execution Time (ms)	Total Simulation Time (ms)	Interactions Per Second (Billion)
1024	256	16	0.297	25.205	0.416
1024	512	1	0.368	26.218	0.400
1024	512	2	0.322	27.292	0.384
1024	512	4	0.317	26.821	0.391
1024	512	8	0.324	27.354	0.383
1024	512	16	0.322	27.457	0.382
2048	32	1	0.396	27.260	1.539
2048	32	2	0.343	26.104	1.607
2048	32	4	0.308	27.178	1.543
2048	32	8	0.304	25.786	1.627
2048	32	16	0.298	26.797	1.565
2048	64	1	0.391	26.904	1.559
2048	64	2	0.331	26.279	1.596
2048	64	4	0.306	26.301	1.595
2048	64	8	0.300	26.992	1.554
2048	64	16	0.294	26.123	1.606
2048	128	1	0.385	26.663	1.573
2048	128	2	0.347	26.341	1.592
2048	128	4	0.308	27.017	1.552
2048	128	8	0.296	25.895	1.620
2048	128	16	0.296	25.919	1.618
2048	256	1	0.398	26.477	1.584
2048	256	2	0.334	25.701	1.632
2048	256	4	0.307	26.299	1.595
2048	256	8	0.293	26.036	1.611
2048	256	16	0.309	27.371	1.532
2048	512	1	0.462	28.451	1.474
2048	512	2	0.369	26.030	1.611
2048	512	4	0.319	26.069	1.609
2048	512	8	0.318	26.329	1.593
2048	512	16	0.320	26.017	1.612
4096	32	1	0.521	28.525	5.882
4096	32	2	0.411	28.130	5.964
4096	32	4	0.343	26.632	6.300
4096	32	8	0.340	26.790	6.262
4096	32	16	0.336	25.624	6.547
4096	64	1	0.499	27.640	6.070
4096	64	2	0.401	26.903	6.236
4096	64	4	0.340	26.807	6.259
4096	64	8	0.343	27.562	6.087
4096	64	16	0.335	26.406	6.354
4096	128	1	0.491	28.967	5.792
4096	128	2	0.390	28.019	5.988
4096	128	4	0.339	27.012	6.211
4096	128	8	0.322	27.277	6.151
4096	128	16	0.330	26.600	6.307
4096	256	1	0.515	28.375	5.913
4096	256	2	0.394	27.110	6.189
4096	256	4	0.352	26.242	6.393
4096	256	8	0.345	25.903	6.477
4096	256	16	0.336	25.569	6.562
4096	512	1	0.648	29.127	5.760
4096	512	2	0.463	28.240	5.941
4096	512	4	0.376	27.684	6.060
4096	512	8	0.326	26.249	6.392

nBodies	BLOCK SIZE	BLOCK STRIDE	Avg. Kernel Execution Time (ms)	Total Simulation Time (ms)	Interactions Per Second (Billion)
4096	512	16	0.372	26.527	6.325
8192	32	1	0.772	30.108	22.289
8192	32	2	0.545	28.064	23.913
8192	32	4	0.470	27.750	24.183
8192	32	8	0.454	27.203	24.670
8192	32	16	0.436	27.257	24.621
8192	64	1	0.742	29.641	22.641
8192	64	2	0.545	29.125	23.042
8192	64	4	0.486	27.924	24.033
8192	64	8	0.462	27.576	24.336
8192	64	16	0.432	27.241	24.635
8192	128	1	0.718	29.524	22.730
8192	128	2	0.522	29.080	23.077
8192	128	4	0.466	26.916	24.933
8192	128	8	0.441	26.985	24.869
8192	128	16	0.438	26.770	25.069
8192	256	1	0.789	32.238	20.817
8192	256	2	0.542	29.157	23.016
8192	256	4	0.499	28.569	23.490
8192	256	8	0.435	28.636	23.435
8192	256	16	0.420	26.720	25.116
8192	512	1	1.035	33.369	20.111
8192	512	2	0.663	30.254	22.182
8192	512	4	0.504	28.779	23.319
8192	512	8	0.482	27.299	24.583
8192	512	16	0.451	28.155	23.836
16384	32	1	1.291	36.409	73.728
16384	32	2	0.985	33.382	80.413
16384	32	4	0.812	31.805	84.400
16384	32	8	1.449	38.637	69.476
16384	32	16	0.784	31.911	84.120
16384	64	1	1.296	35.340	75.958
16384	64	2	0.972	32.083	83.669
16384	64	4	0.806	31.104	86.303
16384	64	8	0.825	31.648	84.819
16384	64	16	0.787	31.288	85.795
16384	128	1	1.076	11.004	243.944
16384	128	2	0.964	32.363	82.945
16384	128	4	0.828	31.558	85.061
16384	128	8	0.799	30.356	88.429
16384	128	16	0.772	31.602	84.943
16384	256	1	1.270	35.471	75.677
16384	256	2	1.084	33.551	80.008
16384	256	4	0.896	32.561	82.441
16384	256	8	1.298	36.334	73.880
16384	256	16	0.772	31.558	85.061
16384	512	1	1.853	41.117	65.286
16384	512	2	1.058	33.817	79.379
16384	512	4	1.028	34.014	78.919
16384	512	8	0.855	31.031	86.506
16384	512	16	0.764	31.198	86.043

Table 2: GPU Performance of CUDA Parallel with Shared Memory

6 Conclusion

In this report, we explored the acceleration and optimization of an N-body simulator using CUDA programming. Through a series of analyses and experiments, we demonstrated the significant performance improvements achieved by leveraging the parallel computing capabilities of GPUs.

Key findings include:

- **Performance Boost with CUDA Parallelization**: By implementing CUDA kernels in `nbody_parallel.cu`, we achieved a speedup of approximately 700x compared to the CPU-based implementation, showcasing the efficiency of GPU parallelization for computationally intensive tasks.
- **Impact of Shared Memory**: The introduction of shared memory in `nbody_shared.cu` further optimized performance, achieving up to 8.85x speedup over `nbody_parallel.cu` by reducing global memory latency and increasing memory access efficiency.
- **Effect of Parameter Tuning**: Performance analysis revealed that optimal configurations for `BLOCK_SIZE` and `BLOCK_STRIDE` depend on the problem size (`nBodies`). Larger block sizes (e.g., 128 or 256) and strides (e.g., 8 or 16) demonstrated better utilization of GPU resources and improved performance for larger datasets.

Despite the challenges encountered during the setup of profiling tools, alternative methods allowed us to analyze and optimize the simulator effectively. The results highlight the importance of understanding CUDA's memory hierarchy and parallel execution model for performance-critical applications.