PARALLELIZING 2D COLLISION DETECTION AND RESOLUTION

Kewei Han [1] **Jiya Zhang** [2]

1 SUMMARY

We parallelized a compute-intensive 2D collision detection application in particle simulation scenes using CUDA and compared multiple spatial partitioning strategies: Static Grid, Quadtree, and Semi-Static Quadtree. Our deliverables include implementations of these methods in both sequential and CUDA-accelerated versions, benchmarked on the GHC cluster. Results demonstrate that a **static grid** with CUDA excels in large, uniform distributions, while a **semi-static Quadtree** with CUDA provides the best overall performance for uneven workloads. Our project includes configurable runtime options and a detailed performance evaluation for both balanced and imbalanced workload scenarios, highlighting the trade-offs between computation overhead and workload adaptability.

Repository can be found at: Github.

2 BACKGROUND

Our project focuses on the computational challenges of parallelizing a 2D collision detection simulation involving the movement and interaction of 15,000+ particles. This problem is fundamental to numerous applications, including physics engines, particle simulations, and game systems.

The input and output of the program are as follows:

- **Input**: A set of particles, each defined by position (x, y), velocity (vx, vy), and bounds $(width, height)$
- **Output**: Updated particle states with modified position (x, y), velocity (vx, vy), and with unchanged bounds $(width, height)$

The output of the program is the same entities with updated velocities and positions based on whether the two entities have collided. This computation is repeated at high frequency with many frames per second to achieve smooth real-time simulation results.

In the naive implementation, at every frame the boundaries of each particle can be checked against every other particle in the entire scene to determine which particles collide. Since this is a purely $O(n^2)$ process, it does not scale for scenes with higher entity counts and can quickly reach over 1000ms frame time for only a thousand entities.

An optimization of this process that vastly speeds up collision checks is implementing a coarse-grained, broad-phase collision check such that we only check pairs of particles that are located in a similar region to each other.

In practice, this means that for every frame a spatial structure is constructed such that a scene of entities is evenly divided into grid cells. Each grid cell represents an area in the 2D scene and is a container to references to particles in that area. After such a structure is constructed, the number of collision checks needed can be drastically reduced.

2.1 Data Structure

From a high level, the notable data structures are:

- `Entity`: contains information about position, velocity, and collider bounds.
- `Cell`: contains references to entities located in a similar region.
- `Grid`: a collection of cells that represent the entire scene.

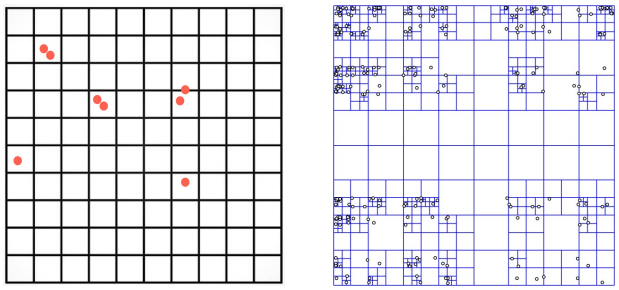
We also have a collision resolution step that takes in a grid and resolves collisions between all pairs of entities in each cell. It modifies the entity's position and velocity if its collider bounds overlap, thereby resolving the collisions.

A `grid` can be implemented in one of the following ways:

- `StaticGrid`: Divides the scene space into equal-sized cells (see Figure 9(a)).
- `Quadtree`: Divides the scene such that a cell contains no more than a certain number of entities in its regions, and if it does, the cell will split into four smaller cells to satisfy that constraint (see Figure 9(b)).

[1]Carnegie Mellon University [2]Carnegie Mellon University. Correspondence to: Kewei Han <keweihan@andrew.cmu.edu>, Jiya Zhang <jiyaz@andrew.cmu.edu>.

15-418/618 Parallel Computer Architecture and Programming, Fall 2024. Course Final Project.



(a) Static Grid (b) Quadtree

Figure 1. Possible `Grid` implementations.

Both methods generate a valid collection of `Cells` but will differ in the space each cell represents and thus the distribution of particles between cells.

2.2 Single Frame

Considering a scene of multiple entities, at a high level, the algorithm does the following.

- Iterate over entities and construct a grid of entity references such that each cell in the grid represents a small area in the scene. [1]
- For each cell in the grid, iterate through all possible entity pairings within the cells and check for and resolve collisions.

Pseudocode (Figure 2) gives a more detailed overview of the process of identifying overlapping entities within grid cells and resolving collisions between them.

Algorithm 1 Update Grid and Resolve Collisions

```
1:  for entity in entities do
2:      Determine grid cells the entity overlaps with
3:      Insert entity into respective grid cells
4:  end for
5:  for cell in grid do
6:      for entityA in cell do
7:          for entityB in cell do
8:              if isColliding(entityA, entityB)
                then
9:                  Update positions and velocities
10:                 resolveCollision(entityA,
                    entityB)
11:             end if
12:         end for
13:     end for
14: end for
```

Figure 2. The algorithm for updating the grid and resolving collisions in a 2D simulation.

3 Approach

3.1 Data Structure Mapping

A 2D scene can be partitioned into up to thousands of individual cells. The parallelization opportunity is to assign to each thread a `Cell` over which to iterate and resolve collisions. Given the high potential thread count, we decided to explore parallelization of the resolution step using CUDA on the Gates Hillman Center (GHC) computer cluster.

The starter code uses a game engine project SimpleECS as a base for development. It comes with a sequential implementation of entities, physics, collision detection, as well as the ability to visualize simulation scenes. For the purpose of collecting data from GHC, the application was modified to support running without visualization.

The application was adapted to accommodate parallelization in two main ways:

- Conversion of a complex entity object into a plain old data structure representation of an entity to allow copying it to the CUDA kernel (flattening).
- Moving the resolution logic involving collision detection and physics calculations into the CUDA device code to enable parallel resolution.

Both steps are encapsulated by a `CudaResolve` class.

To simplify development and allow for a more direct comparison between the two different grid implementations of `StaticGrid` and `Quadtree`, the grids were developed such that both must produce a vector of `Cells`. This vector is passed to `CudaResolve`, which maps each `Cell` to a CUDA thread for parallelized collision resolution.

3.1.1 Static Grid

A static grid implementation is set up so that the programmer can specify the size of the cell to divide the scene. A smaller cell size will result in cells on average containing fewer entities, and thus improving resolution time at the cost of increased overhead to construct the grid. Since threads are mapped to the number of cells, cell size becomes the handle for the number of threads used for parallelization in our implementation.

3.1.2 Quadtree

Our Quadtree implementation went through several iterations before obtaining a sufficiently efficient version for testing.

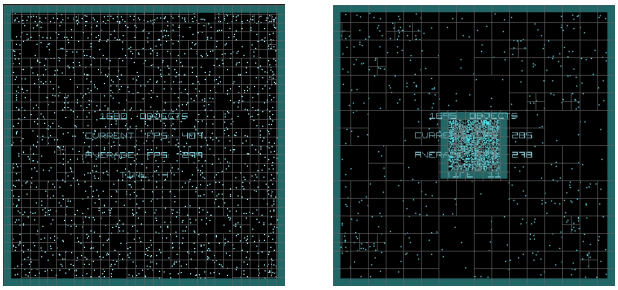
[1] Semi-static assignment only performs this step every few frames.

The initial exploratory implementation stored the Quadtree nodes in dynamic memory. After verifying that this implementation works, we implemented a flattened Quadtree version that continuously stored nodes to improve performance.

From the reference literature, we have found multiple variations of Quadtrees that apply for different use cases. For instance, Quadtrees may be implemented to optimize for retrieval of objects given an arbitrary bounding-box region, avoid duplicate references in the case of large objects, or only sort points instead of bounding boxes, none of which are relevant for this collision resolution scheme. We opt for an implementation of a Quadtree that optimizes the retrieval of its underlying cell structures so that it can be passed to 'CudaResolve'. This involved duplicate references of entities that intersect different cells and storage such that only leaf nodes contain entities.

The end result was a version of Quadtree whose performance is comparable to or better than that of the reasonably well-optimized static grid implementation.

3.2 Scenes



(a) Even density (b) Uneven density

Figure 3. Visualized scenes with Quadtree gridlines

Two scenes were prepared to evaluate and compare the performance of our spatial partitioning implementations under varying workload distributions. The first scene features an **evenly distributed particle configuration** (Figure 3(a)), where particles are uniformly spread across the simulation space, providing a balanced workload. The second scene introduces an **imbalanced high-density variation** (Figure 3(b)), with particles concentrated in specific regions and sparse elsewhere, simulating an uneven workload.

4 Optimization Strategy

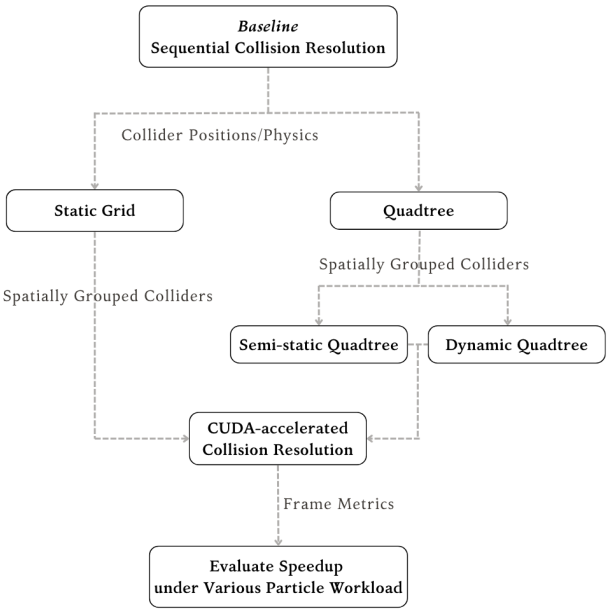


Figure 4. Optimization strategy flowchart

The grid construction step and collision resolution step are both similarly computationally intensive operations.

This project parallelizes the collision resolution step so that each thread maps to a cell of entities. As pictured in 4. We then explore the different methods of work distribution and identify which one results in ideal performance. We consider three different approaches to distribution.

- **Static assignment**: Implemented with a static grid. Each thread is responsible for collision resolutions in the same space or region across frames.
- **Dynamic assignment**: Implemented with a Quadtree. Every frame each thread is responsible for a region of space such that the region has a maximum limit of entities, thereby improving distribution.
- **Semi-static assignment**: Implemented with a Quadtree. Same as dynamic assignment except Quadtree construction occurs infrequently.

5 Results

Results are considered successful and we find that the features in data collected can generally be explained.

The application is created to allow for the selection of the work distribution algorithm during runtime. Experiment requests involved configuring variable values in the entry script `collisionStress.cpp` and library file

`CollisionSystem.cpp` before compiling and then running the binary. If compiled to use a headless renderer, the program returns the average frame length every five seconds to the command line.

We performed two separate analyzes of our program to examine both speed-up with respect to scaling number of entities and performance of the different work distribution schemes in the special case scene in *Figure 2*.

5.1 Parallelization Speedup over Entities

This analysis examines how the speedup (relative to the Static Grid Sequential baseline) of the different implementations scales with the number of entities in a scene where the entities are equally distributed.

In this experiment, the Quadtree and static grid configurations are fixed to the following configuration.

- **Dynamic/Semi-static Quadtree**: `maxDepth=10, maxEntitiesPerCell=20`
- **Static Grid**: `cellWidth`= 26, `cellHeight`=26

The experiments were carried out using workloads of varying numbers of entities: 10, 100, 1000, 10,000, and 50,000. These sizes allow us to analyze both small and large workloads and understand scalability. We use the GHC cluster as the targeted machine. *Figure 5* is the speed-up graph.

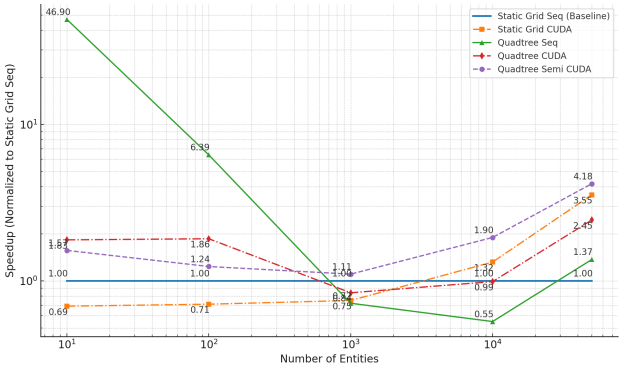


Figure 5. Performance evaluation count based on number of entities

5.1.1 *Result Analysis on Problem Size*

Compared to the baseline for sequential collision resolution of the static grid, its parallelized optimization with CUDA consistently improves as the workload size increases. At 50,000 entities, it achieves a 3.55x speedup relative to the baseline. However, in smaller workloads (10 to 100 entities), CUDA overhead dominates, leading to minimal gains (speedup 0.69–0.71).

Overall, Quadtree implementations with or without parallelization perform better with small workloads because of the flexibility on dynamic workloads. At 10 entities, Quadtree Sequential achieves a remarkable 46.9x speedup due to its adaptive spatial partitioning and minimal computational cost. For larger workloads, parallelization with CUDA is preferred, as semistatic and dynamic CUDA implementation achieves a speedup of 4.18x and 2.45x with 50,000 entities. Among all optimizations, Quadtree Semi with CUDA has the best overall performance for large workloads: by combining GPU acceleration with adaptive partitioning, this approach balances flexibility and computational efficiency.

In short summary, small and sparse workloads benefit from Quadtree methods, while large, uniform workloads favor Static Grid CUDA due to simpler parallelism. A hybrid adaptive method, semi-static Quadtree, can dynamically balance between Static Grid and Quadtree strategies and provide optimal performance for varying workload distributions.

5.1.2 *Analysis on Performance Limitations*

For small workloads (10 to 100 entities), the overhead of GPU initialization, kernel launches, and host-device communication limits performance, leading to suboptimal speedup for CUDA implementations. Also, considering the imbalanced workload, the Quadtree-based methods are sensitive to uneven particle distributions. Although they adapt well to sparse regions, managing tree structures introduces irregularities in workload balancing, which reduces GPU efficiency. Quadtree CUDA accessing from irregular memory access due to the hierarchical tree structure can cause cache misses and increased latency.

5.2 Performance on Particle Distribution

This analysis examines how various implementations perform in terms of frame time when the workload in entity count is kept constant but thread count is varied. Note that thread count is controlled by the configurations of the grid structures and how many `Cells` it outputs.

Both scenes contain 1700 entities, but exhibit different distribution of entities that allow for interesting discussion.

5.2.1 *Balanced Workload*

The key observation in the results of a balanced workload in *Figure 6* is that all three implementation strategies exhibit very similar performance times. At 0.8 - 1.3ms per frame, the scene will typically run at around 1000fps regardless of implementation. Evidently 1700 entities when distributed evenly is relatively trivial computation task, especially when considering the program is capable of performing at 50000 entities at sub-1000ms frame times.

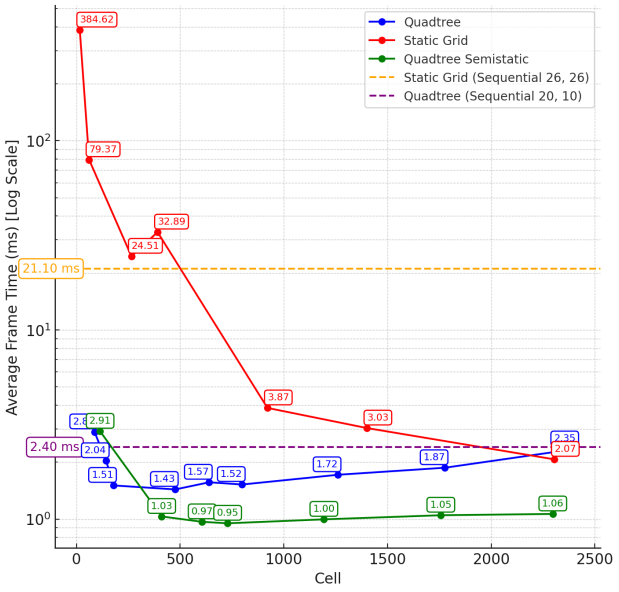


Figure 6. Balanced workload scene frame times

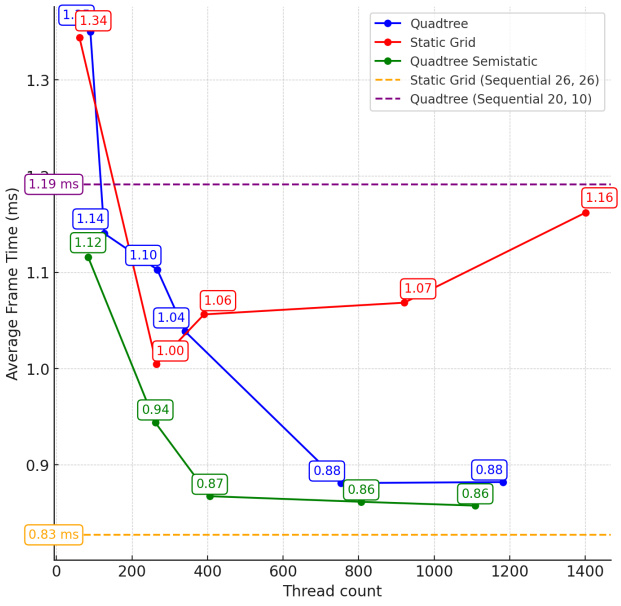


Figure 7. Imbalanced workload scene frame times

Another observation is that all parallel implementations perform noticeably worse than their sequential counterparts due to the low entity count. As discussed in 5.1, at low entity counts, the ratio of calculation of collision resolution to overhead is low, and thus it is normal to observe slowdown.

5.2.2 *Imbalanced Workload*

Results are more interesting when we simulate a scene where there is a region of high concentration of particles and also regions where the particles are very sparse.

The key observation is that a Quadtree performs **significantly better** than a static grid for an unbalanced distribution of workload in most thread counts due to a more balanced workload. In this scene, a Quadtree implementation can achieve better performance with less computation power than a static grid at 1.43 ms per frame with $\tilde{4}00$ threads compared to the static grid 2.07 ms per frame at $\tilde{2}000$ threads.

This scene generally demonstrates a situation where using a static grid can result in extreme workload imbalance for most cell sizes, which can cause very noticeable slowdowns. A single frame may take up to 300ms to resolve in this unbalanced scene, whereas it would instead take 3ms to resolve in a balanced scene when given the same computational resources at 60 threads (or 128 x 128 grid size).

With the Quadtree implementation, the work is distributed more evenly across the board by the nature of the data structure. By not allowing more than a set number of colliders in a given cell, we in effect limit the maximum work imbalance allowed and dynamically distribute work to CUDA threads.

As noted in 2.3, the Quadtree strategy can be further optimized with a semi-static implementation. While the plain Quadtree implementation updates the grid every frame, a semi-static approach updates the grid only once every 10 frames. Depending on the definition of correctness, this sacrifices minimal to zero correctness in the program, especially for scenes with slow moving particles in exchange for noticeable performance gains as evident in the graph.

5.3 General Speedup Limitations

5.3.1 *Non-limitations*

Before discussing limitations, it is worth noting some speculation on which concepts are likely not candidates for speedup limitation. For instance, warp divergence is likely unavoidable due to the unpredictable nature of how many collisions will occur in a given cell. Even if it can be mitigated, the data in Figure 2 indicates that the kernel execution step is not the bottleneck. There is also little opportunity for shared memory optimizations as the collision scenes we test are such that most particles are contained entirely within their own cell and so the work for threads is largely independent.

5.3.2 *Existing Limitations*

A semi-static Quadtree grid already addresses a number of performance bottlenecks such as unbalanced workload and grid construction overhead, but bottlenecks still exist.

Overall program speedup is primarily limited by other, non-collision related elements of the program that takes a con-

stant amount of time to run and reduce arithmetic intensity. If visualization of the scene is enabled, the obvious bottleneck would be scene-rendering-related logic. We see in *Table*1 that the pure collision calculation steps take up only 37% of total execution time, while non-related rendering functions will take up the remainder.

Function Name (Abbreviated)	Total Time(unit, %)
+ mainLoop	183865 (96.63%)
– renderGUI	81088 (42.62%)
– invokeCollisions	71758 (37.71%)
– renderObjects	16455 (8.65%)
– other	4735 (2.49%)

Table 1. Performance Profiling Results for visualized CUDA semi-static Quadtree in a balanced 20,000 particle scene

Function Name (Abbreviated)	Total Time (unit, %)
+ invokeCollisions	63601 (86.26%)
+ flattenCopyToDevice	42308 (57.38%)
– flatten	21935 (29.75%)
– copyToDevice	6036 (8.19%)
[...]	[...]
– kernelResolvePhysics	10580 (14.35%)
‘	10421 (14.13%)

Table 2. Performance profiling results of CUDA semi-static Quadtree without renderer in a balanced 20,000 particle scene

When rendered headless, we find that the breakdown shows that the flattening is the primary performance bottleneck. Flattening is the step before copying to device of gathering information about an entity from host and converting into a plain old data object that can be easily copied to device. This step is necessary because the design of the starting project focuses on generalization by expressing entities as a collection of components with a pooled entity component system, and so information like velocity or collider bounds is not stored together. This can be addressed if the project were designed in such a way that the information needed for collision resolution is already in a plain old data format, eliminating the need for a flattening step. However, this performance gain may come at a sacrifice of generality in the program itself, a tradeoff that is seen often in the world of software design in other contexts like programming languages.

Another notable obstacle to speed-up is the `copyToDevice` function, which involves purely copying data from host to device, taking up nearly as much time as the computation itself. Our project maintains entity information on host as the source of truth, which means every frame entity information is copied to the device, computation occurs, and then copied out. If grid construction can be made to occur on the device we can potentially implement the program such that data only need to be copied in once and the source of truth becomes data on the device to only be copied out of into host if needed. This again is a trade-off between performance and generality.

5.4 Choice of Machines

GPU implementation of parallelization did result in noticeable, and so it can be considered a correct choice of hardware for exploration of parallelization. However, the results also showed that the speedup benefits level off early on at $\tilde{4}00$ threads. Given that GPUs have the capacity to support far more physical threads, it would be an interesting area to explore whether CPU parallelization can achieve the same or better results.

5.5 Conclusion

We parallelized a 2D collision detection and resolution algorithm using CUDA, comparing three spatial partitioning strategies: Static Grid, Quadtree, and Semi-Static Quadtree. Benchmarking on the GHC cluster revealed that the Semi-Static Quadtree with CUDA delivers the best overall performance for imbalanced workloads, balancing adaptive partitioning with reduced grid construction overhead. For large, uniform workloads, Static Grid with CUDA outperformed other methods due to its simplicity and low overhead. Overall, a static Quadtree used in conjunction with CUDA parallelized collision detection and resolution performs better than other options across the board. Quadtree structures are also generally more versatile in being able to handle different scenes with varying particle density because of their dynamic nature.

Performance profiling highlighted bottlenecks in data preparation, particularly flattening and host-to-device memory transfers, limiting speedup. Addressing these could unlock further gains, though at the cost of generality. Our findings underscore the importance of matching spatial partitioning strategies to workload characteristics, with quadtree-based methods excelling in dynamic scenarios and static grid favoring uniform distributions.

This work demonstrates the value of combining GPU parallelization with adaptive partitioning for efficient and scalable real-time simulations.

6 Distribution of Total Credit

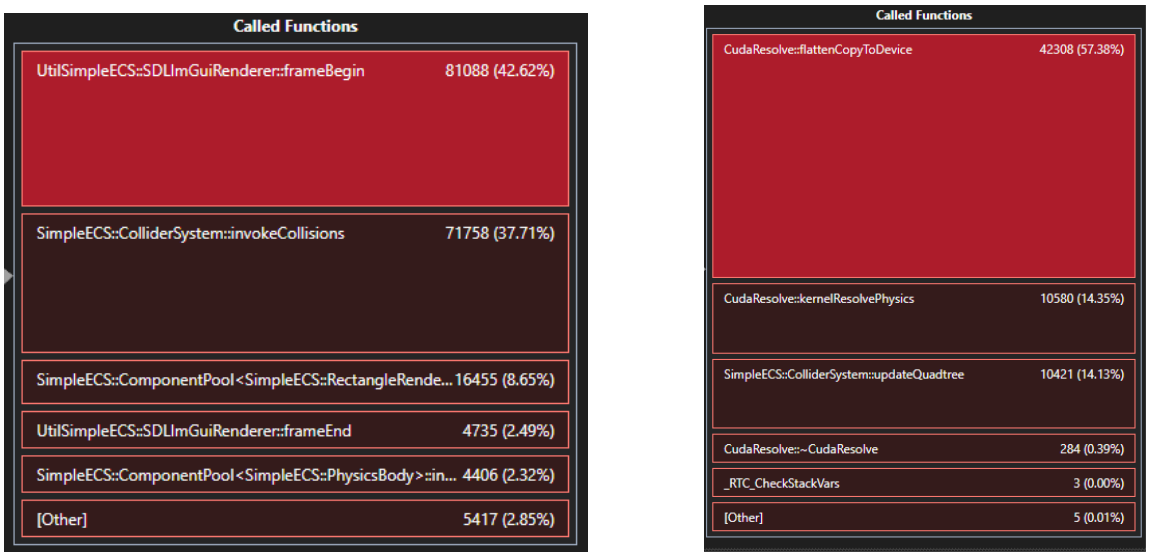
Both authors contributed equally to this project.

Work Task	Kewei Han	Jiya Zhang
Project Setup	X	
CUDA Resolution Parallelization	X	X
Quadtree Implementation		X
Data Collection	X	X
Poster Creation	X	X
Final Writeup	X	X

Table 3. Division of Work Tasks

7 Appendix

Figure 8. Visualization of 50,000 particle scene



(a) Table 1 reference (b) Table 2 reference

Figure 9. Performance profiling references

REFERENCES

Finkel, R. and Bentley, J. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 03 1974. doi: 10.1007/BF00288933.

Wikipedia. Quadtree — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Quadtree&oldid=1254556944, 2024.