Optimizing SPH Fluid Simulations with GPU Parallelization in CUDA

Kai-Wei Lin
Department of Computer Science and Engineering
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
john03690248@gmail.com

Shun-Yu Yang
Department of Computer Science and Engineering
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
aebandafg@gmail.com

Chung-Wei Hung
Department of Computer Science and Engineering
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
k0310507@gmail.com

I. Abstract

In smoothed particle simulation, we aim to accelerate fluid particle simulations. The primary goal of our project is to optimize the computation of particles within the smoothed particle radius in smoothed particle simulations. To achieve this, we implemented the Uniform Grid Search algorithm, which reduces the need to search all particles by limiting the search to those within 9 neighboring grid cells.

Using the perf tool, we identified the bottleneck of the program as the process of updating the grid structure, which requires re-sorting the particles based on their grid locations. To address this, we experimented with different sorting algorithms, such as bitonic sort, merge sort, counting sort, and bucket sort. Additionally, these sorting algorithms were optimized using CUDA, OpenMP, and pthreads to compare their acceleration performance. Finally, we determined that the best performance was achieved with bucket sort implemented using CUDA.

II. Introduction

In recent years, smoothed particle hydrodynamics (SPH) [1] has become increasingly prominent in fields such as computational fluid dynamics [2], computer graphics [3], and engineering. Unlike traditional grid-based approaches [4], particle-based techniques naturally handle complex free-surface flows, large deformations, and intricate fluid-solid interactions. Their adaptability makes them highly attractive for a wide range of real-world applications, including engineering design optimization, virtual prototyping, and realistic visual effects in film and video games.

However, as these simulations grow in scale and complexity—often involving millions of interacting particles—the computational burden becomes substantial. A key challenge lies in efficiently determining particle neighborhoods, which is essential for accurately modeling the forces and interactions in SPH. Naively searching all particles to find neighbors leads to prohibitively high computational costs, making it imperative to devise strategies that limit these searches to smaller, more relevant subsets. State-of-the-art solutions commonly rely on spatial data structures, such as uniform grids or trees [5], to reduce neighbor-finding overhead.

Despite these advancements, even optimized neighborhood searches face bottlenecks. Maintaining and updating the spatial indexing structures as particles move remains computationally expensive. Sorting-based approaches—crucial for grouping particles into grid cells or buckets—are integral to this process. Traditional sorting algorithms like merge sort or counting sort, while effective on CPU architectures, often struggle to achieve the scalability and speed required by large-scale SPH simulations. High-performance frameworks and accelerators, including GPU-based parallelization techniques, have emerged as a promising avenue to tackle these challenges.

Against this backdrop, our work focuses on streamlining the particle-sorting phase of the simulation. By comparing multiple sorting algorithms and their parallel implementations (including CUDA, OpenMP, and pthreads), we identify strategies that minimize sorting time and ultimately boost the overall simulation throughput. The motivation for this research is twofold: first, to address the bottleneck in sorting-driven spatial indexing for SPH simulations, and second, to enable more complex and realistic fluid scenarios to be run within manageable computational budgets. Achieving this balance opens the door to more refined modeling, greater physical fidelity, and richer dynamic interactions within the fluid simulation landscape.

III. Proposed Solution

This experiment evaluates the performance of various sorting algorithms and their parallelized implementations in Smoothed Particle Hydrodynamics (SPH) simulations. Sorting algorithms including Bitonic Sort, Counting Sort, Merge Sort, and Bucket Sort were implemented and parallelized using Pthreads, OpenMP, and CUDA to facilitate efficient neighbor searches. The experimental setup involved generating particle data at varying scales. Performance metrics such as execution time, speedup, and memory efficiency were measured and analyzed to identify the optimal algorithm and parallelization

strategy. This methodology ensures a comprehensive evaluation of sorting techniques within SPH simulations, highlighting their trade-offs and potential for optimization.

A. Bitonic Sort

- **Bitonic Sort (CUDA):** Bitonic Sort is well-suited for execution on GPUs due to its inherent parallel nature and structured computation pattern. CUDA parallelizes the Bitonic Sort algorithm's comparison and swap operations. Each thread independently determines if it needs to swap its element with its comparison partner.

B. Counting Sort

- **Counting Sort (Pthreads):** In Pthreads, the parallelization focuses on the counting step of the Counting Sort algorithm. The input particles array is divided among multiple threads, where each thread counts the occurrences of cell-key values in its assigned portion of the array.
- **Counting Sort (OpenMP):** In the OpenMP implementation of Counting Sort, parallelization is applied to the counting step of the algorithm. The loop that iterates over the particle vector is parallelized using OpenMP, where multiple threads process different parts of the array to update the count array based on the cell-key values.
- **Counting Sort (CUDA):** The "countingSortKernel" kernel counts the frequency of each cell key by incrementing the count array using "atomicAdd" to avoid race conditions.

C. Merge Sort

- **Merge Sort (Pthreads):** In the Pthreads implementation, the input data is divided into multiple blocks, with each block being sorted concurrently by a separate thread. Thread synchronization is managed using `pthread_join`, ensuring all threads complete their tasks before the merging process begins. Once the parallel sorting is finished, the sorted blocks are sequentially merged to produce the final sorted array.
- **Merge Sort (OpenMP):** The OpenMP implementation divides the data into blocks and sorts each block in parallel. During the merging phase, blocks are merged in parallel as well, with the block size doubling in each iteration until the entire array is sorted. OpenMP dynamically assigns threads to balance the workload across available CPU cores, effectively maximizing CPU utilization and improving performance.
- **Merge Sort (CUDA):** In the CUDA implementation, the merging step is parallelized using CUDA kernels, with each thread handling a specific range of subarrays. The merged block size doubles in each iteration to ensure complete sorting of the array. GPU memory is utilized to accelerate computations, and the results are transferred back to the host for final adjustments.

D. Bucket Sort

- **Bucket Sort (Pthreads):** In the Pthreads implementation, particles are divided among threads, with each thread assigning particles to buckets based on their `cell_key` within the thread's designated range. Each thread operates independently on its assigned range, efficiently utilizing CPU cores for parallel computation. After all threads have completed their tasks, the contents of all buckets are merged to form the final sorted array, achieving global order.
- **Bucket Sort (OpenMP):** In the OpenMP implementation, each thread uses private local buckets to assign particles based on their `cell_key`, ensuring thread-safe operations and minimizing contention. Once the local buckets are filled, they are merged into global buckets using critical sections to maintain data integrity while allowing concurrent execution. OpenMP dynamically distributes the workload across threads, effectively leveraging multicore hardware to accelerate the particle sorting process.
- **Bucket Sort (CUDA):** The CUDA implementation assigns particles to buckets based on their `cell_key`, utilizing the GPU's parallel computing capabilities to perform efficient bucket sorting. Offsets for each bucket are computed to ensure particles are correctly placed in their sorted positions, with atomic operations maintaining accuracy in the multi-threaded environment. The entire process, including particle distribution, bucket filling, and result generation, is executed on the GPU, maximizing performance and minimizing data transfer overhead.

IV. Experimental Methodology

A. Objective

The primary goal of this experiment is to compare the performance of various sorting algorithms under different parallelization techniques and to select the fastest sorting algorithm for integration into our project. The comparison focuses on execution time, with parallelization techniques including CUDA, Pthreads, and OpenMP, measured against the serial implementation. After identifying the fastest sorting algorithm on CUDA, it is integrated into the project and its performance is compared with the original serial sorting implementation.

B. Input Data

The dataset consists of particles with key values randomly generated in the range $[0, \text{particle count}]$, matching the random particle generation method used in the project. The particle counts range from 2^{15} to 2^{24}, ensuring scalability and relevance to real-world scenarios.

C. Testing Procedure

The experiment involves the following steps:

- Measure the execution time of each sorting algorithm (Counting Sort, Merge Sort, Bitonic Sort, and Bucket Sort) under different parallelization techniques (CUDA,

Pthreads, and OpenMP) as well as the serial implementation.
- Identify the fastest sorting algorithm when implemented with CUDA.
- Integrate the selected sorting algorithm into the project and compare its execution time with that of the original serial sorting implementation within the project context.

D. Experimental Environment

The experiments were conducted on the following hardware and software setup:

- **CPU:** Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
- **GPU:** NVIDIA Corporation GP106 [GeForce GTX 1060 6GB] (rev a1)
- **RAM:** 12GB
- **Operating System:** Debian GNU/Linux 12 (Bookworm)

E. Limitations

One limitation encountered during the experiments is that the Bitonic Sort algorithm can only operate on particle counts that are powers of two. If the particle count is not a power of two, the algorithm produces an error. This restriction was accounted for in the design of the test cases.

V. Experimental Results

A. Counting Sort

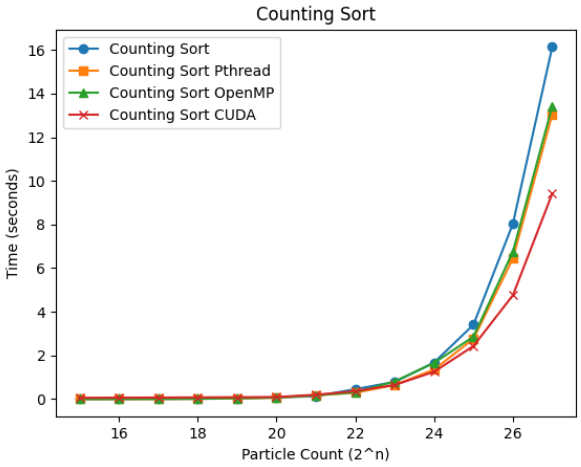

Fig. 1. Execution Time Comparison of Counting Sort Implementations

The experimental results for Counting Sort, as shown in Fig. 2, demonstrate that the CUDA implementation achieves the best performance, particularly at larger particle counts, due to its efficient parallelization and optimized use of GPU resources. While the Pthreads and OpenMP implementations also exhibit significant speedups compared to the serial implementation, they perform similarly across varying particle sizes. The serial Counting Sort shows a dramatic increase in execution time as the particle count grows, highlighting its scalability limitations. Overall, the results emphasize the advantage of GPU parallelization with CUDA for large-scale particle data, while Pthreads and OpenMP provide viable options for CPU-based parallelism.

B. Bitonic Sort

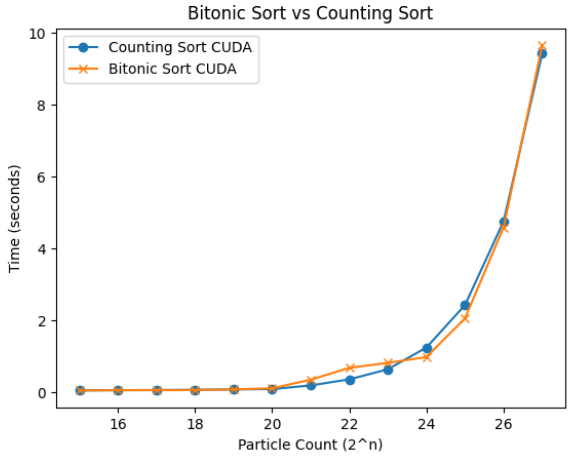

Fig. 2. Performance Comparison of Counting Sort and Bitonic Sort Using CUDA

The experimental results for the comparison of Counting Sort and Bitonic Sort, as shown in Fig. 3, demonstrate that the CUDA implementation of Counting Sort achieves the best performance at larger particle counts, owing to its efficient parallelization and optimized use of GPU resources. While both algorithms exhibit similar execution times for smaller particle sizes, Counting Sort outperforms Bitonic Sort as the particle count increases, due to its simpler computational structure and reduced synchronization overhead. These results highlight the scalability advantage of CUDA-based parallelization for large-scale particle data, emphasizing Counting Sort's suitability for GPU-accelerated SPH simulations.

C. Merge Sort

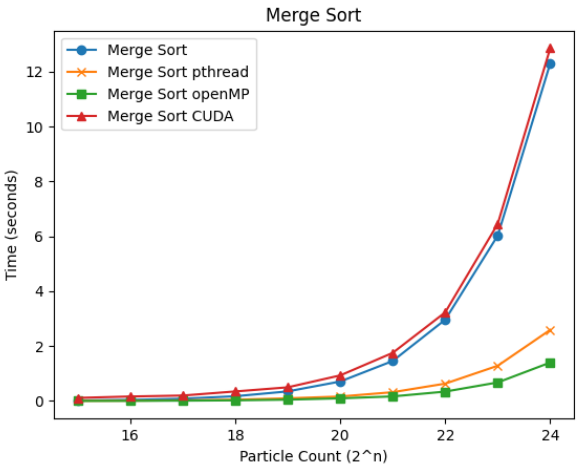

Fig. 3. Execution Time Comparison of Merge Sort Implementations

The experimental results for Merge Sort, as shown in Fig. 3, reveal that both Pthreads and OpenMP exhibit significant and increasingly noticeable performance improvements as the particle count grows. Additionally, OpenMP demonstrates slightly faster execution times compared to Pthreads, indicating that both are viable methods for accelerating Merge Sort on CPUs.

However, the CUDA implementation shows no notable performance improvement compared to the serial Merge Sort as the particle count increases. In fact, it performs slightly slower than the serial version. This could be attributed to several factors: (1) the initial merge intervals are very small, which reduces the effectiveness of parallelization; (2) frequent memory exchanges between the host and device; and (3) synchronization overhead. These challenges suggest that Merge Sort might not be well-suited for acceleration using CUDA, whereas CPU-based parallelism with OpenMP or Pthreads remains a practical approach.

D. Bucket Sort

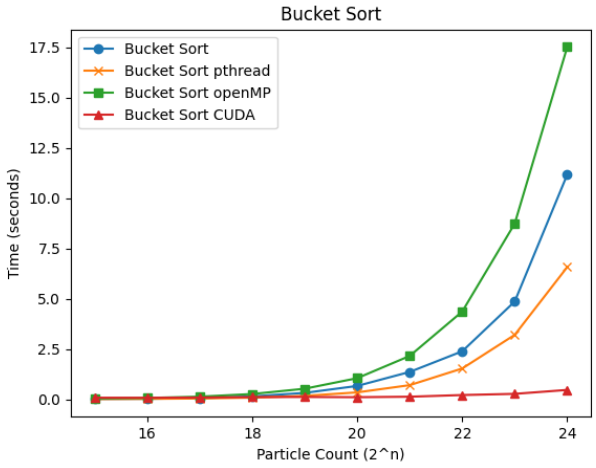

Fig. 4. Execution Time Comparison of Bucket Sort Implementations

The experimental results for Bucket Sort, as illustrated in Fig. 4, indicate that Pthreads effectively accelerates the algorithm as the particle count increases. In contrast, OpenMP performs slower than the serial implementation. This can likely be attributed to the use of `#pragma omp critical`, which introduces significant synchronization overhead. Further adjustments to the implementation might be required to optimize OpenMP's performance for Bucket Sort.

CUDA, on the other hand, achieves highly efficient acceleration and is the fastest among all methods. Its ability to distribute workloads evenly across the GPU cores makes it an ideal choice for Bucket Sort. Overall, among the three methods, CUDA is the most suitable option for accelerating Bucket Sort, particularly for large particle datasets.

E. Overall Sorting Comparison ith CUDA

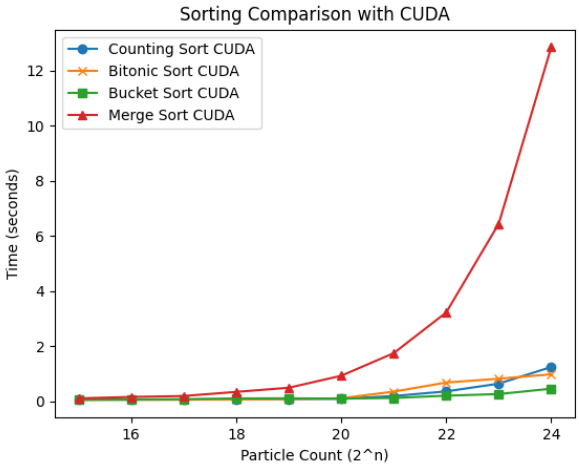

Fig. 5. Overall Sorting Algorithm Performance Comparison with CUDA

Since our primary goal is to leverage CUDA for acceleration, we compared the performance of four sorting algorithms—Counting Sort, Bitonic Sort, Merge Sort, and Bucket Sort—when implemented with CUDA, as shown in Fig. 5. The results indicate that Merge Sort is the least suitable for CUDA acceleration, exhibiting poor performance due to the factors discussed earlier.

In contrast, Counting Sort, Bitonic Sort, and Bucket Sort all achieve substantial speedups on CUDA, with Bucket Sort being the fastest among them. Its exceptional performance makes Bucket Sort the most suitable candidate for integration into our project.

F. Integration into the Project

	Implementation	Execution Time(ms)
1	Serial version	7172
2	Parallel acceleration using Bucket Sort with CUDA	178

Fig. 6. Overall Program Speedup Achieved with CUDA-Integrated Bucket Sort Compared to the Serial Version

Finally, we integrated the Bucket Sort algorithm into our project to evaluate its overall performance. As shown in Fig. 6, when the particle count reaches 2^{15}, the speedup achieved is approximately:

$$\text{Speedup} = \frac{7172.81}{178.522} \approx 40.17$$

This result demonstrates a nearly 40x acceleration compared to the baseline implementation. The significant improvement

in performance highlights the success of our approach and validates the effectiveness of using CUDA-accelerated Bucket Sort for large-scale particle data processing in our project.

VI. Related Work

Research on fluid simulation has a long history, particularly with the use of Smoothed Particle Hydrodynamics (SPH).

[6] Monaghan (1992) provided a classic overview of the fundamental principles and applications of SPH, highlighting its importance in astronomy and fluid dynamics. With advancements in computational power, researchers began exploring how to leverage Graphics Processing Units (GPUs) to accelerate SPH simulations.

[7] Akinci et al. (2012) introduced a GPU-based SPH method that successfully reduced the computation time for particle interactions, achieving real-time simulation of large-scale fluid fields.

Furthermore, [8] Käser and W. D. M. (2006) discussed the developmental trends of SPH methods in their review, emphasizing the importance of parallel computing for enhancing simulation efficiency.

More recently, [9]Gupta et al. (2015) performed a performance analysis on GPU-implemented SPH fluid simulations, demonstrating the potential of parallel computing techniques.

Our work will focus on utilizing CUDA for task allocation and improving the computation of particle interactions, aiming to achieve significant performance enhancements on general-purpose GPU and further advancing the application of fluid simulations.

VII. Conclusion

Our research presented an in-depth evaluation of sorting algorithms and their parallelization techniques to address computational bottlenecks in Smoothed Particle Hydrodynamics (SPH) simulations. We explored four sorting algorithms—Bitonic Sort, Counting Sort, Merge Sort, and Bucket Sort—implemented using three parallelization techniques: CUDA for GPU acceleration, OpenMP for shared-memory parallelism, and Pthreads for multi-threading. Our experiments demonstrated that the performance of each algorithm varied significantly depending on the chosen parallelization strategy and the scale of particle datasets.

Among the tested methods, Bucket Sort implemented with CUDA exhibited the best performance, achieving a nearly 40x speedup over the baseline serial implementation. This exceptional result can be attributed to CUDA's efficient parallel workload distribution and its ability to minimize synchronization and memory transfer overhead. While Counting Sort and Bitonic Sort also showed strong scalability with CUDA, Merge Sort was found to be less effective due to frequent memory exchanges and synchronization costs, highlighting its unsuitability for GPU acceleration in this context. On the other hand, OpenMP and Pthreads implementations demonstrated moderate improvements, making them viable for CPU-based parallelism.

By integrating the CUDA-accelerated Bucket Sort into our SPH simulation framework, we successfully reduced computation time for neighbor searches and enabled efficient handling of large-scale particle datasets. These findings not only optimize SPH simulations but also underline the importance of selecting algorithm-parallelization combinations tailored to the specific computational characteristics of fluid simulations.

References

[1] Koschier, D.; Bender, J.; Solenthaler, B.; Teschner, M. Smoothed particle hydrodynamics techniques for the physics based simulation of fluids and solids. *arXiv preprint* arXiv:2009.06944, 2020.

[2] Micky Kelager. 2006. Lagrangian fluid dynamics using smoothed particle hydrodynamics. Master's thesis. University of Copenhagen: Dept. of Computer Science.

[3] O Connor, J.; Rogers, B.D. A fluid–structure interaction model for free-surface flows and flexible structures using smoothed particle hydrodynamics on a GPU. *J. Fluids Struct.* 2021, *104*, 103312.

[4] Monaghan, J. J. (2015). Smoothed Particle Hydrodynamics in Astrophysics. *Annual Review of Astronomy and Astrophysics, 52*(1), 81–113. https://doi.org/10.1146/annurev-astro-082214-122309.

[5] NVIDIA Corporation. (n.d.). *CUDA Particle Simulation Documentation*.

[6] Monaghan, J. J. (1992). "Smoothed Particle Hydrodynamics." Annual Review of Astronomy and Astrophysics.

[7] Akinci, N., Chandra, R., Liu, H. (2012). "Flexible and efficient simulation of incompressible fluids." ACM Transactions on Graphics (TOG), 31(4), 1-10.

[8] Käser, M., W. D. M. (2006). "Smoothed Particle Hydrodynamics: A review." Journal of Computational Physics, 217(2), 623-634.

[9] V. K. Gupta, R. R. (2015). "SPH-based fluid simulations on GPU." Journal of Computational Physics, 295, 104-122.