CUDA Optimization for Particle-In-Cell (PIC) Simulations

Salvatore Biamonte 264177, Federico Di Franco 263678, Domenico Visciglia 263631

University of Calabria, Department of Mathematics and Computer Science

1 Introduction

The Particle-In-Cell (PIC) methodology is a cornerstone numerical approach in computational physics, particularly for simulating plasma and particle systems. In this study, we present a CUDA-based implementation to accelerate key PIC kernels and analyze their performance using GPU parallelization. The primary objective is to demonstrate how optimized kernels can significantly improve the simulation execution time.
All optimizations presented in this work were developed starting from the baseline version, rather than incrementally building upon successive optimizations. This approach ensures a clear and independent evaluation of the impact of each optimization technique on performance.

2 Performance Evaluation

We benchmarked all kernels on an **NVIDIA V100** GPU using NVIDIA Nsight Compute. The analysis focuses on the following metrics:

- **Execution Time:** Total duration required for kernel execution.
- **Arithmetic Intensity:** A measure of computational operations relative to memory access.
- **Performance:** Measured in FLOPS (floating-point operations per second).
- **Warp Occupancy:** Percentage of active threads relative to hardware limits.

Our investigation highlights the impact of kernel optimizations on these metrics, with a particular focus on ‘updateParticleVelocityKernel’ and its role in reducing the overall runtime.

The execution time was obtained by running each kernel four times and selecting the best execution time. For the performance metrics reported in Tables 2 and 3, each configuration was profiled ten times for each particle species, resulting in a total of twenty runs per configuration. This methodology ensures robust and representative performance measurements.

3 Primary Kernel: Update Particle Velocity

3.1 Overview

The `updateParticleVelocityKernel` is a fundamental step in the PIC simulation loop. This kernel interpolates electric field values at particle positions and updates their velocities using these values and a time step dt.

Each GPU thread processes a particle independently, exploiting CUDA’s parallelism to handle millions of particles simultaneously. Interpolation involves calculating weights based on the particle’s position relative to grid nodes, ensuring accurate velocity updates.

3.2 Performance Metrics

The kernel demonstrates excellent warp occupancy and scalability, but its performance is memory-bound due to frequent memory accesses during interpolation.

Table 1. Performance Metrics for updateParticleVelocityKernel (standard implementation)

Block Size	Time(μs)	AI(FLOP/byte)	Perf.(FLOP/s)	Occupancy(%)
128x1x1	153.79	1.79	1.363.628.797.336	67.62
256x1x1	153.34	1.79	1.367.612.687.813	65.09
512x1x1	154.11	1.79	1.360.797.342.192	62.85
1024x1x1	179.42	1.79	1.168.824.683.431	43.32

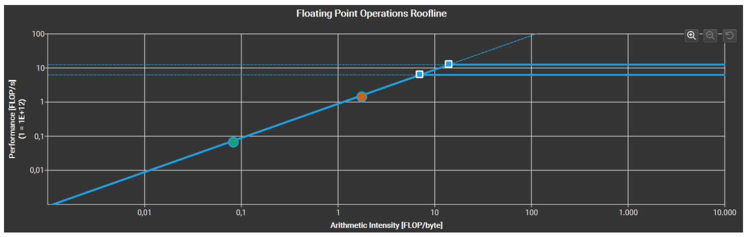


Fig. 1. Roofline Model for `updateParticleVelocityKernel` with 256x1x1 block configuration

4 Particles-to-Grid Kernel: Comparison and Optimization

The `particles2GridKernel` maps particle charge densities onto a structured grid. While straightforward, frequent atomic operations in the baseline version hinder performance. We evaluate three optimized strategies that aim to minimize atomic operations:

4.1 Baseline Implementation

The default implementation performs atomic additions directly on grid cells, incurring significant memory contention.

4.2 Privatization Technique

This strategy introduces local arrays within each thread block, where intermediate particle contributions are accumulated. Atomic updates manage concurrent writes to these private copies, reducing contention since these operations are confined to the thread block. Global atomic updates occur only during the final merging process, when the accumulated local sums are committed to the original array.

4.3 Thread Coarsening

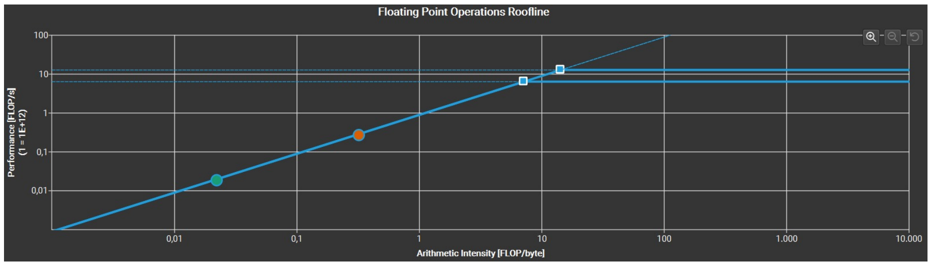
In the coarsening optimization, each thread processes multiple particles. This reduces thread count and mitigates contention, improving memory locality and arithmetic intensity.

4.4 Aggregation Strategy

Here, multiple updates to the same grid node are combined into a single atomic operation. This approach significantly reduces the total number of atomic operations, delivering the largest performance improvement.

Table 2. Performance Metrics for Particles-to-Grid Kernel Variants (standard implementation)

Variant	Block Size	Time(ms)	AI(FLOP/byte)	Perf.(FLOP/s)	Occupancy(%)
Baseline	128x1x1	0,705	0,91	202.096.093.643	89,65
Privatization	128x1x1	4,16	0,04	34.231.174.160	74,01
Coarsening	128x1x1	0,549	0,32	259.610.677.235	30,91
Aggregation	128x1x1	0,684	2,20	208.120.356.959	69,09
Baseline	256x1x1	0,708	0,90	201.165.153.773	88,97
Privatization	256x1x1	2,18	0,08	65.299.420.948	73,84
Coarsening	256x1x1	0,549	0,32	259.489.688.919	31,14
Aggregation	256x1x1	0,713	2,21	199.883.329.593	66,46
Baseline	512x1x1	0,707	0,88	201.583.925.419	85,63
Privatization	512x1x1	1,20	0,16	118.499.600.957	72,97
Coarsening	512x1x1	0,570	0,34	250.078.598.697	31,64
Aggregation	512x1x1	0,672	2,22	211.902.383.330	61,83
Baseline	1024x1x1	0,638	0,85	223.289.387.939	85,03
Privatization	1024x1x1	0,868	0,30	164.181.194.943	49,14
Coarsening	1024x1x1	0,619	0,33	230.153.973.338	49,28
Aggregation	1024x1x1	0,675	2,54	211.038.991.803	43,51



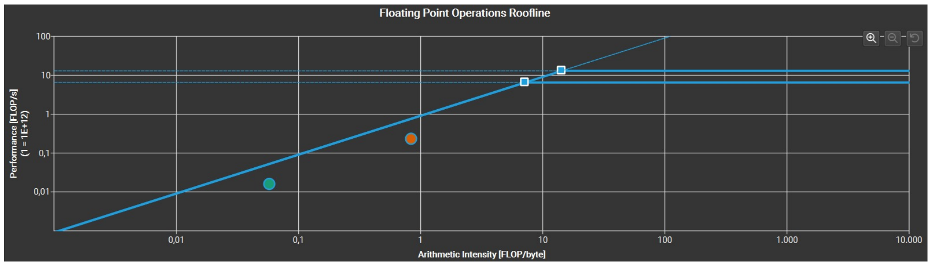


Fig. 3. Roofline Model for Baseline `particles2GridKernel` with 1024x1x1 block configuration (standard implementation)

Table 3. Performance Metrics for Particles-to-Grid Kernel Variants (unified memory implementation)

Variant	Block Size	Time(ms)	AI(FLOP/byte)	Perf.(FLOP/s)	Occupancy(%)
Baseline	128x1x1	0,763	0,91	186.704.669.293	69,61
Privatization	128x1x1	4,16	0,04	34.274.106.675	74,12
Coarsening	128x1x1	0,604	0,32	235.620.206.294	30,89
Aggregation	128x1x1	0,704	2,20	202.426.721.199	69,33
Baseline	256x1x1	0,796	0,91	178.927.495.481	67,82
Privatization	256x1x1	2,19	0,08	64.955.669.622	73,76
Coarsening	256x1x1	0,606	0,32	235.159.961.989	31,12
Aggregation	256x1x1	0,728	2,21	195.746.176.832	67,32
Baseline	512x1x1	0,721	0,91	197.639.542.106	64,45
Privatization	512x1x1	1,20	0,16	118.370.492.413	72,83
Coarsening	512x1x1	0,624	0,32	228.161.655.483	31,53
Aggregation	512x1x1	0,670	2,21	212.600.229.095	65,13
Baseline	1024x1x1	0,761	0,95	187.293.444.897	45,73
Privatization	1024x1x1	0,851	0,30	167.370.556.849	49,20
Coarsening	1024x1x1	0,657	0,32	216.632.623.285	48,49
Aggregation	1024x1x1	0,687	2,53	207.412.926.056	45,79

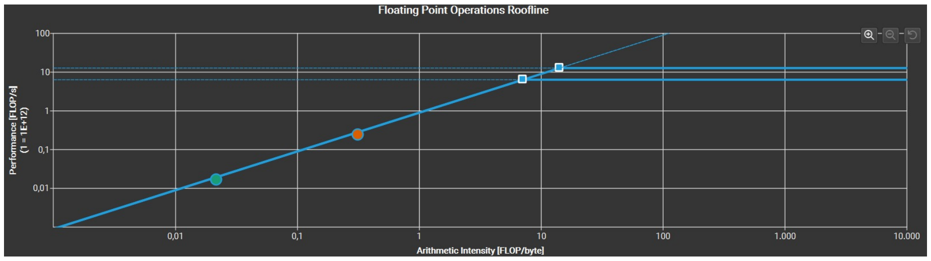


Fig. 4. Roofline Model for `particles2GridKernelCoarsening` with 128x1x1 block configuration (unified memory implementation)

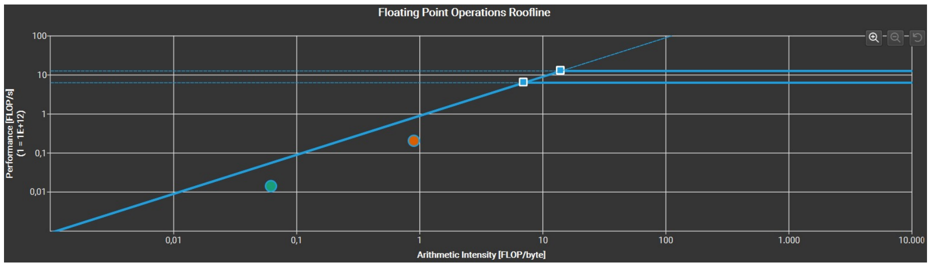


Fig. 5. Roofline Model for Baseline `particles2GridKernel` with 512x1x1 block configuration (unified memory implementation)

5 Particles2GridKernel: Comparison and Optimization

The `particles2GridKernel` accumulates particle contributions (e.g., charge density) onto a structured grid. A direct implementation requires atomic operations to increment grid cells, leading to contention and reduced performance. We consider four variants—Baseline, Privatization, Coarsening, and Aggregation—to mitigate these issues.

5.1 Baseline Implementation

In the baseline approach, each thread processes a single particle and directly updates the corresponding grid nodes via `atomicAdd()` operations provided by CUDA. This method achieves high occupancy, as each thread requires minimal resources and contributes actively to the computation. However, the frequent use of `atomicAdd()` on shared grid nodes results in severe contention, where multiple threads attempt to update the same memory locations simultaneously. This contention leads to significant waiting times for threads, reducing the effective performance despite the seemingly favorable occupancy metrics.

From the performance metrics in Table 2, the baseline implementation achieves an execution time of 0.705 ms for a block size of 128x1x1, with an arithmetic intensity (AI) of 0.91 and a performance of 202.1 GFLOP/s, coupled with an occupancy of 89.65%. While these results indicate reasonable throughput, the high number of atomic operations limits the kernel's scalability and efficiency. Despite these limitations, the baseline serves as a robust starting point for evaluating further optimizations, such as privatization, coarsening, and aggregation.

5.2 Privatization

In the privatization approach, we replicate the charge density array `rhos` within each thread block. This means that, for each block, the threads perform atomic additions to a private copy of the array, thus reducing contention on the global array. Once a block has processed its assigned particles, the partial results accumulated in its private array are merged back into the global charge density array.

From Tables 2 and 3, it is evident that this method, while reducing contention on global memory, does not always translate into better performance. For instance, in the standard implementation, the privatization approach results in significantly longer execution times compared to the baseline (e.g., 4.16 ms vs. 0.705 ms for block size 128x1x1). The arithmetic intensity (AI) also drops to very low values (0.04 for block size 128x1x1), indicating an increased memory overhead due to the additional operations needed to merge the private copies back into global memory.

While privatization shows slightly improved occupancy compared to the unified baseline in some cases (e.g., 74.12% vs. 69.61% with block size 128x1x1), the overall performance improvement is not consistent. The additional logic and memory usage associated with maintaining and merging private copies often negate the expected gains from reduced contention on global atomic operations.

5.3 Coarsening

In the coarsening optimization, each thread processes multiple particles instead of just one (in both tables 2 and 3 the kernel was profiled with each thread managing 32 particles). This approach reduces the total number of threads launched by assigning a range of particle indices (`particleStart` and `particleEnd`) to each thread. For each particle in this subrange, the kernel calculates the corresponding grid cell indices, computes weighting factors, and performs atomic updates to the appropriate grid nodes.

The data from Tables 2 and 3 demonstrate the effectiveness of coarsening. For example, in the standard implementation with block size 128x1x1, the execution time drops from 0.705 ms (baseline) to 0.549 ms, making it the best optimization when it comes to time. Similarly, in the unified memory implementation, the execution time reduces from 0.763 ms to 0.604 ms, reflecting an increase in time performance. However, coarsening can lead to reduced occupancy (e.g., 30.91% for block size 128x1x1 in the standard implementation), as fewer threads are used and a reduction in AI also occurs.

Despite the lower occupancy, the overall performance improvement stems from the drastic reduction in global atomic contention. The approach balances the trade-off between computational load per thread and atomic operation efficiency, making it highly effective for kernels suffering from contention in the baseline implementation.

5.4 Aggregation

The aggregation optimization (`particles2GridKernelAggregation`) seeks to reduce the number of global atomic operations by consolidating multiple contributions to the same grid cell into a single update. Each thread processes one particle at a time and tracks the last grid cell it updated using a `prevIndex` variable. Contributions are stored incrementally in an `accumulator`, and an atomic update is only issued when the index changes or when all contributions for the current particle have been processed.

From Tables 2 and 3, the aggregation technique shows mixed results. In the standard implementation, it achieves slight improvement in time execution and a significantly higher arithmetic intensity compared to the baseline (e.g., for block size 128x1x1 0.684 ms vs. 0.705 ms and 2.20 FLOP/B vs. 0.91 FLOP/B). This improvement stems from the reduction in the number of atomic operations, which mitigates contention. However, the performance gain diminishes for larger block sizes due to the additional logic and branching required to manage the `accumulator`.

In the unified memory implementation, aggregation shows similar trends, with execution times close to the baseline and consistent improvements in arithmetic intensity. For block size 256x1x1, the time is 0.796 ms compared to 0.728 ms in the baseline, and the AI improves from 0.91 to 2.21.

While aggregation effectively reduces atomic contention, its benefits are tempered by the increased complexity of each thread's logic, making it most advantageous in scenarios where frequent atomic updates severely limit performance.

6 Evaluation of Standard vs Unified Memory particles2GridKernel Implementations

We also compare the implementations of standard and unified memory for the `particles2GridKernel` variants. Standard memory tends to deliver lower latency and more predictable performance, as data reside directly on the GPU. Unified memory, designed to simplify programming, can introduce additional overheads due to on-demand data migration and related latency, which may impact execution times.

6.1 Comparison of Metrics

From the results presented in Tables 2 and 3, unified memory implementations generally show slightly higher arithmetic intensity (AI) for some variants, such as privatization and aggregation, due to efficient data reuse. However, the overall execution time and FLOP/s performance are typically lower compared to standard memory. This is primarily because unified memory incurs additional overhead from implicit data migrations and synchronization between host and device, especially at larger block sizes (e.g., 1024x1x1), where latency effects become more pronounced. Furthermore, unified memory implementations can experience reduced effective occupancy due to these overheads, limiting their ability to fully utilize GPU resources.

6.2 Best Optimization Strategy

Among the optimization strategies evaluated, thread coarsening emerges as the most consistently effective approach. By allowing each thread to process multiple particles, coarsening significantly reduces global atomic contention and improves execution time and overall kernel performance. For instance, in both standard and unified memory implementations, coarsening achieves notable performance gains by minimizing contention and efficiently utilizing GPU resources.

While privatization and aggregation techniques can improve AI in some cases, their additional logic and memory overhead often negate the expected performance benefits, especially in unified memory implementations. The baseline kernel, despite its high occupancy, remains less efficient due to the heavy reliance on atomic operations, which introduce contention and delay.

In general, standard memory implementations outperform unified memory due to their lower latency and direct access to GPU memory. Unified memory, however, offers advantages in terms of programming simplicity and portability, making it suitable for scenarios where ease of development is prioritized over peak performance. For maximizing kernel performance in Particle-In-Cell (PIC) simulations, standard memory with careful application of thread coarsening and other optimizations remains the optimal choice.

7 Comparison between Serial and Parallel Implementations

In this section, we compare the performance, in terms of simulation execution time, of the serial and parallel implementations for Particle-In-Cell (PIC) simulations.

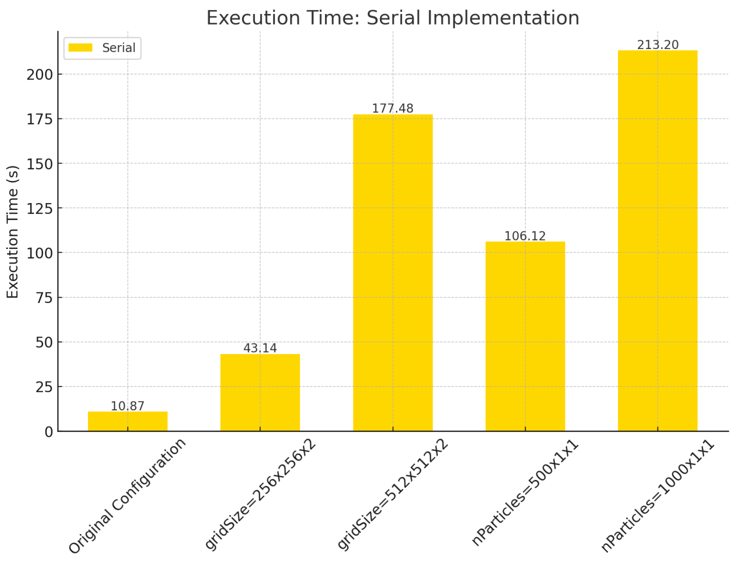


Fig. 6. Execution times of serial implementation, with varying configuration parameters

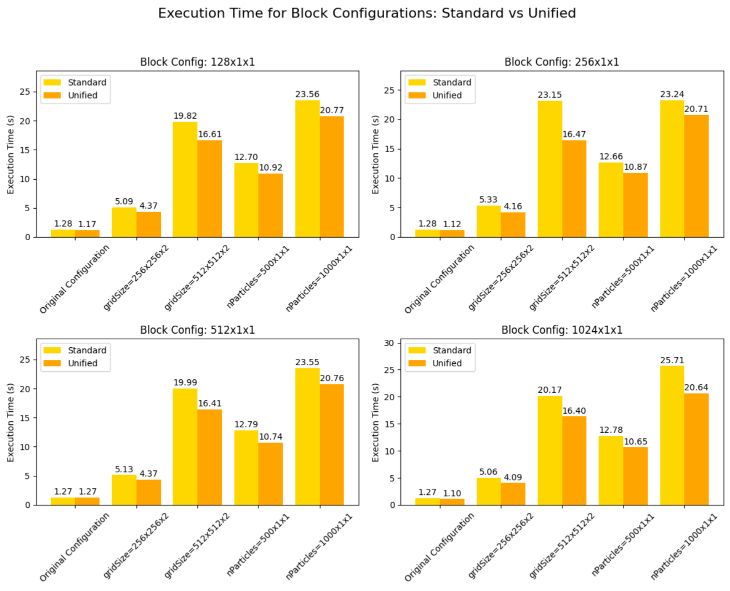


Fig. 7. Comparison between execution times of both standard and unified `baseline` implementations, with varying configuration parameters

To achieve performance improvements, we relied on the baseline implementation for the `particles2GridKernel`, focusing on improving execution time. This strategy was consistently applied across both standard and unified memory implementations, as it provided a straightforward and efficient approach to balance computational overhead and memory performance without introducing additional external factors.

Observations:

- The serial implementation displayed significantly higher execution times across all configurations. As the simulation complexity increased (e.g., larger grids or more particles per cell), the gap between the serial and parallel implementations widened substantially.
- For the `particles2GridKernel`, the standard memory implementation was faster than the unified memory implementation across all tested configurations. This can be attributed to the lower latency and direct data access of standard memory, which minimizes the overhead associated with implicit data migration and synchronization.
- However, when considering the total execution time of the entire simulation, the unified memory implementation outperformed the standard implementation. This result is likely due to unified memory's ability to simplify memory management and reduce the cost of explicit memory transfers and host-device synchronization for other kernels, thereby compensating for its slower performance in `particles2GridKernel`.

We particularly focus on the configurations with $n_x = n_y = 256$ and $n_x = n_y = 512$ since increasing the grid resolution helps achieve a finer and cleaner simulation. Larger grid sizes allow for better spatial resolution, which is critical for capturing detailed particle dynamics and minimizing numerical artifacts, leading to a more accurate representation of the physical phenomena.

By leveraging CUDA's strengths and focusing on an optimized baseline implementation, we achieved substantial speedups over the serial baseline, demonstrating the feasibility of running high-resolution PIC simulations within practical time frames.

8 Conclusion

This study highlights the effectiveness of CUDA optimizations in accelerating key kernels within Particle-In-Cell (PIC) simulations. By leveraging GPU parallelism and exploring optimization strategies, we demonstrated significant improvements in computational performance.

The ‘updateParticleVelocityKernel‘ showcased efficient utilization of CUDA's parallel architecture, achieving excellent warp occupancy and scalability. Despite being memory-bound, it effectively minimized simulation runtime by processing millions of particles simultaneously.

For the ‘particles2GridKernel‘, our analysis of optimization strategies including privatization, thread coarsening, and aggregation—revealed that coarsening consistently delivered the best results in terms of execution time across both standard and unified memory implementations. Coarsening minimized global atomic contention by consolidating operations, making it the most effective in reducing runtime. However, the aggregation optimization demonstrated a significantly higher arithmetic intensity (AI), making it an attractive choice for scenarios where computational density is prioritized over execution time. While the standard memory approach generally outperformed unified memory in the execution time of ‘particles2GridKernel‘, the unified memory implementation demonstrated superior performance in the total execution time of the entire simulation. This was primarily due to its ability to simplify memory management and reduce synchronization overhead for other kernels, compensating for its slightly slower performance in individual kernel execution.

Overall, these findings emphasize the importance of tailored memory access patterns, reduced atomic operations, and thread-level optimizations in GPU-based PIC simulations. Such strategies not only enhance computational efficiency but also enable scalability for larger and more complex particle systems. Future work could focus on hybrid approaches combining the strengths of standard and unified memory models and the various optimizations to further optimize performance for diverse simulation workloads.