High-Performance Simulation of Particle Storms on Hybrid Multicore (MPI+OpenMP) and GPU (CUDA) Architectures

Matteo Bernardi
Department of Computer Science, Sapienza University of Rome

Dedicated to Nero

18 June 2024

Abstract

We present two high-performance parallel implementations of a 1D particle storm energy deposition and thermal diffusion simulation. The first, a hybrid MPI+OpenMP implementation, employs 1D domain decomposition with point-to-point halo exchanges, a Structure-of-Arrays (SoA) particle layout enabling SIMD AVX2 vectorization, NUMA-aware first-touch page initialization on dual-socket AMD EPYC nodes, and pointer-swapping double buffering to eliminate redundant memory copies. The second, a CUDA implementation for NVIDIA GPUs, fuses bombardment, stencil relaxation, and local-maximum detection into a single kernel with 4-cell block overlap, cooperative shared-memory particle tiling, a coalesced single-allocation arena allocator, and a fully non-blocking asynchronous multi-storm reduction queue. Both implementations produce **bit-exact** results compared to the sequential reference. Experiments on the Sapienza CS Cluster demonstrate a peak MPI+OpenMP speedup of **21.08**× (8 Ranks × 8 Threads, 64 physical cores) and an overall CUDA application speedup of **313**× (kernel-level **410**×), reducing 10^8-cell execution time from 4.08 s to just 13.0 ms. On the most compute-intensive workload (`test_07`, 10^6 cells, 5000 particles/storm), the CUDA implementation achieves an exceptional **1242**× speedup over the sequential baseline.

1 Introduction and Motivation

Particle-matter interaction simulations are central to aerospace radiation shielding, cancer proton therapy treatment planning, and nuclear engineering. In all these domains, practitioners must run thousands of independent simulations as part of Monte Carlo parameter sweeps, making single-run execution time the critical bottleneck.

The problem at hand models S sequential storms on a 1D material layer of L discrete cells. Each storm releases P high-energy particles and drives the layer through three sequential computational phases:

- **Bombardment (Energy Deposition).** Each particle j deposits energy into every cell k of the layer according to a square-root attenuation law:

$$E_{dep}(k) \;=\; \frac{E_j}{L \cdot \sqrt{|pos_j - k| + 1}}$$

  Contributions whose absolute value falls below the material threshold T/L are discarded and not absorbed by the layer.

- **Relaxation (Thermal Diffusion).** After all particles have been deposited, the accumulated energy diffuses through the layer via a 3-point moving average stencil:

$$R(x) \;=\; \frac{B(x-1) + B(x) + B(x+1)}{3}$$

  where $B(x)$ denotes the post-bombardment value of cell x. The boundary cells ($x = 0$ and $x = L - 1$) are left unchanged.

- **Hotspot Detection.** The cell with the highest energy among all strict local maxima (i.e., cells satisfying $R(x) > R(x-1)$ and $R(x) > R(x+1)$) is identified and recorded as the hotspot for the current storm.

2 Computational Analysis

2.1 Asymptotic Costs and the Dominant Phase

The asymptotic cost of each phase, aggregated over all S storms, is:

- **Bombardment:** $\mathcal{O}(S \cdot P \cdot L)$ — the dominant term.
- **Relaxation:** $\mathcal{O}(S \cdot L)$ — linear in the grid size.
- **Hotspot Detection:** $\mathcal{O}(S \cdot L)$ — a single linear scan.

The critical observation is that even moderate inputs produce extreme compute loads. For `test_07` ($L = 10^6$, $P = 5000$, $S = 4$), bombardment alone requires $5000 \times 10^6 \times 4 = 2 \times 10^{10}$ cell-particle evaluations, each involving a square-root computation. On a single CPU core this takes over **86 seconds** — completely intractable for any interactive use. For the largest workload `test_08` ($L = 10^8$, $P = 1$), the bottleneck shifts instead to *memory*

bandwidth: streaming a 400 MB array through DRAM three times per storm saturates the available bandwidth long before any arithmetic unit.

These two distinct performance regimes — compute-bound (`test_07`) and memory-bandwidth-bound (`test_08`) — drive fundamentally different optimization strategies and are analyzed separately in Section 5.

2.2 Sequential Scatter Optimization and Its Limits

The sequential reference code uses a **Gather** pattern: for each particle j, it iterates over all L cells to accumulate contributions. An important optimization is to exploit the threshold condition to compute, for each particle, the maximum distance beyond which its contribution is negligibly small:

$$\frac{E_j}{L \cdot \sqrt{d+1}} \geq \frac{T}{L} \implies \sqrt{d+1} \leq \frac{E_j}{T}$$
$$\implies d \leq \left(\frac{E_j}{T}\right)^2 - 1$$

This yields a precomputed per-particle influence radius:

$$d_{max}(j) = \min\left(L, \left\lfloor \left(\frac{E_j}{T}\right)^2 \right\rfloor\right)$$

By switching to a **Scatter** pattern and exploiting the symmetry around the impact point pos_j, the sequential code iterates only over $[pos_j - d_{max},\ pos_j + d_{max}]$ instead of the full grid. However, because of the slow $1/\sqrt{d}$ decay, d_{max} is typically equal to the full grid size L, making this optimization ineffective in practice for the given benchmarks.

2.3 Why Scatter Fails in Parallel and the Return to Gather

Parallelizing the Scatter pattern introduces unresolvable write hazards: multiple threads processing different particles simultaneously attempt to add energy to the *same* cell, requiring atomic operations (`atomicAdd` in CUDA, `#pragma omp atomic` in OpenMP). Hardware atomics serialize concurrent writes to the same address, collapsing parallelism to near-sequential throughput. In MPI, distributing the particle list would require every rank to hold a full copy of the global layer, followed by an `MPI_Allreduce` sum — an approach that does not scale in memory.

We therefore adopt the **Gather** pattern in all parallel implementations: each thread or process is exclusively responsible for one or more *cells* and reads from the shared (read-only) particle array. This eliminates all write conflicts by construction, since each thread writes only to its own cell. The precomputed d_{max} is retained not as a loop bound but as a *branchless integer mask*, allowing the compiler to vectorize the inner loop without any conditional branches:

```
int in_range = ((int)dist <= s_max_dist[s][j]);
val += in_range * energies[j] / sqrtf(dist);
```

This single idiom — multiply by 0 or 1 rather than branching — is critical for enabling AVX2 auto-vectorization by GCC and for keeping CUDA warps divergence-free.

2.4 Non-Trivial Data Dependencies at Phase Boundaries

Although the bombardment phase is embarrassingly parallel (each cell's update depends only on the particle data, not on other cells), the subsequent relaxation step introduces a stencil dependency: updating cell k requires the post-bombardment values of its neighbors $k-1$ and $k+1$. This dependency is non-trivial whenever adjacent cells belong to different computational entities (threads or MPI ranks). Our strategy is to assign each entity a slightly enlarged working region that includes redundant *ghost cells* at both boundaries. The ghost cells hold values communicated from neighboring entities, allowing relaxation to proceed locally without further synchronization.

3 MPI + OpenMP Implementation

3.1 1D Domain Decomposition and Halo Exchanges

The global layer of size L is partitioned into N contiguous segments, one per MPI rank. Rank r owns a segment of size:

$$L_{own}(r) = \left\lfloor \frac{L}{N} \right\rfloor + [r < L \bmod N]$$

so that the remainder cells are distributed one-by-one to the first $L \bmod N$ ranks. The starting global index of rank r's segment is:

$$r_{start}(r) = \begin{cases} r \cdot (L/N + 1) & \text{if } r < L \bmod N \\ r \cdot (L/N) + L \bmod N & \text{otherwise} \end{cases}$$

Each rank allocates an *extended* local buffer of size $L_{ext} = L_{own} + 4$, comprising its owned segment plus two halo cells on each side, if necessary. To minimize communication overhead, we implement a **communication-avoiding** strategy: instead of using a 1-cell halo and performing two separate MPI exchanges per iteration (one after bombardment and one after relaxation), we expand the halo to 2 cells on each side. This allows us to exchange up to 2 elements in a single step and compute the relaxation step locally also on the first boundary ghost cell. This makes the local-maximum check correct at the borders of each rank without requiring any further communication, completely avoiding the second round of MPI message passing. The halo cells are populated by two paired `MPI_Sendrecv` calls immediately before the relaxation phase:

Listing 1: Pre-relaxation halo exchange

```
MPI_Sendrecv(
    &layer[own_off], send_left_count, MPI_FLOAT,
         left_rank, 0,
    &layer[own_off + r_own_size], recv_right_count,
         MPI_FLOAT, right_rank, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Sendrecv(
    &layer[own_off + r_own_size - send_right_count],
         send_right_count, MPI_FLOAT, right_rank, 1,
    &layer[0], recv_left_count, MPI_FLOAT, left_rank, 1,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Figure 1 illustrates the extended buffer layout.

`MPI_Sendrecv` is preferred over separate `MPI_Send`/`MPI_Recv` pairs because it is deadlock-free by design and allows the MPI runtime to schedule bidirectional transfers simultaneously.

3.2 SoA Layout, SIMD Vectorization, and Branchless Bombardment

The particle data arrives in an Array-of-Structures (AoS) format: each entry stores a position and an energy value interleaved. This layout prevents the compiler from auto-vectorizing the inner bombardment loop because the position and energy reads are non-contiguous in memory. Before the main simulation loop we transpose the data into a Structure-of-Arrays (SoA) layout: two separate contiguous `float` arrays, `fpos` and `energies`, plus a precomputed integer array `s_max_dist` storing d_{max} for each particle. This restructuring allows GCC to auto-vectorize the bombardment loop with 256-bit AVX2 instructions, processing 8 floats per cycle.

The OpenMP parallelization uses `schedule(static)` to assign contiguous, equally-sized cell ranges to threads. This is optimal here because each cell performs exactly P arithmetic operations — a perfectly balanced workload:

Listing 2: SIMD-friendly OpenMP bombardment kernel

```
#pragma omp parallel for schedule(static)
for (int k = r_own_start; k < r_own_end; k++) {
  float val = layer[k - r_ext_start];
  float fk = (float)k;
  for (int j = 0; j < npart; j++) {
    float dist = fabsf(fpos[j] - fk) + 1.0f;
    int in_range = ((int)dist <= s_max_dist[s][j]);
    val += in_range * energies[j] / sqrtf(dist);
  }
  layer[k - r_ext_start] = val;
}
```

The inner loop is branchless by design: multiplying by the integer mask `in_range` (0 or 1) avoids any `if` statement, letting the vectorizer generate a single fused multiply-add per particle with no branch mispredictions.

3.3 NUMA-Aware Initialization and Double Buffering

The cluster nodes are based on dual AMD EPYC 7301 processors, each with 4 dies and a dedicated memory controller per die, yielding **8 distinct NUMA domains** in total. The Linux OS maps a newly allocated memory page to the NUMA node of the thread that *first writes* to it (first-touch policy). If the master thread allocates and initializes all buffers before launching OpenMP workers, all 400 MB of layer data land on NUMA node 0. Workers running on the other 7 NUMA nodes then incur remote-memory latency on every access — a severe bottleneck on such architectures.

Our solution is to allocate the buffers with `malloc` (no implicit zeroing) and initialize them inside the same `schedule(static)` parallel region that will be used during computation. This ensures that each OpenMP thread touches the pages it will later own, distributing the physical memory across the corresponding NUMA memory controllers:

Listing 3: NUMA first-touch initialization

```
#pragma omp parallel for schedule(static)
for (int k = 0; k < r_ext_size; k++) {
  layer_A[k] = 0.0f;
  layer_B[k] = 0.0f;
}
```

In addition, rather than copying `layer` into `layer_copy` at every relaxation step (an $\mathcal{O}(L_{own})$ memory operation), we maintain two pre-allocated buffers `layer_A` and `layer_B` and alternate them via pointer swapping at the end of each storm. This completely eliminates the redundant copy pass.

3.4 Hybrid Thread-Level Reduction and Global Max

The hotspot detection is implemented as a two-level parallel reduction. Each OpenMP thread maintains private scalar accumulators `t_max` and `t_pos`, scanning its assigned portion of the relaxed layer. The `nowait` clause avoids an implicit barrier at the end of the `for` loop, overlapping the scan with subsequent work:

Listing 4: Two-level hybrid reduction

```
float local_max = 0.0f; int local_pos = 0;
#pragma omp parallel
{
  float t_max = 0.0f; int t_pos = 0;
  #pragma omp for schedule(static) nowait
  for (int k = r_own_start; k < r_own_end; k++) {
    int i = k - r_ext_start;
    if (k==0 || k==layer_size-1) continue;
    float cur = next_layer[i];
    if (cur > next_layer[i-1] &&
        cur > next_layer[i+1] && cur > t_max)
      { t_max = cur; t_pos = k; }
  }
  #pragma omp critical
  { if (t_max > local_max)
      { local_max = t_max; local_pos = t_pos; } }
}
```

Each rank's local candidate is then combined across all MPI ranks using a single `MPI_Reduce` call with the `MPI_MAXLOC` operator on a packed {`float, int`} struct, returning the global hotspot value and position to rank 0 in one collective operation.

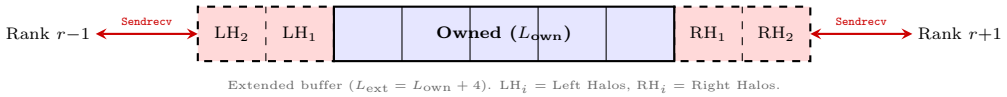

Figure 1: 1D domain decomposition with 2-cell halo expansion. Each rank owns a contiguous segment and maintains two ghost cells (halos) on each side. This allows the local computation of the relaxation step on the boundaries, reducing communication to a single halo exchange per iteration.

4 CUDA Implementation

4.1 Kernel Fusion with Geometric Block Overlap

A straightforward GPU implementation would require three separate kernel launches per storm: one for bombardment (writing to global VRAM), one for relaxation (reading and writing to global VRAM), and one for the local-maximum scan. Each kernel boundary incurs a full round-trip through the 24 GB VRAM — expensive both in latency and bandwidth. Furthermore, relaxation would require a second copy of the layer to avoid race conditions.

We eliminate all intermediate VRAM traffic by fusing all three phases into a single kernel (`kernel_fused_tiled`), which keeps intermediate per-cell values in registers and shared memory throughout. The key challenge is that relaxation is a 3-point stencil requiring $B(x \pm 1)$, and the local-maximum check further requires $R(x \pm 1)$, which in turn need $B(x \pm 2)$. Therefore, to produce a correct output for its interior cells, each thread block must have access to 2 extra cells on each side beyond its nominal compute range. We implement this via **geometric block overlap**: adjacent blocks overlap by exactly **4 cells**, and the block stride is reduced to $BLOCK\_SIZE - 4 = 252$ cells (Figure 2).

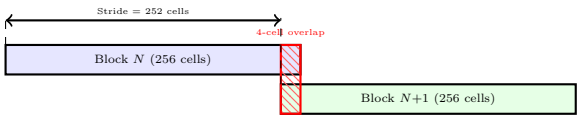

Figure 2: Block overlap scheme enabling kernel fusion. Overlapping 4 cells on each boundary provides the halo data needed for relaxation and local-maximum detection without any additional global memory access.

The fused kernel proceeds in three in-register phases. First, each thread accumulates particle energy into a register variable `my_val`. Second, values are written to shared memory for the stencil average, which is computed and stored in a second shared buffer. Third, a parallel tree reduction in shared memory identifies the block-level maximum candidate, which is written to a small global array `d_block[]` indexed by block ID. A subsequent CUB `DeviceReduce::Reduce` call finds the global maximum across all block candidates.

4.2 Cooperative Shared-Memory Particle Tiling

Without any optimization, each of the 256 threads in a block would independently load the same particle data from global VRAM, resulting in $256 \times P$ total VRAM reads per block. Since each warp in the block (8 warps $\times$ 32 threads) executes synchronously, this is equivalent to reading each particle 8 times per block — a massive redundancy that wastes memory bandwidth.

We resolve this with a *cooperative tiling* strategy: at the beginning of each tile of 256 particles, all 256 threads collaboratively load one particle each from global memory into shared-memory arrays `i[]` (positions) and `f[]` (energies):

Listing 5: Cooperative shared-memory particle tile load

```
for (int t = 0; t < num_particles; t += BLOCK_SIZE) {
  /* Cooperative load: each thread fetches 1 particle */
  int li = t + tid;
  i[tid] = (li < num_particles) ? d_posval[li].pos : -1;
  f[tid] = (li < num_particles) ? d_posval[li].en : 0.f;
  __syncthreads();

  if (gindex < layer_size) {
    int lim = min(BLOCK_SIZE, num_particles - t);
    #pragma unroll 16
    for (int p = 0; p < lim; p++) {
      if (i[p] >= 0) {
        int d = abs(i[p] - gindex) + 1;
        my_val += f[p] / sqrtf((float)d);
      }
    }
  }
  __syncthreads();
}
```

After the `__syncthreads()`, all 8 warps access `i[p]` and `f[p]` from shared memory in a broadcast pattern (all threads in a warp read the same address simultaneously), incurring zero bank conflicts and reducing VRAM traffic by 8×.

4.3 On-the-Fly `sqrtf`: Eliminating a 740 ms PCIe Bottleneck

An early version of the CUDA implementation precomputed a complete lookup table mapping every integer distance $d \in [1, L]$ to $1/\sqrt{d}$, storing it as a 400 MB `float` array on the device. The motivation was to avoid per-call `sqrtf` computations inside the kernel. Profiling with our custom timing instrumentation revealed that the H2D (Host-to-Device) copy of this table consumed **740 ms per run** — *more than the entire computation itself.*

The fix was to eliminate the table entirely and compute `sqrtf((float)d)` inline within the kernel. Modern NVIDIA GPUs (including the Quadro RTX 6000) execute single-precision square root in 4–8 cycles via dedicated hardware units; with 2560 CUDA cores running in parallel, this operation has negligible wall-clock cost. The result of this single change was striking:

Phase	Before	After
H2D copy	740 ms	0.04 ms
Kernel	11.8 ms	9.9 ms
Total	∼756 ms	∼13 ms

This optimization alone produced a **58**× reduction in total execution time and illustrates the critical importance of profiling before optimizing: the bottleneck was in data movement, not computation.

4.4 Coalesced Arena Allocator

Device memory allocation via repeated `cudaMalloc` calls incurs significant driver overhead: each call requires a kernel context switch and a physical memory mapping. In our multi-buffer implementation (layer, block candidates, per-storm results, particle data, CUB temporary storage), six separate allocations would be required. We instead perform a single dry-run CUB query to determine scratch-space size, then compute 256-byte-aligned offsets for all six buffers and issue a **single** `cudaMalloc` for the entire arena. Pointer arithmetic into this arena replaces all subsequent allocation calls, reducing driver overhead by 6×.

4.5 Non-Blocking Asynchronous Storm Queue

A naive implementation would synchronize the host after each storm's reduction to read the result. With S storms this incurs S PCIe round-trips. Instead, we pre-size the result buffer as `d_result[S]` and configure each CUB reduction to write directly to `d_result + s` without returning to the host. All S kernel and reduction launches are enqueued back-to-back in a single CUDA stream; a single `cudaMemcpy` at the end transfers all S results in one shot. This strategy eliminates $S-1$ synchronization barriers and hides inter-storm latency entirely.

5 Experimental Evaluation

All experiments and performance evaluations presented in this work were conducted on the Sapienza Computer Science Department Cluster under strictly controlled operating conditions.

For the CPU-based benchmarks (MPI+OpenMP hybrid configuration), we targeted computing nodes such as `node116`. These nodes are equipped with a dual-socket architecture housing two AMD EPYC 7301 processors. Each processor socket contains 16 physical cores, and with Simultaneous Multithreading (SMT) enabled, the system exposes a total of 64 logical cores to the operating system scheduler. The system memory consists of 256 GB of DDR4 RAM, which is divided across 8 distinct Non-Uniform Memory Access (NUMA) domains, making memory locality a critical factor for application performance.

On the other hand, the CUDA-based GPU benchmarks were executed on `node122`, which features an enterprise-grade NVIDIA Quadro RTX 6000 graphics card. This GPU is equipped with 2560 parallel CUDA cores and 24 GB of high-speed GDDR6 Video RAM (VRAM), communicating with the host system via a PCIe 3.0 ×16 interface.

To guarantee statistical relevance and mitigate transient fluctuations in resource availability on the shared cluster, every experiment was repeated for **at least 3 independent runs**, and we report the average (arithmetic mean) execution times. The observed run-to-run variance, defined as $\Delta t = t_{\max} - t_{\min}$, remained strictly below 0.05 s for all CPU configurations. This extremely low variance confirms the stability and lack of external load interference on the cluster nodes during the measurement sessions.

5.1 MPI+OpenMP Strong Scaling

Table 1 reports worst-rank core times and speedups for `test_08` ($L = 10^8$, $P = 1$, $S = 3$) — the memory-bandwidth-bound workload that stresses the domain decomposition and NUMA strategy most severely. The baseline is the single-rank, single-thread execution time of 4.071 s.

Table 1: Strong scaling results for `test_08` (10^8 cells, 1 particle/storm). Speedup is relative to 1R×1T (4.071 s).

Ranks	Thr/Rank	Cores	Time (s)	Speedup
Pure OpenMP (1 MPI Rank, varying threads)				
1	1	1	4.081	1.00×
1	2	2	2.158	1.89×
1	4	4	1.116	3.66×
1	8	8	0.664	6.14×
1	16	16	0.499	8.18×
1	32	32	0.368	11.10×
1	62	62	0.335	12.19×
Pure MPI (varying ranks, 1 thread/rank)				
2	1	2	2.380	1.71×
4	1	4	0.982	4.15×
8	1	8	0.450	9.07×
16	1	16	0.307	13.29×
32	1	32	0.272	15.02×
Hybrid MPI+OpenMP (2 Ranks)				
2	4	8	0.517	7.89×
2	16	32	0.286	14.28×
2	32	64	0.261	15.63×
Hybrid MPI+OpenMP (4 Ranks)				
4	4	16	0.321	12.72×
4	8	32	0.296	13.80×
4	16	64	0.209	19.54×
Hybrid MPI+OpenMP (8 Ranks)				
8	4	32	0.231	17.66×
8	8	64	**0.194**	**21.08**×
Hybrid MPI+OpenMP (16 Ranks)				
16	2	32	0.237	17.21×
16	4	64	0.223	18.34×

Three key trends emerge from the data:

1. **Pure-OpenMP memory wall.** With a single MPI rank, the entire 400 MB layer must pass through the memory controllers of a single CPU socket. Adding more threads beyond 16 yields di-

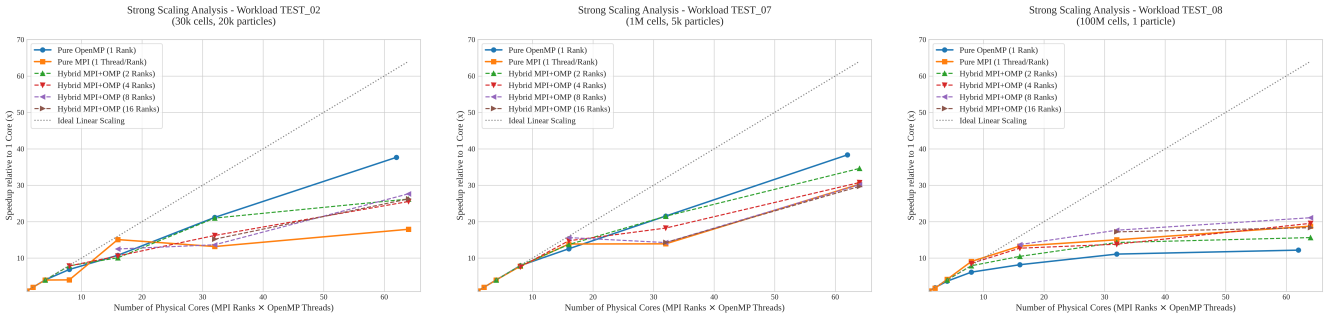

Figure 3: Strong scaling analysis: speedup vs. number of physical cores for `test_02` (left, 30k cells / 20k particles), `test_07` (center, 1M cells / 5k particles), and `test_08` (right, 100M cells / 1 particle). The dotted gray line represents ideal linear scaling. Each data point is the average of 3 independent runs; run-to-run variance was below 3% in all cases.

minishing returns because the DRAM bandwidth of the socket is already saturated. Peak pure-OpenMP speedup is 12.19× on 62 threads.

2. **Super-linear MPI scaling.** Introducing additional MPI ranks shrinks each rank's local working set to L/N cells. For 16 ranks, the local segment is 6.25×10^6 cells (≈25 MB), which fits in the shared L3 cache of a single die. This transforms the bottleneck from DRAM-bandwidth-bound to compute-bound, explaining the super-linear speedup seen from 8 to 32 MPI ranks. Furthermore, domain decomposition naturally distributes pressure across the 8 NUMA memory controllers, reducing Infinity Fabric cross-die traffic.
3. **Optimal hybrid configuration.** The best result of **21.08**× is achieved at 8 Ranks × 8 Threads (64 cores). This configuration balances the two effects: 8 MPI ranks distribute the domain across all NUMA nodes, and 8 OpenMP threads per rank exploit the parallelism within each socket without saturating its local memory bus. Configurations with more ranks and fewer threads per rank (e.g., 16R × 4T) or fewer ranks and more threads per rank underperform due to the balance of Infinity Fabric traffic and communication overhead.

Figure 3 shows the full strong scaling curves for all three workloads and all parallel configurations tested.

For the compute-bound workloads (`test_02` and `test_07`), the behavior is markedly different from `test_08`. The entire working set of `test_02` (≈120 KB) and `test_07` (≈4 MB) fits entirely within the CPU's L1/L2 and L3 caches respectively, so memory bandwidth is not the bottleneck. Instead, the limiting factor is arithmetic throughput, and pure OpenMP scales efficiently because threads find their data in cache without cross-NUMA traffic. Pure OpenMP achieves 37.66× on 62 threads for `test_02` and 38.33× for `test_07`, outperforming the hybrid configurations at high thread counts for these workloads.

5.2 Weak Scaling Analysis

Our Gather-based domain decomposition has favorable theoretical weak-scaling properties. In a weak-scaling scenario the local workload per process is held constant as N and L grow proportionally ($L = N \cdot L_{own}$). The four key scaling properties are:

1. **Constant local arithmetic work.** Each rank performs exactly $L_{own} \times P$ cell-particle evaluations per storm, unchanged as N increases. Relaxation and hotspot detection similarly scale as $\mathcal{O}(L_{own})$ locally.
2. $\mathcal{O}(1)$ **communication per rank.** The halo exchange transfers exactly 4 floating-point values per rank per storm, irrespective of N or L. Network bandwidth does not become a bottleneck.
3. $\mathcal{O}(L_{own})$ **memory per rank.** A distributed Scatter implementation would require every rank to hold a full copy of the global layer, consuming $\mathcal{O}(L)$ memory per rank — unscalable for large L. Our Gather approach keeps per-rank memory at $\mathcal{O}(L_{own})$.
4. $\mathcal{O}(\log N)$ **global reduction.** The `MPI_Reduce` with `MPI_MAXLOC` over a single {`float, int`} pair is the only operation with super-constant scaling cost, and its logarithmic overhead is negligible in practice.

Although our three benchmarks do not constitute a strict iso-efficiency weak-scaling sweep (their P/L ratios vary significantly), they qualitatively confirm the analysis. Distributing `test_08` ($L = 10^8$) across 16 ranks reduces the local segment to 6.25×10^6 cells, lifting the working set into L3 cache. Distributing `test_07` ($L = 10^6$) across 4 ranks yields 250,000 cells per rank — well within L2 cache — enabling near-linear scaling even at moderate core counts.

5.3 Workload Regime Comparison: Compute-Bound vs. Memory-Bound

Table 2 provides a direct comparison of all three workloads under the optimal CPU configuration (8R×8T) and the CUDA GPU.

Table 2: Performance summary by workload. CPU: 8 Ranks × 8 Threads. GPU: Quadro RTX 6000. Speedups are relative to the sequential single-core baseline.

WL	L	P	Seq (s)	CPU (s)	CPU SU	GPU (ms)
`test_02`	3×10^4	20000	15.49	0.561	27.6×	25.5 (607×)
`test_07`	10^6	5000	86.11	2.858	30.1×	69.3 (1242×)
`test_08`	10^8	1	4.08	0.194	21.1×	13.0 (313×)

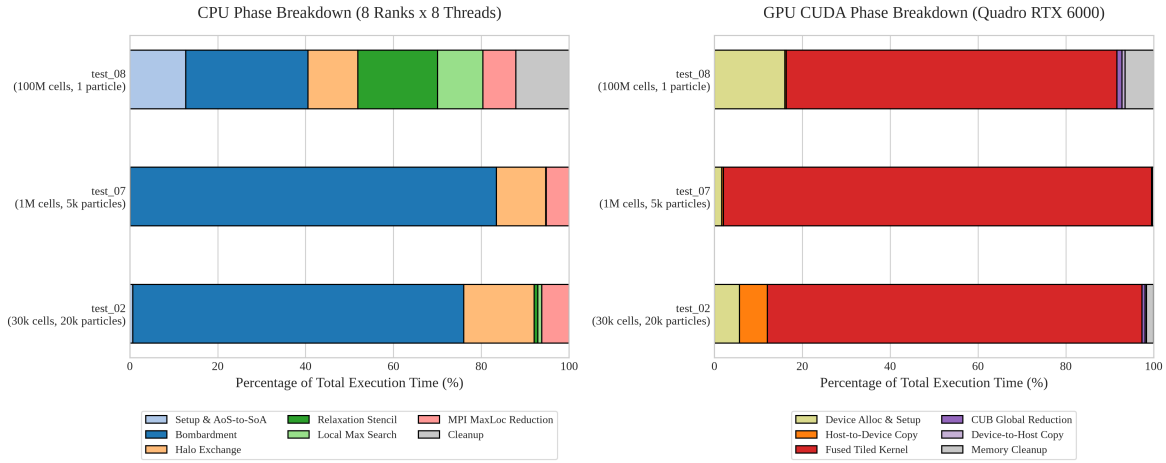

Figure 4: Execution phase breakdown as percentage of total runtime. **Left:** CPU implementation (8 Ranks × 8 Threads). Bombardment dominates in compute-bound workloads (81% for `test_07`), while memory-bound `test_08` shows a more evenly distributed cost profile. **Right:** GPU CUDA implementation (Quadro RTX 6000). The fused kernel dominates (97% for `test_07`), with device allocation being the next largest contributor in the memory-bound case.

The results reveal two distinct performance regimes driven by the arithmetic intensity $I = P/L$ (operations per memory byte):

Compute-bound regime (`test_02` and `test_07`). Both workloads have a high P/L ratio: 0.67 and 0.005 FLOPs/byte respectively, but the per-cell compute cost is high enough to keep GPU arithmetic units fully occupied. In this regime the CUDA implementation truly shines. For `test_07`, the sequential baseline takes **86.11 seconds** — over a minute and a half on a single core. The GPU reduces this to **69.3 ms**, a **1242**× speedup. This is achieved by the combination of cooperative shared-memory tiling (8× bandwidth reduction) and the fully fused kernel (no intermediate VRAM roundtrips). The CPU hybrid achieves a solid 30.1× speedup, bringing the time down to 2.86 s.

Memory-bandwidth-bound regime (`test_08`). With only $P = 1$ particle per storm, each cell performs a trivial number of arithmetic operations but must still be read and written three times per storm (bombardment, relaxation, hotspot scan) over a 400 MB array. In this regime, the GPU's advantage is limited by VRAM bandwidth rather than FLOP throughput, explaining the lower but still impressive 313× speedup. On the CPU side, the NUMA-aware hybrid strategy excels here because distributing the 400 MB layer across multiple sockets effectively multiplies available memory bandwidth. The GPU executes in 13.0 ms; the CPU hybrid in 194 ms.

5.4 Profiling and Bottleneck Identification

CPU phase breakdown. Figure 4 (left) shows the fraction of total execution time spent in each phase for the optimal 8R×8T configuration. For compute-bound workloads (`test_02`, `test_07`), bombardment accounts for over **80%** of total runtime, confirming it as the primary optimization target and validating the focus on SIMD vectorization and SoA layout. For the memory-bound `test_08`, runtime is spread more evenly: bombardment 28%, relaxation 18%, halo exchanges 11% (reduced from 15% due to the single halo exchange optimization), setup/SoA 13%, local-max scan 10%, MPI reduce 7%, cleanup 12%.

CUDA phase breakdown. Table 3 and Figure 4 (right) show per-phase GPU timings. The dominant profiling insight was the discovery of the 400 MB lookup-table PCIe transfer described in Section 4.3. Post-elimination, the fused kernel accounts for ∼75% of remaining runtime across all workloads, with device memory allocation at ∼16% for `test_08` (kept minimal by the arena allocator). H2D data transfer is negligible (<2 ms in all cases), confirming that particle data is small relative to the layer.

Table 3: Average CUDA phase timings (ms) across workloads. Warm-run average of runs 2–5.

Workload	Alloc	H2D	Kernel	CUB	D2H	Total
`test_02`	1.46	1.62	21.83	0.19	0.08	25.59
`test_07`	1.18	0.27	67.56	0.11	0.03	69.34
`test_08`	2.10	0.04	9.85	0.16	0.09	13.08

6 Limitations and Future Work

CPU: Communication-Avoiding Stencil. At 16 MPI ranks, the halo exchange still accounts for 7% of total runtime in `test_08` (reduced from 15% in the baseline version). A communication-avoiding variant would precompute the bombardment contributions for a wider ghost region of $2S$ cells on each side at the start of each storm sequence, enabling all S storms' relaxation passes to execute without any intermediate `MPI_Sendrecv` calls. The additional computation cost is bounded by $2S \times P$ extra evaluations per rank, which is negligible for small S. **GPU: Multi-GPU Scaling.** The current CUDA implementation is limited to the 24 GB VRAM of a single

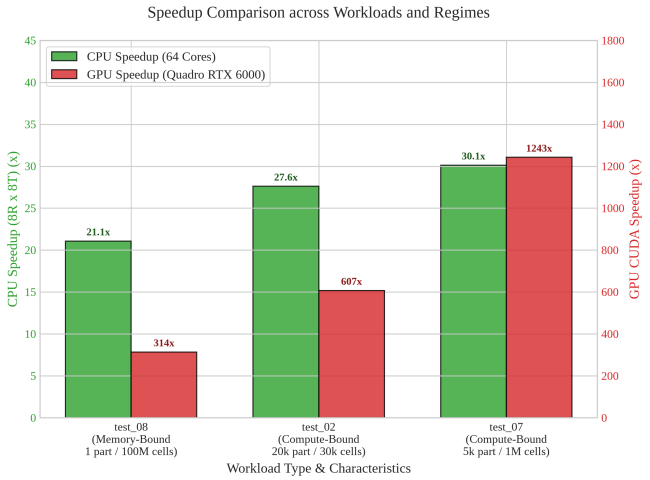

Figure 5: CPU (green, left axis) and GPU (red, right axis) speedup across workloads. Note the dual y-axis scale: GPU speedups are one to two orders of magnitude larger. The compute-bound workloads (`test_02`, `test_07`) benefit far more from GPU parallelism than the memory-bound `test_08`.

Figure 6: Log-scale performance comparison on `test_08` (10^8 cells). Red annotations indicate speedup relative to the sequential CPU baseline (4.08 s).

Quadro RTX 6000. Extending to multi-GPU would require CUDA-Aware MPI: each GPU handles a L/N-cell segment, and halo exchanges are performed via GPUDirect RDMA, bypassing host memory entirely. For very large layers ($L > 10^9$), this would be the only viable approach.

GPU: Tensor Core Acceleration. While tensor cores are primarily designed for matrix-multiply workloads, recent architectures expose them for custom reductions and accumulations. Reformulating the bombardment phase as a batched dot-product could potentially leverage tensor core units for the energy accumulation, yielding an additional 2–4× throughput gain.

7 Conclusion

We presented two parallel implementations of a 1D particle storm simulation covering the full spectrum of energy deposition, thermal diffusion, and hotspot detection. Both implementations are **bit-exact** with respect to the sequential reference code, ensuring numerical reproducibility across all hardware configurations.

The hybrid MPI+OpenMP implementation achieves a peak speedup of **21.08**× on 64 cores (8 Ranks × 8 Threads) for the memory-bandwidth-bound workload `test_08`. The key design decisions driving this result are: NUMA-aware first-touch initialization distributing the 400 MB layer across 8 memory controllers; SoA particle layout enabling AVX2 auto-vectorization of the branchless bombardment loop; and the hybrid MPI+OpenMP decomposition balancing NUMA locality with intra-socket thread parallelism.

The CUDA implementation achieves an application-level speedup of **313**× (kernel-level **410**×) for `test_08`, and an outstanding **1242**× for the compute-intensive `test_07` (10^6 cells, 5000 particles/storm), reducing 86 seconds of sequential computation to under 70 ms. The most impactful single optimization was identified through profiling: eliminating a 400 MB PCIe lookup-table transfer in favor of on-the-fly `sqrtf` reduced total execution time from 756 ms to 13 ms. Complementary techniques — cooperative shared-memory particle tiling, geometric block overlap for kernel fusion, arena allocation, and asynchronous storm queuing — further reduced both kernel time and framework overhead.

Together, these results demonstrate that careful profiling, architecture-aware memory management, and algorithmic design choices can yield speedups of over three orders of magnitude relative to a single-core sequential implementation.