Implementation of the "energy_storms" simulation using Hybrid OpenMP+MPI and CUDA

Alessio Marini
2122855

Aglaia Norza
2114740

Sapienza Università di Roma

February 23, 2026

1 Hybrid OpenMP+MPI Implementation

Our implementation utilizes a Globally Parallel, Locally Parallel strategy by combining MPI and OpenMP.

1.1 1D Domain Decomposition (MPI)

To distribute the workload across the available MPI ranks, we implemented a 1D domain decomposition of the `layer`. The total `layer_size` is partitioned into contiguous blocks, and each rank manages a `local_layer_size` approximately equal to `layer_size / #proc`. Any remainder resulting from the division is distributed among the lower-ranked processes to ensure the maximum load imbalance never exceeds a single cell.

After the bombardment and relaxation stages, the maximum energy value and its corresponding global index are identified using the `MPI_Reduce` collective with the `MPI_MAXLOC` operator. This efficiently reduces the local maxima found by each rank into a single global result on the root rank.

```
MPI_Reduce(&in_max, &out_max, 1,
MPI_FLOAT_INT, MPI_MAXLOC, 0,
MPI_COMM_WORLD);
```

1.2 Further parallelization with OpenMP

Within each MPI rank, we parallelize the workload across CPU cores. To maximize efficiency, we avoid the overhead of repeatedly spawning and destroying threads by encapsulating the main simulation loop within a `#pragma omp parallel` region. This is particularly beneficial in simulations with numerous short-duration storms.

To identify the local maximum after the relaxation phase without resorting to expensive `critical` sections, each thread writes its local peak value to a `ThreadMax` structure that includes a *float* for the value and an *int* for the index of that value. To prevent *false sharing* (where threads on different cores invalidate each other's cache lines), this structure includes 56 bytes of padding (64 - 4 - 4), ensuring each thread's data resides on a unique cache line.

```
typedef struct {
float val; // 4 bytes
int idx; // 4 bytes
char padding
[64 - sizeof(float) - sizeof(int)];
// 54 bytes
} ThreadMax;
```

1.3 Latency Hiding and Thread Synchronization

The relaxation phase involves a 3-point stencil where each point is updated based on its own value and those of its immediate neighbors. To resolve data dependencies at the boundaries of each rank's local block, we implemented a **halo exchange** mechanism, optimised through **latency hiding** with non-blocking primitives (`MPI_Isend` and `MPI_Irecv`)

By restricting inter-process communication to the `master` thread, we maintain a simpler thread-safe execution model while overlapping the data transfer with useful computation. Specifically, the exchange of boundary values is initiated at the start of the relaxation phase; while the network hardware manages the transfer, the CPU threads simultaneously compute the results for all internal cells. Because internal cells do not depend on external halo data, this approach effectively masks the communication latency behind the stencil's computational workload.

To ensure data consistency before finalizing the boundaries, we implement a two-stage synchronization protocol. First, the `master` thread executes an `MPI_Waitall` to confirm that all non-blocking transfers have completed and the halo buffers are fully populated. This is followed by an `omp barrier`, which prevents worker threads from proceeding to the boundary com-

putation until the halo values are globally visible within the rank.

To optimize resource consumption during these synchronizations and to **prevent network noise**, we set `OMP_WAIT_POLICY=passive`; with this rule, idle threads do not waste CPU cycles and do not disrupt MPI communication progress.

```
#pragma omp master
{
    MPI_Isend(&next_layer[0], 1,
    MPI_FLOAT, left_neighbor, TAG_MASSIMO
    , MPI_COMM_WORLD, &reqs[0]);

    MPI_Isend(&next_layer[
    local_layer_size - 1], 1, MPI_FLOAT,
    right_neighbor, TAG_MASSIMO,
    MPI_COMM_WORLD, &reqs[1]);

    MPI_Irecv(&left_val, 1, MPI_FLOAT,
    left_neighbor, TAG_MASSIMO,
    MPI_COMM_WORLD, &reqs[2]);

    MPI_Irecv(&right_val, 1, MPI_FLOAT,
    right_neighbor, TAG_MASSIMO,
    MPI_COMM_WORLD, &reqs[3]);
}

// center computation

#pragma omp master {
    MPI_Waitall(4, reqs,
    MPI_STATUS_IGNORE); }

#pragma omp barrier

// boundary computation
```

1.4 Memory and Cache Optimizations

Optimizing memory accesses was crucial to the implementation's efficiency.

1.4.1 Zero-Copy Pointer Swapping

To minimize the overhead of moving large amounts of data between buffers, we implemented a double-buffering strategy with **pointer-swapping**. This "zero-copy" approach replaces $O(N)$ memory transfers with $O(1)$ pointer updates at the end of each simulation step avoiding an expensive memory ping-pong between buffers.

During the relaxation phase, the system utilizes two distinct buffers: `cur_layer`, which serves as the read-only source representing the post-impact surface, and `next_layer`, where the smoothed values are stored. Upon completion of the relaxation, the roles of these pointers are swapped. This ensures that the correctly relaxed layer of the previous iteration becomes the input for the subsequent storm without the performance penalty of a full `memcpy`.

```
    // O(1)
    float *tmp = cur_layer;
    cur_layer = next_layer;
    next_layer = tmp;
```

1.4.2 SIMD-Aware Vectorization

In addition, we designed our implementation to be **SIMD-aware**, as we integrated vectorization directives across most performance-critical loops. By manually calculating thread chunks as multiples of 16 floats (64 bytes), we allow the compiler to generate efficient SIMD instructions that operate on full cache lines.

To ensure that these vector instructions map perfectly to the CPU's physical cache lines without unaligned access penalties, we utilized `posix_memalign` to ensure that all layer arrays are 64-byte aligned in RAM.

Additionally, we added the `__restrict` keyword to all layer pointers. This informs the compiler that these memory regions do not overlap, allowing it to perform more aggressive loop transformations and instruction reordering that would otherwise be restricted.

```
    float * __restrict cur_layer;

    if (posix_memalign((void**)&cur_layer
    , 64, sizeof(float) *
    local_layer_size) != 0) exit(1);
```

1.4.3 Data Locality

Furthermore, since modern architectures (such as the Department's Cluster) utilize Non-Uniform Memory Access, we implemented a **first-touch** policy [1]. By parallelizing the initial memory allocation and zeroing of the arrays, we ensure that memory pages are allocated in the NUMA domain of the threads that first touch them. Combined with the `OMP_PLACES=cores` and `OMP_PROC_BIND=close` settings, which keep OpenMP threads bound to the same physical cores throughout execution, subsequent accesses remain mostly local to the corresponding socket, improving memory locality.

```
    // inside #pragma omp parallel
    #pragma omp simd
    for (int k = t_start; k < t_end; k++)
     cur_layer[k] = 0.0f;
```

1.4.4 Tiling

Finally, for the bombardment phase, we implemented a **cache blocking** (tiling) strategy with a `TILE_SIZE` of

2048 floats (8KB) [2]. By processing smaller segments of the layer that fit entirely within the L1 cache, we aimed to reduce the frequency of slow main memory fetches, ensuring the "impact" logic operates at register and L1 speeds.

2 Trial and Error

The final implementation is the result of an iterative process where several parallelization strategies were tested and subsequently discarded due to correctness issues or performance bottlenecks. More specifically:

2.1 Storm-Level Parallelism and Non-Associativity

We tried to parallelize the workload by distributing entire storms among MPI processes via round-robin scheduling. However, this altered the summation order of energy impacts: the sequential $((L + e_1) + e_2)$ became $L + (e_1 + e_2)$. Due to **floating-point non-associativity**, this introduced rounding errors that propagated during relaxation, causing the final results to deviate from the verified baseline. A possible "fix" via a chain of dependencies would have serialized the execution, negating any parallel gains.

2.2 Particle-Level Parallelism

We experimented with the idea of assigning individual particles within a single wave to different threads. However, this potentially created race conditions, as multiple particles could impact the same control point or neighborhood simultaneously. Resolving these conflicts would have required expensive atomic operations or critical sections, which significantly degraded performance.

We also considered assigning particles to different processes. In this scenario, each process would maintain a local copy of the layer, but these copies would be incomplete as they would only reflect a portion of the total storm. While these partial layers could be merged using a global reduction, this approach was rejected due to floating-point non-associativity.

2.3 Pre-calculation of Square Roots

During testing, the `sqrtf` function was identified as a potential bottleneck. We tried pre-calculating a **lookup table** for all possible distances. However, for large `layer_size` values, the memory overhead of the table became significant, increasing cache misses and memory traffic. The computational cost of a `sqrtf` (on a modern CPU) was ultimately found to be less than the latency of fetching values from a big, potentially un-cached table.

2.4 Standard Reduction vs. Padded Structures

We initially used standard OpenMP `reduction(max:...)` clauses for the maximum energy search. While correct, this approach was less performant than our final padded `ThreadMax` structure.

2.5 OMP+MPI Performance Analysis

To evaluate the efficiency of our parallelization strategies, we utilized `test_07`, provided with the code. This test is specifically designed to stress the application's memory and scaling capabilities by simulating a substantial domain of $1,000,000$ layer elements undergoing an impact sequence of 5,000 particles per wave. We performed strong scaling tests on this benchmark, utilizing up to 128 physical cores across 4 compute nodes on the Department's Cluster, and tracked execution time to evaluate speedup and parallel efficiency.

We compared four primary execution models:

- **Pure MPI**: 1 thread per rank, mapping one rank to every physical core.
- **Pure OMP**: 1 rank with all available cores.
- **Hybrid Node**: 1 MPI rank per node, 32 OpenMP threads per rank.
- **Hybrid Socket**: 2 MPI ranks per node (one per NUMA socket), 16 OpenMP threads per rank.

2.5.1 Speedup and Efficiency Results

The strong scaling of the application demonstrated exceptional performance, achieving **near-linear speedup** across almost all tested configurations. A summary is provided in Table 1.

Strategy	Cores	Speedup	Efficiency
Pure_MPI	32	32.03x	1.00
Pure_MPI	64	62.90x	0.98
Pure_MPI	128	117.86x	0.92
Hybrid_Node	32	31.79x	0.99
Hybrid_Node	64	63.29x	0.99
Hybrid_Node	128	114.67x	0.90
Hybrid_Socket	32	32.02x	1.00
Hybrid_Soscket	64	60.31x	0.94
Hybrid_Socket	128	113.79x	0.89
Pure_OMP	32	31.78x	0.99

Table 1: Best performing configurations per strategy at peak core counts.

As illustrated in the bottom plot of Figure 1, parallel efficiency remains ideal up to 32 cores. The high parallel efficiency and near-linear speedup observed can be attributed to explicit L1 cache tiling (`TILE_SIZE`

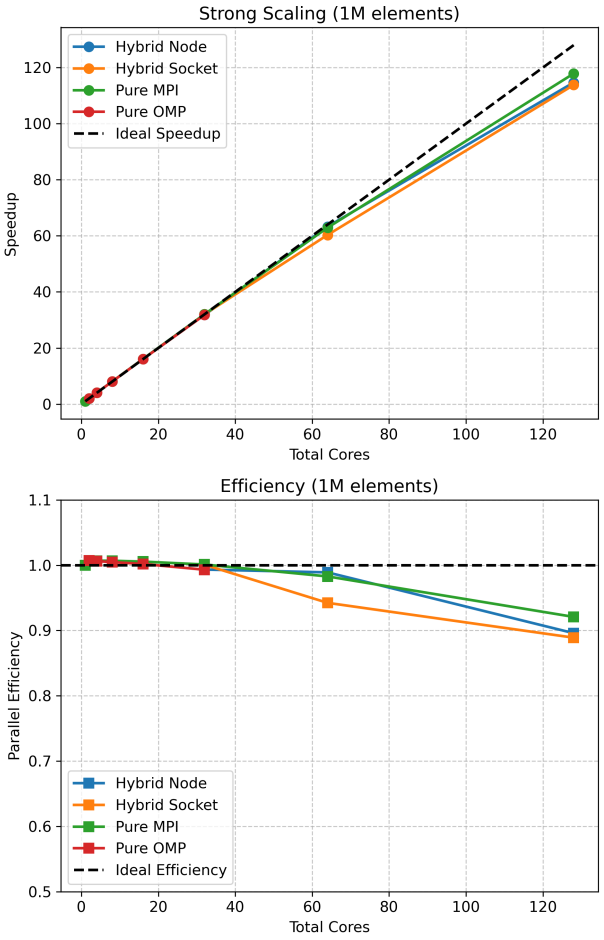


Figure 1: Strong scaling speedup (top) and parallel efficiency (bottom) for the 1,000,000 element workload.

2048) and SIMD-aligned memory structures. Furthermore, the use of padded thread-local reduction arrays (`ThreadMax`) instead of atomic or critical sections effectively minimized false sharing and reduced synchronization overhead in the Hybrid execution models.

2.5.2 Comparative Strategy Analysis

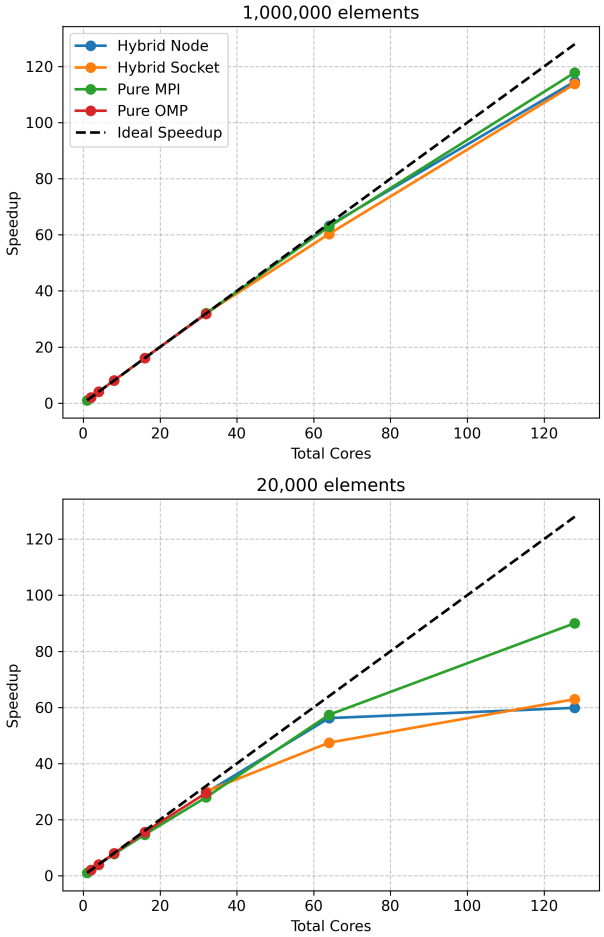
While all strategies scaled well, **Pure MPI** emerged as the most performant, achieving a peak speedup of $117.86x$.

The hybrid models, though robust, were slightly hindered by the additional overhead. At the 128-core limit, Hybrid_Node was ~3% slower than Pure MPI. This delta suggests that the overhead of OpenMP barrier synchronizations and thread management slightly outweighs the cost of local MPI point-to-point messaging.

Notably, we initially believed that achieving these peak speedups across the 128-core cluster required strict spatial process mappings to mitigate memory and network bottlenecks. The validity of this hypothesis, and the application's sensitivity to hardware topologies under sustained workloads, is analyzed in Appendix A.

2.6 Limits of Strong Scaling (Amdahl's Law)

To further investigate the program's scalability and observe the effects of Amdahl's Law, we conducted a secondary strong scaling test using a variant of `test_02` with a small grid of 20,000 cells and 20,000 particles (subsection 2.6).



As the number of physical cores increases in a strong scaling scenario, the problem size per core (`local_layer_size`) shrinks. Eventually, the execution time required to compute the local chunk becomes smaller than the overhead of the parallelization such as MPI processing, OpenMP thread synchronization, and potential particle load imbalances.

As the problem size per core shrinks, the "useful" computation time is rapidly consumed by these overheads:

- at 128 cores on the small workload, **Pure MPI efficiency dropped** to 70%. With each core processing only ~156 elements, the execution time was

no longer dictated by steady physics calculations, but by the costs of 128 MPI halo exchanges and the idle time spent waiting for heavily-loaded cores to finish their localized particle math.

- the **Hybrid_Node** strategy was the most affected, with efficiency plunging to 46%: the cost of synchronizing 32 OpenMP threads via `#pragma omp barrier` becomes more expensive than the actual computation on a tiny local domain.

These results confirm that while our Pure MPI configuration effectively minimizes overhead for minimal workloads, maximizing parallel efficiency ultimately requires scaling the problem size in tandem with the hardware (Weak Scaling).

3 CUDA Implementation

3.1 Thread Organization and Block Size

In our CUDA implementation, since the workload is a 1D simulation array, we found mapping exactly **one thread to one point** in the `layer` to be the most efficient solution to ensure aligned memory accesses.

To determine the optimal thread configuration, we benchmarked various block sizes across two different domain scales: a large layer (1,000,000 cells) and a small layer (20,000 cells). The execution times (in seconds are summarized below:

Block Size	Large (1M)	Small (20K)
32	0.0888	0.0199
64	0.1355	0.0184
128	0.1290	0.0188
256	**0.0615**	**0.0194**
512	0.0824	0.0241
1024	0.0763	0.0312

Table 2: execution time comparison (seconds)

Our testing revealed a divergence in performance behavior based on the domain scale. Smaller block sizes (such as 64) performed best on the smaller domain but suffered bottlenecks on the larger dataset. On the other hand, maxing out the block size at 1024 yielded the absolute fastest time for the 1,000,000-point layer, but caused a noticeable performance penalty on the 20,000-point layer due to thread underutilization and scheduling overhead.

Ultimately, we decided on the ideal "sweet spot", **256 threads per block**. As a multiple of the warp size (32), it provides good Streaming Multiprocessor occupancy, and it offers the best performance balance with respect to both the larger and smaller layer sizes.

The **number of blocks** is calculated dynamically to ensure full coverage of the domain. Using integer division, we apply a ceiling function to account for datasets that are not perfect multiples of the block size:

$$\texttt{num\_blocks} = \frac{\texttt{layer\_size} + \texttt{block\_size} - 1}{\texttt{block\_size}}$$

To prevent illegal memory accesses in cases where the total number of threads exceeds the `layer_size`, we implement a boundary guard within kernels to ensure threads with an `index` $\geq$ `layer_size` remain idle.

```
void update_kernel_small( ... )  {
    int k = blockIdx.x * blockDim.x +
    threadIdx.x;
    if (k < layer_size) { // guard
```

3.2 Kernel 1: Pre-computation and Dual-Path Energy Update

To avoid memory bandwidth and compute bottlenecks during the bombardment phase, we implemented a **pre-calculation** step and a **dual-path** memory strategy.

Before updating the layer, we launch the `prepare_storm_values` kernel, which computes the scaled energy quotient (`energy_den`). Since the expression `((float)p_val * 1000.0f) / (float)layer_size` exclusively depends on each particle's energy and not on the layer cell (keeping it the same for all cells hit by one specific particle), pre-computing it on the GPU prevents hundreds of thousands of threads from wasting arithmetic cycles performing the exact same division later.

To further optimize this process, we utilize a **bit-level packing** technique that repurposes the existing `storm_data` array, avoiding the need for additional memory allocations. By leveraging the fact that the original integer value in the `.y` component is no longer required after the energy calculation, we overwrite it with the floating-point result using the `__float_as_int()` intrinsic.

```
    int2 pv = storm_data[idx];
    int p_val = pv.y;

    float energy_precalc = ((float)p_val
    * 1000.0f) / (float)layer_size;

    pv.y = __float_as_int(energy_precalc)
    ;
    storm_data[idx] = pv;
```

This allows the update kernels to retrieve both the particle position and the pre-computed energy in a single, vectorized 64-bit load. This approach effectively provides the energy data "for free" by **piggybacking on an existing memory transaction**, reducing global memory bandwidth overhead without any loss in precision.

Following pre-calculation, the update phase **diverges based on the storm size**. If a storm fits within the 64KB constant memory limit (defined as `MAX_CONST_POINTS 8100`), its data is moved to `__constant__` memory (`c_storm_posval`) via `cudaMemcpyToSymbol`. While the theoretical limit is 8,192 points ($\frac{64\text{KB}}{\text{8-byte doubles}}$), we intentionally set the threshold to 8100 to provide a safety buffer for compiler-generated constants or kernel parameters that may also reside in the constant memory space. Because all threads in a warp read the exact same storm point simultaneously during the inner loop, **the hardware broadcasts the value** to the entire warp with near-zero latency.

For storms exceeding the 8192-point limit, the data must remain in global memory. To mitigate the resulting latency, the `update_kernel_large` utilizes the `__ldg()` intrinsic [3] to perform vectorized 64-bit loads from the `storm_posval` array. Since this data is marked as `const __restrict__` and remains unchanged throughout the update phase, the `__ldg()` instruction explicitly directs the hardware to route these accesses through the **Read-Only Data Cache**, bypassing the standard L1 cache path, reducing cache pollution and ensuring fast memory accesses even when the data set is too large for constant memory [4].

3.3 Kernel 2: Fused Relaxation and Reduction

Following the energy update phase, the simulation requires a relaxation step (a 3-point stencil average) and a reduction step to find the cell with the maximum energy. Initially, we had implemented these as two separate kernels. However, since separate kernels forced the GPU to write the newly relaxed layer to global memory, only to immediately read it back to find the maximum, we realised that fusing these operations into a single kernel, `avg_max_kernel`, was a better solution.

To optimize the stencil computation, we utilize dynamically allocated shared memory (`extern __shared__ char s_data[]`). The allocated size is exactly `block_size + 2` floats to accommodate the core block elements plus two boundary halo cells. During the first phase of the kernel, threads cooperatively load data from the global source array into this shared memory space. By mapping each thread to a contiguous global memory index (`avg_data[s_idx] = src[gid]`), **global memory reads are coalesced**, maximizing memory bandwidth The halo elements are only fetched by the boundary threads. A `__syncthreads()` barrier is then called to ensure all data is loaded before computing it.

```
    if (gid < layer_size) avg_data[s_idx]
     = src[gid];
    else avg_data[s_idx] = 0.0f;

    // left halo (first thread)
    if (tid == 0) {
        if (gid > 0) avg_data[0] = src[
    gid - 1];
        else avg_data[0] = 0.0f;
    }

    // right halo (last thread)
    if (tid == blockDim.x - 1) {
        if (gid < layer_size - 1)
    avg_data[s_idx + 1] = src[gid + 1];
        else avg_data[s_idx + 1] = 0.0f;
    }
```

Each thread computes the 3-point average, caching the result in a **local register variable**, `my_val`. While this calculated average must still be written back to the global destination array to update the simulation state, retaining it in a register further optimizes the following operations: instead of terminating the kernel and querying `dst[gid]` from global memory in a subsequent launch, threads reuse their register-cached `my_val` to initialize the search for the maximum value.

Furthermore, to minimize the kernel's resource footprint, we reuse the dynamically allocated shared memory space. Once the 3-point average is computed and the data in `avg_data` is no longer needed, we alias the exact same `s_data` pointer to hold the intermediate per-warp maximums and their corresponding indices (`shared_max_val` and `shared_max_index`), significantly reducing the overall shared memory required per block.

3.4 Warp Shuffling & Device-Side Finalization

Finding the maximum energy value and its global index requires a parallel reduction. To keep the execution fast and avoid stalling the pipeline, we went with a multi-step reduction strategy that keeps the CPU out of the loop as much as possible during the main simulation.

For the lowest level of the reduction, we wrote a `warpReduceMax` kernel:

```
    void warpReduceMax(float &val, int &
    idx) {
    unsigned int mask = 0xffffffff;
    for (int offset = 16; offset > 0;
    offset /= 2) {
        float other_val =
    __shfl_down_sync(mask, val, offset);
        int other_idx = __shfl_down_sync(
    mask, idx, offset);
        if (other_val > val) {
            val = other_val;
            idx = other_idx;
        }
    }
}
```

The algorithm is divided into three main phases:

1. **Intra-Warp Phase (Parallel Tree Reduction in Registers)**

   Initially, each thread processes its assigned subset of data and identifies a strictly local maximum. Subsequently, threads within the same warp collaborate using the `__shfl_down_sync` primitive. This instruction allows threads to directly **read values from the registers of other threads in the same warp**, bypassing the memory hierarchy. The reduction follows a tree pattern: by progressively halving the read offset (from 16 down to 1), the operation converges in exactly $\log_2(32) = 5$ steps. At the end of this phase, the leader thread of each warp holds the maximum value and its corresponding index for that specific warp. Since warp execution is intrinsically synchronous in SIMT mode, this phase operates efficiently without the need for explicit synchronization barriers.

2. **Inter-Warp Communication (Writing to Shared Memory)**

   Because registers are strictly private to each warp, inter-warp communication must be routed through shared memory. However, as a direct result of the preceding intra-warp reduction, only the leader thread of each warp needs to write its partial result to the shared memory array. Assuming a maximum block size of 1024 threads (32 warps), this **dramatically reduces the shared memory footprint** to just 32 elements per block. This pattern also effectively avoids shared memory bank conflicts (as there is no simultaneous access to different addresses within the same warp) and significantly reduces overall memory traffic. Immediately following these write operations, we invoke a single `__syncthreads()` barrier to guarantee that all partial results are visible across the entire block.

3. **Final Resolution (Single Warp Reduction)**

   In the final phase, the first warp of the block (Warp 0) assumes responsibility for reducing the array stored in shared memory. The active threads of Warp 0 load the partial maxima, while any excess threads are logically masked by loading a neutral value (`-FLT_MAX`). Warp 0 then executes one last call to the intra-warp tree reduction. Upon completion, the global thread 0 of the block holds the absolute maximum energy value and its index for the entire thread block, which is then written back to global memory.

This implementation achieves exceptionally **low latency** by performing the majority of data exchanges at **register speed**, alongside high **synchronization efficiency** by requiring only a single `__syncthreads()` instruction for the entire block reduction operation.

3.5 Trial and Error

Separate vs. Fused Kernels: as discussed in subsection 3.3, when we first ported the code to CUDA, we kept the relaxation and reduction phases as two separate kernels.

Host vs. Device Reduction: at first, we thought it would be simpler to let the CPU find the absolute maximum for each storm. We used `cudaMemcpy` to pull the partial block maximums back to the host at the end of every iteration, and also experimented with mapped pinned memory via `cudaHostAllocMapped`. We soon realized that this implementation would force a hard synchronization point. Writing the `find_max` kernel solved this problem. By moving that final reduction step to the device, we successfully eliminated the explicit `cudaDeviceSynchronize()` and the expensive device-to-host memory transfers from the main simulation loop.

3.6 CUDA Performance Analysis

We observed that our CUDA implementation is mostly **compute-bound**. To confirm this, we conducted two primary scaling benchmarks: one maintaining a **fixed layer size** of 1,000,000 cells while **increasing the storm size**, and another maintaining a **fixed storm size** of 5,000 particles while **scaling the layer dimensions**. For both tests, we measured the total execution time alongside the pure computation time, allowing us to isolate the overhead introduced by `cudaMalloc` and `cudaMemcpy` operations.

3.6.1 Fixed Layer

Storm Size	Total Time	Compute Time	Overhead
5,000	0.0778	0.0761	0.0017
10,000	0.1379	0.1361	0.0017
50,000	0.5754	0.5737	0.0017
250,000	2.7322	2.7303	0.0019

Table 3: Execution time comparison for varying Storm Size (seconds) with a fixed layer of 1.000.000 cells

The results indicate a near-constant overhead across all configurations. This overhead is primarily composed of:

- **Device Allocation**: `cudaMalloc` calls for the simulation layer incur a fixed latency penalty, as the layer dimension remains constant.
- **Host-to-Device Transfers**: `cudaMemcpy` is used to transfer the storm particle data (`d_storm_staging`) to device memory. Although scaling the storm from 5,000 to 250,000 particles increases the transfer payload from approximately 40 KB to 2 MB (for the `int2` array), the increased

volume remains negligible relative to the available host-to-device bandwidth, resulting in only a marginal increase in total overhead.

Furthermore, the data demonstrates that **the implementation scales linearly with respect to the computational workload**. The total computational work of the core update kernel is $\mathcal{O}(\text{layer\_size} \times \text{storm\_size})$, while, with our one-thread-per-cell mapping, the *parallel time complexity* is $\mathcal{O}(\text{storm\_size})$. Consequently, bringing the storm size from 50,000 to 250,000 particles represents a 5x increase. Considering the isolated compute times, the execution duration scales from 0.5737s to 2.7303s, which is exceptionally close to perfect linear scaling ($0.5737 \times 5 = 2.8685s$).

Finally, the scaling results highlight the effectiveness of our dual-kernel memory architecture. The 5,000-particle test falls below our 8,000-particle threshold and utilizes ultra-fast `__constant__` memory via the `update_kernel_small` kernel. From 10,000 onwards, the tests exceed the constant memory's capacity, and are routed through the Read-Only Data Cache via the `__ldg()` intrinsic in `update_kernel_large`. The fact that performance continues to scale linearly without significant bottlenecks suggests that our read-only cache strategy successfully mitigates the latency penalty of leaving constant memory.

3.6.2 Fixed Storm Size

Layer Size	Total Time	Compute Time	Overhead
10,000	0.0088	0.0084	0.0004
50,000	0.0097	0.0093	0.0004
100,000	0.0181	0.0177	0.0004
500,000	0.0713	0.0699	0.0013
1,000,000	0.1383	0.1365	0.0018
5,000,000	0.5799	0.5768	0.0031
10,000,000	1.0511	1.0464	0.0047

Table 4: Execution time comparison for varying Layer Size with a fixed storm of 10,000 particles (in seconds)

Analyzing the results for the smaller layer dimensions (10,000 to 50,000 cells), we observed that a 5x increase in problem size yields only a marginal increase in execution time. This plateau indicates that the hardware is not fully saturated, meaning the **execution time is primarily dominated by fixed kernel launch latencies and scheduling costs** rather than the arithmetic workload itself.

However, once the layer size reaches 100,000 cells, the hardware becomes fully saturated, and the execution time scales linearly with respect to the workload. This can be observed in the tests with 5,000,000 and 10,000,000 cells, where a doubling of the domain size results in a proportional doubling of the compute time.

Finally, unlike the fixed-layer benchmark, **the measured overhead in this scenario increases as the problem size scales**. This is due to the escalating latency of device memory allocations (`cudaMalloc`) required for progressively larger arrays. Despite this growth, the **overall parallel efficiency remains high**; in the 10,000,000-cell test, the overhead constitutes less than 0.5% of the total execution time.

4 CUDA vs MPI

We compared the steady-state execution times of our 128-core CPU configuration (4 nodes, Pure MPI, our fastest combination) against a single CUDA GPU. We used repeated workloads to expose specific hardware bottlenecks:

- **Compute-Bound (`test_02`):** 30,000 cells, 20,000 particles. Tests ALU throughput.
- **Memory-Bound (`test_08`):** 100,000,000 cells, 1 particle. Tests memory bandwidth.
- **Balanced (`test_07`):** 1,000,000 cells, 5,000 particles. Tests memory and compute balance.

Workload	MPI (128 Cores)	CUDA (1 GPU)
`test_02`	23.26 s	**18.94 s**
`test_08`	**2.87 s**	4.03 s
`test_07`	25.12 s	**11.96 s**

Table 5: Steady-state execution times: 128-core MPI vs CUDA.

4.1 Performance Analysis

Compute-Bound: The GPU outperformed the 128-core CPU by roughly 22%. This gain is driven by the GPU's massive arithmetic throughput efficiently executing dense inner-loop math. Pre-calculating storm energies to eliminate redundant divisions further amplified this raw compute advantage.

Memory-Bound: The MPI implementation took the lead by roughly 40%. Running across 4 discrete nodes provides a massive advantage in aggregated memory bandwidth and Last-Level Cache (LLC) capacity. Furthermore, while the CPU performs in-place updates, the GPU's performance is bottlenecked by host-to-device PCIe transfer latencies for each storm, alongside the memory bandwidth overhead of the double-buffering required during the relaxation phase.

Balanced Workload: The GPU achieved a definitive victory, running over 2× faster than our MPI code. The 5,000-particle storm allows the GPU to leverage `__constant__` memory broadcasts for near-zero latency reads. Furthermore, the GPU bypasses the fixed overhead of MPI halo exchanges by processing the domain globally with fast intra-warp shuffling.

A Topology and Process Mapping Analysis

To evaluate the application's behavior under different mapping configurations, we analyzed execution times on a simulated 100,000,000-cell array with a minimal computational payload (1 particle per wave, `test_08`).

Initial **single-iteration tests** yielded highly **erratic** sub-second execution times (0.015s to 0.070s). This initially **suggested severe topological sensitivities**, implying Block mapping dominated Pure MPI, while Inter-Cyclic mapping was necessary to prevent memory saturation in Hybrid configurations. However, by **scaling** the workload to 100, 200, and up to 300 iterations to amortize overhead costs, we found that under sustained execution, spatial process mapping has a **negligible impact** on performance.

A.0.1 Cold-Start vs Steady-State

We attribute the erratic single-iteration behavior to page fault overhead and NUMA first-touch placement, since the timer includes the first write to the arrays. In short runs, we believe this initialization cost dominates the trivial computational payload. When the execution is extended to **300 iterations**, this effect is fully amortized and the performance gap between intra-node (Block) and inter-node (Inter-Cyclic) rank placement drops below 0.1%. Moreover, each iteration exchanges only a single halo element per boundary, making the communication volume negligible compared to the memory traffic over the 100,000,000-cell domain. Under these **steady-state** conditions (once the system has "warmed up" and initial setup overhead is cleared), no measurable performance difference between mappings is observed.

Configuration (128 Cores)	**Block** (s)	**Inter-Cyclic** (s)
Pure MPI 128 ranks × 1 thread	2.878	2.875
Hybrid Socket 8 ranks × 16 threads	3.168	3.167
Hybrid Node 4 ranks × 32 threads	3.216	3.211

Table 6: 300-Iteration execution times (`test_08`). Variance across all mappings is $< 0.1\%$.

A.0.2 Partial Saturation

This invariance holds even at partial cluster capacity (64 cores). Despite our initial hypothesis that Inter-Cyclic mapping would alleviate Last-Level Cache (LLC) contention, the 300-iteration results showed no practical gain (e.g., Block and Inter-Cyclic varied by only 0.004s in a Hybrid Socket setup).

We believe this to be due to the workload's extremely low arithmetic intensity. Because we only have 1 particle but 100,000,000 cells to update, the CPU spends nearly all its time waiting for the next chunk of the array. It finishes the actual math almost instantly and then goes back to waiting. Consequently, the time required to complete the memory-intensive compute loops far exceeds the time needed for MPI background communication, making the choice between local and remote rank placement irrelevant.

A.0.3 Compute-Bound Workload (`test_02`)

To ensure these conclusions were not limited to memory-bound scenarios, we tested a heavily compute-bound workload (1000 iterations of `test_02`, with a `layer_size` of 30,000 and 20,000 particles to compute). By increasing arithmetic intensity and shrinking the domain size, the primary bottleneck shifts from memory controllers to the CPU's ALUs, while simultaneously increasing the frequency of MPI halo exchanges.

We initially hypothesized that external cluster network latency would penalize Inter-Cyclic mapping in this scenario. However, execution times showed no significant variation: at 128 cores, Pure MPI configurations varied by only 0.3s (23.26s vs 23.56s), and Hybrid Node configurations varied by 0.24s (23.78s vs 24.02s).

We attribute this uniformity to the nature of the parallel overhead. Processors spend the vast majority of their time executing dense particle physics calculations. The physical interconnect latency differences introduced by topological mapping constitute a negligible fraction of the total execution time. Any physical transmission delays are overshadowed by the dense ALU workload and the fixed software costs of synchronization, rendering the spatial location of communicating ranks irrelevant.

A.0.4 Conclusion on Topological Sensitivity

Our tests (across both extremes of the workload spectrum) suggest that the application's performance shows no measurable sensitivity to OS-level spatial process mapping on the tested system. In low-arithmetic, large-grid scenarios (`test_08`), execution is bounded by universal memory bandwidth. In high-arithmetic, small-grid scenarios (`test_02`), execution is bounded by CPU compute capacity. In both extremes, the **location of communicating ranks has no meaningful impact on overall throughput**.

References

[1] Ruud van der Pas. *Performance Tuning of OpenMP Programs*. Slide 22: The First-Touch Placement Policy. Available at: `https://www.openmp.org/w`

p-content/uploads/SC18-BoothTalks-vanderPas.pdf

[2] Cornell University CS3410. *Assignment: Matrix Multiplication and Tiling*. Available at: https://www.cs.cornell.edu/courses/cs3410/2024fa/assignments/cacheblock/instructions.html

[3] NVIDIA Corporation. *CUDA C++ Programming Guide: Read-Only Data Cache Load Function*. Available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#read-only-data-cache-load-function

[4] Stack Overflow. *What is the difference between __ldg() intrinsic and a normal execution?* Available at: https://stackoverflow.com/questions/26603188/what-is-the-difference-between-ldg-intrinsic-and-a-normal-execution

[5] NVIDIA Corporation. *Faster Parallel Reductions on Kepler (Warp Shuffle)*. Available at: https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/