High Performance Computing Systems
Project 5

Tsogkas Panagiotis **Nikas Ioannis Iason** **Tsiantos Dimitrios**

January 2026

Contents

1 The Algorithm **3**

2 CPU Parallelization using OpenMP **4**

3 Initial GPU Version **4**
- 3.1 Parallelization Strategy 4
- 3.2 Kernel Separation 4
- 3.3 Global Memory and Special Functions 4

4 CUDA Streams **5**

5 Changing Data Structure to Structure of Arrays **5**

6 Applying tiling in shared memory **7**

7 Read-Only Data Cache and Fused Multiply Add **8**

8 Coarsening **8**

9 Multiple GPUs **9**

10 Launch Bounds **9**

11 Eliminating divergence **10**

12 Results and measurements **10**

1 The Algorithm

The N-body problem concerns the simulation of the dynamical evolution of a system of N particles that interact with each other through physical forces. In our case, we examine gravitational interaction according to Newton's Law of Universal Gravitation.
The force $\vec{F}_{ij}$ exerted on body i by body j is given by:

$$\vec{F}_{ij} = G\frac{m_i m_j}{|\vec{r}_{ij}|^3}\vec{r}_{ij} \tag{1}$$

where:

- G is the gravitational constant.
- m_i, m_j are the masses of the bodies.
- $\vec{r}_{ij} = \vec{r}_j - \vec{r}_i$ is the position vector from i to j.

The total force $\vec{F}_i$ acting on body i is the sum of the individual forces from all other $N-1$ bodies:

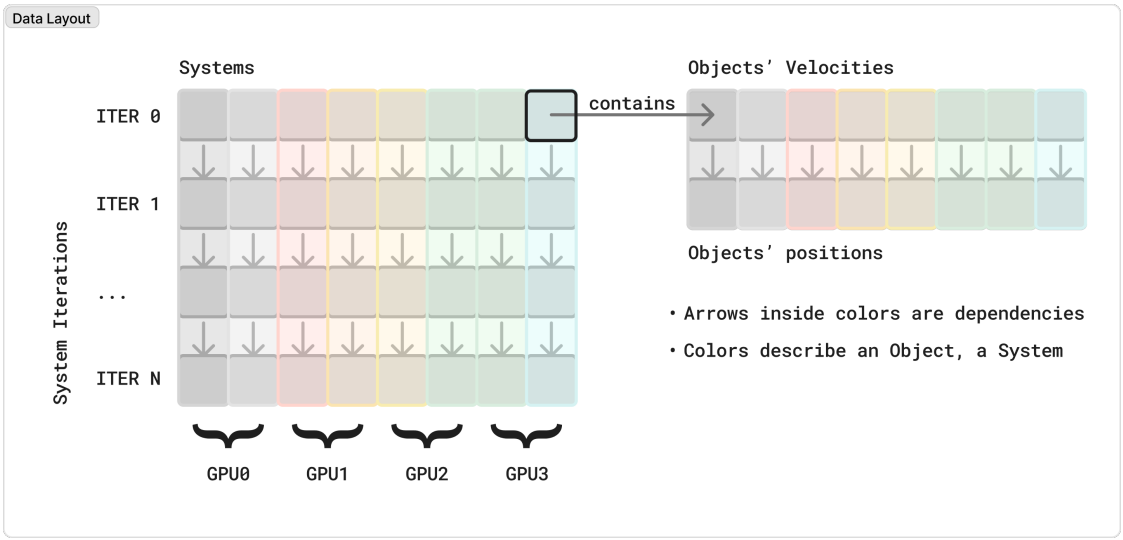
$$\vec{F}_i = \sum_{j=1, j\neq i}^{N} \vec{F}_{ij} \tag{2}$$

The simulation of the system's time evolution is achieved by dividing time into small, discrete steps of duration dt. At each time step, the algorithm is executed in two distinct phases for all bodies:

1. **Interaction and Velocity Computation**
   The total force acting on each body from all others is computed. Immediately after calculating the force (and thus the acceleration), the velocity of the body is updated. This is the most computationally demanding part of the simulation, as it requires reading all data for each body.

2. **Position Update**
   After the new velocities for all bodies have been computed, the second phase updates the coordinates of each body based on its current velocity.

This flow is repeated for the desired number of simulation steps, as shown in the following diagram:



2 CPU Parallelization using OpenMP

Parallelization was applied at the outer level of the simulation, namely the loop that iterates over independent systems, and not at the inner loop over bodies. With this choice, we achieve low overhead since frequent creation and destruction of threads is not required. Additionally, we avoid the need for synchronization, which would reduce performance, since the systems are completely independent from each other and there are no race conditions. Dynamic scheduling is applied to balance the workload across threads. Furthermore, by using `pragma omp simd`, which utilizes SIMD instructions to perform operations on wider registers, we applied instruction-level parallelism.

Finally, the original code used an **Array of Structures** layout, where the data of each body were grouped into a struct. This approach creates large memory strides when reading specific fields (e.g., only the x coordinates). To optimize performance, we converted the structure to a **Structure of Arrays (SoA)**. In this way, all x values are stored contiguously in memory. This improves locality, as the cache is filled with data that are more likely to be reused. This optimization is also applied later in the GPU versions for similar reasons.

Table 1: Average of 25 runs with removal of minimum and maximum measurement

Version	**Throughput** (Billion Interactions/s)	**Standard Deviation** (σ)	**Execution time** (s)	**Improvement**
Sequential CPU	0.1482	0.13	293.255	x1
OpenMP CPU	10.6469	0.0689	4.039	x71.84

3 Initial GPU Version

In our first GPU implementation, the goal was to transfer the algorithm from the CPU to the CUDA environment in the most straightforward way. The strategy is based on mapping one thread per body.

3.1 Parallelization Strategy

Unlike the OpenMP implementation where we parallelized over systems, here we exploit the massive parallelism of the GPU to compute interactions within each system. We create a grid of threads where each thread is exclusively responsible for one body of a galaxy. Each thread iterates over all N bodies of the system, reads their positions, and accumulates the forces. Since each thread writes to a distinct memory location, no synchronization mechanisms are required during result storage.

3.2 Kernel Separation

The simulation is split into two separate kernels, `bodyForceKernel` and `integrateKernel`. This separation is necessary because it acts as a synchronization barrier. We must ensure that velocities for all bodies have been computed before proceeding to update their positions for the next step.

3.3 Global Memory and Special Functions

In this version, all data (positions and velocities) reside in the GPU's global memory. Each thread needs to read the positions of all N bodies from global memory in order to compute the force on its own body. This results in N reads for each of the N threads, i.e., a total of N^2 global memory accesses, which has high latency and therefore leads to reduced performance. This constitutes the main bottleneck of this implementation.

For computing the inverse square root, we replaced the standard division `1.0f / sqrtf(...)` with the intrinsic function `rsqrtf()`. This function is executed directly by the GPU's Special Function Units (SFUs) and is significantly faster, with minimal loss of accuracy.

Table 2: Average of 25 runs with removal of minimum and maximum measurement

Version	Throughput (Billion Interactions/s)	Standard Deviation (σ)	Execution time (s)	Improvement
Sequential CPU	0.1482	0.13	293.255	x1
OpenMP CPU	10.6469	0.0689	4.039	x71.84
Naive GPU	29.2351	1.3894	1.473	x197.26

4 CUDA Streams

To fully utilize GPU resources, we use streams to execute multiple kernels concurrently on the device. Since bodies belonging to different systems have no dependencies, the use of streams allows the GPU scheduler to execute kernels in parallel.

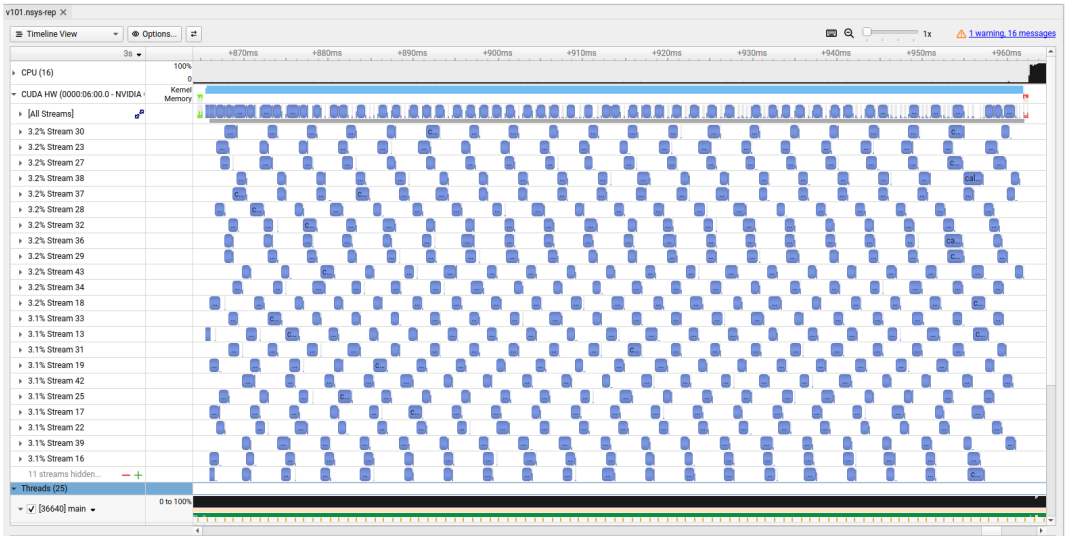


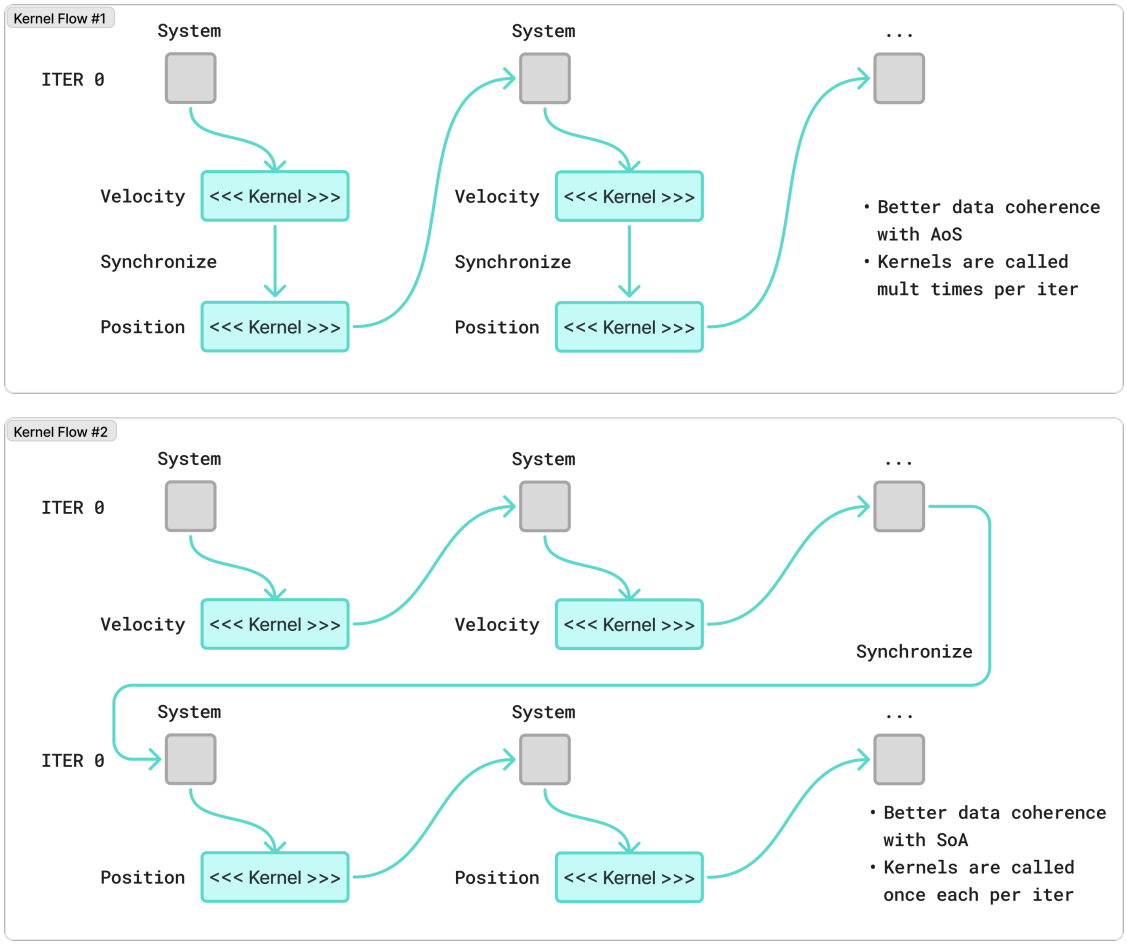
Figure 1: Gantt chart of kernel execution. Each blue square represents one execution of the `bodyForceKernel` kernel.

Table 3: Average of 25 runs with removal of minimum and maximum measurement

Version	Throughput (Billion Interactions/s)	Standard Deviation (σ)	Execution time (s)	Improvement
Sequential CPU	0.1482	0.13	293.255	x1
OpenMP CPU	10.6469	0.0689	4.039	x71.84
Naive GPU	29.2351	1.3894	1.473	x197.26
Streams	40.0133	0.4182	1.0735	x269.99

5 Changing Data Structure to Structure of Arrays

For the kernel structure, we arrived at two versions, from which we selected Kernel Flow #2, since Kernel Flow #1 requires more frequent synchronizations. This structure proves to be more efficient when combined with data organization in the form of Structure of Arrays (SoA), and for this reason the specific data representation was adopted.



As in the OpenMP version, we changed the data structure from Array of Structures to Structure of Arrays. In the CUDA environment, this change is even more important due to the way the GPU global memory operates.

When a warp reads data from global memory, the GPU attempts to serve all requests with a single memory transaction, provided that the addresses are contiguous. In the AoS case, the position x data for thread k and thread $k+1$ are separated by the size of the entire `Body` struct. This leads to inefficient bandwidth utilization (strided access). In contrast, in the SoA case, the x data are stored in contiguous memory locations. When thread k reads `soa.x[k]` and thread $k+1$ reads `soa.x[k+1]`, the addresses are adjacent. This enables **coalesced accesses**, greatly reducing the number of memory transactions.

It should be noted that the use of SoA introduces overhead on the host for the initial and final data conversion. However, the gain from faster memory access on the GPU outweighs this cost, especially for a large number of iterations.

Table 4: Average of 25 runs with removal of minimum and maximum measurement

Version	**Throughput** (Billion Interactions/s)	**Standard Deviation** (σ)	**Execution time** (s)	**Improvement**
Sequential CPU	0.1482	0.13	293.255	x1
OpenMP CPU	10.6469	0.0689	4.039	x71.84
Naive GPU	29.2351	1.3894	1.473	x197.26
Streams	40.0133	0.4182	1.0735	x269.99
Structure of Arrays	40.9047	0.4630	1.0502	x276.01

6 Applying tiling in shared memory

Given the way data are accessed, ideally we would like to place all the data of a system in shared memory, so that we operate exclusively on a single copy residing there. In practice, however, each kernel does not have sufficient shared memory to support such an approach. For this reason, the problem is decomposed into tiles, allowing the reuse of a shared memory block in each execution.

Figure 2 illustrates the way data are managed within the context of a kernel. Specifically, two distinct phases can be identified. In the first phase, each thread loads the data of one body into shared memory. In the second phase, the threads use the information stored in shared memory to compute partial results for the variables F_x, F_y, and F_z.

These partial results are accumulated as all tiles are processed sequentially, and the final result is written back to global memory only after the computation for all tiles has been completed.

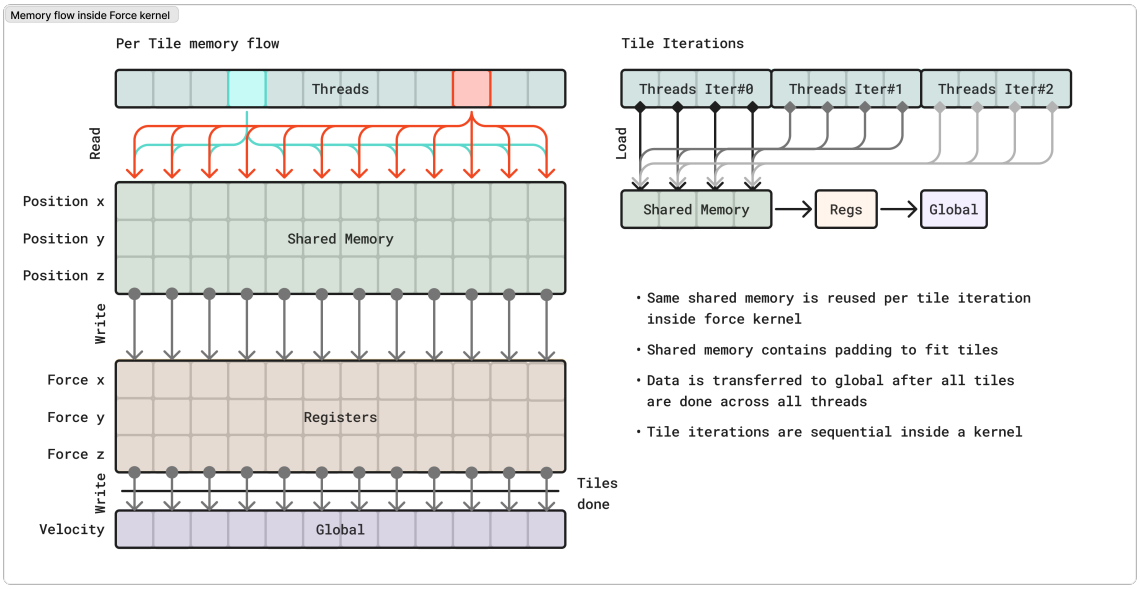


Figure 2: Shared memory flow

With this method, we arrive at an implementation in which each kernel invocation computes the velocities of the bodies of a system in blocks of size `BLOCK_SIZE`. Data access is performed over multiple grids, allowing the problem to be processed in segments.

These kernels use streams, so that they can be executed concurrently. The data access pattern is depicted in Figure 3.

Table 5: Average of 25 runs with removal of minimum and maximum measurement

Version	**Throughput** (Billion Interactions/s)	**Standard Deviation** (σ)	**Execution time** (s)	**Improvement**
Sequential CPU	0.1482	0.13	293.255	x1
OpenMP CPU	10.6469	0.0689	4.039	x71.84
Naive GPU	29.2351	1.3894	1.473	x197.26
Streams	40.0133	0.4182	1.0735	x269.99
Structure of Arrays	40.9047	0.4630	1.0502	x276.01
Tiling	54.6848	0.9807	0.7856	x368.99

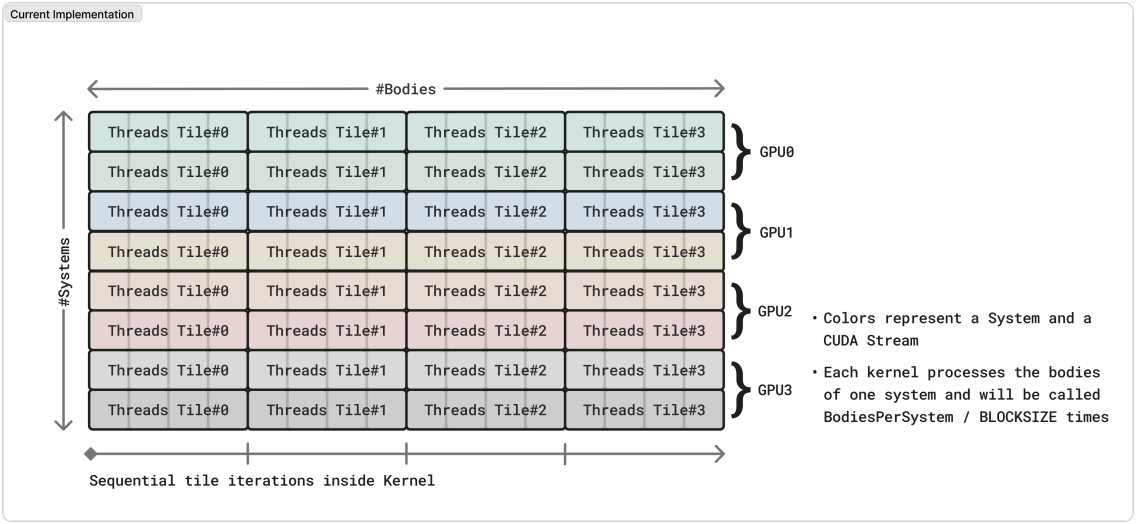


Figure 3: Data access pattern

7 Read-Only Data Cache and Fused Multiply Add

For reading the positions of the bodies from memory, we utilize the device's Read-Only Data Cache through the use of the `__ldg()` function. Since the positions of the bodies are only read from memory, storing them in the read-only cache uses a separate bus for transferring data to the SMs, thereby saving space and bandwidth from the L1 cache.

An additional performance improvement is provided by the fused multiply add instruction using the `fmaf()` function. This function performs multiplication and addition in a single instruction and is used both in the force computation kernel and in the position update computation.

Table 6: Average of 25 runs with removal of minimum and maximum measurement

Version	**Throughput** (Billion Interactions/s)	**Standard Deviation** (σ)	**Execution time** (s)	**Improvement**
Sequential CPU	0.1482	0.13	293.255	x1
OpenMP CPU	10.6469	0.0689	4.039	x71.84
Naive GPU	29.2351	1.3894	1.473	x197.26
Streams	40.0133	0.4182	1.0735	x269.99
Structure of Arrays	40.9047	0.4630	1.0502	x276.01
Tiling	54.6848	0.9807	0.7856	x368.99
Special Functions	84.6815	1.9990	0.5073	x571.40

8 Coarsening

By assigning more work to each thread, performance improves significantly. This is due to better utilization of data within the kernel and increased instruction-level parallelism (ILP). Instead of a thread computing the force for a single body, it can compute the force for two or more bodies within one kernel. This provides better memory accesses, since each set of elements loaded from shared memory is reused for multiple force computations simultaneously, reducing memory accesses. It is important that memory accesses are coalesced, taking the i-th element from each block. The scheduler can now pipeline instructions within a kernel. The disadvantage is the increased register usage per kernel call, but the aforementioned advantages outweigh this cost, resulting in reduced execution time. After experimentation, we observed that the optimal coarsening factor is two bodies per thread.

Table 7: Average of 25 runs with removal of minimum and maximum measurement

Version	Throughput (Billion Interactions/s)	Standard Deviation (σ)	Execution time (s)	Improvement
Sequential CPU	0.1482	0.13	293.255	x1
OpenMP CPU	10.6469	0.0689	4.039	x71.84
Naive GPU	29.2351	1.3894	1.473	x197.26
Streams	40.0133	0.4182	1.0735	x269.99
Structure of Arrays	40.9047	0.4630	1.0502	x276.01
Tiling	54.6848	0.9807	0.7856	x368.99
Special Functions	84.6815	1.9990	0.5073	x571.40
Coarsening	72.1287	1.6262	0.5957	x486.69

9 Multiple GPUs

For the use of multiple GPUs, we tested implementations using both CUDA streams and OpenMP threads, and ultimately chose OpenMP for simplicity. Specifically, we define a parallel region with a number of threads equal to the number of available GPUs, up to the maximum limit of `GPU_MAX`.

The logic among threads is identical, and the total workload is divided into equal parts, corresponding to the number of GPUs being used. This is feasible because the computation of the n-body problem across independent systems is embarrassingly parallel. As a result, the problem can be partitioned into equal-sized segments, each managed by a different CPU thread, with each thread bound to a separate GPU.

It is worth noting that on systems with CUDA 13.0, the NVCC compiler supports the `cudaInitDevice()` instruction, which has been replaced by `cudaSetDevice()`, followed by a call to `cudaFree(0)`. According to relevant forum reports, this call forces the creation of a CUDA context, in a manner analogous to the `cuDevicePrimaryCtxRetain()` instruction.

Table 8: Average of 25 runs with removal of minimum and maximum measurement

Version	Throughput (Billion Interactions/s)	Standard Deviation (σ)	Execution time (s)	Improvement
Sequential CPU	0.1482	0.13	293.255	x1
OpenMP CPU	10.6469	0.0689	4.039	x71.84
Naive GPU	29.2351	1.3894	1.473	x197.26
Streams	40.0133	0.4182	1.0735	x269.99
Structure of Arrays	40.9047	0.4630	1.0502	x276.01
Tiling	54.6848	0.9807	0.7856	x368.99
Special Functions	84.6815	1.9990	0.5073	x571.40
Coarsening	72.1287	1.6262	0.5957	x486.69
Multiple GPUs	322.2653	19.5558	0.1338	x2174.52

10 Launch Bounds

To maximize kernel performance, the `__launch_bounds__()` directive was used. During compilation (using the `-Xptxas -v` flag), it was observed that this configuration leads to increased register usage per thread compared to configurations targeting maximum occupancy, such as setting the minimum number of blocks per SM to 8.

Although reducing the minimum number of blocks per Streaming Multiprocessor decreases theoretical occupancy, in practice it allows the compiler to avoid register spilling. The increased availability of registers enables threads to execute faster, compensating for the loss of block-level parallelism and ultimately leading to improved overall execution time.

Below is an example implementation with a coarsening factor of 4 and different `launch_bounds` directives, which resulted in approximately halving performance. The performance difference is directly explained by

the ptxas information, where the appearance of extensive register spilling is clearly visible in the first case.

Directive `__launch_bounds__(THREADS_PER_BLOCK, 4)`

```
    136 bytes stack frame, 188 bytes spill stores, 180 bytes spill loads
ptxas info    : Used 64 registers, used 1 barriers, 136 bytes cumulative stack size,
                3072 bytes smem, 412 bytes cmem[0]
ptxas info    : Compile time = 24.877 ms
```

Directive `__launch_bounds__(THREADS_PER_BLOCK, 2)`

```
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 128 registers, used 1 barriers,
                3072 bytes smem, 412 bytes cmem[0]
ptxas info    : Compile time = 25.730 ms
```

11 Eliminating divergence

In the kernels, we use various conditions to check whether indices are within bounds. This, however, introduces divergence, which we address by adding padding. The padding technique is applied to align the number of bodies with the product of the block size and the coarsening factor. In this way, all threads within warps always execute the same instructions, without checking whether they are within bounds.

12 Results and measurements

To measure the performance of each optimization we implemented, we ran each version 25 times, taking the average throughput while removing the highest and lowest outlier. The results are shown below:

Table 9: Average of 25 runs with removal of minimum and maximum measurement

Version	**Throughput** (Billion Interactions/s)	**Standard Deviation** (σ)	**Execution time** (s)	**Improvement**
Sequential CPU	0.1482	0.13	293.255	x1
OpenMP CPU	10.6469	0.0689	4.039	x71.84
Naive GPU	29.2351	1.3894	1.473	x197.26
Streams	40.0133	0.4182	1.0735	x269.99
Structure of Arrays	40.9047	0.4630	1.0502	x276.01
Tiling	54.6848	0.9807	0.7856	x368.99
Special Functions	84.6815	1.9990	0.5073	x571.40
Coarsening	72.1287	1.6262	0.5957	x486.69
Multiple GPUs	322.2653	19.5558	0.1338	x2174.52
Divergence	431.2731	1.0297	0.0997	x2910.07

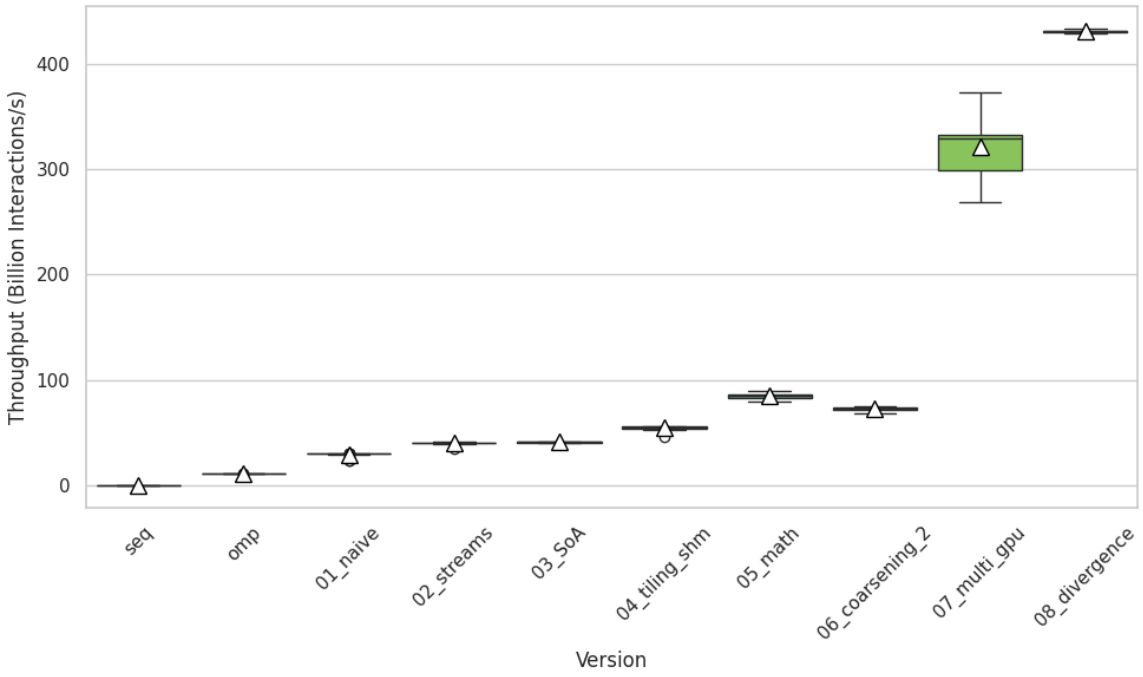


Figure 4: Comparison of the different versions