ECE415: High Performance Computing Systems
Lab5 - Parallelization and optimization of NBody simulation on GPU.

Contents

Introduction

The code simulates the gravitational interaction between stars. It uses a brute force approach where every body calculates the force exerted by every other body. The complexity is $O(N^2)$. For every time step we perform num_systems $\times$ bodies_per_system^2 interactions. The function bodyForce is the computational hotspot. It contains a nested loop where the floating point math happens. In the initial provided code the CPU processes one galaxy (system) at a time sequentially. Since the galaxies are independent and the bodies within a galaxy can be updated simultaneously.

The sequential code took more than 3 minutes to execute so we won't use it to compare performance or correctness.

Hardware and Compiler Flags

All experiments were executed on the `csl-venus` node. The system configuration (reported by `lshw`) is summarized below:

- **CPU:** 2 $\times$ Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz
- **System Memory:** 128 GiB RAM
- **GPU:** 2 $\times$ NVIDIA GK210GL (Tesla K80)

The OpenMP (CPU) implementation was compiled using the Intel oneAPI C/C++ compiler (`icx`) with OpenMP enabled:

```
CC     = icx
CFLAGS = -Wall -qopenmp
```

Listing 1: OpenMP build flags.

The CUDA implementation was compiled with `nvcc`. We enabled aggressive optimization and fast math (for optimizations 5 to 9):

```
CC = nvcc
CFLAGS = -O3 -use_fast_math -Xcompiler -fopenmp
```

Listing 2: CUDA build flags.

CPU parallelization with OpenMP

OpenMP was introduced at the galaxy level by parallelizing the loop that iterates over independent systems.

```
    /* Time-steps */
    for (iter = 1; iter <= nIters; iter++) {
        /* Galaxies */
        #pragma omp parallel for
      for (sys = 0; sys < num_systems; sys++) {
          /* Calculate offset for the galaxy */
          // private pointer for each thread. One thread per system
            Body *system_ptr = &data[sys * bodies_per_system];

          /* Compute forces & integrate for the galaxy */
          bodyForce(system_ptr, dt, bodies_per_system);
          integrate(system_ptr, dt, bodies_per_system);
        }
    }
```

Each iteration of this loop processes a different galaxy which has a complete data independence between threads. We can parallelize each galaxy loop because:

- No synchronization is required
- No shared data structures are modified concurrently
- Each thread operates on a private subset of the global body array

A private pointer (system_ptr) is used so that each thread accesses only the memory region corresponding to its assigned galaxy.

Algorithm Correctness

The internal computation of forces (bodyForce) and position updates (integrate) is unchanged from the sequential implementation. Now the parallel version preserves the same numerical method and computational order within each galaxy.

Due to the floating point arithmetic and non associativity of additions, small numerical deviations are expected. To verify correctness, the final positions of selected bodies (System 0, Bodies 0 and 1) are compared against reference values obtained from the sequential implementation.

Transferring the computation on GPU

0.1 Design Overview

In the second stage of the implementation, the computationally N Body simulation was offloaded to the GPU using CUDA. The goal was to exploit massive data parallelism by mapping the independent computations of bodies and galaxies onto GPU threads and thread blocks.

The GPU implementation parallelization strategy:

- One CUDA thread block per galaxy
- 1024 threads per block (max for devive) each responsible for a subset of bodies

Device Computation

Only two memory transfers are required:

- Host → Device copy of the initial system state
- Device → Host coy of the final results

```
// Launch kernel
void run_device(int num_systems, int bytes, int nIters, int bodies_per_system, \
                float dt) {
    .
    .
    .

    // Copy data on device
    runtime_error = cudaMemcpy((Body *)d_data, (Body *)h_data, bytes, \
                               cudaMemcpyHostToDevice);
    CHECK_CUDA_FAIL(runtime_error);

    // kernel launch
    device_nbody<<<grid_dims, block_dims>>>(d_data, nIters, bodies_per_system, dt);
    cudaDeviceSynchronize();
    runtime_error = cudaGetLastError();
    CHECK_CUDA_FAIL(runtime_error);

    // Copy device output back on host
    runtime_error = cudaMemcpy((Body *)h_device_out, (Body *)d_data, bytes, \
                               cudaMemcpyDeviceToHost);
    CHECK_CUDA_FAIL(runtime_error);

}
```

The force computation and integration steps were writen as device functions operating on a single body:

- device_bodyForce() computes the force acting on one body by iterating over all bodies of the same galaxy
- device_integrate() updates the position of the same body based on its velocity

Each thread independently executes these functions for its assigned bodies. A __syncthreads() barrier is placed at the end of each simulation time step to ensure that all velocity and position updates are completed before proceeding to the next iteration.

This synchronization is required to preserve correctness, as the force computation at the next time step depends on the updated positions of all bodies within the galaxy.

```
// kernel
__global__ void device_nbody(Body *d_data, int nIters, int bodies_per_system, float dt)
    {
    int iter, i;
    int sys = blockIdx.x;
    int stride = blockDim.x;

    /* Calculate offset for the galaxy */
    Body *system_ptr = &d_data[sys * bodies_per_system];

    /* Time-steps */
    for (iter = 1; iter <= nIters; iter++) {
        // 1 thread for bodies_per_system / THREADS_PER_BLOCK bodies
        for (i = threadIdx.x; i < bodies_per_system; i += stride) {
          /* Compute forces & integrate for the body */
          device_bodyForce(system_ptr, dt, bodies_per_system, i);
          device_integrate(system_ptr, dt, i);
        }
        // sync threads in same block
        __syncthreads();
    }
}
```

Each CUDA block is assigned to a single galaxy using the block index:

```
int sys = blockIdx.x;
Body *system_ptr = &d_data[sys * bodies_per_system];
```

The threads within a block cooperatively process all bodies of the assigned galaxy using a strided loop:

```
for (i = threadIdx.x; i < bodies_per_system; i += blockDim.x)
```

With THREADS_PER_BLOCK = 1024, each thread is responsible for bodies_per_system / 1024 bodies and that way we have full coverage of the system even when the number of bodies exceeds the number of threads per block.

```
__device__ void device_bodyForce(Body * p, float dt, int n, int bodyIdx) {
    int j;
    float Fx, Fy, Fz, dx, dy, dz, distSqr, invDist, invDist3;

    Fx = 0.0f;
    Fy = 0.0f;
    Fz = 0.0f;

    for (j = 0; j < n; j++) {
      dx = p[j].x - p[bodyIdx].x;
        dy = p[j].y - p[bodyIdx].y;
        dz = p[j].z - p[bodyIdx].z;
        distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
        invDist = 1.0f / sqrtf(distSqr);
        invDist3 = invDist * invDist * invDist;

        Fx += dx * invDist3;
        Fy += dy * invDist3;
        Fz += dz * invDist3;
    }

    p[bodyIdx].vx += dt * Fx;
    p[bodyIdx].vy += dt * Fy;
    p[bodyIdx].vz += dt * Fz;
}
```

```
__device__ void device_integrate(Body * p, float dt, int bodyIdx) {

  p[bodyIdx].x += p[bodyIdx].vx * dt;
    p[bodyIdx].y += p[bodyIdx].vy * dt;
```

```
    p[bodyIdx].z += p[bodyIdx].vz * dt;
}
```

Correctness Verification

To validate the GPU implementation the final positions of selected bodies are compared with the results of the OpenMP CPU implementation with an absolute difference bellow 0.001.

0.2 Performance

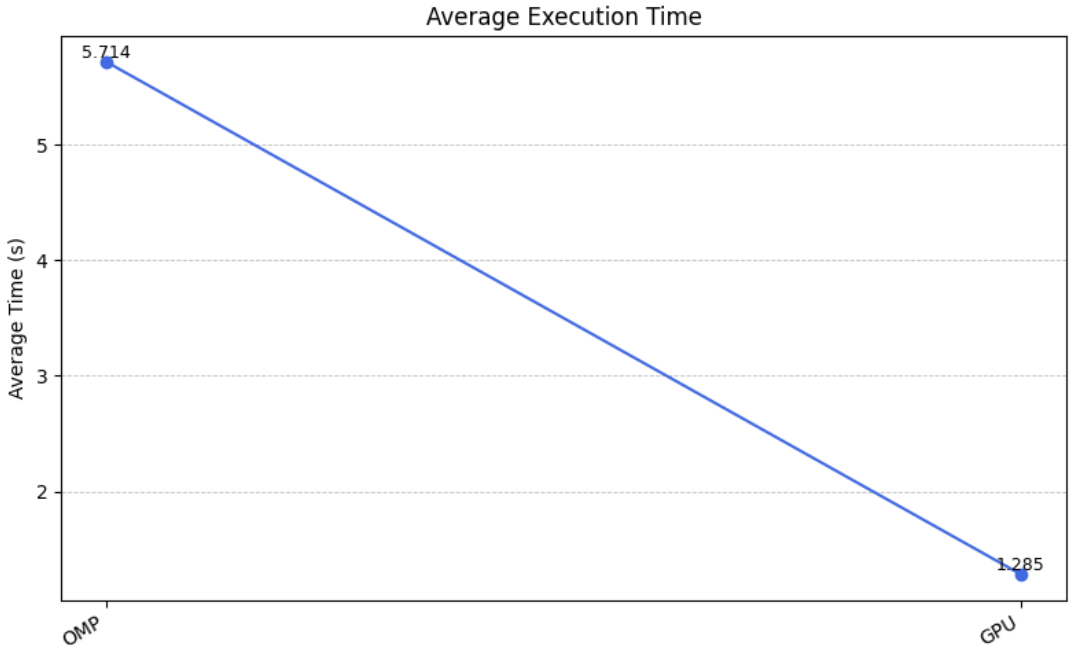

Figure 1: Execution time of GPU compared to OMP.

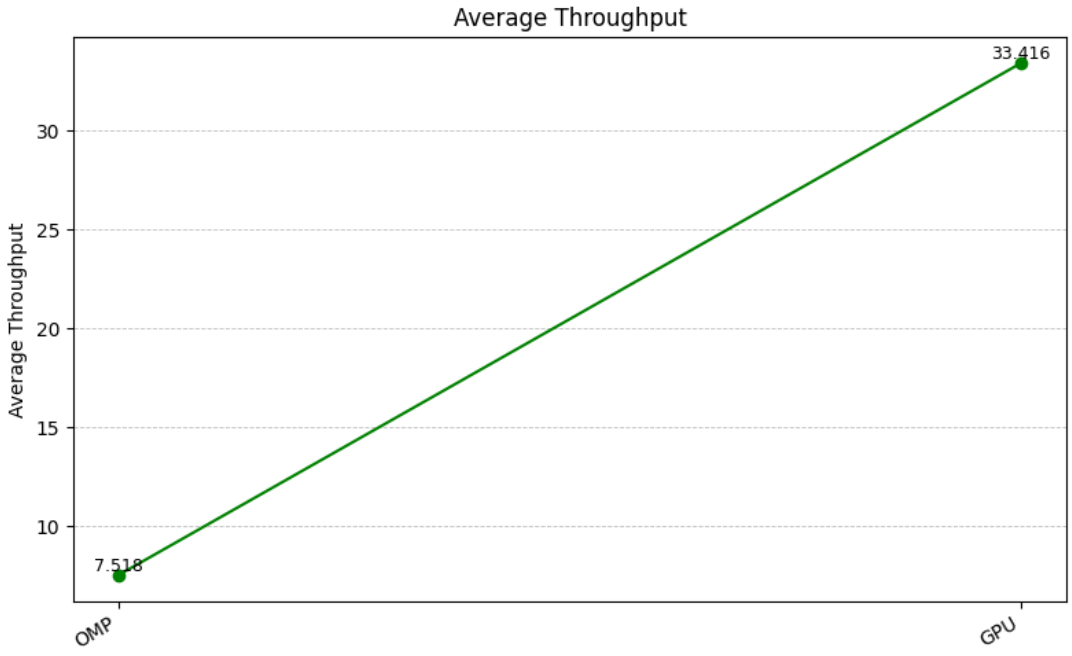

Figure 2: Throughput of GPU compared to OMP.

1 Optimization 1: Data Layout Redesign

The initial working implementation stores each body as a single array of structures (AoS): `Body{x,y,z,vx,vy,vz}`. Both the CPU reference and the GPU kernel operate on a single contiguous `Body*` buffer, and host to device and device to host transfers copy the entire AoS array.
To improve memory access efficiency and enable better coalescing, the data layout was redesigned by separating positions and velocities into distinct arrays.

1.1 Split SoA: Positions + Velocities

Optimization 1 refactors the layout into two separate arrays (a split SoA design):

- Positions: `coords_T{x,y,z}` stored in `coords_T* all_coords`
- Velocities: `velocity_T{vx,vy,vz}` stored in `velocity_T* all_velocities`

The computation model is unchanged (one CUDA block per system, strided loop over bodies, `__syncthreads()` per timestep) but the kernel arguments and memory transfers are updated to reflect the new layout.

```
typedef struct {
    float x, y, z;
} coords_T;

typedef struct {
    float vx, vy, vz;
} velocity_T;

typedef struct {
    coords_T *all_coords;
    velocity_T *all_velocities;
} Universe;

Universe h_data, h_device_out, omp_out, d_data;
```

Each galaxy occupies a contiguous segment of both arrays, preserving spatial locality while we have independent access to position and velocity data. Specifically on device_bodyForce() every thread accesses the coordinates of every body in the system in x, y, z order. Using an array of coordinates memory accesses are sequential without skiping memory addresses in the between.
This redesign allows GPU threads to load only the data they require, reducing unnecessary memory traffic.

1.2 CPU and GPU Adaptation

Both the OpenMP and CUDA implementations were adapted to operate on the new data layout. For the CPU implementation, separate pointers are used for coordinates and velocities:

```
coords_T *coords_ptr = &omp_out.all_coords[sys * bodies_per_system];
velocity_T *veloc_ptr = &omp_out.all_velocities[sys * bodies_per_system];
```

Similarly, the GPU kernel computes offsets independently for the two arrays:

```
coords_T *coords_ptr = &d_all_coords[sys * bodies_per_system];
velocity_T *veloc_ptr = &d_all_velocs[sys * bodies_per_system];
```

This design ensures that all memory accesses remain aligned and contiguous across threads.

1.3 GPU Code changes

The device force computation and integration kernels were modified to operate on separate coordinate and velocity arrays:

- device_bodyForce() reads only position data while updating velocity values
- device_integrate() updates positions using the previously computed velocities

To reduce redundant memory accesses the coordinates of the currently processed body are cached in a local register:

```
coords_T curr = coor_p[bodyIdx];
```

Essentially it reduces repeated global memory loads within the inner interaction loop.

```
    __device__ void device_bodyForce(coords_T * coor_p, velocity_T * vel_p, \
                                float dt, int n, int bodyIdx) {
    .
    .
    .

    coords_T curr = coor_p[bodyIdx];
    for (j = 0; j < n; j++) {
      dx = coor_p[j].x - curr.x;
        dy = coor_p[j].y - curr.y;
        dz = coor_p[j].z - curr.z;
        distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
        invDist = 1.0f / sqrtf(distSqr);
        invDist3 = invDist * invDist * invDist;

        Fx += dx * invDist3;
        Fy += dy * invDist3;
        Fz += dz * invDist3;
    }
    .
    .
    .
}
```

```
    __device__ void device_integrate(coords_T * coor_p, velocity_T * vel_p, \
                                float dt, int bodyIdx) {

    coor_p[bodyIdx].x += vel_p[bodyIdx].vx * dt;
    coor_p[bodyIdx].y += vel_p[bodyIdx].vy * dt;
    coor_p[bodyIdx].z += vel_p[bodyIdx].vz * dt;
}
```

```
// kernel
__global__ void device_nbody(coords_T *d_all_coords, velocity_T * d_all_velocs, int
    nIters, \
                             int bodies_per_system, float dt) {
    int iter, i;
    int sys = blockIdx.x;
    int stride = blockDim.x;

    /* Calculate offset for the galaxy */
    coords_T *coords_ptr = &(d_all_coords[sys * bodies_per_system]);
    velocity_T *veloc_ptr = &(d_all_velocs[sys * bodies_per_system]);

    /* Time-steps */
    for (iter = 1; iter <= nIters; iter++) {
        // 1 thread for bodies_per_system / THREADS_PER_BLOCK bodies
        for (i = threadIdx.x; i < bodies_per_system; i += stride) {
          /* Compute forces & integrate for the body */
          device_bodyForce(coords_ptr, veloc_ptr, dt, bodies_per_system, i);
          device_integrate(coords_ptr, veloc_ptr, dt, i);
        }
        // sync threads in same block
        __syncthreads();
    }
}
```

1.4 Performance

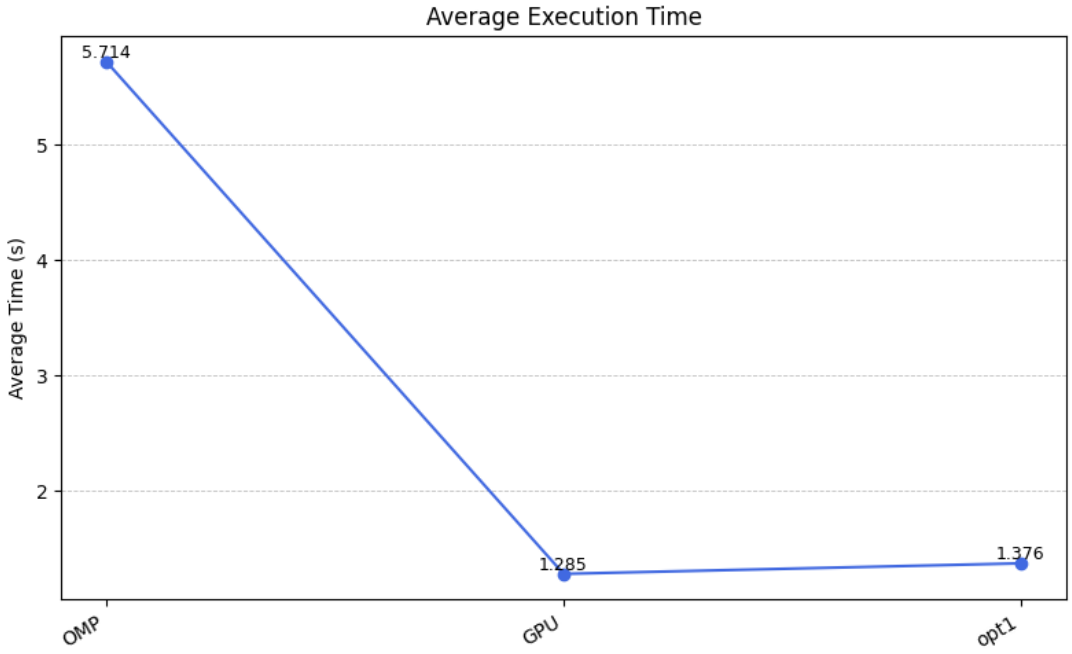

Figure 3: Execution time of opt1 compared to previous versions.

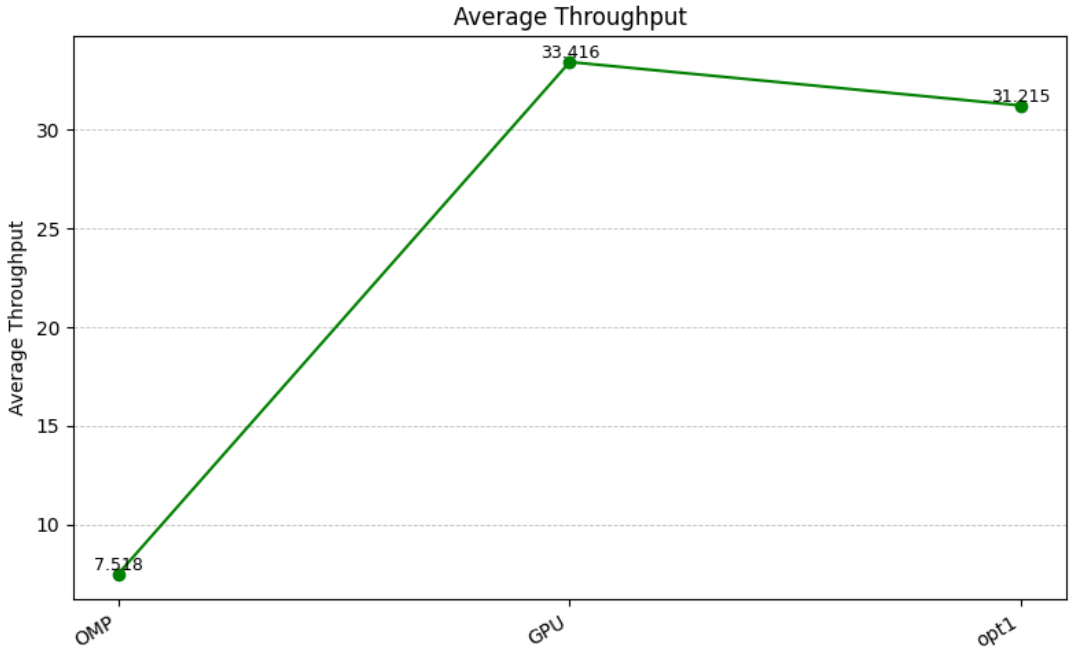

Figure 4: Throughput of opt1 compared to previous versions.

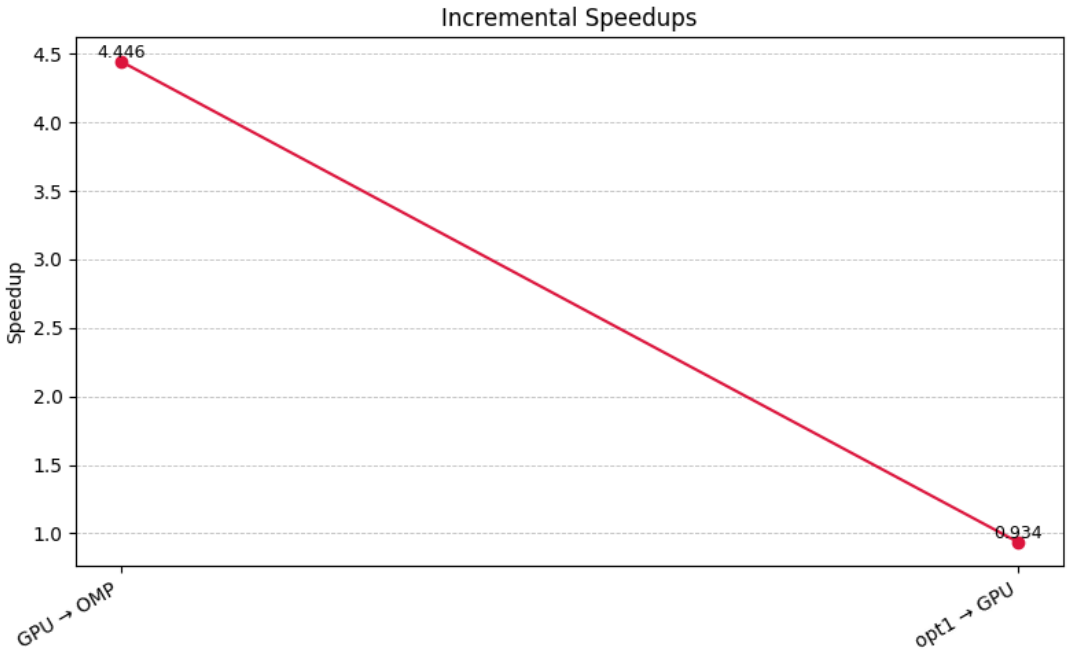

Figure 5: Speedup of opt1 compared to previous versions.

2 Optimization 2: Shared Memory with Tiling

In the initial GPU implementation each thread repeatedly accessed global memory to load the positions of all bodies in the same galaxy. This results in excessive global memory traffic and it also limits the performance.
To reduce redundant memory accesses and better utilize the GPU memory a tiling strategy combined with shared memory was implemented because shared memory accesses are much faster.

2.1 Tiling

On the Tesla K80 every thread block has 48KB of shared memory. Each thread block one galaxy with 8K bodies is assinged. For each body we need 6 floats (x, y, z, vx, vy, vz). In c sizeof(float) = 4 Bytes. For each thread block:

$$bodies\_per\_system * 6 * sizeof(float) = 8K * 6 * 4 = 196,608 \text{ Bytes}$$

Are required to store the whole system. This cannot fit into shared memory and the solution is tilling, load only a subset of the system on shared memory.

2.2 Tiling Strategy

Each galaxy is processed by a single CUDA block and the interaction computation is divided into tiles of fixed size:

```
1 #define TILE_SIZE 512
```

The total number of bodies per galaxy is partitioned into bodies_per_system / TILE_SIZE tiles. For each tile:

- A subset of body coordinates is loaded from global memory into shared memory
- All threads in the block reuse this data to compute partial force contribution
- This approach dramatically reduces the number of global memory loads

2.3 Shared Memory Usage

Shared memory is allocated per block in the kernel to store the coordinates of the current tile:

```
__shared__ coords_T shared_coords[TILE_SIZE];
```

Threads cooperatively fill the shared buffer. A synchronization barrier is pleced so that all shared memory loads complete before computation begins:

```
if (threadIdx.x < TILE_SIZE) {
    s_coor[threadIdx.x] = coor_p[threadIdx.x + tile_offset];
}
__syncthreads();
```

2.4 Implementation

An additional barrier is used for threads to finish the computation before loading the next tile.

```
__device__ void device_bodyForce(coords_T *coor_p, velocity_T *vel_p, coords_T *s_coor,\
                                 float dt, int n, int bodyIdx) {
    int i, j, k;
    int tiles_per_sys, tile_offset;
    float Fx, Fy, Fz, dx, dy, dz, distSqr, invDist, invDist3;
    coords_T curr_coords = coor_p[bodyIdx];

    Fx = 0.0f;
    Fy = 0.0f;
    Fz = 0.0f;

    tiles_per_sys = n / TILE_SIZE;
    // per tile calculation
    for (i = 0; i < tiles_per_sys; i++) {
        tile_offset = i*TILE_SIZE;

        // load shared memory
        if (threadIdx.x < TILE_SIZE) {
            s_coor[threadIdx.x] = coor_p[threadIdx.x + tile_offset];
        }
        __syncthreads();

        // main computation
        for (j = 0; j < TILE_SIZE; j++) {
            dx = s_coor[j].x - curr_coords.x;
            dy = s_coor[j].y - curr_coords.y;
            dz = s_coor[j].z - curr_coords.z;
            distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
            invDist = 1.0f / sqrtf(distSqr);
            invDist3 = invDist * invDist * invDist;

            Fx += dx * invDist3;
            Fy += dy * invDist3;
            Fz += dz * invDist3;
        }
        __syncthreads();
    }

    vel_p[bodyIdx].vx += dt * Fx;
    vel_p[bodyIdx].vy += dt * Fy;
    vel_p[bodyIdx].vz += dt * Fz;
}
```

2.5 Tiles per System Decision

After implemention we tested differ tile sizes to find the optimal.

Table 1: Effect of tile size on performance

TILE_SIZE	Throughput (GInter/s)
2048	26.142
1024	27.178
512	27.295
256	27.161
128	26.833

From experimenting, we ended up using 512.

2.6 Performance

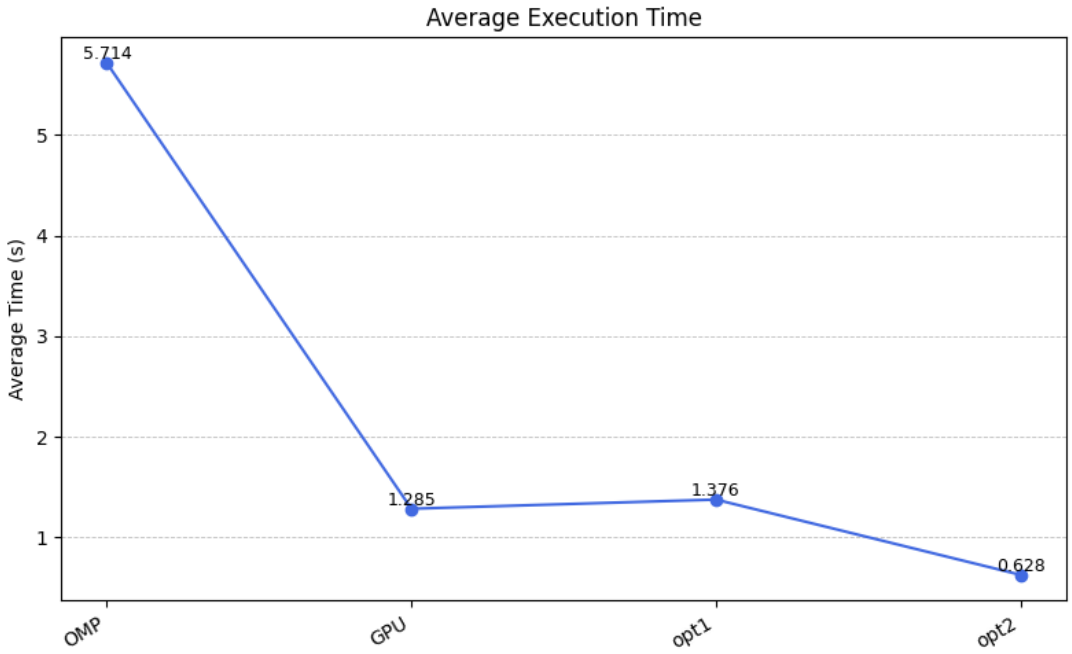

Figure 6: Execution time of opt2 compared to previous versions.

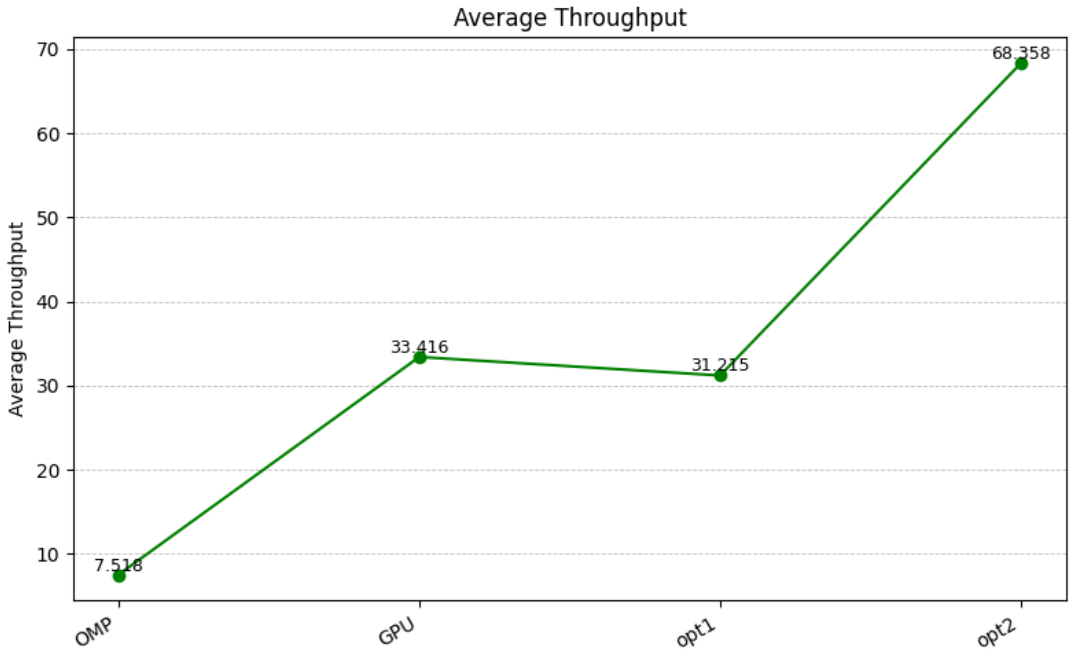

Figure 7: Throughput of opt2 compared to previous versions.

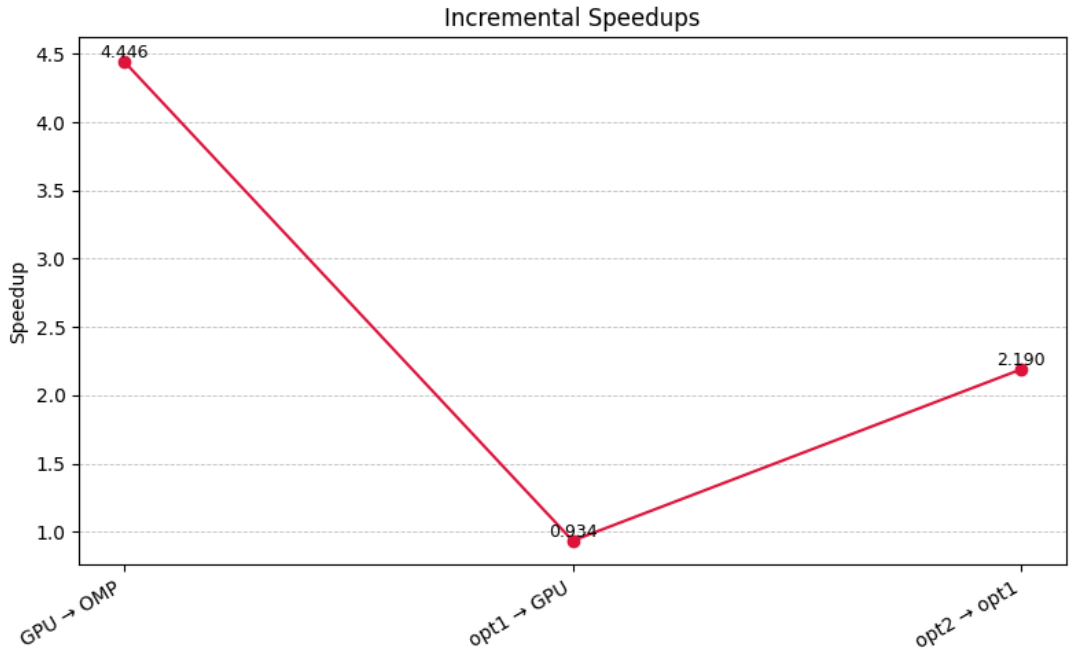

Figure 8: Speedup of opt2 compared to previous versions.

3 Optimization 3: Loop Unrolling

After introducing shared memory tiling, the innermost loop of the force computation iterates over all bodies within a tile and performs floating point expensive operations which are square roots and multiple multiplications computations.
To reduce loop overhead and increase instruction level parallelism manual loop unrolling was applied.

3.1 Implementation

The force computation loop over the shared memory tile:

```
for (j = 0; j < TILE_SIZE; j++) {
    ...
}
```

was unrolled using the following:

```
#pragma unroll 8
```

This instructs the compiler to unroll the loop by a factor of 8 and generates multiple iterations of the loop body per iteration of the control logic
We experimented with number of unrolling to end up with 8.

Table 2: Effect of times unrolling on performance

TILE_SIZE	Throughput (GInter/s)
2	24.626
4	27.152
8	27.457
16	26.151

So now the Force computation kernel is:

```
__device__ void device_bodyForce(coords_T *coor_p, velocity_T *vel_p, coords_T *s_coor,
    \
                                 float dt, int n, int bodyIdx) {
        .
        .
        .

        // main computation
        #pragma unroll 8
        for (j = 0; j < TILE_SIZE; j++) {
            dx = s_coor[j].x - curr_coords.x;
            dy = s_coor[j].y - curr_coords.y;
            dz = s_coor[j].z - curr_coords.z;
            distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
            invDist = 1.0f / sqrtf(distSqr);
            invDist3 = invDist * invDist * invDist;

            Fx += dx * invDist3;
            Fy += dy * invDist3;
            Fz += dz * invDist3;
        }
        __syncthreads();
       (rest of code)
}
```

3.2 Performance

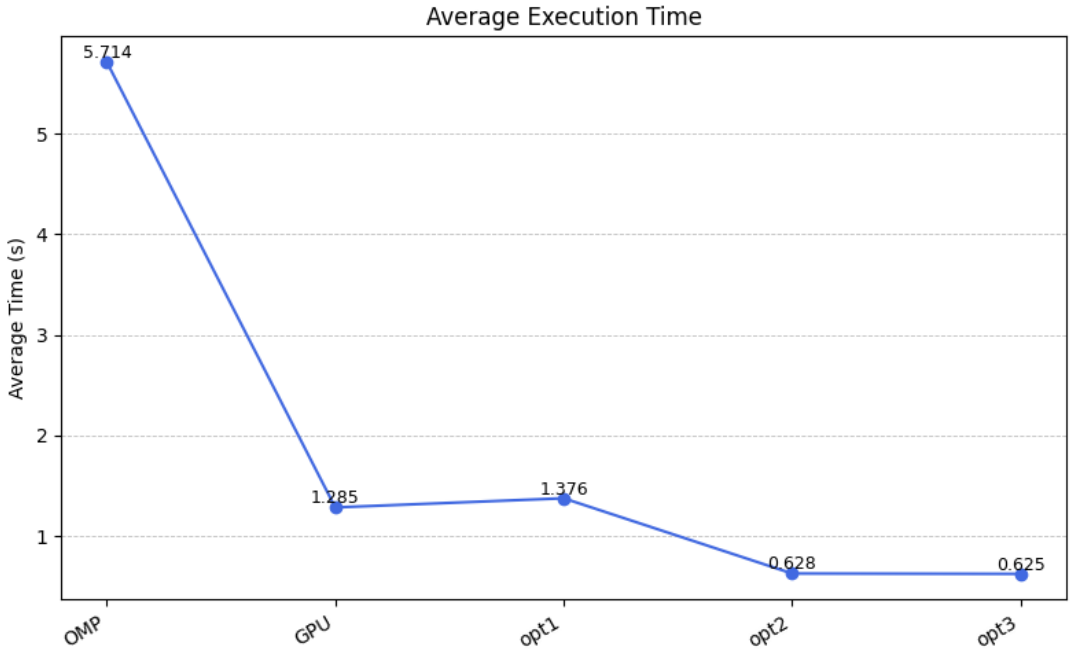

Figure 9: Execution time of opt3 compared to previous versions.

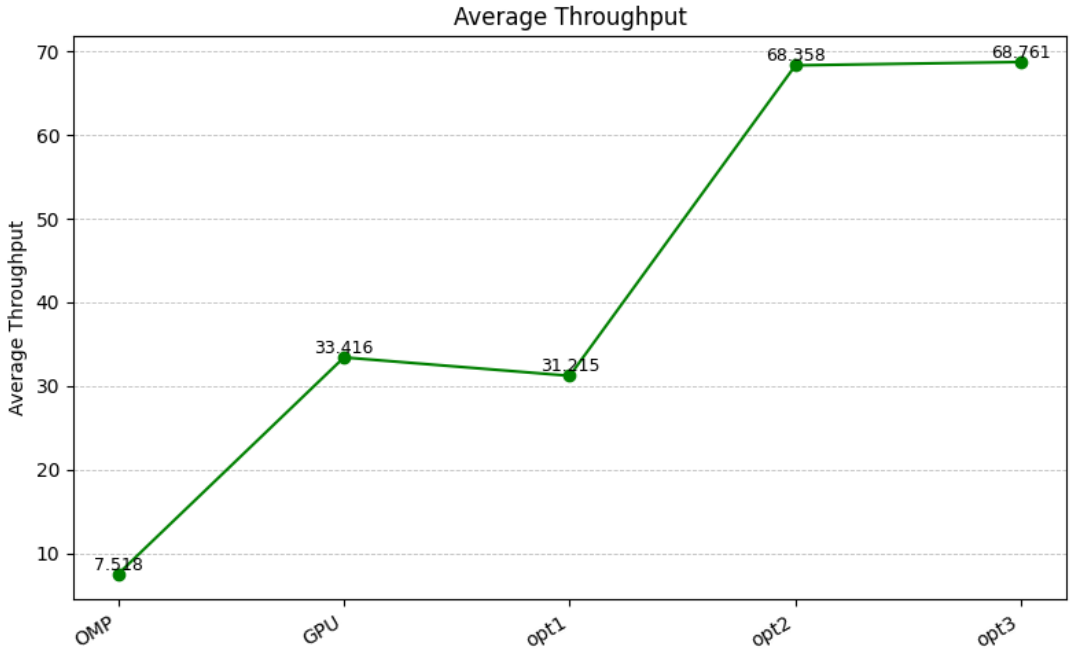

Figure 10: Throughput of opt3 compared to previous versions.

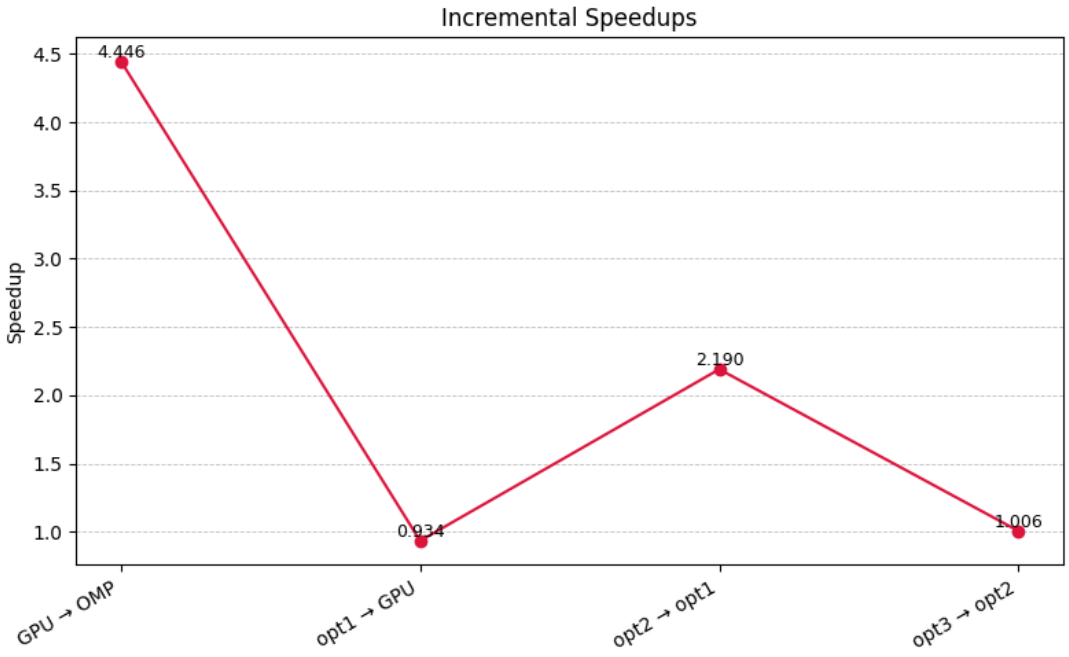

Figure 11: Speedup of opt3 compared to previous versions.

4 Optimization 4: Streams

In the previous GPU implementation all galaxies (systems) were processed through a single execution using the default stream. Although the systems are independent this provides limited control over how work is issued to the GPU.
We introduced **CUDA streams** to enable **concurrent GPU work** across multiple independent galaxies. Instead of submitting all systems through a single execution queue, we assign each galaxy to a stream and enqueue its transfers and kernel launches into that stream. This allows the GPU runtime to processes multiple galaxies at the same time.

4.1 Stream creation

A fixed number of streams is created once during initialization:

```
#define NUM_STREAMS 16
cudaStream_t streams[NUM_STREAMS];

for (int s = 0; s < NUM_STREAMS; s++) {
    cudaStreamCreate(&streams[s]);
}
```

4.2 Mapping galaxies to streams

Each system is mapped to a stream:

```
int stream_id = sys % NUM_STREAMS;
```

Rather than processing all galaxies in one global launch, we handle **each galaxy separately**. For every system `sys` we compute its base offset in the global SoA arrays and issue the corresponding memory transfers and kernel launches to `streams[stream_id]`. This makes each galaxy an independent stream workload that can run alongside others.

- Transfer this systems coordinates and velocities to the device
- Run the simulation timesteps for this system
- Transfer the final coordinates back to the host

```
for (int sys = 0; sys < num_systems; sys++) {
    int stream_id = sys % NUM_STREAMS;
    int offset = sys * bodies_per_system;

    // enqueue transfers for this system
    cudaMemcpyAsync(d_all_coords + offset,
                    h_data.all_coords + offset,
                    bytes_coor_system,
                    cudaMemcpyHostToDevice,
                    streams[stream_id]);

    cudaMemcpyAsync(d_all_velocities + offset,
                    h_data.all_velocities + offset,
                    bytes_vel_system,
                    cudaMemcpyHostToDevice,
                    streams[stream_id]);

    // enqueue compute for this system
    for (int iter = 0; iter < nIters; iter++) {
        update_velocities<<<grid_dims, block_dims, 0, streams[stream_id]>>>(
            d_all_coords + offset,
            d_all_velocities + offset,
            sys, bodies_per_system, dt);

        update_positions<<<grid_dims, block_dims, 0, streams[stream_id]>>>(
            d_all_coords + offset,
            d_all_velocities + offset,
            sys, bodies_per_system, dt);
    }

    // enqueue output transfer for this system
    cudaMemcpyAsync(h_device_out.all_coords + offset,
                    d_all_coords + offset,
                    bytes_coor_system,
                    cudaMemcpyDeviceToHost,
                    streams[stream_id]);
}

// wait for all queued work in all streams to complete
cudaDeviceSynchronize();
```

4.3 Correctness via stream ordering

Within each stream CUDA guarantees in order execution. For each galaxy:

1. velocity updates complete before position integration within the same timestep, and
2. all timesteps execute sequentially in the intended order

A final `cudaDeviceSynchronize()` is used to ensure all streams have finished before timing is recorded and results are accessed on the host.

4.4 Performance

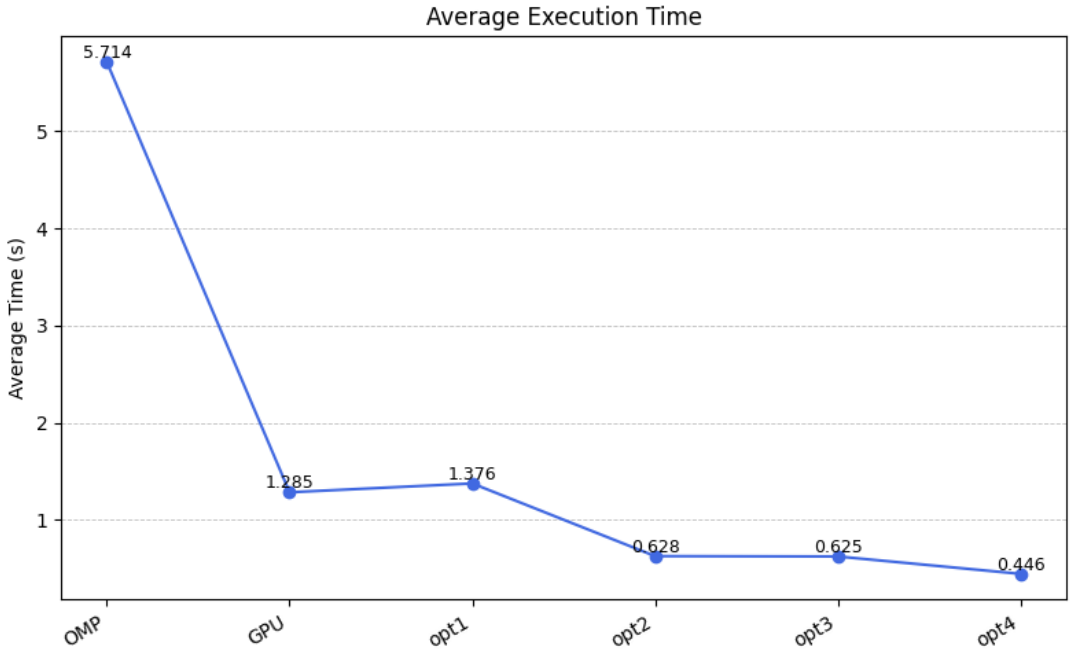

Figure 12: Execution time of opt4 compared to previous versions.

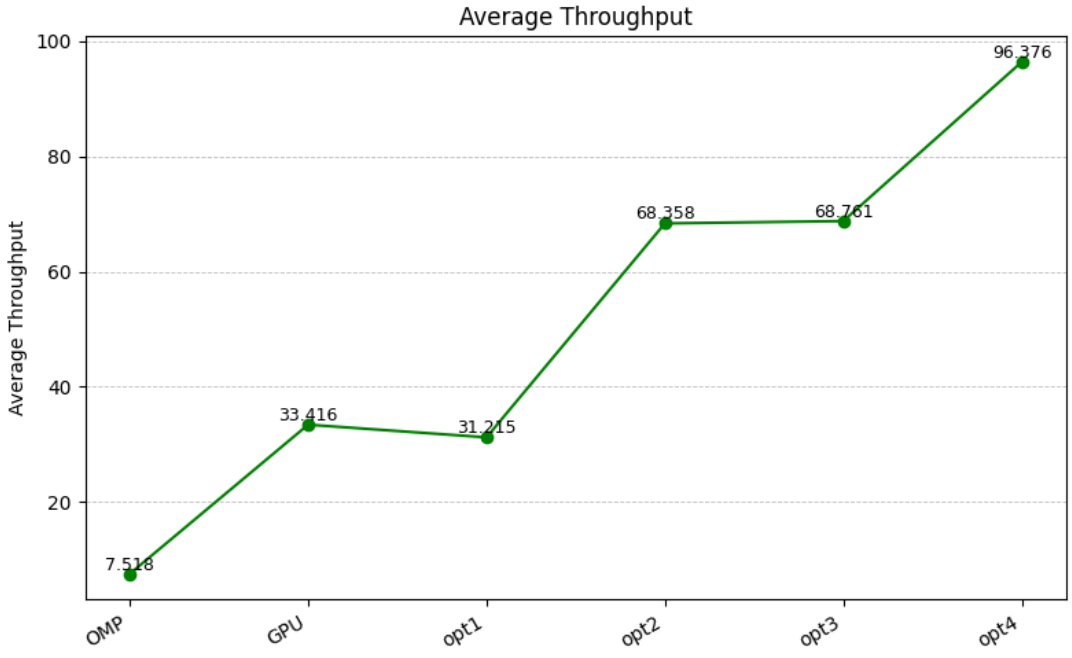

Figure 13: Throughput of opt4 compared to previous versions.

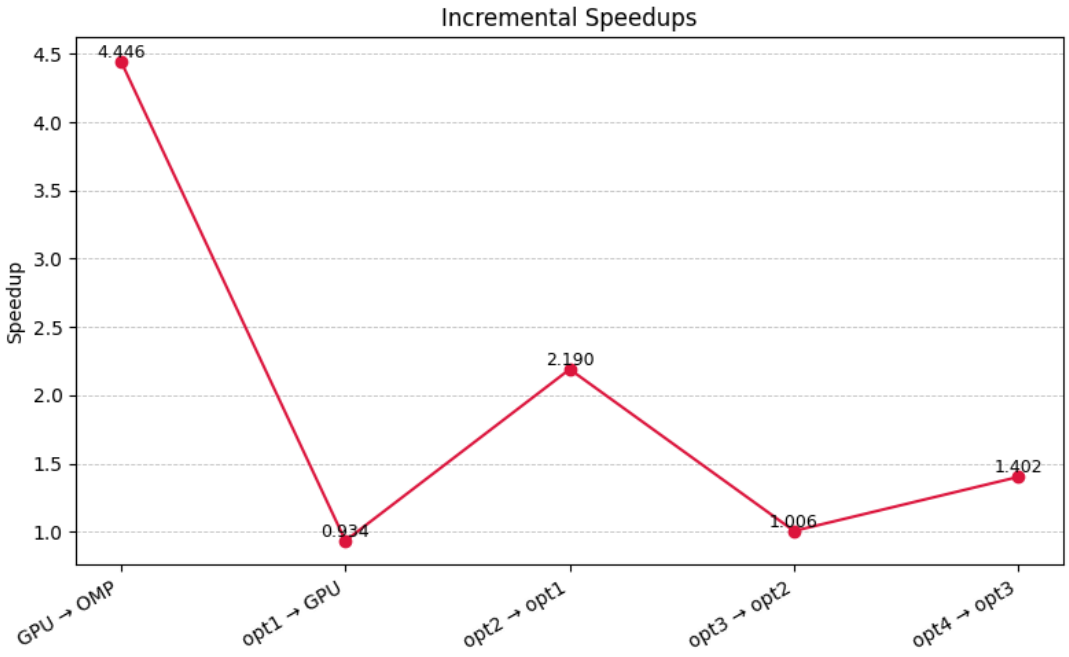

Figure 14: Speedup of opt4 compared to previous versions.

5 Optimization 5: Read Only Caching and Fast Math

After Optimization 4 (streams,multiple blocks per system, two kernel split) most of the runtime is spent in the force evaluation inside `update_velocities`. Even with shared memory tiling and loop unrolling, each body still performs $O(N)$ interactions and the inner loop contains expensive operations (square root and division) while also reading large amounts of coordinate data repeatedly.
In this optimization we implement these three improvements inside the velocity update kernel:

- **More efficient global coordinate loads** using the GPU read only cache.
- **Lower arithmetic cost** in the inner loop by replacing `sqrtf` and division with `rsqrtf`.
- **Correct tiling for any** N (no assumption that `bodies_per_system` is divisible by `TILE_SIZE`).

5.1 (A) Read only loads for coordinate data (`const`, `__restrict__`, `__ldg`)

In Optimization 4, coordinate loads were performed normally from global memory. In Optimization 5 since the coordinates are read only during force evaluation we update the kernel to:

```
__global__ void update_velocities(const coords_T *__restrict__ coords_sys,
                                  velocity_T *__restrict__ velocs_sys,
                                  int bodies_per_system, float dt);
```

`const` allows the compiler to treat loads as read only and `__restrict__` informs the compiler that `coords_sys` and `velocs_sys` do not alias and thus enabling more aggressive optimization.
During the tile load into shared memory we use `__ldg()` to load through the read only cache:

```
tmp.x = __ldg(src + 0);
tmp.y = __ldg(src + 1);
tmp.z = __ldg(src + 2);
```

This reduces memory latency for the coordinate values that are reused heavily in the $O(N^2)$ interaction loop.

5.2 (B) Fast math: replacing `sqrtf` with `rsqrtf` and using FMA

In Optimization 4 the inverse distance was computed as:

```
invDist = 1.0f / sqrtf(distSqr);
```

In Optimization 5, we replace this with:

```
invDist = rsqrtf(distSqr);
```

`rsqrtf` computes an approximation of $1/\sqrt{x}$ with significantly lower latency than `sqrtf` followed by division. Given the allowed correctness tolerance (10^{-3}) this approximation remains acceptable and the CPU produced results match the GPU results.
Additionally the squared distance is computed using fused multiply add operations (FMA) to reduce instruction overhead:

```
distSqr = fmaf(dx, dx, fmaf(dy, dy, fmaf(dz, dz, SOFTENING)));
```

5.3 (C) Robust tiling bounds (correct for any `bodies_per_system`)

Optimization 4 assumed that `bodies_per_system` is divisible by `TILE_SIZE` (the number of tiles was computed as `n / TILE_SIZE`). In Optimization 5 we remove this limitation by using ceiling division:

```
const int tiles_per_sys = (bodies_per_system + TILE_SIZE - 1) / TILE_SIZE;
```

For the last tile we limit the valid range:

```
int tile_end = min(TILE_SIZE, bodies_per_system - tile_offset);
```

This prevents out of bounds loads and works even if `bodies_per_system` changes.

5.4 Performance

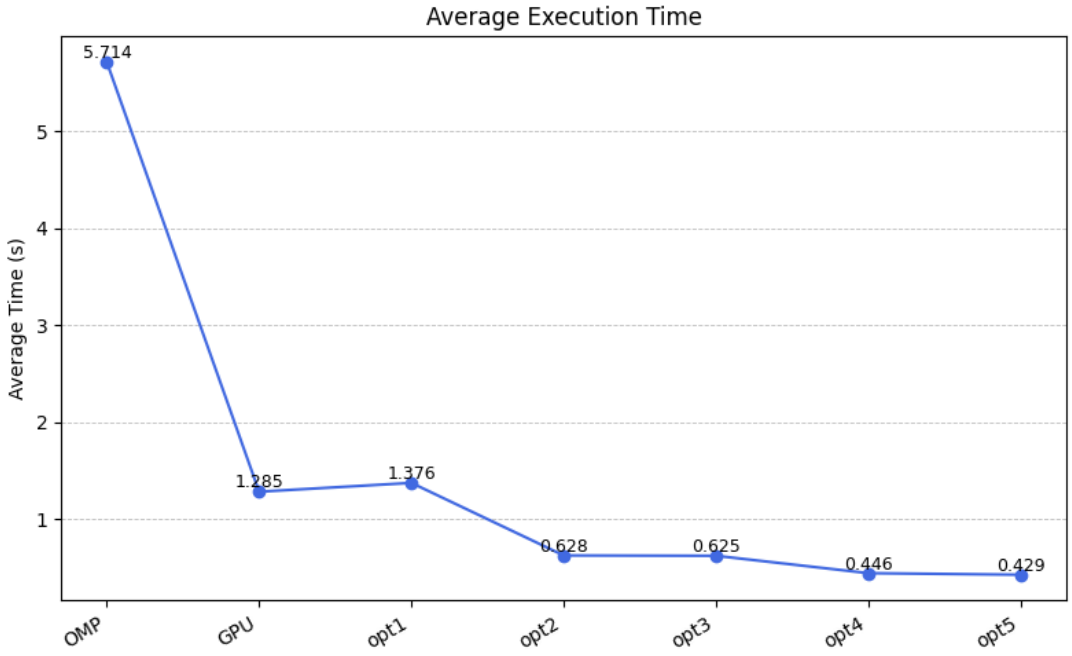

Figure 15: Execution time of opt5 compared to previous versions.

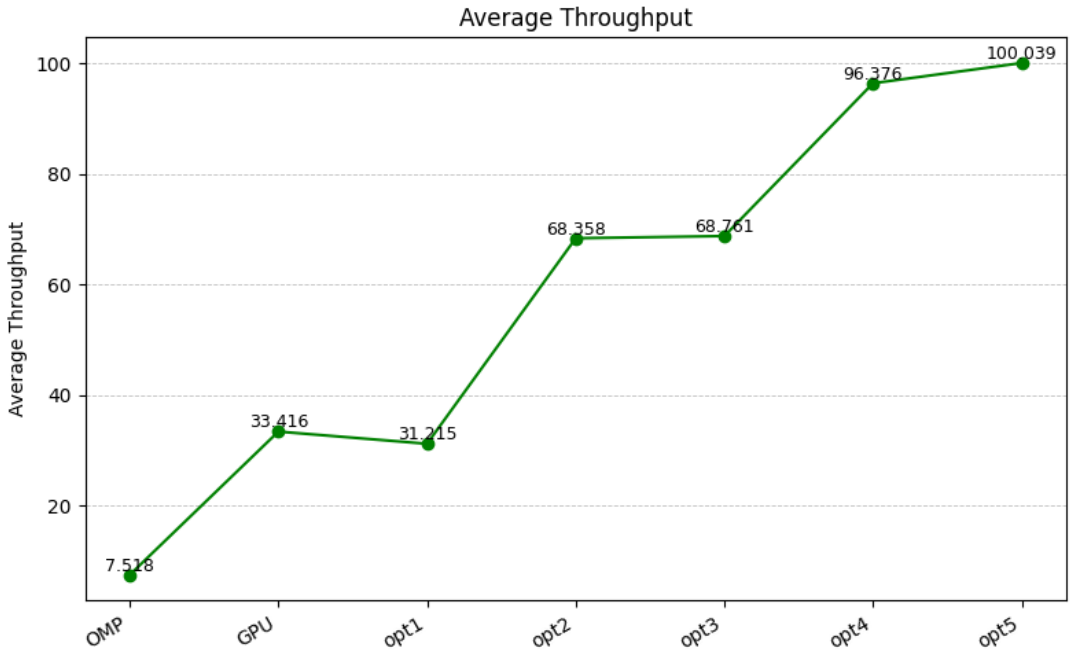

Figure 16: Throughput of opt5 compared to previous versions.

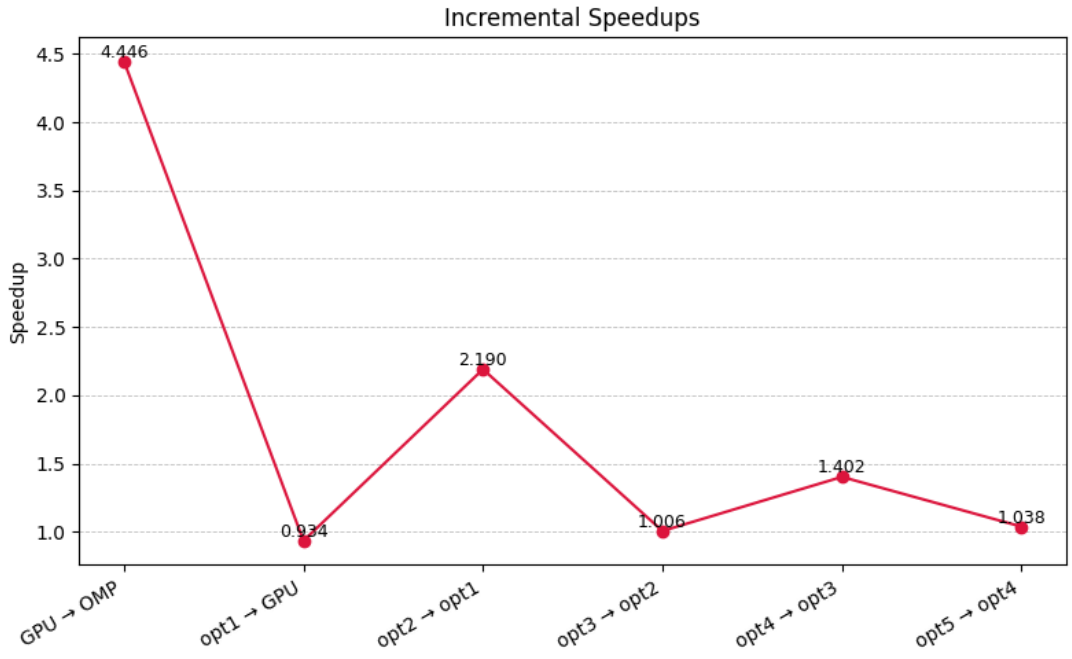

Figure 17: Speedup of opt5 compared to previous versions.

6 Optimization 6: Overlapping Transfers and Compute with CUDA Streams

6.1 Motivation

After Optimization 5 the GPU kernels were significantly faster (read only loads, fast math, robust tiling). Now that we have extracted all the performance from the kernels we focus on the data flow and executing the simulation sequentially per galaxy still leaves performance on the table: host-to-device (H2D) copies,

kernel execution and device-to-host (D2H) copies create idle gaps if they are performed synchronously.

Since the GPU provides two DMA copy engines, it can perform *bidirectional* transfers concurrently: one engine for the Host 2 Device (H2D) copies while the other services Device to Host (D2H) copies. Combined with concurrent kernel execution on the SMs this enables a three-way overlap (H2D + Compute + D2H) across different streams.

Optimization 6 focuses on **overlapping memory transfers with computation** by using multiple CUDA streams. Since galaxies (systems) are independent, they can be processed concurrently: while one galaxy is executing its kernels on the GPU another galaxy can be transferring data.

6.2 Stream based design overview

We create a fixed pool of streams:

```
#define NUM_STREAMS 16
cudaStream_t streams[NUM_STREAMS];
```

Each galaxy is assigned to a stream:

```
int stream_id = sys % NUM_STREAMS;
```

All work for a given galaxy is enqueued into its assigned stream in this order:

1. H2D copy of coordinates and velocities (`cudaMemcpyAsync`)
2. `nIters` time steps (kernel launches for `update_velocities` and `update_positions`)
3. D2H copy of final coordinates (`cudaMemcpyAsync`)

Because operations in the same stream execute in order the correctness is preserved for each galaxy and because different streams can run concurrently the GPU can overlap transfers and kernel execution across galaxies.

6.3 Asynchronous memory transfers and pinned host memory

To enable asynchronous transfers host arrays are allocated with pinned (page-locked) memory using `cudaHostAlloc`. This allows DMA engines to transfer data without paging overhead and without synchronization:

```
cudaHostAlloc((coords_T**)&h_data.all_coords, bytes_coords, cudaHostAllocMapped);
cudaHostAlloc((velocity_T**)&h_data.all_velocities, bytes_veloc, cudaHostAllocMapped);
cudaHostAlloc((coords_T**)&h_device_out.all_coords, bytes_coords, cudaHostAllocMapped);
```

Each galaxy (system) is assigned to one of the available streams:

```
for (int sys = 0; sys < num_systems; sys++) {
    int stream_id = sys % NUM_STREAMS;
    int offset    = sys * bodies_per_system;

    // H2D copies
    cudaMemcpyAsync(d_all_coords + offset,
                    h_data.all_coords + offset,
                    bytes_coor_system,
                    cudaMemcpyHostToDevice,
                    streams[stream_id]);

    cudaMemcpyAsync(d_all_velocities + offset,
                    h_data.all_velocities + offset,
                    bytes_vel_system,
                    cudaMemcpyHostToDevice,
                    streams[stream_id]);

    // nIters time steps ordered after H2D)
    for (int iter = 0; iter < nIters; iter++) {
        update_velocities<<<grid_dims, block_dims, 0, streams[stream_id]>>>(
            d_all_coords + offset,
            d_all_velocities + offset,
            bodies_per_system,
            dt);
```

```
25
26          update_positions<<<grid_dims, block_dims, 0, streams[stream_id]>>>(
27              d_all_coords + offset,
28              d_all_velocities + offset,
29              bodies_per_system,
30              dt);
31      }
32
33      // D2H copy after kernels in the same stream)
34      cudaMemcpyAsync(h_device_out.all_coords + offset,
35                      d_all_coords + offset,
36                      bytes_coor_system,
37                      cudaMemcpyDeviceToHost,
38                      streams[stream_id]);
39  }
40
41  // Wait for all work in all streams to complete
42  cudaDeviceSynchronize();
```

6.4 Overlapping behavior (what runs concurrently)

With multiple streams the GPU is able to overlap:

- H2D transfers of galaxy s+1 with kernel execution of galaxy s
- D2H transfers of galaxy s with kernel execution of galaxy s+1

At the end we wait for all queued work across all streams:

```
1  cudaDeviceSynchronize();
```

This ensures all transfers and kernels are complete before reading results.
Bellow is a figure on how overlapping works conceptually, (note: durations are not accurate)

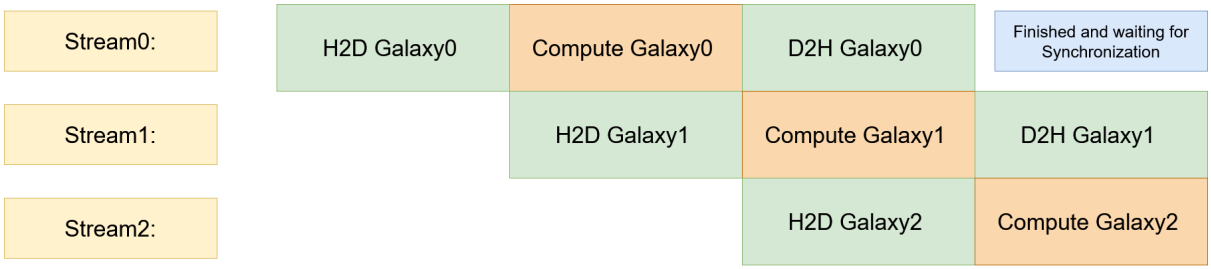

Figure 18: Conceptual Overlapping Streams

We also had to test if the overlap actually works and we found:

Table 3: Measured per galaxy stage times

Stage (per galaxy)	**first galaxies**	**later galaxies**
H2D transfer (ms)	0.127 (1st), 0.047 (2nd)	≈ 0.033
Compute (ms)	69.618 (up to ~3rd)	$\approx 59 \rightarrow 45$
D2H transfer (ms)	0.032	$0.032 \rightarrow 0.027$

So we can conclude that overlapping makes sense and the memory transfers are completely hidden.

6.5 Performance

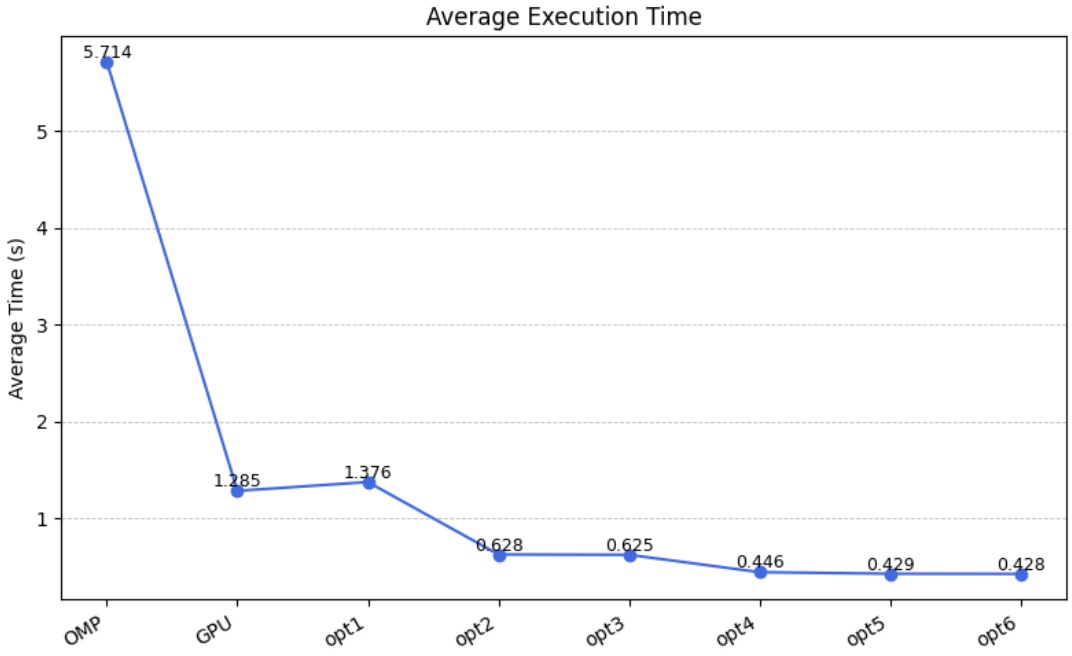

Figure 19: Execution time of opt6 compared to previous versions.

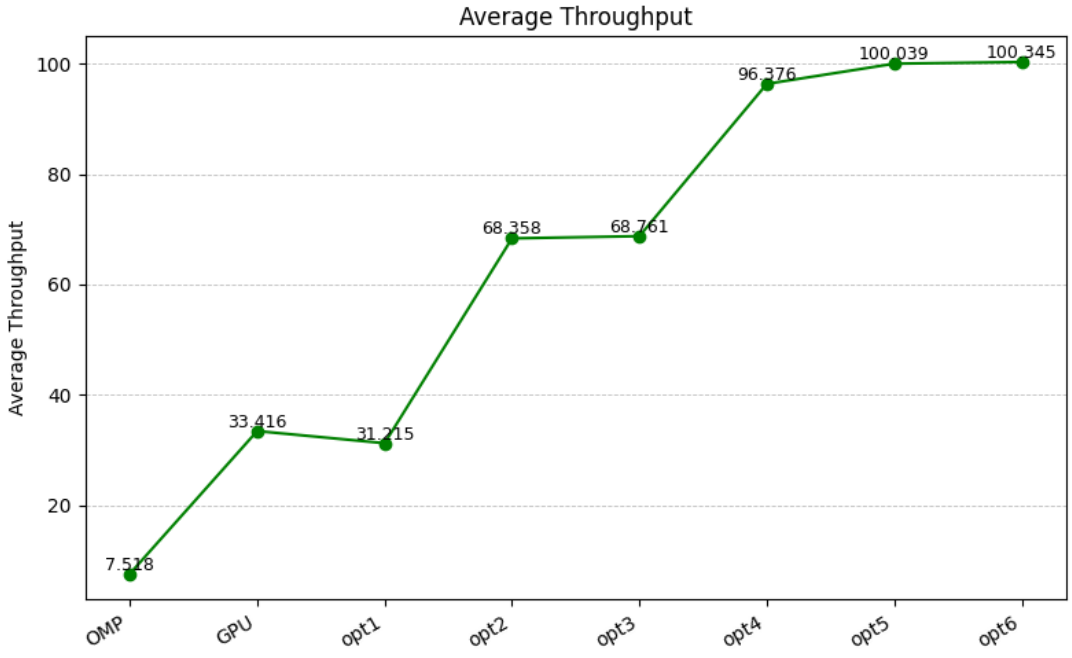

Figure 20: Throughput of opt6 compared to previous versions.

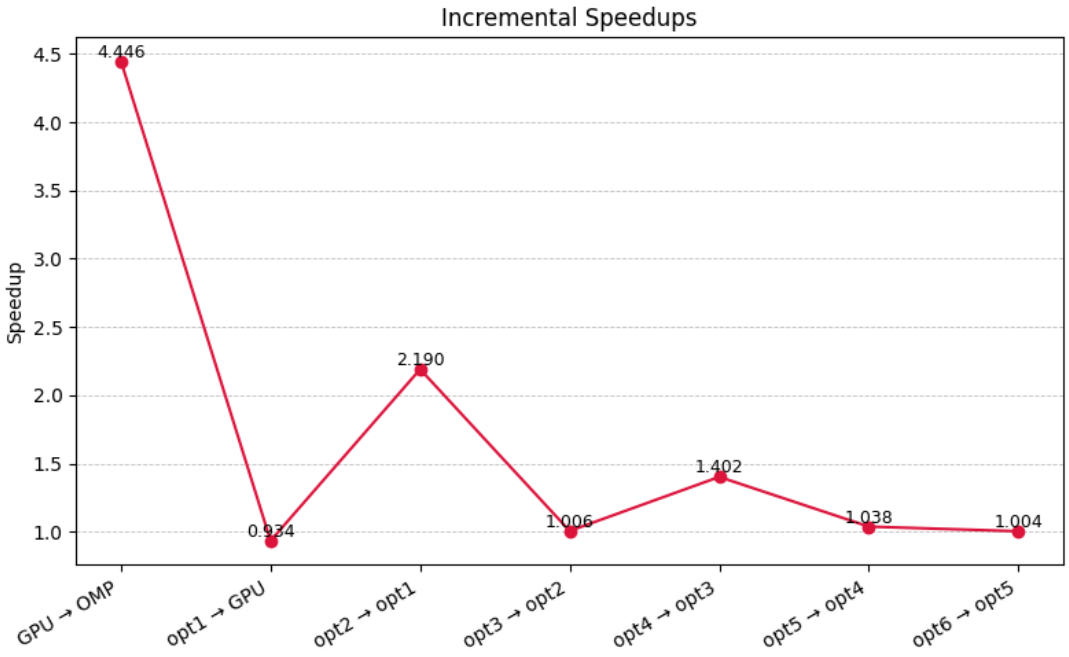

Figure 21: Speedup of opt6 compared to previous versions.

7 Optimization 7: Further stream optimization

As we observe from table bellow the kernel execution time per galaxy is orders of magnitude larger than the transfer times.

Table 4: Measured per galaxy stage times

Stage (per galaxy)	**first galaxies**	**later galaxies**
H2D transfer (ms)	0.127 (1st), 0.047 (2nd)	≈ 0.033
Compute (ms)	69.618 (up to ∼3rd)	$\approx 59 \rightarrow 45$
D2H transfer (ms)	0.032	$0.032 \rightarrow 0.027$

In the worst case a single host to device transfer takes approximately 0.127 ms (≈ 0.130 ms). With 32 galaxies the total worst case H2D time would be:

$$32 \times 0.130 \text{ ms} = 4.16 \text{ ms}$$

which is still much smaller than the compute time of a single galaxy (about 70 ms for the first galaxies). Therefore while the GPU is computing one galaxy, the system can concurrently transfer the input data of many other galaxies effectively doing *prefetching* for the next work. Essentially we are enqueuing all H2D copies early so that transfer overhead is hidden behind computation once the pipeline is filled.
Let's see how the overlapping has changed from the previous optimization:

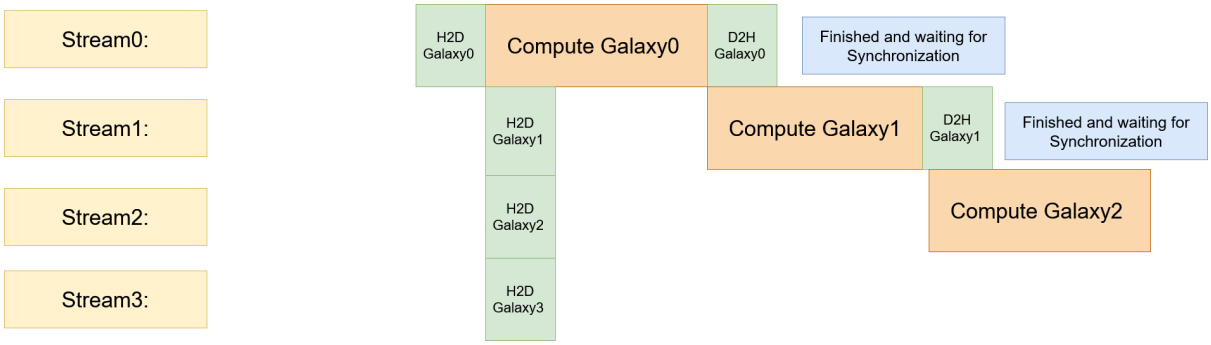

Figure 22: Conceptual Optimized Overlapping Streams

In this version we enqueue all host to device (H2D) transfers first (for all galaxies and all streams). Since the per galaxy compute time is much larger than the transfer time the H2D transfers for the remaining galaxies can be performed concurrently while the GPU is computing the first galaxy. As a result the H2D cost is completely hidden behind computation, with the main visible overhead occurring only during the transfer for the first galaxy. executions).

Similarly, device to host (D2H) copies of completed galaxies can overlap with the computation of other galaxies in different streams. Since our GPU has two copy engines, H2D and D2H transfers can also overlap each other while kernels execute and hiding completly the transfer overhead.

7.1 Code changes from the previous optimization

The main change compared to the previous stream version is the scheduling on the host side. Instead of processing each galaxy start to end (H2D → kernels → D2H) inside one loop, the work now is enqueued in three phases:

```
// Phase 1: enqueue all H2D copies
for (int sys = 0; sys < num_systems; sys++) {
    int stream_id = sys % NUM_STREAMS;
    int offset    = sys * bodies_per_system;

    cudaMemcpyAsync(d_all_coords + offset,
                    h_data.all_coords + offset,
                    bytes_coor_system,
                    cudaMemcpyHostToDevice,
                    streams[stream_id]);

    cudaMemcpyAsync(d_all_velocities + offset,
                    h_data.all_velocities + offset,
                    bytes_vel_system,
                    cudaMemcpyHostToDevice,
                    streams[stream_id]);
}
```

```
// Phase 2: enqueue all computations (kernels)
for (int sys = 0; sys < num_systems; sys++) {
    int stream_id = sys % NUM_STREAMS;
    int offset    = sys * bodies_per_system;

    for (int iter = 0; iter < nIters; iter++) {
        update_velocities<<<grid_dims, block_dims, 0, streams[stream_id]>>>(
            d_all_coords + offset,
            d_all_velocities + offset,
            bodies_per_system,
            dt);

        update_positions<<<grid_dims, block_dims, 0, streams[stream_id]>>>(
            d_all_coords + offset,
            d_all_velocities + offset,
            bodies_per_system,
            dt);
    }
```

```
}
```

```
// Phase 3: enqueue all D2H copies
for (int sys = 0; sys < num_systems; sys++) {
    int stream_id = sys % NUM_STREAMS;
    int offset    = sys * bodies_per_system;

    cudaMemcpyAsync(h_device_out.all_coords + offset,
                    d_all_coords + offset,
                    bytes_coor_system,
                    cudaMemcpyDeviceToHost,
                    streams[stream_id]);
}

// wait for all streams
cudaDeviceSynchronize();
```

Each galaxy is assigned to a fixed stream via `stream_id = sys % NUM_STREAMS`. CUDA guarantees in order execution within a stream and the above three phase enqueue still preserves the per galaxy order: H2D → kernels → D2H, while overlapping transfers and compute across different streams.

Attempted Optimization: Kernel Fusion with Double Buffering (Lower Throughput)

After optimizing the force kernel (tiling, readonly loads, fast inverse square root) and improving overlap with CUDA streams we tried fusing the two kernels per iteration into a single kernel and using double buffering for coordinates. The main goals were:

- reduce kernel launch overhead by launching one kernel per iteration instead of two
- increase control over the update order inside a time step
- avoid additional global synchronization points between separate kernels which we believed that would make the bigger difference

Approach

We allocated a second coordinate buffer d_all_coords2 and replaced the two kernels update_velocities and update_positions with a single fused kernel compute_kernel. At each iteration we alternate input and output coordinate buffers:

```
coords_T *coords_in  = (iter % 2 == 0) ? (d_all_coords  + offset)
                                       : (d_all_coords2 + offset);
coords_T *coords_out = (iter % 2 == 0) ? (d_all_coords2 + offset)
                                       : (d_all_coords  + offset);

compute_kernel<<<grid_dims, block_dims, 0, streams[stream_id]>>>(
    coords_in, coords_out,
    d_all_velocities + offset,
    bodies_per_system, dt);
```

The fused kernel computes forces using coords_in, updates its own velocities and writes the updated positions into coords_out. Shared memory tiling is still used to reduce global memory traffic.

Outcome

This approach resulted in **lower throughput** compared to the best previous version, so we consider it a failed optimization attempt. This was expected for two main reasons:

- **Avoiding extra global synchronization was not beneficial in our case.** We expected the boundary between `update_velocities` and `update_positions` to be a major overhead. That boundary is only one kernel launch separation per iteration and its cost is tiny compared to the tens of milliseconds spent in force evaluation. Removing this synchronization point did not yield a measurable gain and the overall throughput decreased (by 10GInt/s).

- **Reducing kernel launches (2 → 1 per iteration) did not matter.** Removing one launch per step has a negligible impact on the overall runtime and throughput because the most time is spent on the $\mathcal{O}(N^2)$ force computation.
- **Update order didn't help with speedup** The fused kernel computes forces, updates velocities and integrates positions in one place the biggest cost is still the force loop. The additional bookkeeping (extra pointers and a larger kernel body) and the 'local data' did not reduce the expensive part of the computation.

7.2 Performance

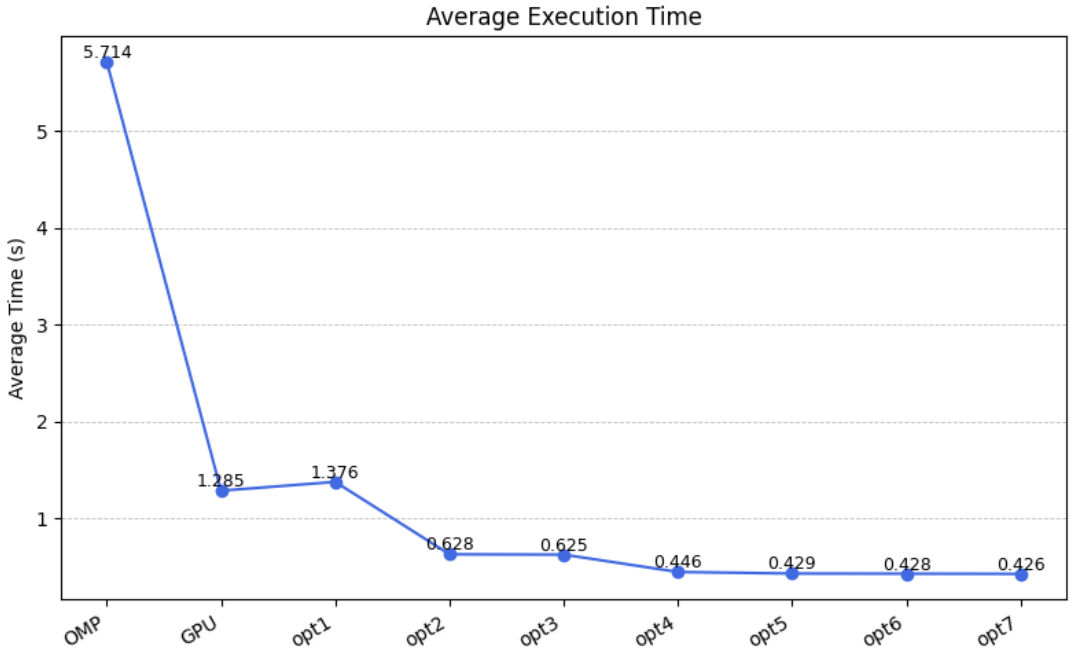

Figure 23: Execution time of opt7 compared to previous versions.

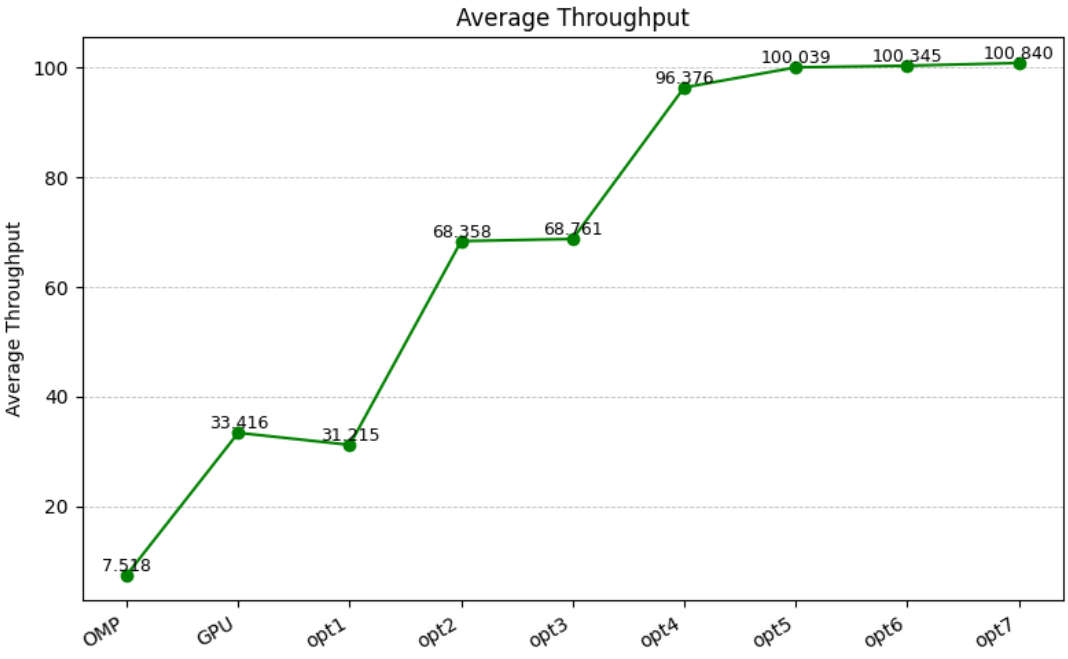

Figure 24: Throughput of opt7 compared to previous versions.

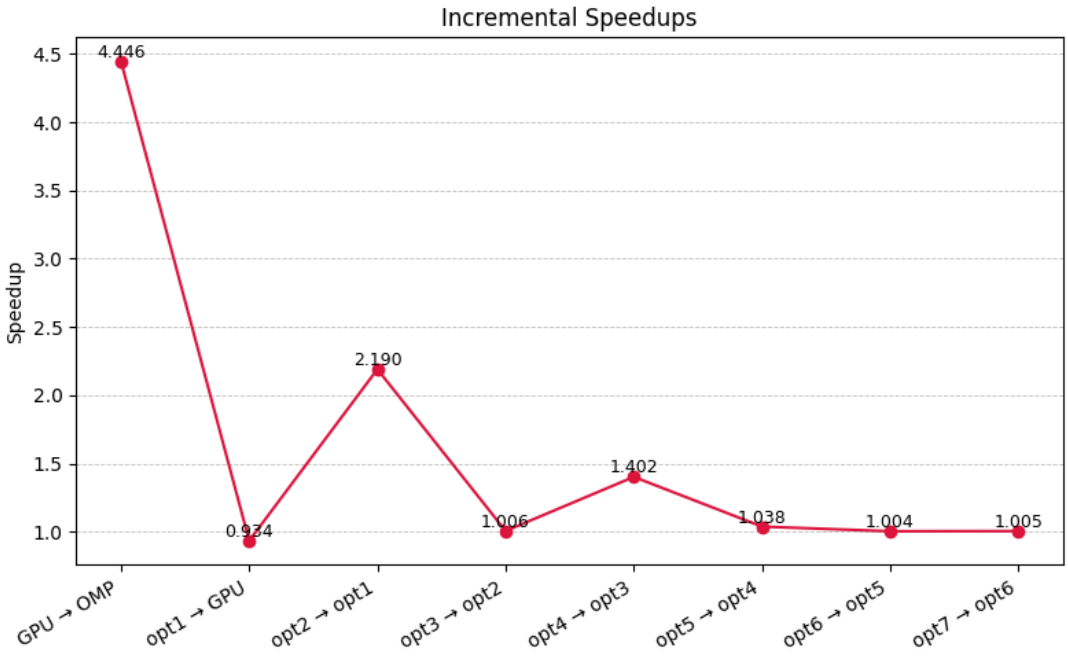

Figure 25: Speedup of opt7 compared to previous versions.

8 Optimization 8: Multiple GPUs

Each system is independent and since we have to deal with multiple systems we can 'spread' the work to multiple GPUs with each one working independently and concurrently.

8.1 What changes for multiple GPU support

The previous version used:

- one global device buffer d_data for holding all systems on a single GPU
- one global stream pool streams[NUM_STREAMS] created once in main
- a single entry run_device that assumed all work targets the current device

The multiple GPU version has:

- a new struct per_Device for holding the private device pointers and streams
- work partitioning across GPUs (each GPU owns a range of systems)
- a new entry point run_device_multi_gpu that iterates over devices, sets the active device(GPU) via cudaSetDevice and enqueues the same pipeline on each GPU

Also we have to mention that the GPU kernels update_velocities and update_positions remains unchanged and multiple GPU support is done by changing the host side and memory management.

8.2 Per GPU state

We store device pointers and streams in a struct:

```
typedef struct {
    Universe d;                          // device buffers for this GPU
    cudaStream_t streams[NUM_STREAMS];   // streams for this GPU
    int sys_start;                       // global system start index
    int num_systems;                     // number of systems assigned to this GPU
} DeviceContext;
```

Each GPU allocates only the memory needed for its own work:

$$\texttt{local\_bodies} = \texttt{num\_systems\_local} \times \texttt{bodies\_per\_system}.$$

8.3 Partitioning Systems Across GPUs

We query the number of GPUs using `cudaGetDeviceCount` and split the work:

```
int dev_count = 0;
CUDA_CHECK(cudaGetDeviceCount(&dev_count));

int base  = num_systems / dev_count;
int extra = num_systems % dev_count;

int sys_start = 0;
for (int dev = 0; dev < dev_count; dev++) {
    int nsys = base + (dev < extra ? 1 : 0);
    context[dev].sys_start   = sys_start;
    context[dev].num_systems = nsys;
    sys_start += nsys;
}
```

This ensures all systems are assigned exactly once and the remainder is distributed evenly across the first GPUs.

8.4 New Indexing

A difference from the single GPU version is that each GPU stores only its local systems so we changed it into :

- global system id `global_sys`
- local system id `local_sys`

The offsets are:

$$\texttt{host\_off} = \texttt{global\_sys} \cdot \texttt{bodies\_per\_system}, \qquad \texttt{dev\_off} = \texttt{local\_sys} \cdot \texttt{bodies\_per\_system}$$

Transfers use `host_off` on pinned host arrays and `dev_off` on device arrays

8.5 Per GPU Streams

In the previous single GPU version streams were created once globally:

```
for (int i = 0; i < NUM_STREAMS; i++) cudaStreamCreate(&streams[i]);
```

In the multiple GPU version streams are created per device after selecting the device:

```
CUDA_CHECK(cudaSetDevice(dev));
for (int s = 0; s < NUM_STREAMS; s++) {
    CUDA_CHECK(cudaStreamCreate(&context[dev].streams[s]));
}
```

We then reuse the same pipeline (H2D, Compute, D2H) but execute it independently on each GPU.

```
for (int local_sys = 0; local_sys < context[dev].num_systems; local_sys++) {
    int global_sys = context[dev].sys_start + local_sys;
    int stream_id  = global_sys % NUM_STREAMS;

    int host_off = global_sys * bodies_per_system;
    int dev_off  = local_sys  * bodies_per_system;

    cudaMemcpyAsync(context[dev].d.all_coords + dev_off,
                    h_data.all_coords + host_off,
                    bytes_coor_system, cudaMemcpyHostToDevice,
                    context[dev].streams[stream_id]);

    cudaMemcpyAsync(context[dev].d.all_velocities + dev_off,
                    h_data.all_velocities + host_off,
                    bytes_vel_system, cudaMemcpyHostToDevice,
                    context[dev].streams[stream_id]);
}
```

We keep the same stream mapping but we use global_sys to have a distribution independent of GPU partitioning.

```
for (int local_sys = 0; local_sys < context[dev].num_systems; local_sys++) {
    int global_sys = context[dev].sys_start + local_sys;
    int stream_id  = global_sys % NUM_STREAMS;
    int dev_off    = local_sys * bodies_per_system;

    for (int iter = 0; iter < nIters; iter++) {
        update_velocities<<<grid, block, 0, context[dev].streams[stream_id]>>>(
            context[dev].d.all_coords + dev_off,
            context[dev].d.all_velocities + dev_off,
            bodies_per_system, dt);

        update_positions<<<grid, block, 0, context[dev].streams[stream_id]>>>(
            context[dev].d.all_coords + dev_off,
            context[dev].d.all_velocities + dev_off,
            bodies_per_system, dt);
    }
}
```

Each system now uses a single stream on a different GPU and all the operations happen in order so we keep the same (H2D → kernels → D2H).

Phase 3: Enqueue all D2H (per GPU) As in the single GPU version, we only copy back positions (output validation compares coordinates):

```
for (int local_sys = 0; local_sys < context[dev].num_systems; local_sys++) {
    int global_sys = context[dev].sys_start + local_sys;
    int stream_id  = global_sys % NUM_STREAMS;

    int host_off = global_sys * bodies_per_system;
    int dev_off  = local_sys  * bodies_per_system;

    cudaMemcpyAsync(h_device_out.all_coords + host_off,
                    context[dev].d.all_coords + dev_off,
```

```
                    bytes_coor_system, cudaMemcpyDeviceToHost,
                    context[dev].streams[stream_id]);
}
```

8.6 Synchronization and Cleanup

In the single GPU version, a single `cudaDeviceSynchronize()` was used. With multiple GPUs we must synchronize each device:

```
for (int dev = 0; dev < dev_count; dev++) {
    cudaSetDevice(dev);
    cudaDeviceSynchronize();
}
```

The CHECK_CUDA_FAIL macro frees single GPU global buffers and resets the device. This is not correct for multiple GPUs (it may free the wrong device pointers or only reset one device). We replaced it with CUDA_CHECK macro that reports the error location and aborts:

```
#define CUDA_CHECK(call) do { ... } while(0)
```

Lastly cleanup is done at the end by iterating over devices, destroying the per device streams and freeing the per device allocations.

8.7 Performance

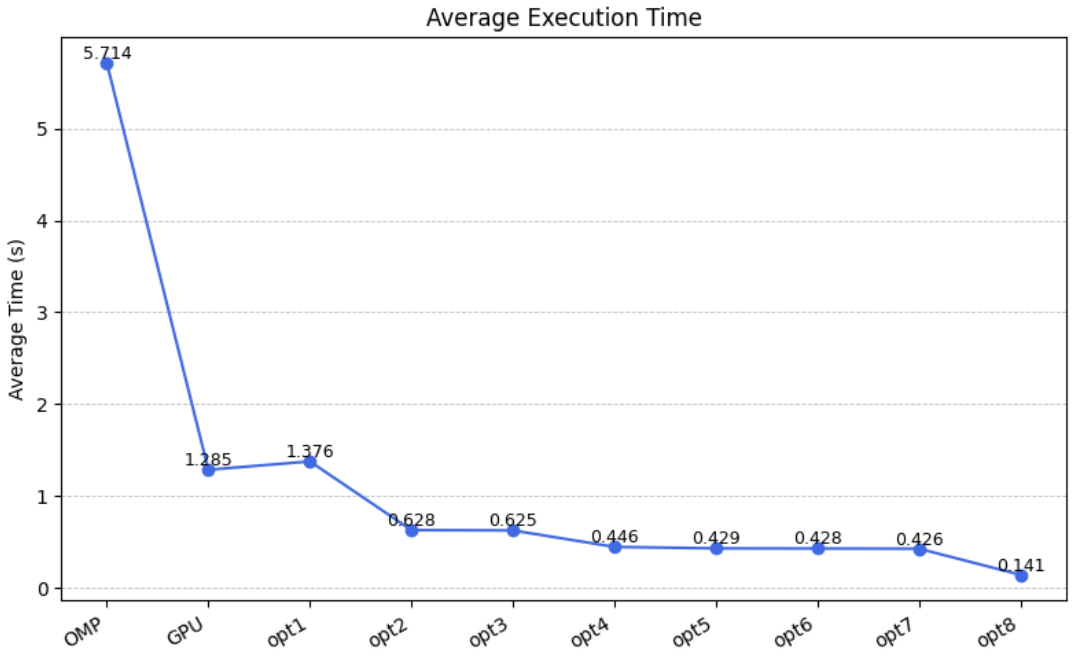

Figure 26: Execution time of opt8 compared to previous versions.

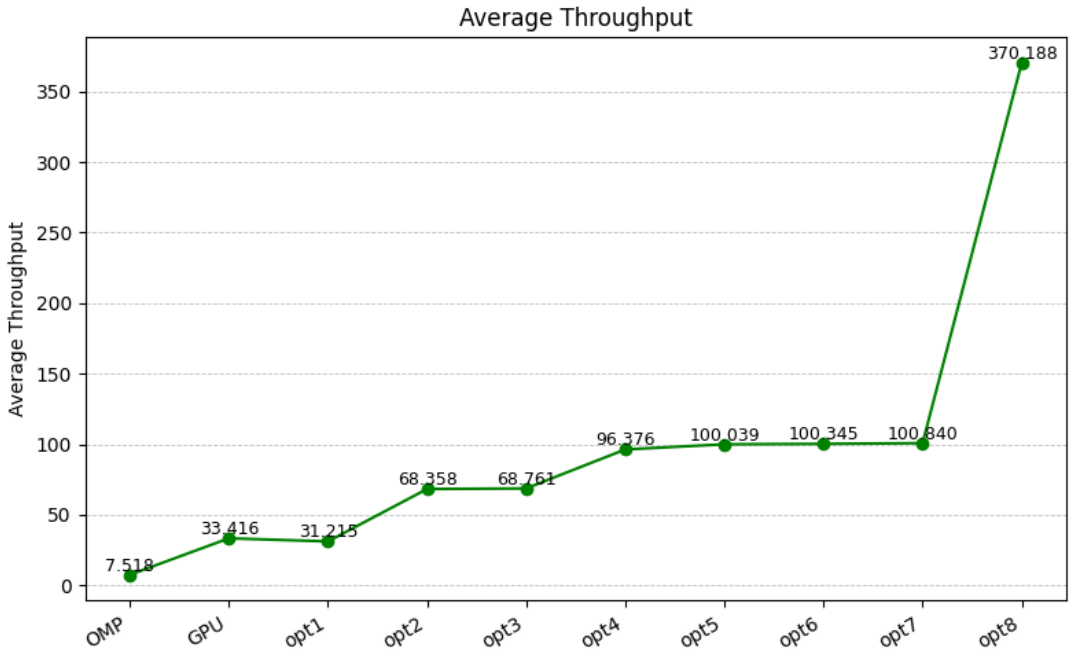

Figure 27: Throughput of opt8 compared to previous versions.

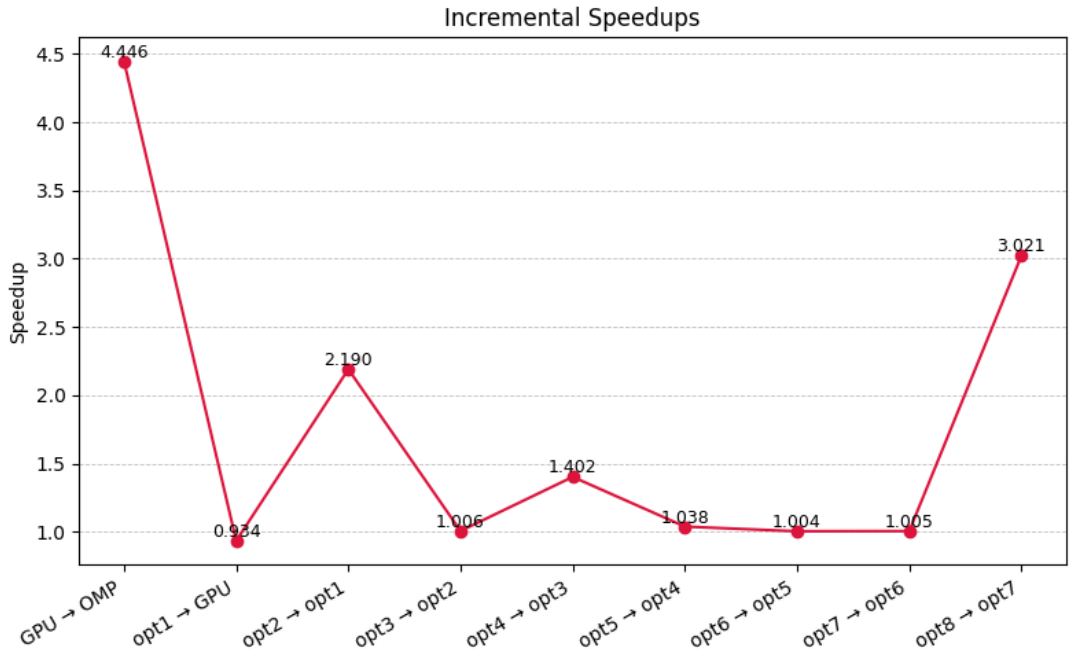

Figure 28: Speedup of opt8 compared to previous versions.

9 Optimization 9: Array of Structures → Structure of Arrays (SoA)

Previously each GPU stored:

`coords_T[local_bodies]`, `velocity_T[local_bodies]`.

Now each GPU stores six `float` arrays:

```
typedef struct {
    float *x, *y, *z;
    float *vx, *vy, *vz;
} Body_Arrays;
```

This improves memory coalescing: threads in a warp read contiguous `x[]` values (and similarly for `y,z,vx,vy,vz`) instead of strided struct fields.

9.1 AoS vs SoA example

Lets see an example of how the different AoS and SoA are on memory:

Table 5: Memory layout for Aos and SoA

Body index	**AoS:** `coords_T all_coords[i]`	**SoA: separate arrays**
0	`[x0 y0 z0]`	`x[0]=x0, y[0]=y0, z[0]=z0`
1	`[x1 y1 z1]`	`x[1]=x1, y[1]=y1, z[1]=z1`
2	`[x2 y2 z2]`	`x[2]=x2, y[2]=y2, z[2]=z2`
3	`[x3 y3 z3]`	`x[3]=x3, y[3]=y3, z[3]=z3`

Table 6: Address pattern when loading `x` for consecutive bodies.

Thread	**AoS load**	**SoA load**
$t = 0$	`&all_coords[0].x`	`&x[0]`
$t = 1$	`&all_coords[1].x`	`&x[1]`
$t = 2$	`&all_coords[2].x`	`&x[2]`
$t = 3$	`&all_coords[3].x`	`&x[3]`
Stride between threads	`sizeof(coords_T)` (12 bytes)	`sizeof(float)` (4 bytes)

In AoS each thread in the warp accesses x with a stride of sizeof(coords_T) which is is not coalescing. In SoA the warp accesses x[t] and has fully coalesced global loads.

9.2 Host Packing and Unpacking

Because the input dataset and CPU validation use array of structs(AoS) we pack into structure of arrays(SoA) on the host:

```
for (int i = 0; i < total_bodies; i++) {
        body_array.x[i]  = h_data.all_coords[i].x;
        body_array.y[i]  = h_data.all_coords[i].y;
        body_array.z[i]  = h_data.all_coords[i].z;
        body_array.vx[i] = h_data.all_velocities[i].vx;
        body_array.vy[i] = h_data.all_velocities[i].vy;
        body_array.vz[i] = h_data.all_velocities[i].vz;
    }
```

After the run we copy back `x,y,z` and unpack into `h_device_out.all_coords` for printing and validation:

```
for (int i = 0; i < total_bodies; i++) {
        h_device_out.all_coords[i].x = body_array.x[i];
        h_device_out.all_coords[i].y = body_array.y[i];
        h_device_out.all_coords[i].z = body_array.z[i];
    }
```

Note: this packing and unpacking adds additional overhead that did not exist in the previous optimization. In our timing measurements we only measure the GPU compute region.

9.3 Streams and Overlap

The stream pipeline (asynchronous H2D, kernel launches and D2H per system) is the same but the previous optimization assigned systems to streams using the global system index

`cudaMemcpyAsync + stream_id = global_sys % NUM_STREAMS.`

In this optimization stream assignment is done by using the local system index on each GPU:

`cudaMemcpyAsync + stream_id = local_sys % NUM_STREAMS.`

9.4 Kernel Changes: Force Kernel

The new velocity update kernel uses shared memory tiling with structure of arrays:

- `THREADS_PER_BLOCK=256`
- `TILE_SIZE=256`
- shared arrays: `shared_x/y/z[TILE_SIZE]`
- fast math: `rsqrtf` and `fmaf`

The cooperative tile load:

```
int tile_idx = tile * TILE_SIZE + threadIdx.x;
if (tile_idx < bodies_per_system) {
    shared_x[threadIdx.x] = x[pointer_index + tile_idx];
    shared_y[threadIdx.x] = y[pointer_index + tile_idx];
    shared_z[threadIdx.x] = z[pointer_index + tile_idx];
}
__syncthreads();
```

The inner loop reuses the shared tile:

```
float distSqr = fmaf(dx, dx, fmaf(dy, dy, fmaf(dz, dz, SOFTENING)));
float invDist = rsqrtf(distSqr);
float invDist3 = invDist * invDist * invDist;
fx = fmaf(dx, invDist3, fx);
```

9.5 Performance

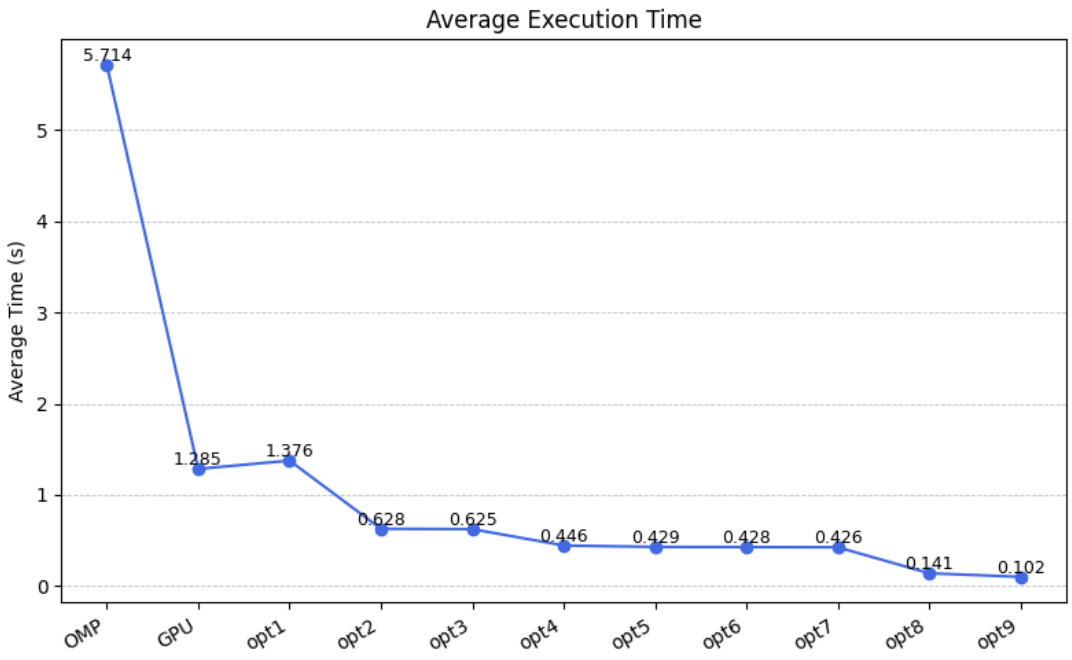

Figure 29: Execution time of opt9 compared to previous versions.

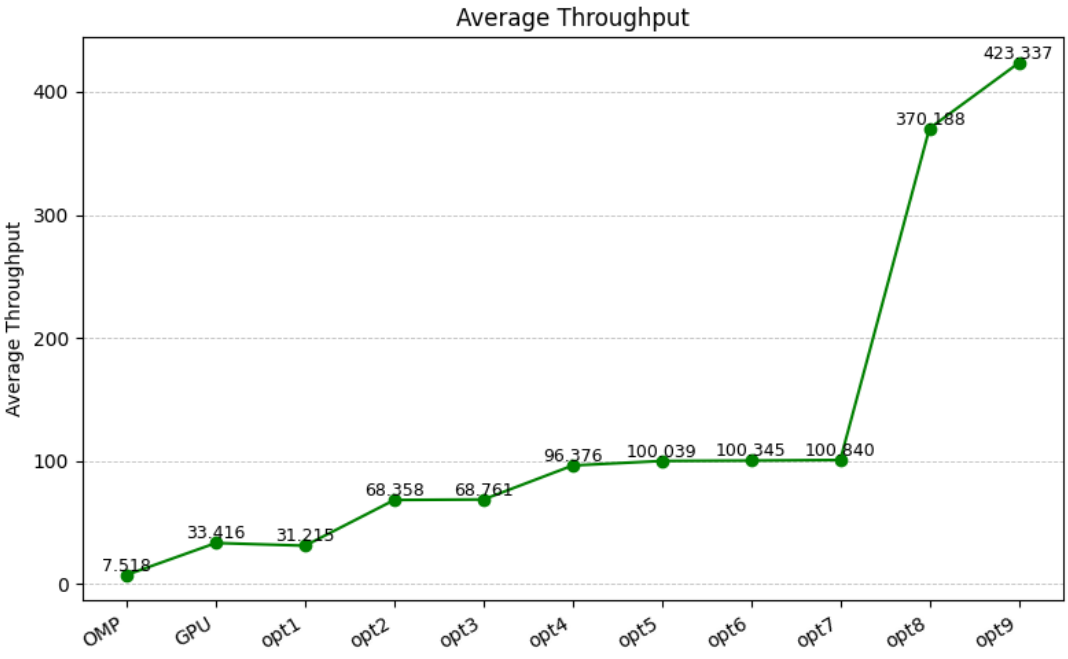

Figure 30: Throughput of opt9 compared to previous versions.

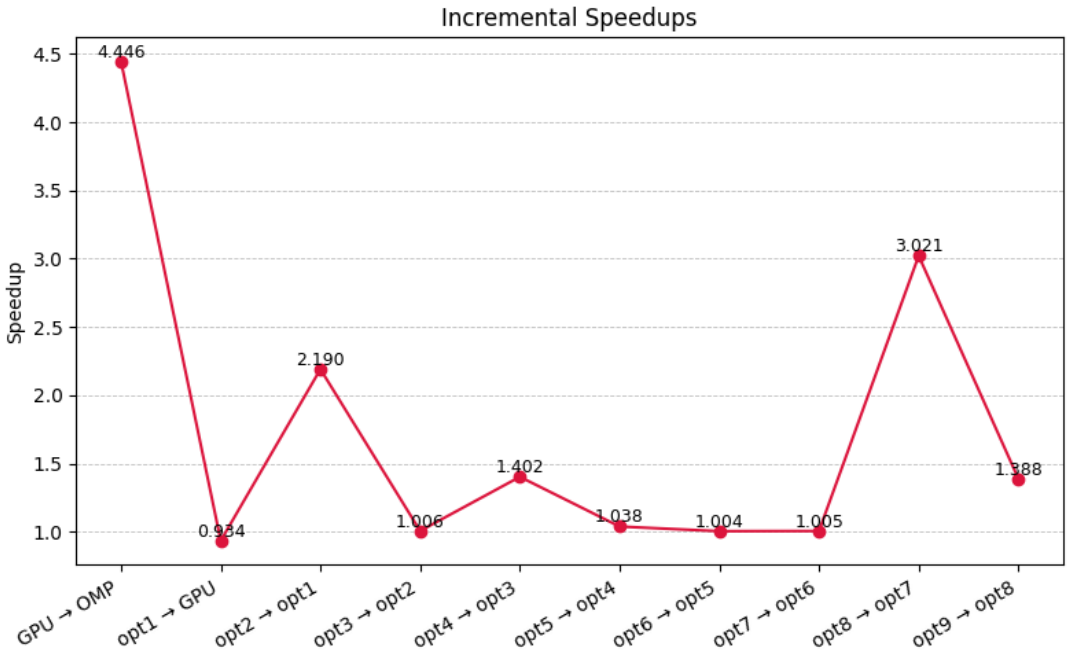

Figure 31: Speedup of opt9 compared to previous versions.

10 Final Results

Optimization	Avg. Time (s)	Speedup	Avg. Throughput
CPU OpenMP	5.7137	1.00	7.5175
GPU	1.2852	4.44	33.4165
Data Layout Redesign	1.3762	4.15	31.2147
Shared Memory with Tiling	0.6283	9.09	68.3581
Loop Unrolling	0.6248	9.15	68.7606
Streams	0.4457	12.82	96.3765
Read only Caching and Fast Math	0.4295	13.30	100.0388
Overlapping Transfers and Compute with CUDA Streams	0.4280	13.35	100.3450
Further Stream Optimization	0.4260	13.41	100.8395
Multiple GPUs	0.1410	40.53	370.1884
Array of Structures → Structure of Arrays (SoA)	0.1016	56.25	423.3374

Table 7: Performance comparison and speedup relative to OMP

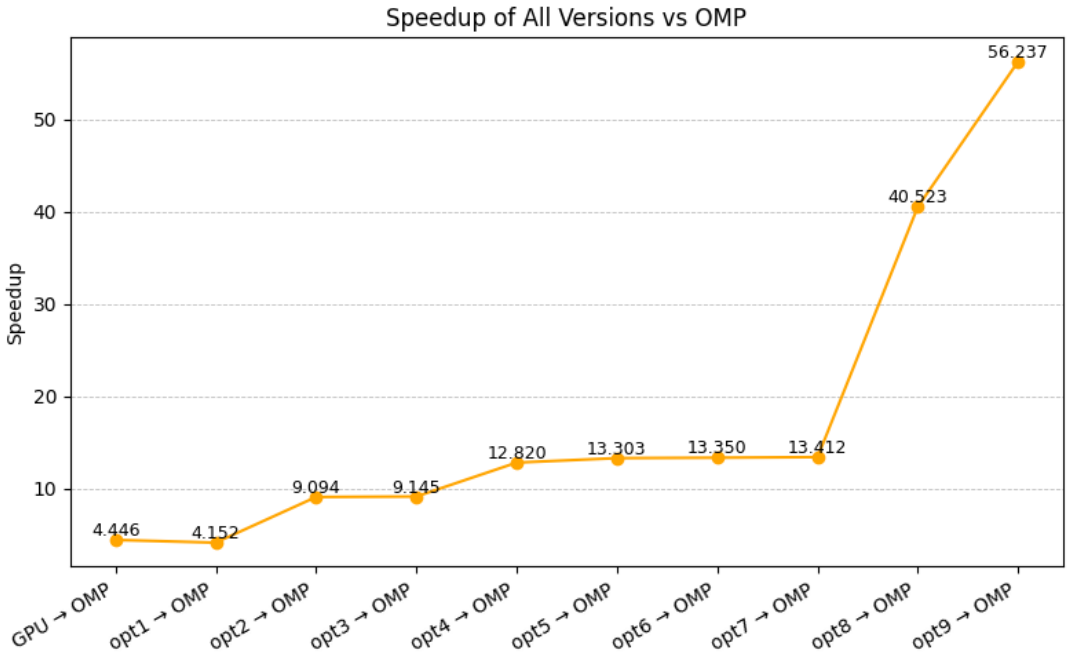

Figure 32: Speedups of all versions compared to OMP.