Move U'r Body (MUrB): Optimizing an N-Body Simulation

UM5IN160 - PACC Project - Sorbonne Université

Giacomo Pauletti, Alberto Taddei

January 26, 2026

Abstract

This report presents a set of CPU, GPU and heterogeneous optimizations applied to the MUrB n-body simulator. The objective is to increase throughput, measured primarily in frames per second (FPS), while keeping numerical results consistent with the provided golden model using Catch2 validation tests, and some stricter ones. Starting from the reference implementation, we progressively improved memory layout, arithmetic cost, SIMD utilization via MIPP, and thread-level parallelism with OpenMP. We also implemented a heterogeneous offload strategy controlled by an environment parameter that selects which portion of bodies is accelerated on the GPU. Experimental results on Dalek partitions show large speedups on CPU (up to $\sim 64\times$ over the naive baseline for a representative workload) and a dramatic gain when fully offloading to the GPU.

In addition, we implemented benchmarking scripts to analyze efficiency and complementary metrics from a green-computing perspective, and we explored physics-oriented studies and algorithmic variants to better characterize the dynamics of the problem. Finally, we implemented a cyberpunk-style visualization whose "vibe" is inspired by the track *Move Your Body* (Eiffel 65).

Contents

1 Introduction

The n-body problem is a classical compute-intensive workload where each body interacts gravitationally with all others. At each iteration, we compute accelerations and then update velocities and positions. The dominant cost is the acceleration computation, which is $\mathcal{O}(N^2)$ per iteration.

MUrB provides a working baseline (golden model) and a visualization mode; for all performance measurements, visualization is disabled (`--nv`).

Project goal. The project goal is to speed up the computation of accelerations (Eq. (1)) by exploiting:

- **Algorithmic optimizations** (reduce arithmetic cost and redundant work),
- **SIMD vectorization** (portable intrinsics via MIPP),
- **Multithreading** (OpenMP),
- **Multi-node distribution** (MPI prototype),
- **GPU tiled kernels** (shared-memory tiling),
- **GPU full-device execution** (minimize transfers / keep computation on device),
- **Heterogeneous CPU+GPU execution** (fractional offload controlled at runtime).

Governing equations. Given N bodies with masses m_j and positions $\mathbf{q}_j(t)$, the softened acceleration of body i is:

$$\mathbf{a}_i(t) \approx G \sum_{j=1}^{N} m_j \frac{\mathbf{r}_{ij}}{\left(\|\mathbf{r}_{ij}\|^2 + \varepsilon^2\right)^{3/2}}, \qquad \mathbf{r}_{ij} = \mathbf{q}_j(t) - \mathbf{q}_i(t). \tag{1}$$

Validation philosophy. Every new implementation is validated against the golden model with Catch2 unit tests, comparing SoA arrays (masses, radii, positions, velocities) and ensuring the relative error stays within configured tolerances. Implementations that are not validated against the basic Catch2 test sets (e.g., `[dmb]`) are not considered.

2 Description of the Optimizations

This section describes the main ideas behind each implementation. The baseline naive implementation is assumed given. We focus on the acceleration kernel, which dominates runtime.

2.1 CPU: sequential optimization (`cpu+optim`)

The naive version computes all pairs (i, j) and uses expensive operations (`pow`, repeated computation of ε^2, etc.). The `cpu+optim` implementation improves both arithmetic and memory behavior:

- **SoA layout for better locality and vectorization readiness.** Positions and masses are accessed as contiguous arrays (`qx[]`, `qy[]`, `qz[]`, `m[]`), reducing strided loads typical of AoS.
- **Reduced arithmetic cost.** Replace `pow` with explicit multiplies and compute `soft2 = soft*soft` once.

- **Exploit symmetry (Newton's third law).** Iterate only over $j > i$ and update both bodies in one interaction: this halves the pair evaluations and reduces redundant work. Conceptually:

$$\mathbf{a}_i \mathrel{+}= f(i,j)\,\mathbf{r}_{ij}, \qquad \mathbf{a}_j \mathrel{-}= f(j,i)\,\mathbf{r}_{ij}.$$

- **Temporary arrays.** Accumulation uses separate arrays `temp_ax/ay/az` to keep memory writes simple and contiguous.

A minimal illustrative snippet showing the symmetry idea:

Listing 1: `cpu+optim`: symmetric pair update (illustrative)

```
for (unsigned long i = 0; i < n; ++i) {
  for (unsigned long j = i + 1; j < n; ++j) {
    // dx,dy,dz, dist2, inv, inv3 ...
    const T fij = s * m[j]; // contribution to i from j
    const T fji = s * m[i]; // contribution to j from i
    ax[i] += fij * dx; ay[i] += fij * dy; az[i] += fij * dz;
    ax[j] -= fji * dx; ay[j] -= fji * dy; az[j] -= fji * dz;
  }
}
```

2.2 CPU: SIMD vectorization with MIPP (`cpu+simd`)

The `cpu+simd` version uses MIPP to exploit SIMD instructions in a portable way. Compared to `cpu+optim`, the goal is to compute multiple j-interactions at once while preserving the same numerical model.

- **Vectorizing the inner loop.** For a fixed i, the interactions with j bodies are computed in SIMD blocks of size $V =$ `mipp::N<T>()`.
- **FMA and reduced temporary storage.** Use `mipp::fmadd` to fuse multiply-add operations where possible.
- **Unrolling.** The SIMD inner loop is unrolled (e.g., $\times 4$) to increase ILP and reduce loop overhead.
- **Fast reciprocal square root.** For `float`, `mipp::rsqrt` is used, optionally refined with one Newton–Raphson step (compile-time macro `MURB_SIMD_RSQRT_NR`).
- **Tail handling.** When N is not a multiple of V, remaining iterations are computed in scalar mode.

Characteristic snippet (vector block, FMA):

Listing 2: `cpu+simd`: SIMD block accumulation (illustrative)

```
const int V = mipp::N<T>();
for (unsigned long j = 0; j < n_vec; j += V) {
  mipp::Reg<T> qjx = &qx[j], qjy = &qy[j], qjz = &qz[j], mj = &m[j];
  mipp::Reg<T> dx = qjx - qix, dy = qjy - qiy, dz = qjz - qiz;

  mipp::Reg<T> dist2 = mipp::fmadd(dx, dx,
                       mipp::fmadd(dy, dy,
                       mipp::fmadd(dz, dz, soft2)));
  mipp::Reg<T> inv = mipp::rsqrt(dist2);
  mipp::Reg<T> inv3 = (inv * inv) * inv;
  mipp::Reg<T> fac = G * mj * inv3;
  ax = mipp::fmadd(fac, dx, ax);
  ay = mipp::fmadd(fac, dy, ay);
  az = mipp::fmadd(fac, dz, az);
}
```

2.3 CPU: OpenMP multithreading (`cpu+omp`)

The `cpu+omp` version parallelizes the outer loop over i, combining SIMD inside each thread. The key idea is that each i writes only `acc[i]`, so the kernel is race-free with a simple data-parallel decomposition.

- **Parallelization strategy.** Each thread computes accelerations for a subset of bodies i.
- **Scheduling and affinity.** We used `schedule(static)` and explicit affinity settings (e.g., `OMP_PROC_BIND=close`, `OMP_PLACES=cores`) to reduce migration and improve reproducibility.
- **Tunable macros.** Compile-time or CMake-defined macros control: `MURB_OMP_UNROLL`, `MURB_OMP_PREFETCH_AHEAD`, `MURB_OMP_RSQRT_NR`.
- **Prefetching.** Software prefetch is used ahead of the current SIMD block for `qx/qy/qz/m`.

Characteristic snippet (outer-loop OpenMP):

Listing 3: `cpu+omp`: parallel outer loop (illustrative)

```
#pragma omp parallel for schedule(static)
for (unsigned long i = 0; i < n; ++i) {
  // compute a_i by looping over all j (SIMD inside)
  acc[i].ax = ax_i;
  acc[i].ay = ay_i;
  acc[i].az = az_i;
}
```

2.4 Multi-node MPI prototype (`mpi`, not benchmarked in results)

A multi-node implementation (`SimulationNBodyMultiNode`) was developed to explore distributed memory scaling. The approach partitions bodies across MPI ranks using `counts/displs`. At each iteration:

- **Global state exchange.** Each rank gathers the global positions and masses using `MPI_Allgatherv`.
- **Local compute range.** Each rank computes accelerations for its local $[i_0, i_1)$ range (optionally with OpenMP).
- **Consistency for integration.** Accelerations are gathered (or made available) so that the integrator updates states consistently.

Characteristic snippet (global gather + local range):

Listing 4: `mpi`: Allgatherv + local compute range (illustrative)

```
const int i0 = displs[rank];
const int i1 = i0 + counts[rank];

// Gather global state (positions, masses) on all ranks
MPI_Allgatherv(qx + i0, counts[rank], MPI_FLOAT,
              qx_all, counts, displs, MPI_FLOAT, MPI_COMM_WORLD);
// ... same for qy_all, qz_all, m_all

for (int i = i0; i < i1; ++i) {
  // compute a_i using gathered global arrays
}
```

This prototype is correct and useful for understanding communication patterns, but it is not included in the performance results section of this report. When run on a single node with multiple ranks, it measures inter-process parallelism plus communication overhead, not true multi-node scaling.

2.5 GPU: CUDA

(`gpu+tile+full`) The approach for the GPU implementation, based on CUDA C++, is the best performing implementation we have. Unfortunately, we hadn't time to implement a sophisticated technique such as Barners-Hut algorithm, so our approach remains the direct one with complexity $\mathcal{O}(N^2)$.

2.5.1 Implementation details

The development has been performed in the following steps:

- **Sequential code optimization**. We first developed `cpu+optim`, reducing as much the computations done in the inner loop, mainly by avoiding function calls and useless computations. This code, then, has been transposed to GPU by assigning to each thread one body. This bijective relation thread-body has revealed to be as good as another approach, used when the number of bodies is high, in which one thread calculates the accelerations for multiple bodies.
- **Reversed square root**. We used the CUDA function `rsqrt` which performs very fast $\frac{1}{\sqrt{x}}$. We used it to compute $1/\sqrt{\text{rijSquared} + \text{softSquared}}$. This gave us a very important gain in the FPS rate.
- **Tiling** The next major step is tiling. Each thread of a block loads in shared memory one element, with a coalescent access to global memory. The best block dimension and tiling size was found to be of 1024.
- **CUDABodies** Finally, we implemented a child class of Bodies, CUDABodies, which has dedicated space in GPU global memory. This is a good improvement for the $200k$ bodies benchmark, but with larger number of bodies the quadratic complexity of the kernel dominates the memory transfer.
- **Compile flags** Last but not least, a great improvement has been obtained by using the cmake flag `-DCMAKE_BUILD_TYPE=Release`, which enables strong compiler optimizations.
- **Loop unrolling** We noticed that the register pressure was not that high, so we decided to unroll the inner loop of a factor of 32 with `#pragma unroll 32`. This enable a better latency covering hence better performances

We implemented 2 variations of `gpu+tile+full`, namely `gpu+tile+full` and `gpu+tile+full200k`. The first one uses the mapping thread-body one-to-many. So it tries to give more load to the single kernel and not to spawn too many blocks. The idea beneath this is that when the number of bodies is massive, the number of blocks is huge and the overhead of allocating and deallocating resources for a block is visible. With this version we aim to have all the 128 SM of the GPU always working.

The other version, on the other hand, is aimed to best perform to a medium-sized amount of bodies. It appears to be slightly better with the mapping thread-body one-to-one.

2.5.2 Discussion on optimality of the kernel

We believe we implemented the best kernel possible withouth changing algorithm, i.e. without moving to Barners-Hut or Multipole methods.

Other approaches possible were to consider the matrix (or, more formally, tensor) of accelerations where $a_{i,j}$ is the acceleration $\vec{a}_{ij}$ that j applies to j, to build it (at least theoretically) and then to reduce it to find the total acceleration that each bodies perieves. Reduction is a procedure which is has extensively studied on GPU and can be implemented very efficiently. However we do not believe that this approach is the best one. We preferred our approch, in which each body is assigned to one and only one thread and hence all the threads perform independent sums; it is an embarassingly parallel problem.

Finally, the parameter involved in our kernel (ex: tile size) and the operations done in the inner loop has been extensively fine tuned. For example, different ordering of instruction and different functions and intrinsics, such as the not-so-known `rnorm3d`. Another example is that we premultiplied the masses by the gravitational constant G and we stored them in the GPU memory, specifically in the `devGM` array. This reduces by 1 flop the amount of work done in the inner loop.

2.6 Heterogeneous CPU+GPU execution (`hetero`)

The heterogeneous implementation (`SimulationNBodyHetero`) offloads a fraction of bodies to the GPU while the remainder is computed on CPU (OpenMP). A key design choice is to keep a single global SoA state on host and use explicit CUDA memory transfers.

- **Fractional offload parameter.** The number of GPU-accelerated bodies is `cut` $= \lfloor f \cdot N \rfloor$, controlled at runtime by `MURB_HETERO_GPU_FRACTION` (clamped to $[0, 1]$).
- **Activation threshold.** GPU is used only if `N` $\geq$ `MURB_HETERO_MIN_N` and a CUDA device is available.
- **GPU kernel tiling in shared memory.** The CUDA kernel loads a tile of bodies into shared memory (`MURB_HETERO_BLOCK`, default 256) and iterates over tiles to compute the full interaction sum for each i in $[0, \texttt{cut})$.
- **Asynchronous transfers and stream.** Inputs are copied H2D with `cudaMemcpyAsync` into a non-blocking stream. The resulting accelerations are copied back D2H and merged into the global acceleration array.
- **CPU remainder.** Bodies $i \in [\texttt{cut}, N)$ are computed on CPU with OpenMP in parallel.

Characteristic snippet (CPU/GPU split):

Listing 5: `hetero`: fractional offload split (illustrative)

```
const unsigned long cut = (unsigned long)std::floor(f * (double)N);

// GPU part: i in [0, cut)
cudaMemcpyAsync(d_qx, qx, N*sizeof(T), cudaMemcpyHostToDevice, stream);
kernel<<<grid, block, 0, stream>>>(..., N, 0, cut, soft2);
cudaMemcpyAsync(ax_gpu, d_ax, cut*sizeof(T), cudaMemcpyDeviceToHost, stream);

// CPU remainder: i in [cut, N)
#pragma omp parallel for schedule(static)
for (unsigned long i = cut; i < N; ++i) {
  // CPU compute a_i
}
cudaStreamSynchronize(stream);
// merge GPU accelerations into global array
```

Important note on performance interpretation. When $f = 1.0$, the acceleration computation is fully offloaded to GPU, and the CPU part becomes negligible (except position/velocity update). For intermediate f, PCIe transfers and CPU/GPU overlap determine the optimal point.

Available GPU tags. The following `--im` tags are available for GPU-oriented implementations:

- `gpu+tile`
- `gpu+tile+full`
- `gpu+tile+full200k`
- `gpu+tracking`
- `gpu+leapfrog`

3 Computer Architecture

All experiments were executed on the Dalek cluster. Performance results depend on core types, SMT, memory hierarchy, and GPU availability. We report the partitions used and the most relevant characteristics.

3.1 CPU testbed: `iml-ia770` (Intel Core Ultra 9 185H)

The `iml-ia770` partition is based on Intel's Meteor Lake generation (2023) featuring a heterogeneous CPU:

- **Performance cores (P-cores):** 6 Redwood Cove P-cores with 2-way SMT.
- **Efficiency cores (E-cores):** 8 Crestmont E-cores with 1-way SMT.
- **Low-power E-cores (LP-E):** 2 Crestmont LP-E cores with 1-way SMT.

The following `lscpu -e` excerpt illustrates the heterogeneous core/cache topology and frequencies:

Listing 6: Excerpt of `srun -p iml-ia770 lscpu -e`

```
CPU NODE SOCKET CORE L1d:L1i:L2:L3 ONLINE    MAXMHZ   MINMHZ       MHZ
  0    0      0    0 16:16:4:0        yes 4800.0000 400.0000 2278.3521
  1    0      0    1 8:8:2:0          yes 5100.0000 400.0000  400.0000
  2    0      0    1 8:8:2:0          yes 5100.0000 400.0000  400.0000
  ...
 20    0      0   14 64:64:8          yes 2500.0000 400.0000 1099.2760
 21    0      0   15 66:66:8          yes 2500.0000 400.0000 1099.7050
```

Mixed core types imply different frequencies and per-core performance. OpenMP scheduling and binding may influence whether work lands primarily on P-cores or spreads to E/LP-E cores. This is relevant for the following green computing study, where different core classes may lead to different energy/performance trade-offs.

3.2 CPU+GPU testbed: `az4-n4090` (AMD Zen 4 CPU, Nvidia RTX 4090)

The `az4` GPU-oriented partitions are based on AMD Zen 4 (2022). In particular, `az4-n4090` provides:

- **CPU:** AMD Ryzen 9 7945HX, 16 $\times$ Zen 4 cores with 2-way SMT.
- **GPU:** Nvidia GeForce RTX 4090.

3.3 Compilation and run methodology

- Build type: `Release` (aggressive compiler optimization).
- Precision: `fp32` (`NBODY_FLOAT` definition).
- Visualization disabled: `--nv`.
- FLOPs metric enabled: `--gf` (reported as GFLOP/s by MUrB).

Compilation. We use CMake to configure a `Release` build. For the CPU-focused experiments we enable fast-math (project option `ENABLE_FAST_MATH`) to allow faster floating-point transformations where acceptable for this project (numerical consistency is still enforced by validation). For GPU/heterogeneous tests we also enable CUDA.

Listing 7: Typical build (Release + CUDA + fast-math)

```
module purge
module load easytools
module load openmpi/5.0.8

rm -rf build
cmake -S . -B build \
  -DCMAKE_BUILD_TYPE=Release \
  -DENABLE_MURB_CUDA=ON \
  -DENABLE_FAST_MATH=ON
cmake --build build -j4

# Run validation tests (example)
srun -p az4-n4090 ./build/bin/murb-test
```

Run methodology (CPU, OpenMP, MPI). We bind CPU execution to cores. For OpenMP we set affinity and scheduling explicitly for reproducibility:

Listing 8: OpenMP environment (typical)

```
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
export OMP_DYNAMIC=FALSE
export OMP_PLACES=cores
export OMP_PROC_BIND=close
export OMP_SCHEDULE=static
export OMP_WAIT_POLICY=ACTIVE
```

Commands use cases

Listing 9: CPU single-thread runs on `iml-ia770`

```
# naive
srun -p iml-ia770 -n 1 --cpus-per-task=1 --cpu-bind=cores \
  ./build/bin/murb -n 30000 -i 200 --nv --im cpu+naive --gf

# optim
srun -p iml-ia770 -n 1 --cpus-per-task=1 --cpu-bind=cores \
  ./build/bin/murb -n 30000 -i 200 --nv --im cpu+optim --gf

# simd
srun -p iml-ia770 -n 1 --cpus-per-task=1 --cpu-bind=cores \
  ./build/bin/murb -n 30000 -i 200 --nv --im cpu+simd --gf
```

Listing 10: OpenMP run (best configuration example: 12 threads) on `iml-ia770`

```
# OpenMP runtime configuration (in the submission shell)
export OMP_NUM_THREADS=12
export OMP_DYNAMIC=FALSE
export OMP_PLACES=cores
export OMP_PROC_BIND=close
export OMP_SCHEDULE=static
export OMP_WAIT_POLICY=ACTIVE

# Run on Dalek (iml-ia770)
srun -p iml-ia770 -n 1 --cpus-per-task=12 --threads-per-core=1 --cpu-bind=cores --export=ALL \
  ./build/bin/murb -n 30000 -i 200 --nv --im cpu+omp --gf
```

Listing 11: GPU run example on `az4-n4090`

```
srun -p az4-n4090 -n 1 --cpus-per-task=1 --cpu-bind=cores \
  ./build/bin/murb -n 200000 -i 200 --nv --im gpu+tile+full --gf
```

Listing 12: MPI prototype run (4 ranks) on `iml-ia770` (single node)

```
srun -p iml-ia770 -n 4 --cpus-per-task=1 --threads-per-core=1 --cpu-bind=cores \
  ./build/bin/murb -n 30000 -i 200 --nv --im mpi --gf
```

4 Experimentation Results

This section reports Frames Per Second (FPS) and GFLOP/s for representative experiments. The focus is on:

- improvement across CPU implementations,
- scaling with threads,
- scaling with problem size N,
- heterogeneous CPU+GPU fraction sweep.

Remark on reproducibility. All results reported here correspond to one run per configuration unless stated otherwise. Some variability is expected on shared systems. When an outlier was observed, we repeated runs and report the median.

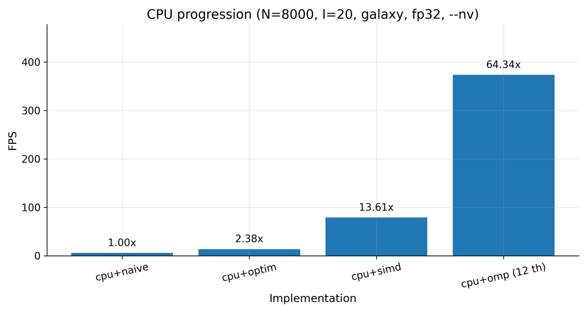
4.1 CPU results on `iml-ia770`

4.1.1 Progression of CPU optimizations (fixed workload)

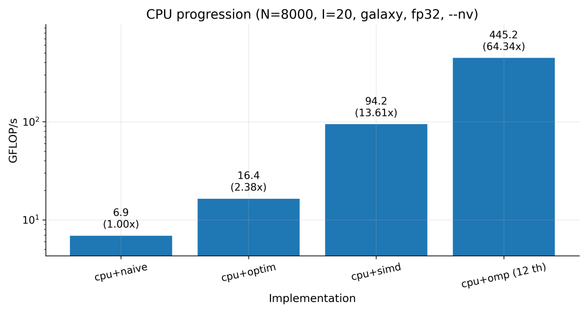
Run: $N = 8000$, $I = 20$, scheme=`galaxy`, `fp32`, `--nv` on `iml-ia770`.

Table 1: CPU optimization progression (same input).

Implementation	Time (ms)	FPS	GFLOP/s	Speedup vs naive
cpu+naive	3445.84	5.804	6.9	1.00x
cpu+optim	1450.26	13.791	16.4	2.38x
cpu+simd	253.171	78.998	94.2	13.61x
cpu+omp (12 th)	53.548	373.497	445.2	64.34x



(a) FPS by implementation.



(b) GFLOP/s by implementation.

Figure 1: CPU optimization progression ($N = 8000$, $I = 20$).

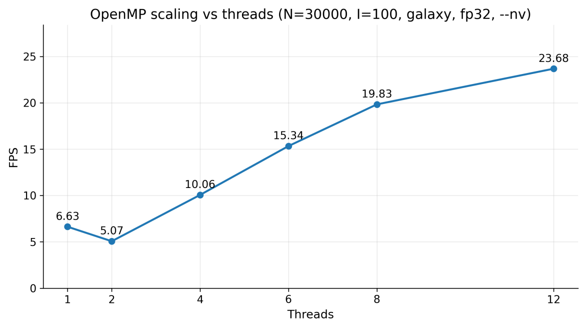
Discussion. `cpu+optim` provides a clear gain by reducing arithmetic and exploiting pair symmetry. `cpu+simd` adds a large boost via SIMD parallelism, and `cpu+omp` adds thread-level scaling on top, reaching $\sim 64\times$ speedup over the naive baseline for this workload.

4.1.2 OpenMP scaling vs number of threads

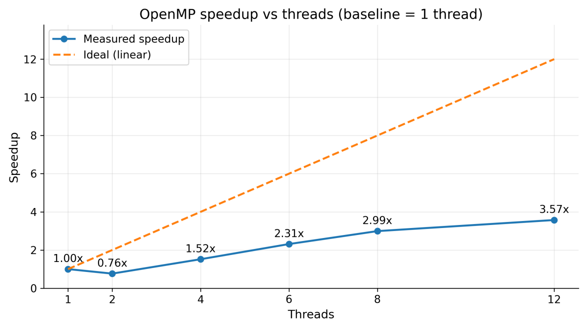
Run: $N = 30000$, $I = 100$, scheme=`galaxy` on `iml-ia770`.

Table 2: OpenMP scaling vs threads (speedup baseline = 1 thread).

Threads	Time (ms)	FPS	GFLOP/s	Speedup
1	15081.2	6.631	111.2	1.00x
2	19738.4	5.066	84.9	0.76x
4	9935.38	10.065	168.7	1.52x
6	6520.51	15.336	257.1	2.31x
8	5042.09	19.833	332.5	2.99x
12	4222.8	23.681	397.0	3.57x



(a) FPS vs threads.



(b) Speedup vs threads (baseline = 1 thread).

Figure 2: OpenMP scaling on `iml-ia770` for $N = 30000$, $I = 100$.

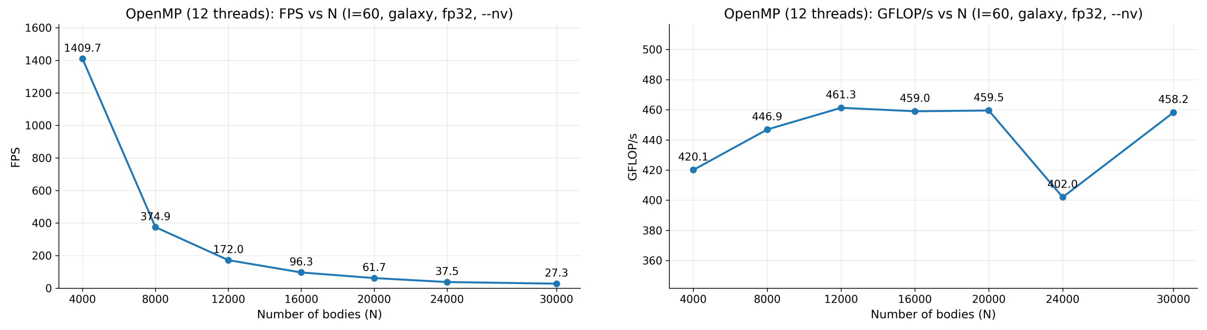
Discussion and the 2-thread anomaly. The 2-thread configuration is slower than 1 thread. This can happen due to system noise. A standard mitigation is to repeat each measurement several times and report the median, and to compare affinity policies (e.g., `OMP_PROC_BIND=close` vs `spread`).

4.1.3 Scaling with number of bodies (N-scaling)

Run: `cpu+omp`, threads=12, $I = 60$, scheme=`galaxy` on `iml-ia770`.

Table 3: N-scaling with fixed thread count (12 threads).

N	Time (ms)	FPS	GFLOP/s
4000	42.561	1409.74	420.1
8000	160.053	374.876	446.9
12000	348.868	171.985	461.3
16000	623.311	96.260	459.0
20000	972.780	61.679	459.5
24000	1601.44	37.466	402.0
30000	2195.25	27.332	458.2



(a) FPS vs N. (b) GFLOP/s vs N.

Figure 3: N-scaling for `cpu+omp` (12 threads, $I = 60$).

Discussion. As expected for an $\mathcal{O}(N^2)$ kernel, FPS decreases as N grows, but GFLOP/s stays roughly stable around ~ 450 GFLOP/s for most sizes. This indicates the implementation is compute-bound in that region. The dip at $N = 24000$ may be caused by cache effects or frequency variability.

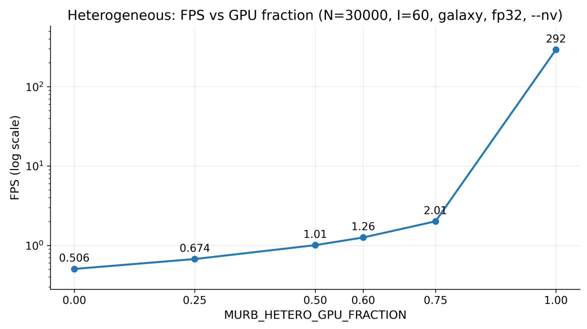
4.2 Heterogeneous CPU+GPU results on `az4-n4090`

4.2.1 Fraction sweep (`MURB_HETERO_GPU_FRACTION`)

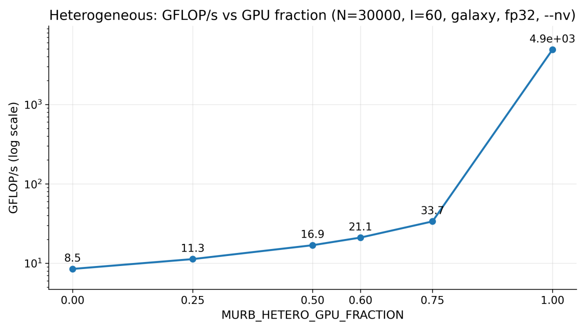
Run: $N = 30000$, $I = 60$, scheme=`galaxy`, `hetero`, 12 CPU threads on `az4-n4090`.

Table 4: Heterogeneous fraction sweep on `az4-n4090`.

GPU fraction	Time (ms)	FPS	GFLOP/s
0.00	118566	0.506	8.5
0.25	89066.1	0.674	11.3
0.50	59498.6	1.008	16.9
0.60	47610.2	1.260	21.1
0.75	29810.6	2.013	33.7
1.00	205.376	292.147	4897.5



(a) FPS vs fraction (log scale).



(b) GFLOP/s vs fraction (log scale).

Figure 4: Heterogeneous offload sweep on `az4-n4090` ($N = 30000$, $I = 60$).

Discussion. Partial offload yields modest improvements up to $f = 0.75$, suggesting that PCIe transfers and CPU-side work dominate until most of the workload is moved to GPU. At $f = 1.0$ the acceleration computation becomes fully GPU-driven and the throughput jumps dramatically, reaching 292 FPS and $\sim$ 4.9 TFLOP/s. This result strongly supports the idea that a *fully offloaded* acceleration kernel is preferable to mixed CPU+GPU execution for this configuration.

4.3 GPU implementations

In Table 5 are reported some performance experiments. Usually `gpu+tile+full` is better performant on 500000 bodies but the table tell otherwise.

Table 5: GPU implementation performances (number iteration: 200)

N	implementation	Time (ms)	FPS
200000	`gpu+tile+full200k`	4941.3	40.5
200000	`gpu+tile+full`	4985.3	40.1
500000	`gpu+tile+full200k`	26787.4	7.5
500000	`gpu+tile+full`	26772.1	7.8

4.4 Green computing analysis

In modern High-Performance Computing, energy efficiency is becoming as critical as raw performance. To complement throughput-oriented benchmarks (FPS / GFLOP/s), we performed a green computing study on the `iml-ia770` partition, which is particularly relevant thanks to its hybrid CPU architecture (Intel Core Ultra 9 185H, Meteor Lake).

4.4.1 Measurement methodology

To measure energy consumption and efficiency on specific core subsets, we developed a small benchmarking suite based on two Python scripts:

1. `measure_energy.py`. This script automates Slurm job submission and uses the Dalek's energy monitoring tool `node-conso` to sample power sensors (e.g., CPU, RAM, motherboard; and GPU if present) over a fixed measurement window (5 seconds). Crucially, it pins the `murb` process to selected cores via `taskset` (e.g., cores 0–11 for P-cores, 12–19 for E-cores), preventing thread migration during the measurement.
2. `parse_energy_log.py`. This script parses raw logs produced by `node-conso`, integrates power over time to obtain total energy (Joules), and combines it with the throughput (FPS) printed by MUrB to compute efficiency metrics such as **FPS/Watt** and **Joules per frame**.

Metrics. Let $P(t)$ be the measured total system power (W). We compute:

$$E = \int_{t_0}^{t_1} P(t)\,dt \quad \text{(Joules)}, \qquad \overline{P} = \frac{E}{t_1 - t_0} \quad \text{(Watts)}.$$

Given the measured throughput FPS, we report:

$$\text{Efficiency} = \frac{\text{FPS}}{\overline{P}} \text{ (FPS/W)}, \qquad \text{Energy cost} = \frac{\overline{P}}{\text{FPS}} \text{ (J/frame)}.$$

4.4.2 Experimental results

We evaluated `cpu+omp` on the three core classes. The results are summarized in Table 6.

Core type	**Throughput** (FPS)	**Mean power** (Total system W)	**Efficiency** (**FPS/W**)	**Energy cost** (J/frame)
P-cores (Perf.)	**26.89**	46.15	**0.583**	**1.72**
E-cores (Eff.)	10.53	44.84	0.235	4.26
LP-E cores	2.01	52.75[1]	0.038	26.21

Table 6: Energy efficiency comparison on `iml-ia770` (`cpu+omp`, $N = 30000$).

4.4.3 Discussion

A key finding is that, for this compute-bound FP32 workload, the "Performance" cores are also the most energy-efficient.

- **Race-to-idle effect.** P-cores consume slightly more instantaneous power than E-cores, but complete the work $\approx 2.5\times$ faster, reducing the total energy per frame (1.72 J vs 4.26 J).
- **Floating-point throughput gap.** The n-body kernel is dominated by dense floating-point arithmetic ($\mathcal{O}(N^2)$ interactions, heavy use of vector math). E-cores are optimized for efficient throughput on general tasks, but they typically provide lower FP32/vector performance than P-cores at comparable power.
- **Practical takeaway.** For heavy scientific kernels such as n-body, "green computing" often means minimizing time-to-solution with the fastest compute resources, rather than running on low-power cores for longer.

Limitations. This analysis uses total system power (not only CPU package power) and a fixed measurement window. While this is appropriate for comparing configurations on the same node, it may over-emphasize idle/background components when the workload is slow (notably for LP-E cores).

4.5 Artist Award: "Cyberpunk" N-Body animation

For the Artist Award, we reinterpreted the project's theme song, *"Move Your Body"* (Eiffel 65), not merely as background music, but as a driving concept for the visualization itself. The goal was to transform a standard scientific simulation into an immersive rhythm-inspired cyberpunk "music video" rendering.

[1]LP-E measurements are more sensitive to background/system noise because the execution time is longer, so non-CPU components can dominate the average.

4.5.1 From static to dynamic: visual transformation

The original MUrB visualization is functional but intentionally minimal. We introduced three main changes:

- **Background atmosphere.** The original environment uses a transparent black clear color, producing a void-like look. In `OGLSpheresVisu.cpp`, we modified `glClearColor` to a deep cyber-space blue (e.g., `(0.02, 0.02, 0.05)`) to add depth and contrast.
- **Replacing static color tables.** The baseline implementation in `OGLSpheresVisuGS.cpp` relies on static lookup tables (`MAPPING_R`, `MAPPING_G`, `MAPPING_B`) and a hardcoded velocity range, resulting in a fixed palette largely independent of the simulation dynamics. We replaced this logic with a *procedural* color mapping computed each frame from particle speed and a rhythm signal.
- **Dynamic contrast normalization.** To keep visuals consistent across simulation scales, we compute the minimum and maximum velocity magnitude per frame and normalize each particle speed to $t \in [0, 1]$. This ensures stable contrast even when global velocities vary over time.

These changes preserve the scientific meaning of the simulation while providing a distinctive cyberpunk visual identity and a rhythm-inspired dynamic rendering, aligned with the project theme "Move U'r Body".

4.5.2 Audio-visual synchronization (the math)

To create rhythm-synchronized effects, we implemented a lightweight beat model directly in the C++ render loop. The song tempo is approximately 130 BPM, which corresponds to:

$$f = \frac{130}{60} \approx 2.166 \text{ Hz}.$$

Using `glfwGetTime()`, we compute a continuous beat phase:

$$\text{phase} = \text{time} \cdot f \cdot 2\pi.$$

A simple sine wave sin(phase) is visually too smooth to resemble a percussive "kick". We therefore sharpen the waveform by compressing the peaks with a power transform:

$$\text{beatPulse} = \left(\frac{\sin(\text{phase}) + 1}{2} \right)^8 .$$

Raising the normalized sine to a high power creates short, intense peaks followed by longer off-beat intervals, mimicking a strobe/kick behavior.

4.5.3 Cyberpunk palette and beat-driven highlights

In `OGLSpheresVisuGS.cpp`, we implemented a dynamic gradient and beat-dependent "flash" injection:

- **Base gradient (speed-dependent).** Slow particles ($t \approx 0$) are mapped to dark blue (close to the background), while faster particles interpolate towards electric cyan.
- **Strobe injection (beatPulse).** When beatPulse is high (on the beat), we inject a white-light contribution for sufficiently fast particles (e.g., $t > 0.3$), so that the galaxy core "explodes" in brightness in sync with the rhythm.

Some color components may exceed 1.0 to exaggerate the neon effect depending on the pipeline; if needed, values can be clamped to $[0, 1]$ for strict LDR output.

5 Validating the N-body simulation

Disclaimer: what is discussed in this section hasn't been implemented due to time constraints, nonetheless we believe that the investigation we did is still of relevance. Attempts of implementation have been done and, when so, it will be stated.

5.1 Sources of error and how to improve the solution

We found it interesting to try to give physical arguments on the validity of our simulation. To do so we consider the three following sources of errors:

- **Error in the initial conditions**, which are usually derived from approximate models or observations
- **Floating point error** given by the finitness of the precision
- **Time discretization error**, due to the time discretization performed when computing the integral of the motion equations.

In general, since these errors cannot be avoided completely, the error will be exponentially inflated due to the chaoticity of the system. So, in general, the solution will be, sooner or later wrong. However, how soon it will be wrong and "how wrong" it will be, can be somewhat controlled. This is what we briefly investigate in this section.

We are not interested in the first source of error, it is not something we, authors of this report, can do much about.

Concerning the other two sources, the error they produce can be reduced.

5.1.1 Handling the floating point error

For the floating point error, the idea is naive and consists in using extended-precision, for example 128bit or more. Furthermore, particular functions such as `rsqrt` should be avoided, since they lose few digits of precision. For example, in the Nvidia programming guide[2], it is written that `rsqrt` loses 2 Units in Last Place (ULP) of precision, that is to say the last two digits of the mantissa are not valid.

In the discipline of floating-point error analysis, the summation is a well-studied problem and its error propagation has been extensively studied. The final error can be due to 2 main agents:

- **cancellation error**, which happens when two very close floating point numbers $x \approx y$ are subtracted, filling then the mantissa of "garbage". In particular, it happens when the *condition number* of $\sum p_i$ is high. Since we haven't investigated on this, we don't know whether this is the case or not, so we neglected this error in our analysis.
- **accumulation error**, which is simply the accumulation of many errors

We focus on the accumulation error. In particular, it is well known that the *recursive summation* algorithm, that is to say the naive summation which accumulates all the data on a unique variable, has bad accuracy in this regard. Better algorithms are *Ogita, Rump, Oishi compensated summation* or the *FABSum algorithm* [1]. Due to time reason, we were not able to experiment in this direction but we might do so in the future.

Last point about floating point error is that an attempt to study the error with the CADNA library [3], unsuccessful due to compilation problems. This would have allowed us to estimate at which iteration the data would have become numerical noise (error).

[2]https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/mathematical-functions.html#non-standard-cuda-mathematical-functions

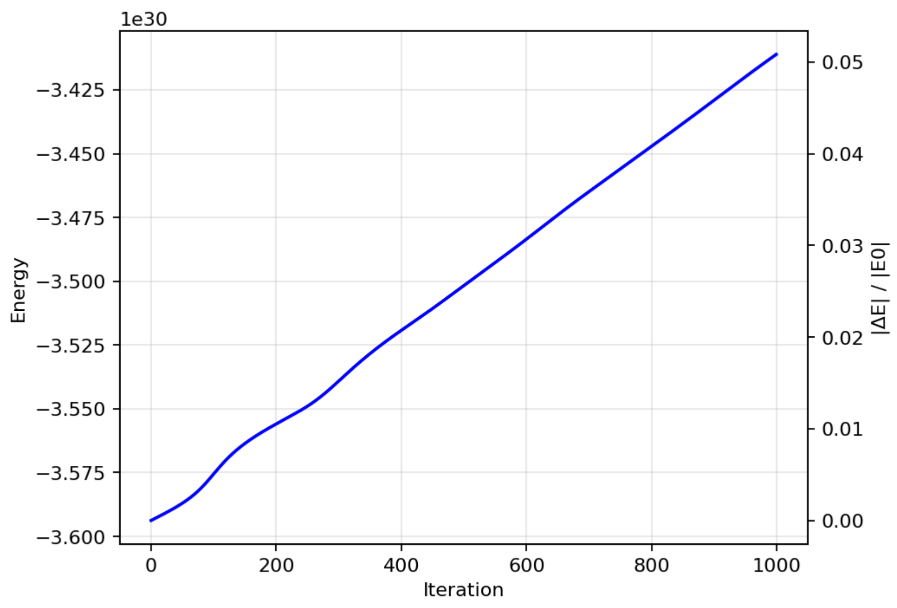


Figure 5: Non preserving energy of default esplicit-euler scheme of MUrB

5.1.2 Handling time discretization error

A first important remark is that, as stated in one study [2], usually the discretization error dominates the floating point error.

The same study states that the discretization error is responsible for introducing violation of physical low. In particular, energy and angular momentum might not be conservated.

An example of this is given in the Figure 5 where the error of the default esplicit-euler scheme imployed in MUrB is shown.

There are particular choices of integrators which tend to preserve better this quantities. An example of such methods is the category of symplectic and time-reversible operators. A prominent example of this category, often used in N-body simulations, is the `leapfrog scheme`, which integrates the motion equations providing x_n and v_n for $n \in \{1, \ldots, N\}$. The three-step version is based on the following recurrence:

$$v_{n+1/2} = v_n + a_n \frac{\Delta T}{2}$$
$$x_{n+1} = x_n + v_n \Delta T$$
$$v_{n+1} = v_{n+1/2} + a_{n+1} \frac{\Delta T}{2}$$

This recurrence can be rewritten in a more operative form which is

$$v_{n+1/2} = v_n + a_n \frac{\Delta T}{2}$$
$$v_n = v_{n-1/2} + a_n \frac{\Delta T}{2}$$
$$x_{n+1} = x_n + v_n \Delta T$$

plus some other equations to take care the beginning and the ending of the iterations. There are more complex methods, such as the seven-setp leapfrog or some time adaptive methods, which we will investigate in the future.

Unfortunately we have to report that, due to time reasons, our implementation of the three-step leapfrog scheme is badly implemented and the bodies do not behave correctly neither does the energy. We aim to fix this in the future (we hope).

6 Conclusion

We implemented and validated a progressive optimization pipeline for the MUrB n-body simulator.

GPU results. GPU implementation `gpu+tile+full` is likely to be the be the optimum or close-to optimum kernel for the direct method for simulating the N-body problem. If running on the GPU Nvidia RTX 4090 present on Dalek cluster, the code can is usefull for problems up to 500.000 bodies, after that the code is too slow. Further improvement can be retained only by moving to lower complexity models such as Barners-Hut or Multipole methods.

Main CPU results. On `iml-ia770`, the CPU pipeline yields substantial speedups on a fixed workload ($N = 8000, I = 20$): `cpu+optim` improves over the baseline by reducing redundant work and expensive operations, `cpu+simd` provides a large boost via SIMD vectorization, and `cpu+omp` achieves the highest throughput by combining SIMD with OpenMP parallelism, reaching $\sim 64\times$ speedup over `cpu+naive`.

Scaling behavior. OpenMP scaling is generally positive up to 12 threads, though the 2-thread outlier highlights the importance of repeated runs and careful affinity control on heterogeneous CPUs. N-scaling confirms the $\mathcal{O}(N^2)$ nature of the kernel, while GFLOP/s remains roughly stable across a wide range, indicating efficient utilization of compute resources.

Heterogeneous result. The heterogeneous fraction sweep on `az4-n4090` shows that partial offload offers limited gains, while full offload ($f = 1.0$) yields a dramatic improvement. This strongly suggests that minimizing CPU/GPU splitting overhead (and ideally using a fully GPU-resident kernel) is critical for maximum throughput.

References

[1] Pierre Blanchard, Nicholas J. Higham, and Theo Mary. A class of fast and accurate summation algorithms. *SIAM Journal on Scientific Computing*, 42(3):A1541–A1557, 2020.

[2] Tjarda Boekholt and Simon Portegies Zwart. On the reliability of n-body simulations. *Computational Astrophysics and Cosmology*, 2, 11 2014.

[3] Fabienne Jézéquel and Jean-Marie Chesneaux. Cadna: a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12):933–955, 2008.