University of Thessaly

Parallelization and Optimization of Nbody Simulation on GPU

Student	Garyfallia Anastasia Papadouli — 03533
Student	Filippos Markovitsis — 03488
Student	Israel Sanchez Cabrera — 04557
Professor	Christos Antonopoulos

January 9, 2026

Contents

Introduction

For the final HPC project, every concept learned in this course should be applied to accelerate an N-body simulation. **OpenMP** and **CUDA** techniques must be combined to parallelize the workload, aiming to achieve the **highest possible speedup and throughput**.

An **N-body** simulation models the dynamic evolution of a system of particles, such as stars in a galaxy, under the influence of physical forces like gravity. At every time step, the algorithm calculates the interaction of each particle with every other particle, resulting in a computationally intensive $O(N^2)$ complexity. This **exponential workload** makes it an ideal benchmark for demonstrating the massive speedups achievable through hybrid OpenMP and CUDA parallelization.

Device specifications

Before writing any CUDA code, it is essential to run deviceQuery while connected to the csl-venus, the computer provided by the needs of the assignment. DeviceQuery fetches all the specifications about every CUDA-enabled device on it.

Venus has 2 Nvidia Tesla K80 modules that have 2 GK210 GPUs each, so we can access 4 CUDA-capable devices.

However, we will be limiting the usage of devices to only one by running the following command:

```
export CUDA_VISIBLE_DEVICES=3
```

The output of deviceQuery concerning that particular part of the system is as follows.:

```
  Device 3: "Tesla K80"
CUDA Driver Version / Runtime Version          11.4 / 11.5
CUDA Capability Major/Minor version number:    3.7
Total amount of global memory:                 11441 MBytes (11997020160 bytes)
(013) Multiprocessors, (192) CUDA Cores/MP:    2496 CUDA Cores
GPU Max Clock rate:                            824 MHz (0.82 GHz)
Memory Clock rate:                             2505 Mhz
Memory Bus Width:                              384-bit
L2 Cache Size:                                 1572864 bytes
Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536),
3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
Total amount of constant memory:               65536 bytes
Total amount of shared memory per block:       49152 bytes
Total shared memory per multiprocessor:        114688 bytes
Total number of registers available per block: 65536
Warp size:                                     32
Maximum number of threads per multiprocessor:  2048
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch:                          2147483647 bytes
```

```
Texture alignment:                              512 bytes
Concurrent copy and kernel execution:           Yes with 2 copy engine(s)
Run time limit on kernels:                      No
Integrated GPU sharing Host Memory:             No
Support host page-locked memory mapping:        Yes
Alignment requirement for Surfaces:             Yes
Device has ECC support:                         Enabled
Device supports Unified Addressing (UVA):       Yes
Device supports Managed Memory:                 Yes
Device supports Compute Preemption:             No
Supports Cooperative Kernel Launch:             No
Supports MultiDevice Co-op Kernel Launch:       No
Device PCI Domain ID / Bus ID / location ID:    0 / 133 / 0
```

Some notable specifications are visible through deviceQuery , such as the maximum number of threads per block, which is 1024; the total amount of shared memory per block, which is 49152 bytes; or the warp size, which is 32.

Helper scripts

As a starting point, a sequential implementation of the algorithm is presented along with a python script *(generate_dataset.py)* that generates a dataset equivalent to the specifications of the project. To assist with the goals of the assignments, two more scripts were created. The first one, *verification.py* **compares two binary files** of the exact format of the expected input and output of the project. At first it checks wether the **headers match**, so to avoid comparison of files containing results of different datasets. If they do, it continues checking each final position one by one with a **predetermined tolerance**. The second one, *avg.py* runs the executable for a predetermined amount of iterations (def = 36) and parses the output to find the execution times and throughput. It gets rid of the two highest values and the two lowest ones as outliers and returns the **mean average** and **standard deviation**.

Baseline speed and throughput

In order to be able to measure any speedup or jump in throughput it is imperative that peed and throughput of the **sequential code** is measured. Utilizing *avg.py*, it is shown that it takes around **87.014 seconds** *(stdev: 2.048)* for the baseline code to run on **csl-venus**, with an estimated throughput of **0.4939 Billion Interactions/Second** *(stdev: 0.0113)*.

OpenMP

OpenMP offers substantial performance gains with **remarkably low code intrusion**. By utilizing **simple compiler directives**, developers can instantiate thread pools and distribute workloads across CPU cores without the overhead of manual thread management. To leverage this effectively, we targeted three specific critical sections for optimization:

1. The system traversal loop within the main function.

```
#pragma omp parallel for num_threads(num_threads)
```

2. The inner loop of the bodyForce function, which handles the primary computational bottleneck.

```
#pragma omp unroll partial(32)
```

3. The update loop within the integrate function.

```
#pragma omp unroll partial(16)
```

The outer loop implements **coarse-grained task parallelism**. Since each system acts as an independent simulation universe with its own **isolated memory block** (system_ptr), there are absolutely **no data dependencies** between them. This allows us to map entire systems to separate CPU threads safely.
The inclusion of the num_threads clause is a critical resource management decision. By explicitly defining the thread pool size, likely matching the number of systems or the physical cores available, we prevent the operating system from "oversubscribing" the CPU, creating more threads than it have to. If the number of systems (num_systems) is lower than the available cores, num_systems threads are created. Otherwise, the created threads are as many as the available CPU cores.
The selection of a **partial unrolling** for the loops inside the functions with factors of **32** and **16** (the factors were decided mostly through trial and error) is a strategic optimization designed to maximize instruction-level parallelism (ILP) and reduce loop overhead. The compiler is forced to expand the loop body to process **multiple data points** (32 or 16) within a single iteration block. This drastically reduces the CPU cycles wasted on "**administrative**" tasks like incrementing counters and checking loop termination conditions.
More importantly, this technique exposes a **larger sequence of independent instructions to the CPU pipeline**.

The results of this attempt were immediately visible. With no more than **three** simple directives, the execution time dropped to **5.19 seconds** *(std dev: 0.06)(speedup: x16,76)* and throughput jumped to **8.552** Billion Interactions/second *(std dev: 0.10)*, an increase of 17104%. These are the metrics that will be used for comparison of CPU code to GPU ones.

CUDA

The **N-body algorithm** is inherently suited for CUDA parallelization because it represents a "computational throughput" problem rather than a "latency" problem, aligning perfectly with the architecture of GPUs. The core challenge is the $O(N^2)$ **complexity**, where every particle interacts with every other particle; however, these interactions are **entirely independent during a single time step**. This independence allows us to map each particle to one of the thousands of available CUDA cores simultaneously, executing identical physics instructions on different data points without the performance penalties of thread divergence or complex branch prediction. Additionally, the N-body problem is **arithmetically intense**, requiring complex floating-point calculations relative to the amount of memory accessed. This high ratio of math-to-memory operations plays to the

GPU's strength, allowing the device to **hide memory access latency** by keeping its massive array of ALUs constantly busy. To maximize this potential, our implementation employed shared memory with **tiling**, **elimination of repeated kernel launches**, use of **multiple GPUs** using contexts and pthreads, **memory coalescing**, and **mathematical conversions** to speed up computation.

Shared Memory Tiling & Kernel Fusion

The nested loop structure of the *bodyForce* function creates an opportunity for tiling, as it is a classic solution to this code format. Instead of every thread repeatedly fetching particle data from slow global memory (**DRAM**), the threads in a block cooperate to load a "tile" of data into the on-chip shared memory (**cache**). This creates a high-speed temporary data space that functions like a **manually managed L1 cache**. Inside the computation loop, the code **maximizes arithmetic density**. By storing the current thread's body position in local registers *(temp_x, temp_y)* and accumulating forces in registers *(Fx, Fy)*, we minimize memory traffic further.
Finally, by integrating the velocity and position updates at the very end of the force calculation, we perform kernel fusion. This eliminates the need to write the forces back to memory and read them again in a separate "integrate" kernel. This first attempt at parallelizing the code yielded results of great importance, as the time of execution dropped at**2.1149 seconds** (std dev: 0.14735) and the throughput increased at **20.39B Interactions/Second**, an increase of 138,4%.

Elimination of System counting Loop

A culprit for speed and throughput immediately detected was the need for a nested loop in the main function, which launched a kernel for each system provided in the dataset.

```
...
for (iter = 1; iter <= nIters; iter++) {
  for (sys = 0; sys < num_systems; sys++) {
      /* Calculate offset for the galaxy */
      system_ptr = &data[sys * bodies_per_system];
         ...
```

Each thread identifies its system by using **blockIdx.y** as a vertical coordinate that maps directly to a specific galaxy's data range in the global memory array. This index acts as a multiplier against the total number of bodies per system, allowing **each row of blocks to offset its starting pointer to the correct memory segment.** Consequently, every thread **"detects" its system membership** implicitly through its unique position within the CUDA grid hierarchy.

```
bodyIdx = blockIdx.y*gridDim.x*BLOCK_SIZE + blockIdx.x*BLOCK_SIZE + x;
```

The detection of the system allows for passing the whole data array to the *bodyForce* kernel at once, unlocking the potential for the biggest jump in throughput yet, with **87.129B Interactions/Second**, 1,018.93% more than the openMP code and 327.31% more than the previous optimization. The time needed for the CUDA code to run is for the first time under a second (0.493 seconds, std dev: 0.0050).

Use of multiple devices

With the identification of optimizations to maximize throughput on a singular GPU becoming less intuitive, **dividing the workload to all available devices** became the logical next step. This process included identifying the number of visible devices to create the appropriate number of **pthreads** and establishing to each of them a new **context**. Then, each of the devices received an equal part of the array of length equal to NUM_SYSTEMS / NUM_DEVICES systems, which contained the bodies of each system to calculate.

Assumptions

To improve efficiency and maintain simplicity, several assumptions have been made:

- **NUM_SYSTEMS has to be divisible with NUM_DEVICES.** In the context of this project, four GPUs were available. NUM_SYSTEMS' default value is 32, satisfying this assumption. Setting NUM_SYSTEMS to a number non divisible by NUM_DEVICES will lead to inconsistent results.
- **Creation of CUDA contexrs and cudaDeviceReset() is excluded from performance metrics.** In the first attempt of the optimization at hand, the final execution time would include (for all threads) the following, resulting in the slowest thread accounting for the execution time reported.:

  1. thread creation
  2. CUDA context establishment
  3. device memory allocation and transferring data to device
  4. execution of the kernel
  5. transferring data back to host
  6. device reset
  7. joining all pthreads

  With its value being slightly bigger than the execution time of the slowest thread. The exclusion of context establishment and device reset was deemed necessary because the added overhead was way too large comparatively to the rest of the events (memory allocation, data transfers, actual execution, join). As an indication of the scale of this phenomenon, the total time measured would be around 0.8 seconds, with each thread spending around 0.5 seconds in order to prepare the contexts and 0.1 seconds in order to reset the device (leaving about just 0.2 seconds of actually relevant work!).

  Considering this, the creation of pthreads in which CUDA contexts for each device are established was moved above the generation of the input data, aiming for them to occur in parallel and therefore hiding the overhead. It was also observed that a CUDA context is attached to the thread in which it was created and it cannot be used by another thread, meaning that the launch of a kernel targeted for a context should occur in the same thread in which the context was established. This played a significant role, since it made apparent the importance of some sort of synchronization between the main thread and the threads used for the kernel launches.

Multiple-GPU Code Execution Flow

As implied in the third assumption made for the implementation of the multiple-GPU code, the program was now separated into three phases:

1. **Phase 0:** Overlapping data preparation with CUDA context preparation.
2. **Phase 1:** Data transfer to and from GPU and actual calculations, while main is waiting for the threads to finish. This is the part whose execution time we are calculating.
3. **Phase 2:** Exporting data into an output file, resetting devices and freeing memory.

In **Phase 1**, the program identifies the visible devices and, based on the number found, it creates the appropriate amount of semaphores and additional threads that will start creating CUDA contexts. Then, it follows the normal data preparation flow, storing the data in global variables.

In **Phase 2**, each thread copies in the memory of its corresponding device a part of the simulation data in the way described in the beginning of the subsection. Considering that the copied part has the same format as the data managed in the single GPU code, it proceeds to perform calculations the same way. Finally, the final data is copied in the appropriate indices of an output array. Main thread remains idle until the calculations finish.

Part 3 remains the same as it would be for the single GPU code, with the addition of destroying the created semaphores and resetting, using NUM_DEVICES new threads, the used devices.

Synchronization and Time & Throughput Measurements

Although adding *pthread_join*() resolves any kind of synchronization issue between Phases 1 and 2, synchronization should also be added between Phases 0 and 1. Specifically, between the pthreads created for the GPU workload management and the main thread, ensuring that both data preparation and context establishment are finished before actual calculations begin.

The synchronization was performed using a custom binary semaphores library, implemented within the context of subject ECE321: Concurrent Programming. The semaphores used were:

- *main_sem*: initialized with 0. Main attempts to decrease its values after the data is generated.
- *mtx*: initialized with 1. Used for mutual exclusion, as threads used for GPU tasks are attempting to increment a counter that shows how many of them have established their corresponding CUDA context.
- $*gpu\_semaphores$: An array of NUM_DEVICES semaphores, all initialized with 0. Each pthread attempts to decrease the value of its corresponding semaphore from this array, after it creates successfully the CUDA context.

The idea is that, the pthread that establishes its context last will perform *up*() for the *main_sem*, notifying main that all contexts are ready. Main then initiates the time monitoring and performs *up*() for all of the $*gpu\_semaphores$, allowing the pthreads to launch their corresponding kernel.

As mentioned, time monitoring ends after all pthreads join. An important detail to mention here is that, for this to be achieved, blocking $cudaMemcpy(..., cudaMemcpyDeviceToHost)$ is used, otherwise the execution of the kernels and the data transfers would be asynchronous and, therefore, results would be inconsistent.

The throughput achieved by splitting the execution among devices skyrocketed to a staggering **298.392B Interactions/Second** (std dev: 21.442), with the execution time dropping to an impressive **0.1480 seconds.** (std dev: 0.0036)

Memory Coalescing

Observing the plummeting execution time and vertical increase of throughput after the last optimization, the next steps had to be done precisely and targeted to the specific problems the code suffered from. To achieve that we relied on **Nsight compute**. Specifically, a report was generated running ncu through the commandline.

```
ncu -set full -o report ./nbody
ncu --import report.ncu-rep >> report.txt
```

The output of the profiling was rather illuminating. The initial performance analysis revealed a critical bottleneck in the memory subsystem. The kernel exhibited **inefficient global memory access patterns**, characterized by 5,898,240 excessive sectors, which constituted **83%** of the total memory traffic. This inefficiency stemmed from the use of an Array of Structures layout (struct Body x, y, z...), where threads within a warp accessed **non-contiguous memory addresses**. Consequently, for every 32-byte sector loaded from DRAM, only 5.3 bytes were effectively utilized by the threads. The initially high L1 Hit Rate (73%) was misleading; it indicated **"false hits" on useless data** (e.g., loading vx when only x was needed) rather than efficient data reuse.

```
 Section: Memory Workload Analysis
-------------------------------------- ----------- ------------
Metric Name              Memory              Metric Unit Metric Value
-------------------------------------- ----------- ------------
Local Memory Spilling Requests                                   0
Local Memory Spilling Request Overhead           %               0
Memory Throughput                          Mbyte/s          492.87
Mem Busy                                         %           20.32
Max Bandwidth                                    %           20.30
L1/TEX Hit Rate                                  %           72.97
L2 Hit Rate                                      %           92.55
Mem Pipes Busy                                   %           20.30
-------------------------------------- ----------- ------------

  Section: Memory Workload Analysis Tables
  OPT   Est. Speedup: 33.87%
The memory access pattern for global loads from L1TEX might not be optimal.
On average, only 5.3 of the 32
bytes transmitted per sector are utilized by each thread.
This could possibly be caused by a stride between
threads. Check the Source Counters section for uncoalesced global loads.
```

To address this, the memory layout was refactored to a Structure of Arrays format. By storing components in separate, contiguous arrays, memory accesses became **fully coalesced.** Threads in a warp can now load a continuous block of 32 floats in a single transaction. Subsequent analysis of the

optimized kernel showed a precipitous drop in the L1/TEX Hit Rate to 5.4%. This metric change confirms the elimination of false hits and indicates that the memory controller is now streaming new, useful data efficiently.
Measuring the changed code for speed and throughput revealed a small cut in execution time **(0.1358 sec.), (std dev: 0.0063)** and a bump in throughput **316.940B Instructions/Second** (*std dev: 13.219*) of **6.22%.**

Instruction fusion

With the memory subsystem no longer throttling performance, the bottleneck has shifted to the compute units. The "*Warp State Statistics*" indicates that warps now spend **41.7%** of their time stalled waiting for the Execution Pipe (specifically the FMA/Math units) to become available. Additionally, the "Scheduler Statistics" show that **40.35%** of the time, no warps are eligible to issue instructions because they are all blocked waiting for previous math operations to complete.

To address the compute-bound bottleneck identified, we enabled aggressive floating-point optimizations (Fast Math), specifically targeting the saturation of the execution pipelines. This optimization leveraged the **fmaf** (Fused Multiply-Add) instruction, which collapses separate multiply and add operations into a single-cycle step, drastically reducing the instruction count required for the critical distance calculation ($dx^2 + dy^2 + dz^2$). Additionally, we ensured the use of the **rsqrt()** intrinsic, which maps the inverse square root calculation directly to a dedicated hardware unit, bypassing the higher latency of standard software division routines. These changes directly alleviated the execution pipe stalls by increasing the arithmetic intensity the GPU could sustain, effectively converting the available compute bandwidth into faster simulation throughput. The measurements of execution time and throughput at this stage, characterizing the last of our experiments, indicated another slight improvement in both. The final time measured sits at **0.1100 seconds** (std dev: 0.0046) and the final throughput at **391.427** *(std dev: 15.293).*

Final Takeaways and results

Table 1: Performance Metrics for Different Optimizations

Optimization	Time (sec)	Time Std	Throughput (B Interactions/s)	Throughput Std
OpenMP	5.19	0.06	8.552	0.10
GPU tiling	2.1149	0.14735	20.39	1.414
Elimination of system loop	0.493	0.0050	87.129	0.8644
Use of multiple gpus	0.1480	0.0036	298.392	21.442
SoA	0.1358	0.0063	316.940	13.219
faster math computations	0.1100	0.0046	391.427	15.293

The experimental results obtained by optimizing the N-Body simulation illustrate a dramatic evolution in performance. The baseline single-threaded CPU implementation exhibits the lowest throughput at approximately 0.49B Interactions/s (87.014s). This metric clearly indicates a **compute-bound process** that is highly inefficient for serial execution.

The first major performance leap is observed with the introduction of **OpenMP**, which parallelizes the workload across available CPU cores. This results in a roughly 17× speedup (5.19s), demonstrating the effectiveness of utilizing multi-core architectures. However, this improvement is quickly eclipsed by the transition to GPU acceleration. The initial GPU optimization using tiling exploits shared memory to drastically reduce global memory bandwidth pressure, lowering the execution time to 2.11s.

A critical architectural insight is revealed with the elimination of the system loop. By refactoring the control flow to minimize host-device synchronization and kernel launch overhead, the performance quadruples to 87.1B Interactions/s (0.493s). This substantial jump suggests that for high-performance kernels, **system overhead can often become the dominant bottleneck.** Scalability is effectively demonstrated through the multi-GPU implementation, which nearly triples the throughput to 298.4B Interactions/s. This near-linear scaling confirms the algorithm's suitability for distributed parallelization, as the interaction calculations can be partitioned with minimal inter-device communication. Finally, low-level refinements push the hardware to its limits. Switching to a Structure of Arrays layout **improves memory coalescing,** ensuring optimal memory bus utilization, while the use of *–use_fast_math* and *fmaf* trades negligible precision for **faster hardware instruction execution.** The cumulative effect of these optimizations results in a **peak throughput of 391.4B Interactions/s (0.110s)**. This represents an astonishing $\approx 790\times$ **total improvement over the original serial code**, underscoring the necessity of an approach that combines algorithmic parallelism, memory access optimization, and hardware-specific tuning.

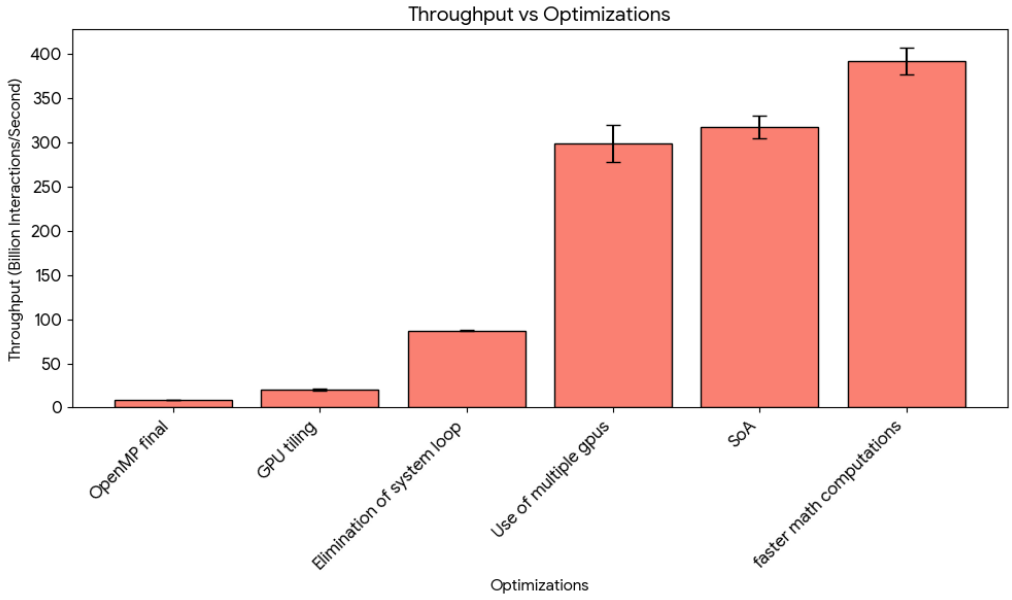

Figure 1: Throughput with different code optimizations

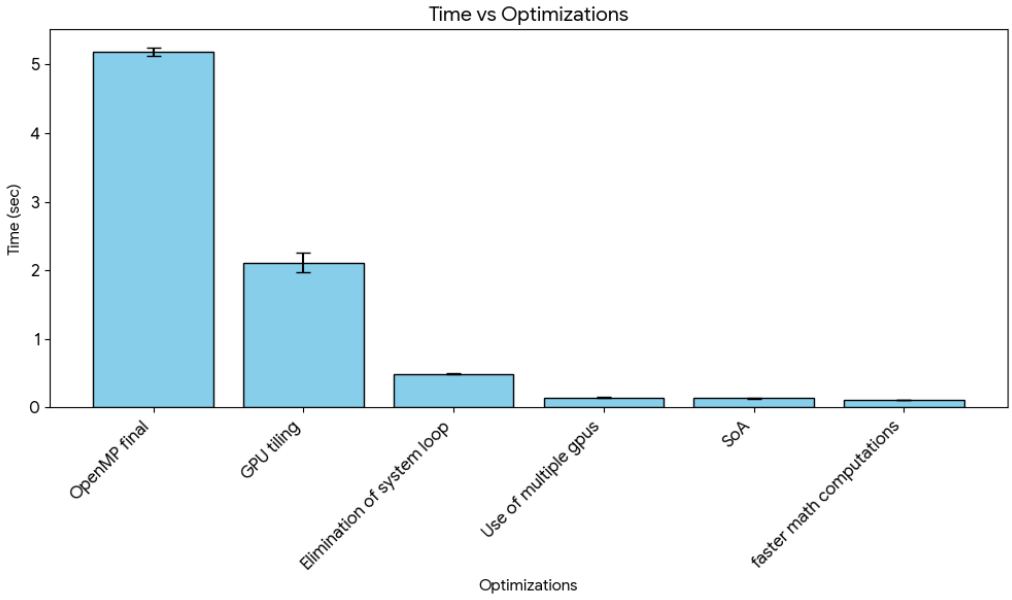

Figure 2: Execution time with different code optimizations