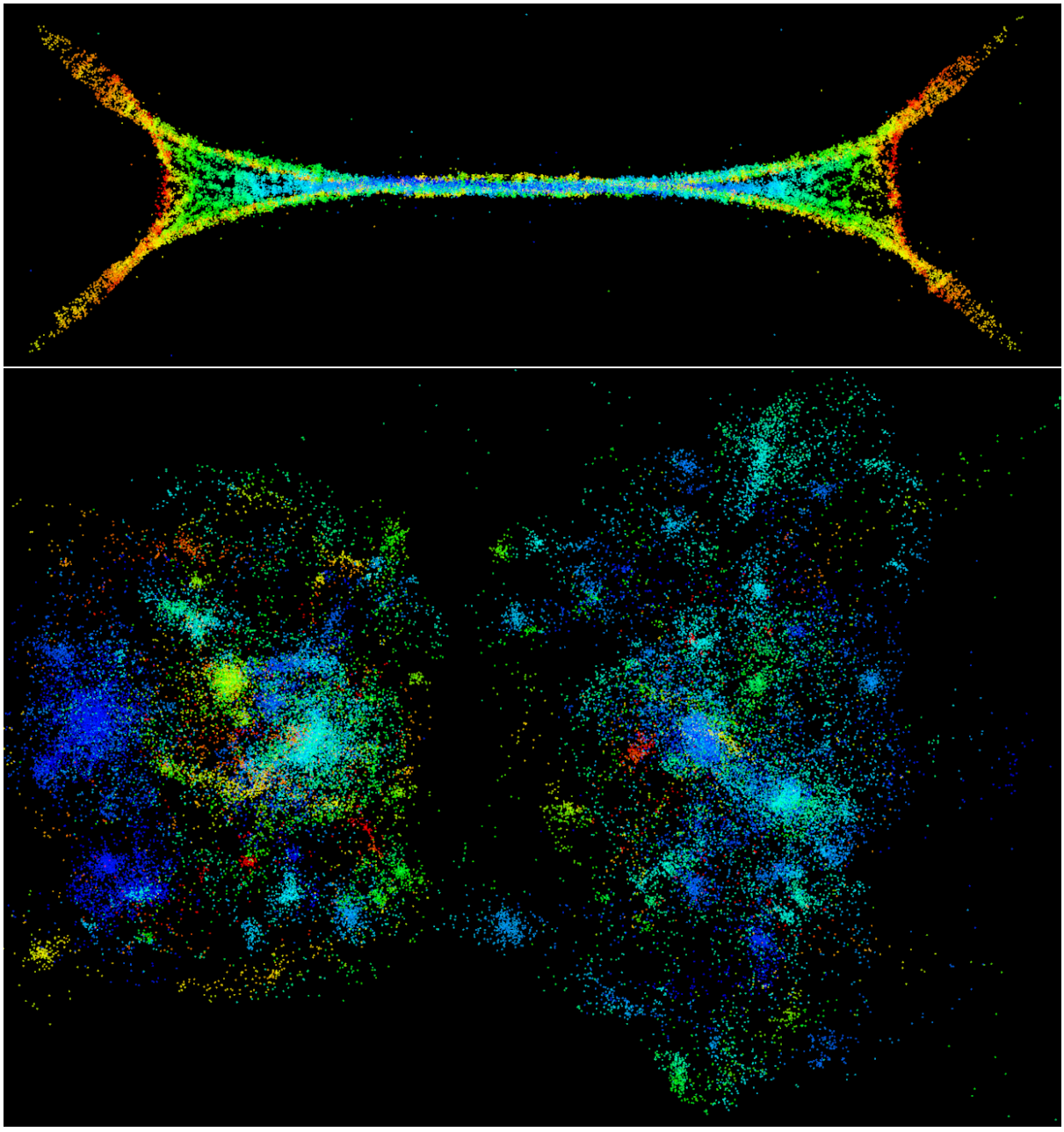
Particles Gravity Simulation

With CUDA and SFML



Michael Marchesan – 945887

Table of Contents

Objective 3

Algorithms 3

Environment 3

- Libraries 3
  - Limitations 3
- Hardware 3
- Benchmarking 3

Underlying structures 4

- CPU-only 4
  - Matrix 4
- CUDA 4
  - vec2 4
  - vector 4
  - matrix 5
  - launcher 5

Rendering and program structure 6

Gravity Simulation implementations 7

- CPU 7
  - Nested Foreach Sequential 7
  - Nested Foreach Parallel 7
  - Forces Matrix Sequential 7
  - Forces Matrix Parallel 7
- GPU 8
  - Nested Foreach 8
  - Forces Matrix 8
  - Nested Foreach SMEM 8
- Performance 9

Objective

The objective of the project is to simulate gravity with multiple particles. More generally, this is a case of an “N-body” problem.

Algorithms

The only way to accurately run an N-body problem, is to iterate each element, for each element, with a complexity of n^2. That means, for each particle, we iterate all the particles and calculate the total gravitational force vector. This solution will be referred to as *Nested foreach*.

Another solution was attempted, but no attempt was done in optimizing it, since It would have ended up being slower than the n^2 solution. It involves evaluating and storing the total gravitational field in each pixel of the world, and then for each particle, applying the force on its corresponding pixel. This solution will be referred to as *Forces matrix*. This is already less accurate, since particles positions can have floating point numbers, whereas pixels are discrete, so the force to apply to a particle would be the force at the closest integer position.

There are, however, different ways to produce a less accurate but more efficient simulation, like storing the particles in a quad tree data structure.

In order to leverage the full potential of parallel computing, I decided to focus on the *Nested foreach* solution.

Environment

Libraries

3 different libraries were used in the project:

- CUDA, used for computation.
- SFML, used for rendering, window management and input management.
- Utils, used for some data structures and compilation utilities.

Limitations

SFML is meant to be used from a CPU “only” program, abstracting all the graphics away. I didn’t invest time in delving into CUDA-OpenGL or CUDA-Vulkan interoperability, because it seemed a huge time investment outside of the scope of this project, and would have involved ditching SFML in favour of writing all the rendering code in pure OpenGL or Vulkan.

That means all the data to render the particles, must go through the CPU side in order to perform rendering. This was somewhat alleviated by using pinned memory.

Hardware

The code was run with an Nvidia 2070 Super GPU and a Ryzen 7 3800x CPU, with 32GB of RAM @3600MHz. Suboptimal CPU cooling has potentially affected CPU implementations benchmarking.

Benchmarking

The program itself keeps track of the average time required by each frame computation, independently for each different implementation of the simulation that is running.

Underlying structures

These are the data structures I made for and used in this project.

CPU-only

Matrix

(See file containers/matrix.h in the utils library)

The class `utils::matrix_dyn<T>` represents a bidimentional matrix, which internally stores data in a linear array. It exposes accessors for both the internal linear storage, which take a single index as parameter, and for bidimensional storage, which take an x, y indices pair as parameters.

Most importantly, it exposes functions to convert a 1d index to and from 2d indices.

Ultimately, it exposes standard functions to enable std functionalities, like ranged loops and std algorithms.

CUDA

These classes enforce modern C++ RAII conventions on memory management for CUDA and expose standard functions to enable std functionalities, like ranged loops, on the GPU side.

vec2

(See file CUDA/vec2.h)

The class `utils::math::vec2` used in this project is an identical copy of the `utils::math::vec2` class in my utils library, with the only difference that methods as marked using CUDA's macros in order to make them available on the GPU side.

It exposes utility functions for performing arithmetic on 2d vectors.

vector

(See file CUDA/Vector.h)

The classes `utils::CUDA::vector`, `utils::CUDA::device_vector` and `utils::CUDA::shared_vector`, are used to simulate `std::vector` on the device side. Their biggest limitation is that they do not allow for storage resizing; however, the name vector was chosen, because they are meant to be mapped to CPU side vectors.

- `utils::CUDA::device_vector` is meant to be used exclusively on the GPU side. It doesn't do any memory management, it just exposes an `std::vector`-like interface. It must be constructed on the CPU side, and then passed to the GPU. The user can construct this data structure, by passing it an `std::vector` as parameter, as long as that vector is using pinned memory for internal storage, which can be achieved by specifying Nvidia's `thrust::system::cuda::experimental::pinned_memory<T>` allocator when declaring the CPU side source vector. In that case, no copy will happen. The `device_vector` will access directly the source vector's data. It can also be constructed by a GPU global memory managing structure (see the next point).
- `utils::CUDA::vector` is a CPU manager for GPU side memory allocation. Its lifetime is tied to a CUDA global memory array. It has utility methods to copy data from a CPU side `std::vector` to the device array and vice-versa. It also exposes a method to get an

`utils::CUDA::device_vector` that points to the managed global memory; that instance is meant to be passed to a kernel for use on the GPU side.

- Finally `utils::CUDA::shared_vector` is a GPU side only data structure meant to abstract shared memory usage. It makes a couple of assumptions for its usage: the `shared_vector` must be constructed with a size equal to the amount of threads in a block. It must be constructed passing a `device_vector` (whether that vector was constructed from pinned memory or global memory is irrelevant). It exposes a method (load) which takes a callable as parameter. That method takes care of loading the source vector in chunks equal to the shared vector size in parallel (with each thread loading a slot of memory) and then running the callable with the data readily available in shared memory.
  It also takes care about not overflowing data and indices on the last iteration, if the source vector's size isn't an exact multiple of the shared vector size.

matrix

(See file CUDA/Matrix.h)

The `utils::CUDA::device_matrix` and `utils::CUDA::matrix` classes act exactly like their vector counterparts, but they work on `utils::matrix` instead of `std::vector`.

I did not make an `utils::CUDA::shared_matrix` class because it was not needed for the project.

launcher

(See file CUDA/Launcher.h)

The launcher class serves no practical function. I made it to hide Nvidia's kernel call syntax, because Microsoft's intellisense was unable to report errors before compile time in lines that involved kernel calls.

```
function<<<blocks, threads, smem>>>(parameters);
```

becomes

```
Launcher<function>{blocks, threads, smem}(parameters);
```

Rendering and program structure

(See files Window.h, Elements.h)

The program revolves around a generalized window manager made with SFML. The window manager can be populated with multiple “updatable”s, which have to expose an update method.

The updatable class takes care of keeping track of the average exacution time of each update call, while the window manager takes care of enabling or disabling individual updatables execution and rendering.

(See file PS_Base)

For the specific application, all the updatables are different implementations of the gravity simulation, called “particle systems”, which inherit from the `Particle_system::base class`, which takes care of initializing the system, allocating memory for the particles and placing them at random locations.

The particle systems all need to store an `std::vector<sf::Vertex>`, which represents the particles to be drawn at screen. For the previously mentioned SFML limitations, that vector must exist on the CPU side of the program. For that purpose, in the GPU implementations I ended up allocating the vertices vector in pinned memory, using Nvidia’s pinned memory allocator, so to make it available to the GPU without making copies. This more than halved the execution time, compared to having to copy particles data from GPU’s global memory to CPU’s memory on each frame, which in low amounts of particles actually made the CUDA implementation slower than the CPU one.

(See files Color_manip.h, Particle_system.h)

Finally, a global `mass` used for gravity computation in all systems, a function that calculates the gravitational attraction between two particles and a function which changes a particle’s colour based on its total velocity are available both on CPU and GPU side.

Gravity Simulation implementations

> Note that the forces matrix implementations are incomplete and have uncorrected bugs (both when snapping the particles into the available space and when calculating forces). However, since those versions are inherently slower, I didn't fix them. I've only kept them as reference.

These are the implementations that are available in the program:

CPU

Nested Foreach Sequential

(See file PS_CPU_Nested_foreach.h)

The most naïve implementation. Prepare a vector of forces to apply; on each frame, for each particle iterate all particles and sum up the gravitational force vectors. Once all the forces are evaluated, iterate on all particles and add the final force vector to their position vector.

Nested Foreach Parallel

(See file PS_CPU_Nested_foreach.h)

Same as before, but makes use of C++'s execution policies to run the code in parallel for each entry of the outermost loop when calculating forces, and for each particle when applying them.

Forces Matrix Sequential

(See file PS_CPU_Forces_matrix.h)

Allocates a matrix of forces that represent the total gravitational force on each pixel of the window. Particles cannot move outside of the original window size.

On each frame, the gravitational force on all entries of the matrix from each particle is summed up; then, for each particle, its position is floored to the nearest integers, and the force calculated at those indices in the forces matrix is applied to the particle.

Forces Matrix Parallel

(See file PS_CPU_Forces_matrix.h)

Same as before, but makes use of C++'s execution policies to run the code in parallel for each entry of the matrix when populating its forces, and for each particle when applying them.

GPU

Note: All GPU implementation use pinned memory to store the vector of particles.

Nested Foreach

(See files PS_GPU_Nested_foreach.cuh and PS_GPU_Nested_foreach.cu)

A simple port of the CPU implementation to the GPU. Instead of having an outer loop on all the vertices, the outer loop entries are split in different CUDA threads.

The vector of forces to apply is exclusively stored on the GPU0's global memory, with no CPU side counterpart, as its values are of no relevance to the CPU side of the program.

Forces Matrix

(See files PS_GPU_Forces_matrix.cuh and PS_GPU_ Forces_matrix cu)

A simple port of the CPU implementation to the GPU. Instead of having an outer loop on all the entries of the forces matrix, the outer loop entries are split in different CUDA threads.

The forces matrix is exclusively stored on the GPU, with no CPU side counterpart, as its values are of no relevance to the CPU side of the program.

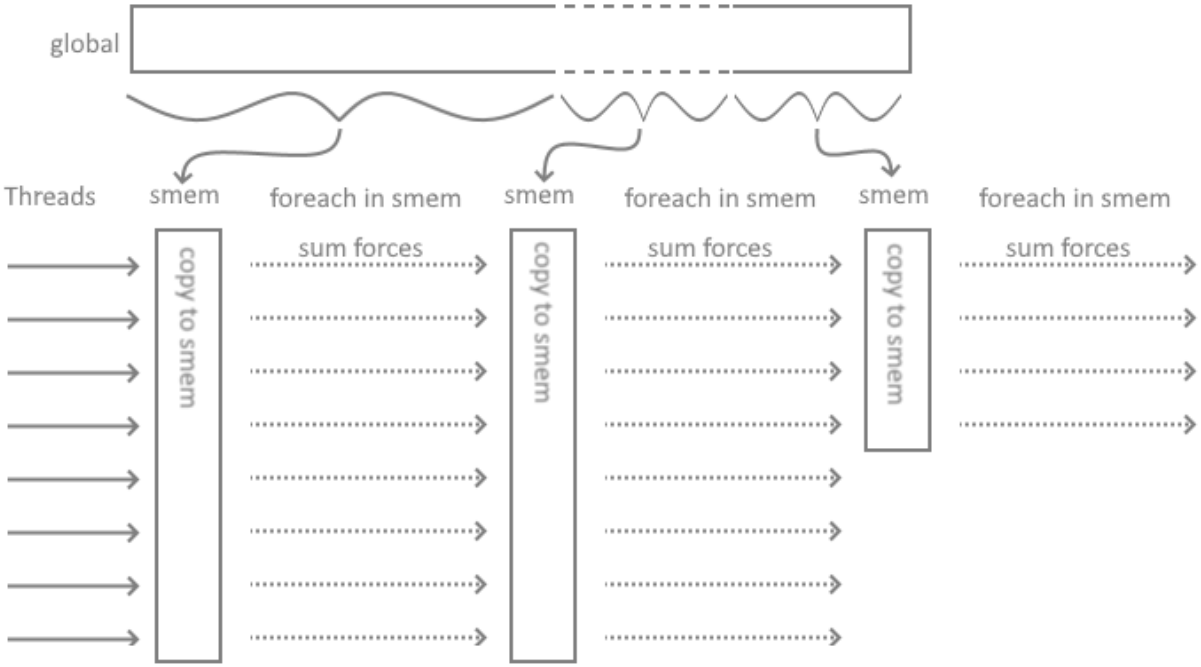
Nested Foreach SMEM

(See files PS_GPU_Nested_foreach.cuh and PS_GPU_Nested_foreach.cu)

This version has no CPU counterpart, as it makes use of shared memory. It makes use of my `shared_vector` data structure to load particles in chucks in order to perform the total force evaluation. The forces applying part has no change compared to the other nested foreach version.

The preloading on shared memory greatly affects the results, with another halving of the computation time per frame.

Here is a visual representation of the usage of shared memory:



Performance

The following table represents the average time take taken by the update calls with a given amount of frames. For the slowest versions I've only run a single frame.

With 50'000 particles

Implementation	Frames	Average time (in nanoseconds)
CPU Nested foreach (sequential)	10	10412234900ns
CPU Nested foreach (parallel)	100	1317002015ns
CPU Forces matrix (sequential)	1	1121476779700ns
CPU Forces matrix (parallel)	1	111852161200ns
GPU Nested foreach	100	78793397ns
GPU Forces matrix	100	1934806767ns
GPU Nested foreach (SMEM)	100	37360690ns

As the results show, the GPU implementations are 2 to 3 orders of magnitude faster than the CPU counterparts. Moreover, the version that makes use of shared memory is twice as fast as the one that directly accesses global memory on each iteration.

Block size

Originally I tried using Nvidia's `cudaOccupancyMaxPotentialBlockSize` to get the theoretical best occupancy; however asking around it seems that this function is at best unreliable, so I ditched it.

In order to find the optimal block size, I wrote an alternative update method that uses incrementally larger block sizes, from 32 onwards, for the first frames, and then settles on the block size that led to the fastest frame evaluation. This code is still available in the GPU Nested foreach implementation; however it still isn't reliable enough: the evaluation time of a single frame can be offset by external factors; this leads that approach to sometimes select a suboptimal block size, and then continue with that size through the whole execution.

However, after multiple runs with that approach, I've manually picked and hardcoded the block size that led to the best timings across multiple tests, which, in the case of 20k, 40k and 50k particles was always 256.