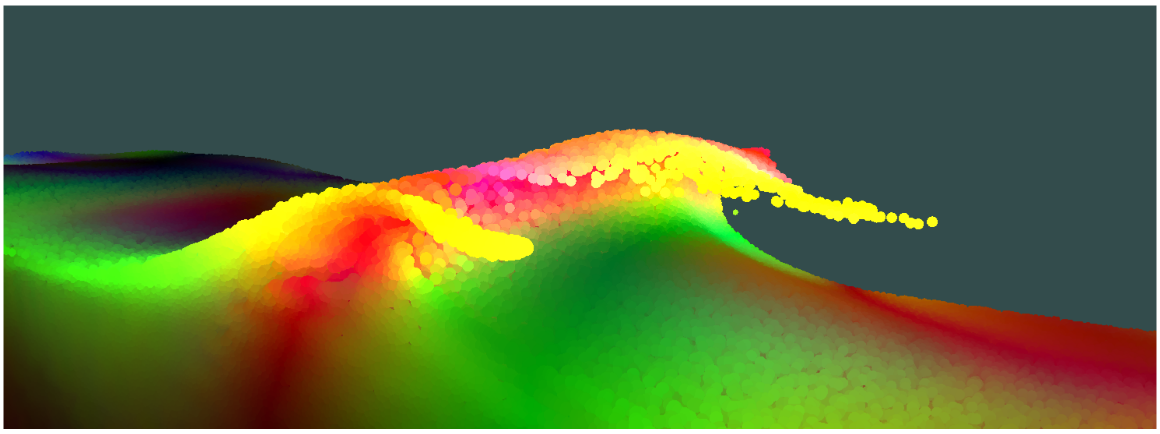
MATERIAL POINT METHOD FOR REAL-TIME FLUID SIMULATION IN CUDA

Final project submission for course: GPU Computing (2025)



Luca Leopardi
Matr. No. 16080A

Università degli Studi di Milano, Dipartimento di Informatica "Giovanni degli Antoni"

Contents

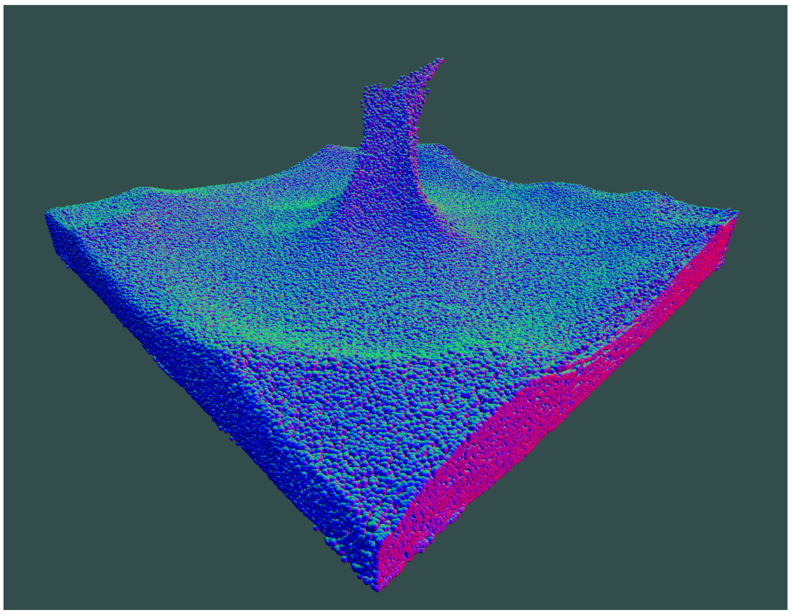
Contents

Chapter 1

Introduction

This project follows the development of a GPU implementation of the Material Point Method (MPM) algorithm. The algorithm is first presented in its general form, along with a sequential CPU implementation that will be used as a performance baseline for the following parallel CUDA version. Several optimization techniques are then explored and their impact measured.

The implementation is realized as part of an application for real-time fluid simulation, to be executed locally on a CUDA-capable machine. Rendering is done with OpenGL.



1.1 How to compile

- Install CUDA Toolkit (v13.1 or above).
- Install VS Code.
- Download Visual Studio Installer and install *Visual Studio Build Tools 2022 - C++ desktop applications* workload with components:
  - MSVC
  - Windows SDK
  - CMake tools
  - vcpkg package manager
- Install vcpkg via git and run setup script. No need for other setup steps.
- Install CMake (v4.0.3 or above).
- Install project's recommended extensions in VS Code (`.vscode/.extensions.json`).
- Edit `CMakePresets.json` file: `CMAKE_TOOLCHAIN_FILE` must point to `[VCPKG_INSTALL_FOLDER`
- In VS Code > CMake side panel:
  - Run Configure and select Visual Studio Build Tools 2022 Preset.
  - Run Build (Release or Debug preset).
  - Launch application or debugger.

1.2 Testing methodology

All results in this report were captured on a machine with the following specifications:

- **CPU:** AMD Ryzen 7 5700X3D (8 core | 3.00 GHz)
- **GPU:** NVIDIA GeForce RTX 3060 (12 GB VRAM) (CC 8.6)
- **RAM:** 16 GB 3200MHz
- **Storage:** SSD

Program benchmarks were captured using Nsight Systems, while precise kernel performance was analyzed using Nsight Compute. Benchmark values were averaged over 5000 frames. Two simulation scenarios were tested: 65536 and 655360 particles, both within a 100x100x100 MPM grid.

Fluid rendering is absent in this project, but particles can be visualized either as points or instanced quad ("impostor") spheres, and with color representing velocity. Benchmarks were run with minimal solid color point rendering per particle: this allows to measure the impact of interfacing with OpenGL, while keeping the rendering cost to a minimum.

Kernel configuration was kept at BlockDim 128 throughout the implementation, while GridDim varied among solutions.

1.3 The Material Point Method

The Material Point Method (MPM) [1] is an algorithm for material simulation, mostly used in offline rendering and scientific modeling of particles, fluids, and elastic materials.

In the context of fluid simulation, the choice of frame of reference categorizes the flow field specification as either Lagrangian or Eulerian: the former follows the change in position of fluid particles, while the latter follows the passage of fluid in specific positions. MPM can be seen as a combination of the two: it follows particles positions, but interactions among them are mediated by a grid of cells at fixed points in space. This way, some physics calculations become easier than in pure Lagrangian methods.

The MPM algorithm consists of the following steps, repeated at each iteration:

1. **Grid reset:** Background grid Cells' values are reset (or initialized, if it's the first step) to zero.

2. **Particles to Grid transfer (P2G):** For each Particle, its mass, momenta, and relevant forces are scattered to its neighboring grid Cells, interpolated by a shape function.

3. **Grid update:** For each Cell, new velocity values are calculated from data gathered from Particles.

4. **Grid to Particles transfer (G2P):** For each Particle, velocity data is gathered from neighboring Cells, interpolated by a shape function.

5. **Particles integration:** Particle position is integrated using updated velocities.

In this project, the Moving-Least-Squares Material Point Method (MLS-MPM) [2] variant of the algorithm is implemented: the interpolation kernel is given by a Moving Least Squares approximation, and material stress is calculated using affine velocity instead of a deformation gradient.

A derivation of the Tait equation of state is used to model stress for the fluid material. Also, a Particle density estimation is added in P2G instead of being derived from their evolving volume.

Simulation boundary conditions are applied both to grid Cells and Particles: edge Cells velocity is zero-ed in the direction of the boundary, and Particle position is clamped to the simulation domain after integration.

The MLS-MPM algorithm, with the number of its per-Cell and per-Particle operations, is well suited for a parallel implementation. In this project, a goal of real-time 60 FPS performance was kept: a <16 ms frame-time would allow the simulation to run every frame at the chosen integration step of 16 ms, necessary to maintain stability.

Chapter 2

Single-thread implementation

For the CPU implementation, `Cell` and `Particle` structs have been created to store the data needed by each in the various steps of the simulation.

```
struct Cell
{
    glm::vec3 velocity;
    float mass;
};

struct Particle
{
    glm::vec3 position;
    glm::vec3 velocity;
    float mass;     // Constant, removed in later versions
    float volume;   // Changed to density in later versions
    glm::mat3 velocity_gradient; // Affine velocity gradient
    ParticleMaterial material;      // Physical characteristics
};
```

A `MPMSimulation` class holds vectors with Cells and Particles data, and has a `step()` method that runs one whole iteration of the MPM algorithm. Each frame delta time is accumulated and, if the 16 ms time-step is reached, it's consumed for one call to `step()`.

Each phase of the MPM algorithm corresponds to a `for-each` loop over all Cells or Particles. Within these, operations that scatter or gather data between the two require a loop over the 3x3x3 Cells surrounding a Particle.

A device-side OpenGL Buffer holds Particles data for rendering. After each simulation step, data is copied from the host-side `MPMSimulation` vector to the device-size Buffer.

```
void MPMSimulation::step()
{
    // 1. Reset grid, zero out mass and velocity for each cell
    for (Cell& cell : _grid) {
        // Reset cell...
    }

    // 2. P2G
    // Scatter particle mass to the grid
    for (Particle& particle : _particles) {
        for (Cell& cell : grid_neighborhood) {
            // Scatter mass (interpolation)...
        }
    }
    //Scatter momentum to the grid
    for (Particle& particle : _particles) {
        for (Cell& cell : grid_neighborhood) {
            // Find volume from mass/density (interpolation)...
        }
        for (Cell& cell : grid_neighborhood) {
            // Scatter momentum (interpolation)...
        }
    }

    // 3. Grid update
    for (Cell& cell : _grid) {
        // Calculate cell velocity, apply boundary condition...
    }

    // 4. G2P
    // Gather velocity from grid
    for (Particle& particle : _particles) {
        for (Cell& cell : grid_neighborhood) {
            // Gather velocity (interpolation)...
        }
        // 5. Particles advection
        // Integrate position, clamp to domain...
    }
}
```

2.1 Benchmark

In the testing scenario of a 100x100x100 grid, simulating 65536 Particles yields an average frame-time of 37,26 ms (27 FPS). The simulation accounts for 35,93 ms of each frame's computation time.

CPU | 100x100x100 grid | 65536 particles

Operation	**Time (ms)**
Grid reset	0,569
P2G + Density estimate	18,662
Grip update	0,97
G2P + Advection	14,935
glBufferData	0,8
Total	**35,39**

As expected the operations with the highest cost are the P2G and G2P steps, which involve the scattering and gathering of data between Particles and Cells. Per-iteration host-to-device data transfer is also costly, though acceptable at this scale of simulation.

Chapter 3

CUDA implementation

For the parallel implementation, `MPMSimulation` was changed to rely on CUDA kernels to handle each phase of the MPM algorithm: `grid_reset` and `grid_update` map each thread to a grid Cell, while `p2g_init`, `p2g`, `particles_estimate_density`, `g2p` and `particles_advect` are mapped to Particles. Kernels were also written for Particles initialization, though not accounted in benchmarks since they only run at tests start.

Between each phase of the MPM algorithm a synchronization barrier is required, so that subsequent calculations use up-to-date data. A barrier is also needed between the mass scatter and Particle density estimate operations in the P2G phase, so it was split into separate kernels: `p2g_init`, to scatter mass to the grid, `particles_estimate_density` to estimate Particle density, and `p2g`, to scatter resulting velocity.

In P2G, when Particles concurrently scatter data to grid Cells, it is also necessary to ensure atomic operations with `atmoicAdd`. This is not needed in G2P, as data from neighboring Cells is only read.

3.1 Naive implementation

As a first parallel approach, a kernel is created for each phase of the MPM algorithm, plus the additional `p2g_init` aand `particles_estimate_density`. Simulation data is now kept on-device, but still needs to be copied to the host to be passed to the OpenGL Buffer for rendering.

3.1.1 Benchmark

Even with the naive approach, the parallel implementation yields a performance gain of around 40x the CPU version, with frame-time averaging at 0,91 (1094 FPS) in the base test scenario. Added together, the memory transfers from CUDA and to OpenGL are now the most expensive operations per iteration. However, with each transfer being around 4 MB in the first test scenario, it can still maintain a high frame-rate.

Naive CUDA | 100x100x100 grid | 65536 particles

Operations and kernels	**Time (ms)**
grid_reset	0,048
p2g_init	0,106
estimate_density	0,052
p2g	0,4
grid_update	0,053
g2p	1,55
particles_advect	0,017
cudaMemCpy	1,013
glBufferData	0,9
Total	**4,139**

However, scaling the simulation to 655360 particles with the second scenario, performance drops to around 35,62 ms (28 FPS). P2G and G2P in particular are the kernels where most of the Particle calculations and memory accesses occur, and it can be noted that their execution time scales roughly linearly with the number of Particles. Data transfer from CUDA and to OpenGL, now at 40 MB per iteration, also take longer.

Naive CUDA | 100x100x100 grid | 655360 particles

Operations and kernels	**Time (ms)**
grid_reset	0,048
p2g_init	1,23
particles_estimate_density	0,527
p2g	3,711
grid_update	0,054
g2p	14,129
particles_advect	0,136
cudaMemCpy	3,513
glBufferData	5,1
Total	**28,448**

3.2 CUDA-OpenGL interoperability

Data transfer between host and device is the most obvious pain point. Keeping CUDA and OpenGL separated, necessitating host data as a temporary stop-through, is wholly avoidable: since simulation happens on the GPU, as does rendering, data can avoid ever passing through the host.

CUDA offers interoperability with numerous graphics API, including OpenGL. It is possible to register OpenGL Buffers as CUDA resources, and mapping them to developer pointers using the appropriate API. Cells and Particles data can be handled as OpenGL Buffers, mapped to CUDA resources for each iteration, and then unmapped for rendering.

```
// Map OpenGL resources to CUDA
Particle* d_particles;
cudaGraphicsMapResources(1, &_particles);
cudaGraphicsResourceGetMappedPointer(
    (void**) &d_particles,
    NULL,
    _particles);

// Run CUDA kernels...

// Unmap resources
cudaGraphicsUnmapResources(1, &_particles);
```

3.2.1 Benchmark

Kernels timings are unchanged, as their code hasn't changed. However, with the cost of OpenGL interoperability being negligible, the data transfer waits are shaved off. Average frame-time is now 21,14 ms (47 FPS), getting closer to the simulation being able to run in real-time.

CUDA-OpenGL interoperability | 100x100x100 grid | 655360 particles

Kernel	Time (ms)
grid_reset	0,048
p2g_init	0,924
particles_estimate_density	0,529
p2g	3,758
grid_update	0,054
g2p	14,356
particles_advect	0,136
Total	**19,805**

3.3 Memory-aligned data

Data can be aligned in 16, 32, 64 or 128 byte segments to coincide with the size of Global Memory transactions and reduce uncoalesced accesses. The `Cell` and `Particle` structs can be aligned as 16 and 64 bytes respectively. `ParticleMaterial` can also be passed as a kernel parameter instead of being stored per-Particle, reducing their struct size. Particle mass, being a constant, is also moved to `ParticleMaterial`.

```
struct alignas(64) Particle
{
    glm::vec3 position;
    glm::vec3 velocity;
    glm::mat3 velocity_gradient;
    float density;
};
```

3.3.1 Benchmark

The simple change to aligned memory much improves G2P, where a lot of memory accesses occur. With frame-time around 9 ms (110 FPS), the simulation can now run in real-time.

Memory-aligned structs | 100x100x100 grid | 655360 particles

Kernel	Time (ms)
grid_reset	0,048
p2g_init	0,997
particles_estimate_density	0,525
p2g	3,763
grid_update	0,054
g2p	8,54
particles_advect	0,133
Total	**14,06**

3.4 AoS vs SoA

To further optimize memory access, data organization can be changed from being an *Array of Structures* to a *Structure of Arrays*: instead of storing arrays of Cell and Particle structs we can store each of their members in separate arrays, with each entity being defined as the members located at the same index in each array. This also allows some kernels to operate on less data, as they don't need to access all the members of the structs.

```
cudaGraphicsResource* _cells_velocities;
cudaGraphicsResource* _cells_masses;
cudaGraphicsResource* _particles_positions;
cudaGraphicsResource* _particles_velocities;
cudaGraphicsResource* _particles_velocity_gradients;
cudaGraphicsResource* _particles_densities;
```

Where applicable, the arrays' data types were still aligned to multiples of 4 bytes to optimize memory accesses.

3.4.1 Benchmark

Performance increases for almost all kernels, and especially for the memory-intensive P2G and G2P. Although rendering is kept as minimal as possible as to not impact benchmarks beyond interoperability costs, the SoA data structure also improves OpenGL performance: total frame-time drops to 2,5 ms (400 FPS).

SoA | 100x100x100 grid | 655360 particles

Kernel	Time (ms)
grid_reset	0,06
p2g_init	0,923
particles_estimate_density	0,266
p2g	2,692
grid_update	0,023
g2p	6,975
particles_advect	0,095
Total	**11,034**

3.5 Reducing memory accesses

After more testing and kernel analysis in Nsight Compute, it is clear that performance is mostly limited by memory operations rather than compute. Kernels operating on Particles access neighboring Cells data depending on their position, which is effectively randomized by the simulation, resulting in uncoalesced Global Memory accesses. This lack of any pattern in Cells data access also impedes the use of Shared Memory. The only applicable amortization is to reduce Global Memory operations as much as possible, and to rely on L1 and L2 cache hits.

Kernels are fused where possible: `particles_estimate_density` is moved to the beginning of `p2g`, and `particles_advect` to the end of `g2p`. This reduces the number of data written between kernels, and allows to remove Particle density as a stored value altogether, as it becomes a temporary variable in `p2g`. However, no further kernel fusions are possible, because of the need of grid-wise synchronization barriers between the MPM phases.

Each kernel is also refactored to store Global Memory values to local registers and delay any write until all operations are completed. This increases the already high register usage, but reduces costly memory operations.

3.5.1 Benchmark

This optimization has expectedly no effect on kernels operating on Cells, where no further memory is accessed beyond the Cell mapped to the thread. However both `p2g` and `g2p`, despite now being more computation heavy, show greatly increased performance. Total frame-time is now 0,96 ms (1044 FPS).

Reduced memory access | 100x100x100 grid | 655360 particles

Kernel	**Time (ms)**
grid_reset	0,066
p2g_init	1,036
p2g	2,678
grid_update	0,024
g2p	0,627
Total	**4,431**

3.6 Grid-stride loop

To further reduce the number of kernel launched, a grid-stride loop can be implemented: instead of the CUDA GridDim covering all data, a fixed size can be set and threads can loop and stride over data until coverage.

```
__global__ void p2g_init(
    glm::aligned_vec3* const particles_positions,
    const unsigned int particles_count,
    const ParticleMaterial particles_material,
    float* const cells_masses,
    const glm::uvec3 grid_size)
{
   for (unsigned int particle_idx = threadIdx.x + blockIdx.x * blockDim.x;
         particle_idx < particles_count;
         particle_idx += blockDim.x * gridDim.x)
    {
        // Kernel operations...
    }
}
```

3.6.1 Benchmark

Upon testing, performance is shown to be worse for Cells kernels, and so grid-stride loops are not implemented. However, for Particles kernels, performance increases slightly: with GridDim equal to the number of Streaming Multiprocessors (polled at program start) occupancy drops from 74% to 8,4%, but compute throughput and L1 cache hits increase. A larger GridDim shows decreased performance.

Total frame-time is reduced to 0,91 ms (1103 FPS). This improvement, while small, proved consistent throughout multiple tests.

Grid-stride loop (Particles only) | 100x100x100 grid | 655360 particles

Kernel	Time (ms)
grid_reset	0,06
p2g_init	0,916
p2g	2,685
grid_update	0,025
g2p	0,738
Total	**4,424**

3.7 Loop unrolling

Given the repeated looping over a fixed number of neighboring Cells in P2G and G2P, loop unrolling can be expected to have an effect. The compiler can be instructed to unroll a loop with the `#pragma unroll` directive without much change to the code.

3.7.1 Benchmark

Tests show loop unrolling to increase registers usage in `p2g` from 56 to 72 per thread, and results prove inconsistent when paired with the grid-stride loop optimization. However, keeping loop unrolling but removing grid-stride loops yields overall better performance: frame-time is 0,79 ms (1259 FPS).

Loop unrolling | 100x100x100 grid | 655360 particles

Kernel	Time (ms)
grid_reset	0,06
p2g_init	0,949
p2g	2,647
grid_update	0,025
g2p	0,598
Total	**4,279**

Chapter 4

Results

The MPM algorithm is well suited for parallel computing for its operations per Cell and Particle, and a GPU implementation leads to drastic improvements in performance compared to the single-thread version.

The simulation is heavily limited by uncoalesced memory operations, inevitable without a radical restructuring of the underlying algorithm and data structures. Still, proper memory management and a reduction in Global Memory accesses lead to notable improvements.

100x100x100 grid | 655360 particles

	Naive	Interop.	Alignment	SoA	Red. mem.	Grid-stride	Unrolling
grid_reset	0,048	0,048	0,048	0,06	0,066	0,06	0,06
p2g_init	1,23	0,924	0,997	0,923	1,036	0,916	0,949
estimate_density	0,527	0,529	0,525	0,266	n.a.	n.a.	n.a.
p2g	3,711	3,758	3,763	2,692	2,678	2,685	2,647
grid_update	0,054	0,054	0,054	0,023	0,024	0,025	0,025
g2p	14,129	14,356	8,54	6,975	0,627	0,738	0,598
advect	0,136	0,136	0,133	0,095	n.a.	n.a.	n.a.
cudaMemCpy	3,513	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
glBufferData	5,1	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
Total (ms)	**28,448**	**19,805**	**14,06**	**11,034**	**4,431**	**4,424**	**4,279**

The CPU implementation cannot achieve the target real-time performance of 60 FPS even in the test scenario with reduced Particles count. On the testing machine, the maximum number of Particles able maintain 60 FPS is around 27k.

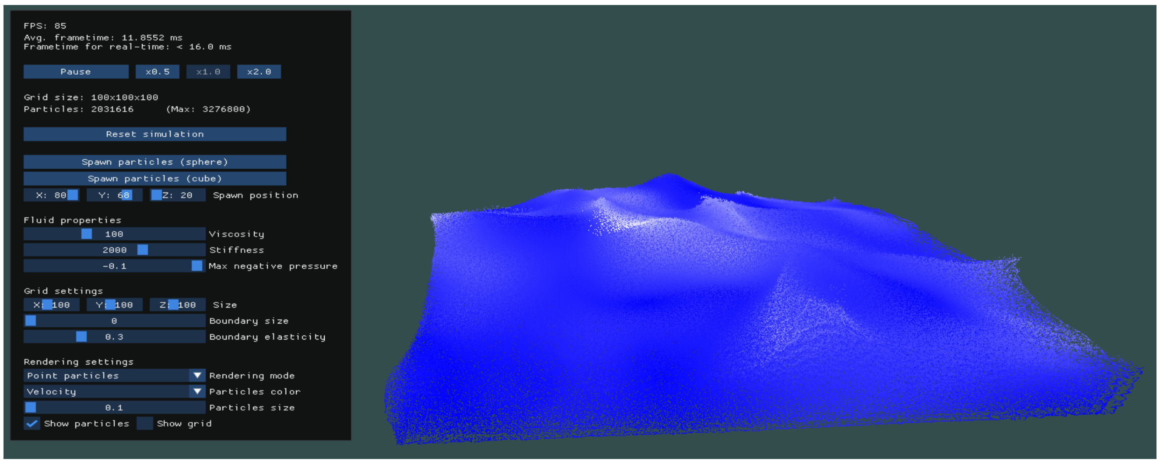
Even a naive GPU implementation, with unnecessary CPU data mediation between CUDA and OpenGL, yields greatly improved results in the same test scenario. With a 40x improvement over the CPU version, this implementation is well beyond capable of running the first test in real-time.

The final, optimized parallel implementation correctly utilizes CUDA-OpenGL interoperability and memory-aligned data, and reduces the costly Global Memory accesses. In the first test scenario the simulation is no longer a bottleneck for rendering: baseline (no simulation) program frame-time is 0,68 ms (1469 FPS), and 0,72 ms (1388 FPS) with 65536 particles. While it is only a 1.27x (51x CPU) performance improvement over the naive implementation, scaling the simulation to 10 times the particles count shows a 45x improvement.

100x100x100 grid | 65536 particles

	Single-thread	Naive CUDA	Final CUDA
grid_reset	0,569	0,048	0,06
p2g_init	8,271	0,106	0,102
estimate_density	n.a.	0,052	n.a.
p2g	10,391	0,4	0,298
grid_update	0,97	0,053	0,02
g2p	14,935	1,55	0,068
advect	n.a.	0,017	n.a.
cudaMemCpy	n.a.	1,013	n.a.
glBufferData	n.a.	0,9	n.a.
Total (ms)	**35,936**	**4,139**	**0,548**
Frame-time	**37,26 ms (27 FPS)**	**0,91 ms (1094 FPS)**	**0,72 ms (1388 FPS)**

The final implementation of the simulation can maintain the real-time target of 60 FPS in a 100x100x100 grid with 2 million particles.


```
FPS: 85
Avg. frametime: 11.8552 ms
Frametime for real-time: < 16.0 ms

Grid size: 100x100x100
Particles: 2031616     (Max: 3276800)

X: 80   Y: 68   Z: 20   Spawn position

Fluid properties
100     Viscosity
2000    Stiffness
-0.1    Max negative pressure

Grid settings
X: 100  Y: 100  Z: 100  Size
0       Boundary size
0.3     Boundary elasticity

Rendering settings
Point particles   Rendering mode
Velocity          Particles color
0.1               Particles size
[✓] Show particles   [ ] Show grid
```

Bibliography

[1] Sulsky D.; Chen Z.; Schreyer H. L. A particle method for history-dependent materials. `https://digital.library.unt.edu/ark:/67531/metadc1385575/`, 1993.

[2] Yuanming Hu, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Transactions on Graphics*, 37(4):150, 2018.