Thesis no: XXXX-20XX-XX

Accelerating IISPH

A parallel GPGPU solution using CUDA

André Eliasson & Pontus Franzén

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Game and Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s):
André Eliasson
E-mail: aael10@student.bth.se
Pontus Franzén
E-mail: pontus.franzen@live.se

University advisor:
Prof. Prashant Goswami
Department of Creative Technologies

Faculty of Computing	Internet	:	www.bth.se
Blekinge Institute of Technology	Phone	:	+46 455 38 50 00
SE–371 79 Karlskrona, Sweden	Fax	:	+46 455 38 50 57

Abstract

Context. Simulating realistic fluid behavior in incompressible fluids for computer graphics has been pioneered with the implicit incompressible smoothed particle hydrodynamics (IISPH) solver. The algorithm converges faster than other incompressible SPH-solvers, but real-time performance (in the perspective of video games, 30 frames per second) is still an issue when the particle count increases.
Objectives. This thesis aims at improving the performance of the IISPH-solver by proposing a parallel solution that runs on the GPU using CUDA. The solution should not compromise the physical accuracy of the original solution. Investigated aspects are execution time, memory usage and physical accuracy.
Methods. The proposed implementation uses a fine-grained approach where each particle is calculated on a separate thread. It is compared to a sequential and a parallel OpenMP implementation running on the CPU.
Results and Conclusions. It is shown that the parallel CUDA solution allow for real-time performance for approximately 19 times the amount of particles than that of the sequential implementation. For approximately 175 000 particles the simulation runs at the constraint of real-time performance, more particles are still considered interactive. The visual result of the proposed implementation deviated slightly from the ones on the CPU.

Keywords: implicit incompressible smoothed particle hydrodynamics, fluid simulation, real-time, GPGPU

Preface

Acknowledgment

We would like to thank Prashant Goswami for the proposed research area and his invaluable knowledge and help.

We would also like thank fellow student Mattias Frid Kastrati for giving feedback and proof-reading our thesis.

Finally we would like to thank our peers who gave us valuable feedback with their oppositions.

About abbreviations

Throughout this thesis certain terms are abbreviated to ease reading. At the first occurrence they are written in full (except from units, such as MB). They are also listed below.

AoS – Array of Structs
API – Application Programming Interface

EOS – Equation Of State

FPS – Frames Per Second

GPGPU – General-Purpose computing on Graphics Processing Unit

ISPH – Incompressible Smoothed Particle Hydrodynamics
IISPH – Implicit Incompressible Smoothed Particle Hydrodynamics

PCISPH – Predictive-Corrective Incompressible Smoothed Particle Hydrodynamics
PPE – Pressure Poisson Equation

PSNR – Peak Signal-to-Noise Ratio

SPH – Smoothed Particle Hydrodynamics
SoA – Structure of Arrays
SSIM – Structured Similarity Index Measure

VRAM – Video Random Access Memory

WCSPH – Weakly Compressible Smoothed Particle Hydrodynamics

List of Figures

List of Tables

List of Algorithms

Contents

Chapter 1

Introduction

Fluids are common elements included in the environments of video game worlds. The behavior of them can be achieved with various techniques, such as animating a height-map. This technique is limited to only simulate the surface of the fluid however and require additional methods for water splashes and sub-surface simulation. To simulate the entire body of a fluid, alternative methods such as smoothed particle hydrodynamics (SPH) [1] can be used.

In video games, real-time performance is assumed to range between 30 to 60 or more frames per second (FPS) to ensure the player an acceptable experience [2]. This infers that the time between two consecutive frames may not exceed at the most 33 ms; which includes the time to compute the scene of the frame and to render it on screen.

Implicit incompressible SPH (IISPH) is a state-of-art SPH-solver which allows for realistic fluid behaviors of incompressible fluids (such as water) [3]. In order for IISPH to be suitable for simulating fluids in games, it would need to meet the requirement for real-time performance. Other limiting factors would be interaction with game objects and computations to be completed within a given time-frame together with other logic, however this is out of the scope of this thesis.

With the increased use of general-purpose computing on GPU (GPGPU) for massive parallel data computation it is a logical strategy to implement IISPH to be run on the GPU to overcome the real-time performance barrier.

1.1 SPH based methods

In this section a brief description on how realistic fluid behavior can be achieved for computer graphics using particles and how it has been improved in terms of performance and compressibility.

1.1.1 Smoothed Particle Hydrodynamics

Realistic fluid simulations can be achieved with the SPH method where the fluid's body is represented as a set of particles. SPH uses various external forces, such as gravity and viscosity, combined with internal pressure forces to calculate the velocities of the particles [1].

The pressure is computed with an equation of state (EOS). An EOS is an equation describing the physical state of a system according to a set of variables known as state variables which have an internal relationship with each other, such as mass, density and pressure. However, this pressure computation also results in forces which penalize compression [3].

SPH allows for easy boundary handling and mass conservation which can accomplish realistic movement and behavior, and effects such as water splashes. SPH is not without its limitations though. Restrictions such as a non-real-time performance [4] and an abundance of efficient incompressible SPH-solvers [3, 4, 5, 6] render the method unsuitable in application areas such as video games.

1.1.2 Weakly Compressible SPH

Much work has been done to improve SPH in terms of visual quality and performance. Weakly compressible SPH (WCSPH) [7] utilizes a stiff EOS in order to address compressibility. Density fluctuations are kept low by assuming a high speed of sound in the medium. Though the computations of SPH and WCSPH are inexpensive per frame, compared to alternative approaches [3]; a higher stiffness requires smaller time-steps (time simulated between two frames) in the physics simulation, which results in that the total number of frames required for a given time-frame increases when compressibility decreases.

1.1.3 Incompressible SPH

It was suggested that by solving a pressure projection equation similar to the Eulerian methods [8], rather than the EOS in SPH and WCSPH, an enforcement of incompressibility could be achieved. Then by projecting either the intermediate velocity field [9] or the variation of particles' density [10] onto a divergence-free space, incompressibility can be enforced through a pressure Poisson equation (PPE). Incompressible SPH (ISPH) allows for use of larger time-steps but at an expense of additional computation cost per time-step (frame).

1.1.4 Predictive-Corrective Incompressible SPH

Further improvements suggest predictor-corrective schemes to combine the advantage of large time-steps from the ISPH method with WCSPH's low computation cost per time-step. The predictive-corrective incompressible SPH (PCISPH) [6] method is such an approach which utilizes a predictor-corrective scheme of the EOS. The velocity of the particle is predicted based on previous velocity and then corrects it based on the new pressure. It is an iterative process that repeats its process for as long as the density error is above a specified user-defined threshold (e.g. if the error margin is within 0.1 % deviation from the rest density). PCISPH outperforms WCSPH in terms of performance in regard to incompressibility with as much as a factor of 55 while approaching incompressibility as well.

1.1.5 Local Poisson SPH

An alternative approach to combine the advantages of WCSPH and ISPH was proposed by He *et al.* [11] with Local Poisson SPH. The PPE is solved by converting the differential PPE to a continuous integral. A discretization method is used to convert this form to a discretized summation in the local pressure integration domain for all particles. The pressure solver is then integrated into a predictive-corrective framework. The result was improvements in both density error and convergence rate of the solver, rivaling the performance of PCISPH.

1.1.6 Implicit Incompressible SPH

Another discretization of the PPE was proposed by Ihmsen *et al.* [3] with IISPH, which is an improvement of the PCISPH-method. An issue with traditional ISPH-solvers is that they do not scale well with large-scale scenarios comprising of millions of particles. IISPH gained a significant speed-up compared to ISPH and PCISPH as a result of a discretization of the PPE, reportedly a speed-up factor of 6.2 and 5.2 times compared to PCISPH for two different scenarios. This resulted in a performance that scaled better with large-scale scenarios while further enforcing incompressibility.

Any further details of the algorithm are being referred to chapter 4.

1.2 Problem

Incompressible SPH-solvers addressed the issue with incompressibility in the fluid. It was further improved with the implementation of the state-of-the-art incompressible SPH-solver IISPH [3]. However, because of it being a sequential implementation running on the CPU, there is still a gap in performance. IISPH cannot by itself be considered a feasible solution for fluid simulations in real-time applications. As a consequence this severely limits the utility of IISPH in certain fields where real-time performance are considered important e.g. video games and virtual reality.

1.3 GPGPU and CUDA

The basics of GPGPU is to utilize the graphics card's processing units to compute general tasks. The three major application programming interfaces (APIs) are NVIDIA® CUDA® [1] [21], Microsoft® Direct Compute and the Apple® [2] and Khronos™ Group OpenCL™ [3]. These APIs allow programmers to utilize the GPU's computational power in areas where massive computations can be executed in parallel. This has resulted in increased performance of computationally heavy applications, such as in analyzing air traffic flows and visualizations of molecules [21].

CUDA was the chosen programming platform because of its computational power, its interoperability with Microsoft® DirectX® [4] rendering API and because of the project outline preferring CUDA. The supporting development languages are C, C++ and Fortran.

CUDA Toolkit 7.0 was used during the implementation and is distributed with supplementary libraries. Thrust [22] is one of those libraries and was originally created by Hoberock and Bell. Thrust is a parallel algorithms library with CUDA interoperability providing parallel implementations of basic algorithms such as sort, transform and reduce.

1.4 Thesis Outline

So far a brief introduction to fluid simulations with various SPH-solvers has been given as well as the problem and a short introduction of CUDA. In chapter 2 a more explanatory description of the problem and the research questions this thesis aims to answer are given. This is followed in chapter 3 by other related work that has aimed at similar areas as this thesis. The IISPH-algorithm is described in detail in chapter 4 and in chapter 5 the implementations of it are covered. Chapter 6 describes the methodology of the experiments with the results summarized in chapter 7. This is followed by a discussion of what was achieved in chapter 8 and finally a short conclusion and future work in chapter 9.

[1] NVIDIA and CUDA are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and/or other countries.

[2] Apple is a trademark of Apple Inc., registered in the U.S. and other countries.

[3] OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.

[4] Microsoft and DirectX are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Chapter 2

Aims, Objectives and Research Question

This thesis aims to improve the performance of the IISPH-algorithm by proposing a parallel implementation which can run on the GPU. The proposed algorithm was implemented and experimented on using CUDA and comparisons was made to a sequential and parallel CPU-implementation.

2.1 Objectives

To achieve this goal a set of sub-tasks was created to track the progress; of which some were required to be completed before others, whereas some could be done in parallel:

- Get accustomed to either the original or a reconstructed implementation of the proposed algorithm in [3]. Two reasons existed for this task:
  - With the actual code available, a detailed study of the algorithm's nature could be acquired which alleviated the workload and understanding of it when proposing the parallel one.
  - To evaluate the performance of the proposed parallel CUDA-implementation it was necessary to gather data and compare test results with a sequential as well as a parallel CPU-implementation in order to make any conclusions regarding the proposed CUDA-implementation.
- With the in-depth study, areas which could be beneficial of running in parallel was identified. Suitable methods of parallelization were proposed for one and each of the parts of the algorithm. If any part had been required to run sequentially it would had been of importance to identify it and decide whether it would cause problems or if it could have been solved with an alternative approach.
- Using the parallel proposal and the knowledge from the sequential CPU-code the CUDA-version was implemented.

- To see a visual representation of the simulations, rendering the particles was required. In the CPU-version the necessary particle data was sent to the GPU for rendering each frame. In the CUDA-version it was desirable to not have to copy this data from the GPU to the CPU and back again; thus connecting CUDA to DirectX to allow it to modify the buffers was desired.
- Creating appropriate scenarios to be run when measuring performance. Aspects such as the number of particles and the layout of test scene itself could be affecting the result.
- Suitable measurements were needed to evaluate two aspects in experiments using the scenarios. The performance in terms of execution time and the visual quality in terms of identical visual outcome were necessary to be examined.
- An analysis of the gathered data was made to evaluate whether a positive result had been achieved or if further improvements were necessary to achieve a sufficient result.

2.2 Research questions

This thesis aims to answer the following research questions:

RQ1. Is it possible to achieve real-time performance of IISPH using CUDA?

1. What sections of the algorithm are the most time consuming?
2. For how many particles can the proposed solution be considered real-time?

RQ2. Will a parallel implementation with CUDA achieve better performance than that of a parallel version on the CPU?

RQ3. Can IISPH gain increased performance utilizing a parallel implementation with CUDA but without compromising the physical accuracy?

Hypothesis:

- It is believed that the memory usage between the implementations will be similar.

Chapter 3

Related Work

Various improvements have been proposed for CPU and GPU oriented approaches of SPH [12]. While some of these have already been covered in section 1.1, this chapter will focus on related parallelization and optimizations of SPH- and ISPH-solvers.

In this thesis the approach of parallelizing IISPH was by using a fine-grained (data parallelism) approach on the GPU. Each particle is run in parallel and they are themselves responsible for calculating their own values. An alternative coarse-grained (task parallelism) approach on the GPU is presented by Goswami *et al.* [13] which utilizes groups of particles using shared memory and CUDA. By dividing the particles into small groups and utilizing shared memory a significant performance increase of SPH was gained.

A recent study by Thaler *et al.* [14] shows a parallel multi-core implementation of IISPH with focus on a fine-grained approach with promising performance results, however without any information about how the visual results compare to a sequential implementation. In this thesis a similar algorithm structure is proposed; with variables being calculated in sections with synchronization in-between when necessary. In their multi-core solution it is necessary to communicate and balance the load efficiently between cores to fully utilize the cores capabilities.

Another shared memory approach with CUDA for PCISPH is presented by Nie *et al.* [15]. A good speed-up was achieved compared to both sequential and multi-core CPU-implementations. They also found that the sorting method for neighbor search to be a bottleneck and provide with a new parallel sorting method to increase performance. The implementation utilizes a structure of arrays (SoA) pattern for particle data due to better coalesced memory access which is crucial to maximize the bandwidth the GPU supports. Using a SoA pattern means that each variable is a separate array stored in a consecutive section of memory.

Chapter 4

IISPH

In this chapter an overview of the IISPH-algorithm will be given and details that may not be obvious will be addressed as well. Any details on the derivation of the algorithm are considered out-of-scope of the thesis and are being referred to the paper by Ihmsen *et al.* [3]. All equations and details in this chapter are cited from the same paper unless stated otherwise.

Algorithm 1: IISPH-algorithm proposed by Ihmsen *et al.* [3].

Procedure *ADVECTION PREDICTION*
- **foreach** *particle* i **do**
  - `compute` $\rho_i(t) = \sum_j m_j W_{ij}(t)$
  - `predict` $\mathbf{v}_i^{adv} = \mathbf{v}_i(t) + \Delta t \frac{\mathbf{F}_i^{adv}(t)}{m_i}$
  - $\mathbf{d}_{ii} = \Delta t^2 \sum_j -\frac{m_j}{\rho_i^2} \nabla W_{ij}(t)$
- **foreach** *particle* i **do**
  - $\rho_i^{adv} = \rho_i(t) + \Delta t \sum_j m_j (\mathbf{v}_{ij}^{adv}) \nabla W_{ij}(t)$
  - $p_i^0 = 0.5 p_i(t - \Delta t)$
  - `compute` a_{ii}(4.5)

Procedure *PRESSURE SOLVE*
- $l = 0$
- **while** $\rho_{avg}^l - \rho_0 > \eta \vee l < 2$ **do**
  - **foreach** *particle* i **do**
    - $\sum_j \mathbf{d}_{ij} p_j^l = \Delta t^2 \sum_j -\frac{m_j}{\rho_j^2(t)} \nabla W_{ij}(t)$
  - **foreach** *particle* i **do**
    - `compute` p_i^{l+1}(4.8)
    - $p_i(t) = p_i^{l+1}$
  - $l = l + 1$

Procedure *INTEGRATION*
- **foreach** *particle* i **do**
  - $\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i^{adv} + \Delta t \frac{\mathbf{F}_i^p(t)}{m_i}$
  - $\mathbf{x}_i(t + \Delta t) = \mathbf{x}_i(t) + \Delta t \mathbf{v}_i(t + \Delta t)$

4.1 Terminology

An overview of the algorithm is presented as pseudo code by Ihmsen *et al.* [3] in Algorithm 1. It consists of three parts labeled as procedure, viz. advection prediction, pressure solve and integration.

Bold variables indicate three dimensional vectors whereas italic variables are considered as scalar values.

The W's are kernel functions used to sample the influence a particle has on a neighboring particle dependent of the distance between the two.

A specific particle in the fluid is identified with an index denoted with the subscript of i. The j denotes one of the neighbors of i and k denotes the neighbors of j. It is worth to note that in the set of neighbors k the particle with index i is included, since if j is a neighbor to i it follows that i is a neighbor of j.

The nabla symbol ∇ indicates the del operator that operates in the scalar field of W to produce a non-normalized direction vector between the particles. In the cases where the results that are to be computed are scalar values the dot product is used which will be mentioned in the subsequent sections.

In the computations a global support radius is needed for various tasks. An illustration of it can be seen in Figure 4.1. It is computed as $n * 2r$, where r is the radius of the particle and $n \in \mathbb{N}$ defining how many particle widths from the center point of the particle neighbors are considered to influence it.

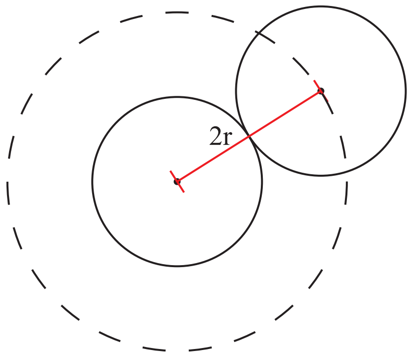

Figure 4.1: *A particle (center circle) with its particle spacing (dashed circle) being equal to twice its radius as illustrated by the red line. The particle to the right has its center point within the particle spacing and is thus considered to be a neighbor.*

4.2 Kernel functions

Three sets of kernels was provided by Prashant Goswami and are used when simulating IISPH: W_{poly6}, W_{spiky} and $W_{\text{viscosity}}$, see Figure 4.2. W_{poly6} is used when calculating density, W_{spiky} is used in pressure related computations and $W_{\text{viscosity}}$

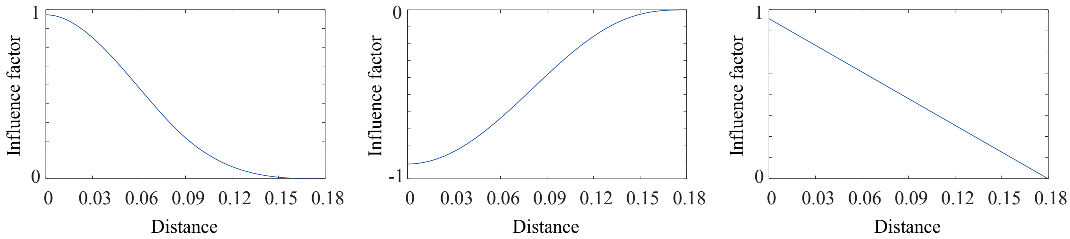

Figure 4.2: *The three kernels visualized with a global support radius of 0.18 m. Left: W_{poly6}. Middle: W_{spiky}. Right: $W_{viscosity}$. In all kernels the influence is decreasing when nearing the support radius. Full influence in the graphs is visualized with 1 or -1 and no influence as 0.*

is used in the particles' viscosity equation. Each kernel is used to determine the influence that a particle provides to a neighbor based on the distance to it; closer particles impart higher influence whereas those beyond the support radius provide none.

Each kernel returns zero if either the provided distance d between two particles is zero (special case) or if the distance is larger than a provided global support radius R (not to be confused with the particles radius which was used to compute the support radius). This effectively limits the number of particles which influence a specific particle.

$$W_{poly6}(d,R) = \begin{cases} 0 & \text{if } d < 0 \text{ or } d \geq R \\ \frac{8(1-6s^2+6s^3)}{\pi R^3} & \text{if } \frac{d}{R} < 0.5 \\ \frac{16(1-s)^3}{\pi R^3} & \text{if } \frac{d}{R} \geq 0.5 \end{cases} \tag{4.1}$$

The W_{poly6} is constructed as a cubic spline [1] according to (4.1), where $s = \frac{d}{R}$. Although it is possible to use alternative kernels, a study by Fulk and Quinn [16] have found that no other kernels give a significantly better result for density computations than the cubic spline.

$$W_{spiky}(d,R) = \begin{cases} 0 & \text{if } d < 0 \text{ or } d \geq R \\ \frac{-8*3465}{512\pi R^5} * (\frac{1-d^2}{R^2})^3 \end{cases} \tag{4.2}$$

The function for W_{spiky} (4.2) follows an ordinary $cos(x)$ pattern. The major difference is that it only returns negative values instead of in the range of $[-1:1]$.

$$W_{viscosity}(d,R) = \begin{cases} 0 & \text{if } d < 0 \text{ or } d \geq R \\ \frac{45(R-d)}{\pi R^6} \end{cases} \tag{4.3}$$

$\text{W}_{\text{viscosity}}$ returns an influence value according to a linear function (4.3).

4.3 Advection prediction

Advection is the conserved movement of a property in a fluid due to its bulk motion. When a fluid is flowing the internal properties such as regional density and pressure fluctuations will follow. As an example, imagine a river of water flowing in one direction. When dripping some color into the water it too will follow downstream rather than instantly disperse in all directions. The first component in IISPH is to predict this physical behavior for each particle.

In order to do this, iterating each particle in two passes is required. In the first pass (lines 3-5 Algorithm 1) density ρ_i, intermediate velocity $\mathbf{v}_i^{adv}$ and displacement factor $\mathbf{d}_{ii}$ is calculated. During the second pass (lines 7-9 Algorithm 1) intermediate density ρ_i^{adv}, initial pressure value p_i^0 and advection factor a_{ii} is computed.

In the first pass of advection prediction the density of a particle ρ_i (line 3) can be calculated as the summation of the products from each of its neighboring particle's mass m_j and a finite support kernel function W_{ij} which uses the distance between particles i and j as an input parameter.

Then an intermediate velocity $\mathbf{v}_i^{adv}$ (line 4) can be predicted as the particle's current velocity $\mathbf{v}_i$ and by taking Newton's second law of motion into account for all external non-pressure forces $\mathbf{F}_i^{adv}$ (such as gravity and viscosity). This is done by combining all those forces into a single net force, divide it with the particle's mass m_i and multiply with the time-step Δt, resulting in a velocity caused by non-pressure forces which is added to the current velocity.

$$\mathbf{F}_i^{visc} = -\sum_j (\mathbf{v}_i - \mathbf{v}_j)(\frac{m_j}{\rho_i})^2 Const_{visc} W_{ij} \tag{4.4}$$

The viscosity force for a particle i is calculated according to Equation 4.4 where $Const_{visc}$ is computed as $2 * sizeFactor$ and the kernel used is $\mathrm{W}_{\mathrm{viscosity}}$.

Displacement is a quantity that measures a particle's distance of movement from its equilibrium position in the fluid as the fluid is transmitting a wave. The last part of the first pass is to compute each particle's displacement factor $\mathbf{d}_{ii}$ (line 5) which is repeatedly used in the following equations. It is the square of the time-step Δt^2 multiplied with the summation of the negative value of each neighbor's mass m_j divided by the particle's calculated density squared ρ_i^2 multiplied with $\mathrm{W}_{\mathrm{spiky}}$ which also uses the distance between i and j as an input parameter.

In the second pass of advection prediction the first variable to be calculated is the intermediate density ρ_i^{adv} (line 7). This incorporates the contribution of advection created through pressure forces. It is the sum of the current density ρ_i and the multiplication of the time-step Δt and a summation of multiplications iterating for each neighbor. The multiplications are between the mass of the neighbor m_j, the dot product of the intermediate velocity vector $\mathbf{v}_{ij}^{adv}$ between i and j and the non-normalized distance vector and finally the $\mathrm{W}_{\mathrm{spiky}}$ previously used with the distance between i and j as input parameter.

The initial pressure value p_i^0 (line 8) is set to 0.5 of the previous frame's pressure value p_i in order to receive optimal convergence. This was empirically proven by Ihmsen *et al.* [3]. In the first frame it is set to 0.

$$a_{ii} = \sum_j m_j(\mathbf{d}_{ii} - \mathbf{d}_{ji})\nabla W_{ij} \tag{4.5}$$

The advection factor a_{ii} which will be needed in the pressure solver is finally computed according to (4.5) where m_j is the neighbor's mass. The displacement factor $\mathbf{d}_{ii}$ is already computed and $\mathbf{d}_{ji}$ can be computed according to (4.6) which is similar to how $\mathbf{d}_{ii}$ was calculated, only that it is not a summation over all neighbors, the mass m_i is the particle's mass and the non-normalized vector to multiply with $\mathrm{W}_{\mathrm{spiky}}$ is spanning from j to i. A dot product is used (4.5) between the difference vector from the displacement factors and the non-normalized distance vector of j and i.

$$\mathbf{d}_{ji} = -\Delta t^2 \frac{m_j}{\rho_j^2} \nabla W_{ij} \tag{4.6}$$

4.4 Pressure solve

The second component of the algorithm is to solve the pressure equation. Similarly to advection prediction it requires to iterate each particle in two passes. In the first pass (line 14 Algorithm 1) the total movement due to pressure forces from neighboring particles $\sum_j \mathbf{d}_{ij} p_j^l$ are computed. The second pass (lines 16-17 Algorithm 1) computes the pressure p_i^{l+1} for the next frame and assigns it. The pressure equation is solved employing relaxed Jacobi, which is an iterative process for approximating a solution to an equation [17]. This makes the nature of the solver iterative, meaning the two passes are iterated in a loop as it converges towards the pressure value. The average density of the next frame can be predicted according to (4.7). Due to this the iterative process can be controlled by terminating when the error in density is less than a threshold η. That is when the difference of the average predicted density ρ_l^{avg} and the fluid's rest density ρ_0 (the average density value of the fluid when perfectly still) is smaller than the threshold value η.

$$\begin{aligned}\rho_i^{l+1} = \rho_i^{adv} + p_i \sum_j m_j(\mathbf{d}_{ii} - \mathbf{d}_{ji})\nabla W_{ij} + \\ \sum_j m_j\Big(\sum_j \mathbf{d}_{ij} p_j - \mathbf{d}_{jj} p_j - \sum_{k \neq i} \mathbf{d}_{jk} p_k\Big)\nabla W_{ij}\end{aligned} \tag{4.7}$$

The sum of movement caused by the pressure of neighboring particles is computed similarly to the displacement factor in advection prediction. It differs in that instead of dividing the neighbor's mass with the particle's density, it is divided by the neighbor's density and that the resulting quotient is multiplied with the neighbor's pressure p_j^l for this iteration.

$$p_i^{l+1} = (1-\omega)p_i^l + \omega \frac{1}{a_{ii}} \Big(\rho_0 - \rho_i^{adv} - \\ \sum_j m_j \Big(\sum_j \mathbf{d}_{ij} p_j^l - \mathbf{d}_{jj} p_j - \sum_{k \neq i} \mathbf{d}_{jk} p_k^l \Big) \nabla W_{ij} \Big) \tag{4.8}$$

Computing the pressure is achieved according to (4.8). The calculation utilizes the current iteration's pressure values p_i^l in order to get the next iteration's values p_i^{l+1}. The first term (1-ω)p_i^l is altering the pressure using a relaxation factor ω. Optimal convergence is achieved by setting it to the value of 0.5. The second term is calculated as the relaxation factor ω divided with the advection factor a_{ii} and multiplied with the deviation from the rest density, computed as ρ_0 minus the intermediate density ρ_i^{adv} and (4.9).

$$\sum_j m_j \Big(\sum_j \mathbf{d}_{ij} p_j^l - \mathbf{d}_{jj} p_j^l - \sum_{k \neq i} \mathbf{d}_{jk} p_k^l \Big) \nabla W_{ij} \tag{4.9}$$

The summation of (4.9) should only add pressure values caused by each neighboring particle. All terms to this have already been computed with one exception. The movement caused by neighbors' pressure forces $\sum_j \mathbf{d}_{ij} p_j^l$ was computed in the previous pass. The product $\mathbf{d}_{jj} p_j^l$ is the product of the neighbor's displacement factor and pressure, both of which are calculated and available to use. The $\sum_{k \neq i} \mathbf{d}_{ij} p_j^l$ is the same as (4.10) which is partially computed.

$$\sum_{k \neq i} \mathbf{d}_{jk} p_k^l = \sum_k \mathbf{d}_{jk} p_k^l - \mathbf{d}_{ji} p_i^l \tag{4.10}$$

The first term in (4.10) is the neighbor's movement caused by pressure forces from its neighbors and too has already been computed in the previous pass. The second term however is the movement of the neighbor caused by the particle's pressure (particle i of which the pressure is being calculated), thus it should be excluded since only pressure values from neighbors to particle i should be considered.

$$\mathbf{d}_{ji} p_i = -\Delta t^2 \big(\frac{m_i}{\rho_i^2(t)} p_i^l \nabla W_{ji}(t) \big) \tag{4.11}$$

This is calculated according to (4.11), where m_i is the particle's mass, ρ_i^2 is the particle's density squared and p_i^l is the particles pressure in the current iteration. The vector to be multiplied with $\mathrm{W}_{\mathrm{spiky}}$ is spanning between j and i. Once the movement caused by the particle itself is excluded the result is multiplied as according to (4.9) with the neighbor's mass m_j, the dot product of the result and the vector between the particle i and neighbor j and $\mathrm{W}_{\text{spiky}}$ function which is using the distance between i and j as parameter. Finally the computed pressure is assigned to the particle, if the value would be negative it is clamped to zero due to negative pressure values not being allowed.

In order to control the compression the density error needs to be calculated as well. As mentioned above it can be solved according to (4.7). Most values are already computed, the intermediate density ρ_i^{adv} was used before when computing the pressure p_i. The second term's summation for each neighbor uses the difference of the displacement factor $\mathbf{d}_{ii}$ (already available) and the neighbors' displacements due to the particle itself $\mathbf{d}_{ji}$ (can be calculated again (4.11) excluding the pressure p_i^l). $\mathrm{W}_{\text{spiky}}$ again utilizes the distance between i and j and the support radius as input. The third term is the exactly the same as (4.9).

4.5 Integration

Once the pressure is computed it is possible to update the particles with correct velocities $\mathbf{v}_i(t+\Delta t)$ and positions $\mathbf{x}_i(t+\Delta t)$ achieved in the integration procedure (lines 21-22 Algorithm 1).

The velocity $\mathbf{v}_i(t+\Delta t)$ is calculated as the intermediate velocity $\mathbf{v}_i^{adv}$ which were calculated in advection prediction and by adding the velocity caused by the pressure force $\mathbf{F}_i^p(t)$ according to Newton's second law of motion. The pressure force $\mathbf{F}_i^p(t)$ should preserve momentum and is obtained according to (4.12) [1, 3].

$$\mathbf{F}_i^p(t) = -m_i \sum_j m_j \Big(\frac{p_i(t)}{\rho_i^2(t)} + \frac{p_j(t)}{\rho_j^2(t)} \Big) \nabla W_{ij}(t) \tag{4.12}$$

The position $\mathbf{x}_i(t+\Delta t)$ is then updated by adding the product of the time-step Δt and the velocity $\mathbf{v}_i(t+\Delta t)$ to the current position $\mathbf{x}_i(t)$.

Chapter 5

Implementation

Three versions of the algorithm were implemented. The first one was a sequential version running on the CPU. The second one was the proposed parallel version running on the GPU. A parallel implementation of the CPU-version was also developed using OpenMP. All three versions are based on the proposed IISPH-algorithm by Ihmsen *et al.* [3], see Algorithm 1, implemented in C++.

DirectX 11.0 is used for rendering, billboards are used to represent each particle. Billboards were used because it made it possible to render more particles due to the small amount of triangles compared to a spherical object. DirectX 11.0 was chosen due to previous experience and it allowed for reusing an already existing graphics engine and basic program structure.

Microsoft Visual Studio® 2013 [1] was used as developing environment because of previous experience have been exclusively utilizing it. Additional software used was the NVIDIA Nsight™ Visual Studio Edition [2] in order to assist with GPU debugging and profiling.

IISPH needs a set of variables whose values are constant throughout the simulation, such as the gravity constant, viscosity constant, time-step etc. These variables can be seen in Figure 5.1. They are collected into a single static class which can be accessed from anywhere within the program.

Figure 5.2 illustrate the information needed for each particle. However, how it was implemented differs between the CPU-versions and the GPU-version. This will be covered in respective implementation details below.

To keep the particles inside the specified simulation domain, a set of particles called boundary particles are utilized. These do not move during the simulation; instead they have a fixed position and act as impenetrable walls, providing the necessary forces to keep the moving fluid particles within the simulation domain.

[1] Visual Studio is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.

[2] Nsight is a trademark and/or registered trademark of NVIDIA Corporation in the United States and/or other countries.

```
// defines particle resolution
float sizeFactor
double globalSupportRadius
// time step
float deltaT
// physical quantities
double fluidParticleSpacing
float gravityConst
float fluidRestDensity
float initialMass
float fluidGasConst
float fluidViscConst
float densityError
// convergence constant
float relaxationFactor
// Iteration values for
// pressure solve loop
uint minIterations
uint maxIterations
```

Figure 5.1: *Constant variables data structure. These variables are used throughout the physics simulation but their values are never modified once initialized.*

5.1 CPU

During the initialization of the simulation a neighbor grid is created which divides the simulation domain into a uniform grid. Each cell receives a unique identifier and calculates which other cells that are neighbors of it and stores theirs identifiers. The cell is thereafter stored in a vector container.

The particles' data are stored in the manner of array of structs (AoS), that is each particle is an object consisting of the data according to Figure 5.2 and the data being localized in a consecutive section of memory.

The particles are created according to a user specified scene as the struct mentioned above. The variables of the algorithm in Algorithm 1 correlate to the ones in the struct as can be seen in Table 5.1. In the creation process all values are set to default values of zero except from the position and whether the particle is a boundary particle or not. The last two variables in Figure 5.2 indicate whether a particle is a boundary particle and at which cell ID (from now on denounced as z-index) the particle is currently residing in. These variables are implementation specific to assist with various tasks, such as finding particle neighbors, hence they are not represented in Algorithm 1.

The following sections describes the reoccurring events for each simulated frame.

```
float3 position
float3 velocity
float density
float pressure
float advection
float densityAdvection
float3 velocityAdvection
float3 displacement
float3 sumPressureMovement

bool isStaticBoundaryParticle
uint index
```

Figure 5.2: *The particle data structure. These are the variables that are stored for each particle. Total memory usage per particle reaches 81 bytes. Note that indexations to identify neighbor particles are not included.*

Struct	Algorithm
position	$\mathbf{x}_i$
velocity	$\mathbf{v}_i$
density	ρ_i
pressure	p_i^{l+1}
advection	a_{ii}
densityAdvection	ρ_i^{adv}
velocityAdvection	$\mathbf{v}_i^{adv}$
displacement	$\mathbf{d}ii$
sumPressureMovement	$\sum_j \mathbf{d}_{ij} p_j^l$

Table 5.1: *Mapping of the variables in the particle data struct to the ones in the IISPH-algorithm.*

5.1.1 Neighbors

Before calculating how the particles are distributed in the grid and finding each particle's neighbors they need to be sorted. This is done according to which cell they belong to by using the cell's identifier, calculated as described in section 3.1 in Goswami *et al.* [13]. After the sorting, the particles belonging to the same cell have been grouped together, see Figure 5.3. Thrust's CPU-version of `sort` is used for the particle sorting.

To make it easier to find a particle's neighbors, each cell stores the container index of the first occurring particle which is inside of the cell and saves how many of the following particles which also exist inside the cell.

To find all neighbors of a particle, it is required to scan the cell the particle resides in and the 26 neighboring cells. Since each cell knows which particles resides in them from previous calculations, it is only a matter of iterating through those particles and check if they are closer or equal to a global support radius, if closer, then a reference to that particle is added to the current particle. Since a

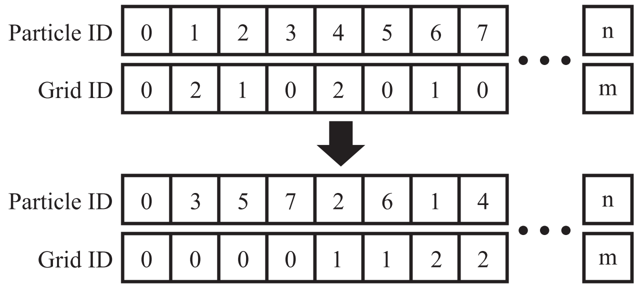

Figure 5.3: *Illustrating how particle's IDs are sorted based on location in the spatial uniform grid. Throughout the simulation, particles will change which cell they are spatially located in (top part). In each frame a sorting occurs grouping particles with the same cell ID together, illustrated with a black arrow. The result is particles stored by cell ID (bottom part).*

particle's neighbors are available, the directions between a particle and its neighbors and the influence of every kernel are computed and stored as well, to later be used during physics computations.

5.1.2 Advection prediction

Advection prediction is divided into two major sections, each section iterating the particles. In both sections a particle's neighbors are fetched once, temporarily stored locally, to later be iterated when calculating density, viscosity and displacement since these calculations require data from surrounding neighbors.

The first section computes the density, intermediate velocity and displacement factor for each particle according to lines 3-5 in Algorithm 1 and stores the resulting values in their respective variables *density*, *velocityAdvection* and *displacement*. When calculating a particle density, its own density contribution is added before iterating over its neighbors adding their contribution to the particle density.

By using the stored results from the first section, the second section calculates each particle's intermediate density, initial pressure and advection factor according to lines 7-9 in Algorithm 1. These are stored in the variables *densityAdvection*, *pressure* and *advection*. The initial pressure can be stored in the *pressure* variable by multiplying itself with 0.5, as mentioned in section 4.3.

5.1.3 Pressure solve

Computing pressure values are done according to section 4.4 with certain modifications. A safeguard was added to the while-loop to prevent it from locking in an infinite loop by not only checking if the average density error is smaller than the acceptable error but also if the iteration counter $l < 50$.

According to Algorithm 1 the pressure should be split into two for-loops, the first one calculates the movement due to neighboring particles' pressure values and stores it in *sumPressureMovement*. The second loop computes the next iteration's pressure before storing it in *pressure*. Both loops are implemented according to the pseudo code in Algorithm 1 with no modifications.

As previously mentioned it is possible to predict the density in the next frame in order to compute the average density error, thus a third for-loop was added. For each particle the density of it in the next iteration is computed as Equation 4.7. Only density values greater than the fluid's rest density are considered as errors. Should the value be less than the rest density, the contribution to the error is clamped to 0.

5.1.4 Integration

The integration works accordingly to the algorithm with a few additions and changes. Collision handling is added to ensure that the particles stay inside the simulation domain and also to avoid getting invalid cell indices due to particles going outside of the uniform grid. It has been modified so particles that are flagged as boundaries are not updating their positions and velocities. Finally in the function the new position and velocity are stored in their respective variables *position* and *velocity*.

5.2 GPU

One of many performance issues that has to be taken into account when designing an algorithm or programming on the GPU is the memory transfer bottleneck occurring when moving memory back and forth to the CPU. One naïve approach would be to do all the computations on the GPU and, to render the results, one would copy the data back to the CPU to be able to update the buffers used when rendering.

However due to the interoperability with DirectX, once the particles are in GPU memory they are never transferred back to the CPU. This effectively removes the memory transfer bottleneck for rendering. In addition to avoiding the slow transfer rates between CPU and GPU, by removing a synchronization point even more time is saved.This also makes it possible to save overall memory since there is no longer a need to have a copy of the data on the CPU and another one on the GPU.

CUDA textures [23] are used throughout the implementation whenever repeated access to an array of data is used within a function. This is due to their fundamental nature of utilizing the GPU's cache to improve the overall speed when repeatedly fetching contiguous data. The gained speed is highly dependent on how the memory is accessed and if the data has enough locality to allow coalescing. This usage is read-only and thus cannot be utilized if an array must be read from and written to during the same kernel.

Furthermore whenever a kernel directly modifies or does computations on a particle or cell, that kernel is launched so every particle or cell gets assigned one thread. For that, the necessary number of CUDA blocks needs to be computed. It is computed as $ceil(t/s)$, where t is the total number of threads needed and s is a specified number of threads for each block. This can create threads that will be unused, this is handled by checking if a threads global id which is calculated as $blockIdx.x * blockDim.x + threadIdx.x$ is lower than the total number of objects available, if so then the thread will continue to run and if not it will terminate.

The number of threads per block is limited by the graphics card's compute capability, e.g. a modern graphics card with compute capability 5.x can have a maximum of 1024 threads for each block, for more details about the different compute capabilities see the CUDA Toolkit Documentation [23]. An overview of kernels used can be seen in Algorithm 2.

Algorithm 2: CUDA physics update

Procedure *NEIGHBOR SORT*
- **Kernel** *calcIndex*
- **Kernel** *sortParticles*
- **Kernel** *sortParticleData*
- **Kernel** *FindGridCellStartEnd*
- **Kernel** *updateNeighbors*

Procedure *ADVECTION PREDICTION*
- **Kernel** *computeDisplacementFactor*
  - **foreach** *particle* i **do**
    - `compute` $\rho_i(t)$
    - `predict` $\mathbf{v}_i^{adv}$
    - `compute` $\mathbf{d}_{ii}$
- **Kernel** *computeAdvectionFactor*
  - **foreach** *particle* i **do**
    - `compute` ρ_i^{adv}
    - `compute` p_i^0
    - `compute` a_{ii}(4.5)

Procedure *PRESSURE SOLVE*
- $l = 0$
- **while** $\rho_i^{err} > \eta \vee l < 2$ **do**
  - **Kernel** *computeSumPressureMovement*
    - **foreach** *particle* i **do**
      - `compute` $\sum_j \mathbf{d}_{ij} p_j^l$
  - **Kernel** *computePressure*
    - **foreach** *particle* i **do**
      - `compute` p_i^{l+1}(4.8)
      - `assign` $p_i(t)$
  - **Kernel** *predictDensity*
    - **foreach** *particle* i **do**
      - `compute` ρ_i^{l+1} (4.7)
  - **Kernel** *calcDensityError*
    - **foreach** *particle* i **do**
      - `reduce` ρ_i^{err}
  - $l = l + 1$

Procedure *INTEGRATION*
- **Kernel** *integration*
  - **foreach** *particle* i **do**
    - `compute` $\mathbf{v}_i(t + \Delta t)$
    - `compute` $\mathbf{x}_i(t + \Delta t)$

5.2.1 Initialization

The CUDA implementation requires initialization and allocation of device memory, which is video RAM (VRAM), from the host. Constant variables used throughout the program as global constants such as the *FluidConstants* are initialized on the host and copied to constant memory on the device.

Unlike the CPU-versions the data of the particles are arranged into SoA in the GPU-version. One advantage of SoA is that it allows for coalesced memory access of consecutive threads. SoA also avoids the issue of padding between elements due to memory alignment which may occur with AoS. The host allocates the arrays in device memory and initializes the particles' starting positions; transferring the position data to the corresponding array in device memory.

In order to render the simulation, data such as the particles' positions and colors are necessary. This information can be shared from CUDA to DirectX without the need to copy the data from the device to the host and back again by using interoperability, this is covered in subsection 5.2.6.

```
uint deviceParticleHash[N]
uint deviceParticleId[N]
uint2 deviceCellGrid[N] //start(x), end(y)

//position(x,y,z), isBoundary(w)
float4 deviceParticlePosition[N]
float4 deviceParticleVelocity[N]
float deviceParticleDensity[N]
float deviceParticlePressure[N]
float deviceParticleDensityAdv[N]
float4 deviceParticleVelocityAdv[N]
float4 deviceParticleSumPressureMovement[N]
//displacement(x,y,z), advection(w)
float4 deviceParticleDisplacement[N]
float deviceParticleDensityNextIteration[N]

uint deviceNeighborGrid[N]
uint deviceParticleNeighbors[N]
```

Figure 5.4: *The different buffers allocated on the device and used in the physics calculations. Buffers beginning with deviceParticle are corresponding to the variables in the particle struct seen in Figure 5.2.*

The initialization begins by constructing the scene of particles precisely as the CPU version does. After the creation the constant variables in Figure 5.1 are transferred to the constant memory on the GPU. It is followed by the initialization of all buffers (see Figure 5.4) used during the simulation by allocating memory and setting default values. The buffers are allocated as mentioned above as contiguous linear memory blocks residing in global memory on the GPU.

The total allocation size depends on the number of particles. For one particle 264 bytes are needed for the physics simulation (104 bytes for the particle data and 160 bytes allocated for neighbors), additional 56 bytes are used when swapping values during the data sort and finally 36 bytes are used for rendering. Note that only 20 bytes (`float4` color and `float` scale) are required for the rendering, the additional 16 bytes (`float4` position) acts as a separator between the rendering and physics. As such a total of 356 bytes are needed for each particle throughout the implementation.

After the buffer initialization a parallel version of `createGridNeighbors` is launched to create and store the grid structure directly on the GPU.

The final step in the initialization is the initial data transfer, the transfer cannot begin before the necessary rendering resources are bound to the physics, see function one in Figure 5.8. When the resources are bound the previously created particles data is transferred to global memory with the addition of rendering data such as color and scale for the two types of particles, boundary and fluid.

5.2.2 Neighbors

The first kernel launch `calcIndex` in the physics loop calculates the z-index for each particle according to section 3.1 in Goswami *et al.* [13] and stores it in *deviceParticleHash*. It also stores an identifier in *deviceParticleId* which is used when sorting the particles in the next kernel.

After the z-index and identifier assignment the particles are grouped together by their z-index value and sorted according to their id using Thrusts `sort_by_key` algorithm. The ids stored are no longer following a sequence, each element is instead indicating which particle that should be stored at that buffer location in the buffers. This is crucial in `sortParticlesData` where the variables used between updates; *position*, *velocity*, *color*, *scale* and *pressure* needs to be relocated accordingly to the new z-index values.

Algorithm 3: FindGridCellStartEnd

```
gridIndex = zValue[index]
sharedIndex[t.x+1] = gridIndex
if index > 0 && t.x == 0 then
    sharedIndex[0] = zValue[index - 1]
end
cell[index] = default
__syncthreads()
if index == 0 || gridIndex != sharedIndex[t.x] then
    cell.x = index
    if index > 0 then
        cell[sharedIndex[t.x]].y = index
    end
end
if index == numParticles - 1 then
    cell[gridIndex].y = index + 1
end
```

`FindGridCellStartEnd` is launched to make finding a particle's neighbors more convenient and efficiently, the goal for this kernel is described in subsection 5.1.1. To avoid the need for iterating every particle and check if it is a valid neighbor, the kernel runs according to Algorithm 3 which is inspired by a particle demo by NVIDIA [24]. Where *index* is the global thread index, $t.x$ is the same as $threadIdx.x$ and *sharedIndex[]* is a shared memory array with an allocated size of CUDA $blocksize + 1$.

The `updateNeighbors` works as described in subsection 5.1.1, with one major difference. Instead of sequentially iterating every particle, each particle is assigned one thread to find all of its neighbors. Apart from the algorithm itself, the way it is stored differs as well; since there is no such thing as a vector container on the GPU, especially not a vector container of vector containers which the CPU version utilizes. The GPU stores the neighbors in *deviceParticleNeighbors* which is allocated as a contiguous memory block with room for the (total number of particles $*$ max number of neighbors allowed) elements. This is a 2D array stored as a 1D array, as such it is accessed as $[x, y] = x *$ max number of neighbors allowed $+ y$.

Because of the contiguous memory allocation and for simplicity it was decided not to store pre-computed frame based values such as distances between particles, which is done in the CPU-implementations.

Algorithm 4: AdvectionPrediction_CPU

```
Procedure ADVECTION PREDICTION
    foreach particle i do
        compute ρ_i(t) = Σ_j m_j W_ij(t)
        predict v_i^adv = v_i(t) + Δt F_i^adv(t)/m_i
        d_ii = Δt² Σ_j −(m_j/ρ_i²) ∇W_ij(t)
    foreach particle i do
        ρ_i^adv = ρ_i(t) + Δt Σ_j m_j(v_ij^adv) ∇W_ij(t)
        p_i^0 = 0.5 p_i(t − Δt)
        compute a_ii(4.5)
```

Algorithm 5: AdvectionPrediction_CUDA

```
Procedure ADVECTION PREDICTION
    Kernel computeDisplacementFactor
        foreach particle i do
            compute ρ_i(t)
            predict v_i^adv
            compute d_ii
    Kernel computeAdvectionFactor
        foreach particle i do
            compute ρ_i^adv
            compute p_i^0
            compute a_ii(4.5)
```

Figure 5.5: *Comparison between CPU- (left) and CUDA-solution (right) for advection prediction. The two kernels in the CUDA-solution correspond to the for-loops in the CPU-solution. A thread will be allocated for each particle i.*

5.2.3 Advection prediction

Due to the necessity of the advection prediction part to run its two loops separately, one after the other, it is required to have a synchronization barrier between the two when developing a parallel version. This can be achieved either by using thread synchronization (using `__syncthreads`) or by launching each loop as a separate kernel since kernel launches are done in sequence on the CPU. Using `__syncthreads` will only allow threads within a block to be synchronized [25], thus only a single block would be possible to be utilized if this method were chosen. Due to this restriction, the separation into two kernel launches was chosen, see Figure 5.5.

Parallelism is achieved by assigning each particle a thread to compute its values and removing the two outer for-loops which the CPU iterated over for each particle. Input-parameters with repeated access to arrays are fetched as textures to alleviate the accessing from global memory. Because of the outputs from `computeDisplacementFactor` are completed before `computeAdvectionFactor` can begin it is safe to access the dependent data using texture fetches.

5.2.4 Pressure solve

The pressure solver can be parallelized with the same strategy as advection prediction, that is, each particle will be assigned a thread and the for-loops is replaced by kernel launches. However, due to that these loops are themselves iterated repeatedly as long as the average density error is greater than an acceptable threshold, this strategy will restrict how much of the pressure solver that is executed on the GPU. Kernel launches are done from the CPU and thus the while-loop and its conditional checks too will be CPU-code. An alternative solution is discussed in subsection 5.3.2.

Algorithm 6: PressureSolve_CPU

```
Procedure PRESSURE SOLVE
    l = 0
    while ρ_i^err > η ∨ l < 2 do
        foreach particle i do
            Σ_j d_ij p_j^l = Δt² Σ_j − m_j/ρ_j²(t) ∇W_ij(t)
        foreach particle i do
            compute p_i^{l+1}(4.8)
            p_i(t) = p_i^{l+1}
        foreach particle i do
            compute ρ_i^{l+1} (4.7)
        foreach particle i do
            reduce ρ_i^err = max(ρ_i^{l+1} − ρ_0, 0)
        l = l + 1
```

Algorithm 7: PressureSolve_CUDA

```
Procedure PRESSURE SOLVE
    l = 0
    while ρ_i^err > η ∨ l < 2 do
        Kernel computeSumPressureMovement
            foreach particle i do
                compute Σ_j d_ij p_j^l
        Kernel computePressure
            foreach particle i do
                compute p_i^{l+1}(4.8)
                assign p_i(t)
        Kernel predictDensity
            foreach particle i do
                compute ρ_i^{l+1} (4.7)
        Kernel calcDensityError
            foreach particle i do
                reduce ρ_i^err
        l = l + 1
```

Figure 5.6: *Comparison between CPU- (left) and CUDA-solution (right) for pressure solve. The four kernels in the CUDA-solution correspond to the for-loops in the CPU-solution. A thread will be allocated for each particle i. The condition check for the while-loop is still executed on the CPU.*

As can be seen in Figure 5.6 another kernel is added beside the ones that computes the movements due to pressure, pressure values and the one that tries to predict the density for the density error. The kernel `calcDensityError` is a small kernel that adds up the estimated density values using Thrust `reduce`.

As with advection prediction, pressure solve kernels use texture fetches to reduce global memory access for input-parameters.

5.2.5 Integration

This procedure is the least changed, see Figure 5.7, due to its simple and straightforward structure. Just as in the CPU-version it has the additional collision handling and the boundary particles do not update their velocities nor positions. It is launched as every other kernel and utilizes CUDA textures whenever possible.

Algorithm 8: Integration_CPU	**Algorithm 9:** Integration_CUDA

Procedure *INTEGRATION*
  foreach *particle i* **do**
    $\mathbf{v}_i(t+\Delta t) = \mathbf{v}_i^{adv} + \Delta t \frac{\mathbf{F}_i^p(t)}{m_i}$
    $\mathbf{x}_i(t+\Delta t) = \mathbf{x}_i(t) + \Delta t \mathbf{v}_i(t+\Delta t)$
	Procedure *INTEGRATION*
	  Kernel *integration*
	    foreach *particle i* **do**
	      `compute` $\mathbf{v}_i(t+\Delta t)$
	      `compute` $\mathbf{x}_i(t+\Delta t)$

Figure 5.7: *Comparison between CPU- (left) and CUDA-solution (right) for integration. The kernel in the CUDA-solution corresponds to the for-loop in the CPU-solution. A thread will be allocated for each particle i.*

5.2.6 DirectX interoperability

```
cudaGraphicsD3D11RegisterResource(cudaGraphicsResource **resource,
ID3D11Resource *pD3DResource, unsigned int flags)
cudaGraphicsUnregisterResource(cudaGraphicsResource_t resource)
cudaGraphicsMapResources(int count, cudaGraphicsResource_t *resources,
cudaStream_t stream = (cudaStream_t)0)
cudaGraphicsUnmapResources(int count, cudaGraphicsResource_t *resources,
cudaStream_t stream = (cudaStream_t)0)
cudaGraphicsResourceGetMappedPointer(void **devPtr, size_t *size,
cudaGraphicsResource_t resource)
```

Figure 5.8: *The interoperability commands used which allow CUDA to manipulate the buffers for DirectX.*

To connect CUDA to DirectX four functions where used, their signatures can be seen in Figure 5.8. The functions always returns a `cudaError_t` indicating if anything went wrong or if the function call succeeded, e.g. an error indicating an invalid value or a unknown error [26].

The first function is used to tell CUDA to register a given resource accompanied with optional *flags*. The *flags* give CUDA a hint to how the resource is going to be used e.g. as a surface reference. However, there are limitations on which Direct3D resources that can be registered, a few of those limitations are resources allocated as shared and the primary render target cannot be registered. The second function is used to unregister a registered resource [26].

The third and fourth functions are used to map and unmap *count* resources to allow access from CUDA. While the resources are mapped, access is granted. However, if DirectX should try to access the resources while it is bound it will cause undefined behavior. Function three guarantees any DirectX API calls are done before any following CUDA work begins, while function four guarantees that all CUDA work is completed before new DirectX calls are issued [26].

The final function is used to get a `void**` pointer which points to the first element in the provided resource. If desired, the accessible size can also be provided. Otherwise providing the function with `nullptr` will block the function from returning a size [26].

These functions are used for each connected Direct3D resource. The resources used are allocated in three structured buffers containing the positions, colors and scales for each particle. Function one is used only once for each resource after the initialization of the physics and graphics, function two is used when the physics destructor is called.

Since function three and four are synchronization points in the program they should not be used in abundance because, if not carefully placed, they could very well limit the program's overall performance. Therefore, in the implementation each of the three resources call function three closely followed by function five at the beginning of each reoccurring update while function four is the final call in each update.

5.3 Alternative approaches

The proposed implementation was one of few alternatives that emerged during the implementation stage. Alternatives ranged from function changes to fundamental structure changes in kernels. Those worth mentioning are the shared memory approach and adjustments due to newer compute capabilities.

5.3.1 Shared memory

A solution based on Goswami *et al.* [13] was implemented. This was done by inserting one stage before advection prediction and after finding all cells start and end particles, this stage copies the buffer containing cells start and end particles indices to a new buffer and modifies it. In the new buffer all the empty cells were removed and the cells with a particle count above 27 was divided into chunks, each containing up to a maximum of 27 particles.

To be able to use the new buffer with the kernels, modifications had to be made accordingly. Kernels were now launched with a fixed number of threads, viz. 27, one for each particle with as many blocks as the new buffer had cells.

All kernels had to be slightly modified to accompany the new solution. The initial process was changed to determine the maximum amount of possible neighbors residing in neighboring cells. Before iterating through the maximum amount of neighbors each thread copies its particle's relevant data into shared memory. During the frame each thread copies a neighboring particle into shared memory and synchronizes the threads directly afterwards without leaving the frame. Finally during the frames each thread iterates over the data residing in shared memory and runs the desired computations such as calculating density for the assigned particle.

5.3.2 Compute Capability 5.X

Due to hardware limitations on the development platform, it was necessary to restrict the solution to compute capability 3.0. This effectively restricted which features in CUDA that could be used.

With the introduction of compute capability 3.5 dynamic parallelism was supported [23]. This allows for a kernel to launch separate kernels that are run in parallel. The control loop for the compression when solving the pressure equation could take advantage of this feature by launching its subsequent kernels (the kernels to compute movement due to pressure, pressure values, calculate a predicted density and the average density error) as child kernels. Thus unnecessary returns to CPU in order to launch kernels could have been avoided.

Another feature set introduced with compute capability 3.5 is the atomic min and max operations. This could have been used in the kernel for finding a cell's start and end indices. With these atomic operations the branching factor could have been reduced by the possibility to eliminate certain *if* statements. This in turn could have led to better performance.

Chapter 6

Experimental Method

Experiments were conducted on all versions by running identical scenarios with implemented test scenes and measuring the computation times required to complete each scenario. Due to the aim of the thesis and that it is commonly done [3, 13, 14], measuring the time required for running the simulation was deemed as the best method to gather relevant data. All scenarios assume identical relative camera position and view direction.

Additionally, the particles' memory usage was gathered as well. Since a single graphics card only has a certain amount of memory it is a limiting factor on how many particles that can be residing in memory simultaneously. For instance a 4 GB graphics card can theoretically hold approximately 12 million particles with the proposed solution.

The SPH CUDA algorithm presented by Goswami *et al.* [13] was modified to the proposed fine-grained solution so the performance cost of adding incompressibility to fluids could be observed.

The experiments were run on two different hardware setups according to Table 6.1.

	Setup 1	**Setup 2**
CPU	Intel® Core™ [1] i7-3770, 3.4GHz, 8MB Cache	Intel® Xeon® [1] E5-1650, 3.2GHz, 12MB Cache
GPU	MSI GeForce GTX 970 Gaming 4G	NVIDIA Quadro K4000
RAM	Corsair Vengeance DDR3, 1600MHz, 16GB	DDR3, 1600MHz, 16GB
OS	Windows® [2] 7 Ultimate 64-bit	Windows® [2] 8.1 Enterprise 64-bit

Table 6.1: *Computer specifications for experiments.*

6.1 Time measurement functions

To measure the time and to get as close accuracy as possible in terms of what was measured, three separate timers were used: one for CPU-code, one for CUDA-

[1] Intel, Intel Core and Xenon are either registered trademarks or trademarks of Intel Corporation in the United States and/or other countries.

[2] Windows is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.

code and one for rendering. Timings of the physics simulation were conducted using two approaches. Both methods computed the time used per frame and calculated an average time usage. The first one created a time stamp right before launching the physics simulation, ran the current frame's simulation and created a new time stamp right after the end of the simulation, which allowed for an evaluation of total time required by the frame. Adding one layer of detail, the second method used the same approach but on comparable (CPU compared to GPU) subsections inside of the physics-function. This allowed for identification of which sections of the physics simulation that were the most computationally expensive. The two approaches were run separately to not cause inaccurate timings due to overhead.

The CPU-version utilizes the QueryPerformanceCounter API available in Microsoft Windows which can provide the frequency of the CPU core as well as the number of clock ticks between two time stamps [27]. The time can thus be evaluated in milliseconds and seconds, and finally converted to FPS.

CUDA provides the methods `cudaEventRecord`, which is used to record an event, and `cudaEventElapsedTime` [28] which evaluates the time between two recorded events. The GPU-timer for CUDA measurements used this.

The time required to render each frame was measured using the `ID3D11Query` interface which is used to query the graphics card for data. Using `D3D11_QUERY_TIMESTAMP` and `D3D11_QUERY_TIMESTAMP_DISJOINT` it is possible to receive reliable time stamps in clock ticks and the GPU core frequency [29, 30] which is used to convert the result similar to how the CPU-timer does.

6.2 Test scenes

Four different test scenes were implemented with varying complexity, *simple*, *breaking dam*, *two blocks* and *gallery*, see Figure 6.1. All scenes are of the same dimensions and use identical setup for static boundary particles and collision boxes. The boundary particles creates a box with an open top domain. This can be seen in Figure 6.2 where the green particles represent the static boundary particles and the blue ones are the fluid particles. The number of particles ranged from 4 000 to 200 000 particles.

In *simple*-scene a block of fluid particles begins by falling towards the bottom of the box before collapsing. In *breaking dam* the fluid particles are stacked directly on the bottom as a block alongside one of the walls, collapsing into the box. The *two blocks*-scene is similar to the *simple*-scene but consist of two smaller separate blocks which fall towards the bottom of the box. The last scene, *gallery*, is more complex. It contains obstructions created with static boundary particles which were used to build a wall and some pillars. A small block of fluid particles begins by falling towards the "ground" before interacting with the walls and pillars.

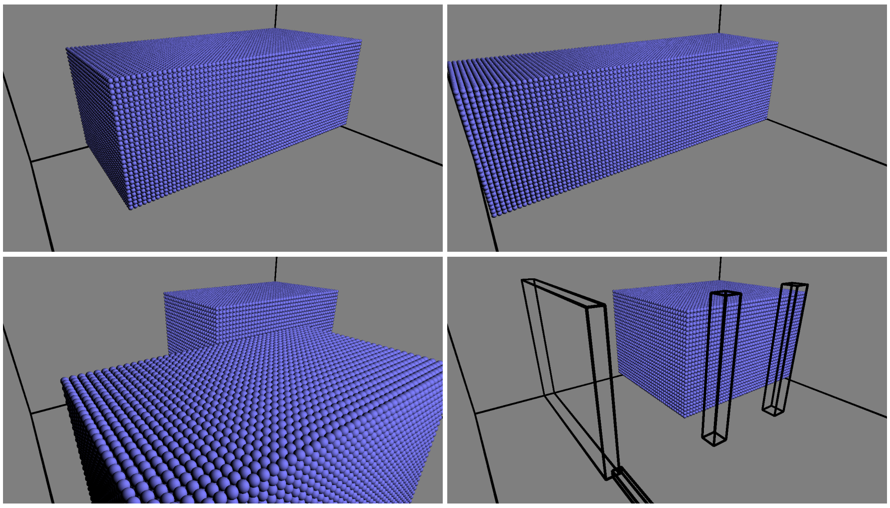

Figure 6.1: *The four test scenes. Top left: simple, 121 000 particles. Top right: breaking dam, 114 000 particles. Bottom left: two blocks, 116 000 particles. Bottom right: gallery, 99 000 particles. Note that boundary particles have been omitted during rendering.*

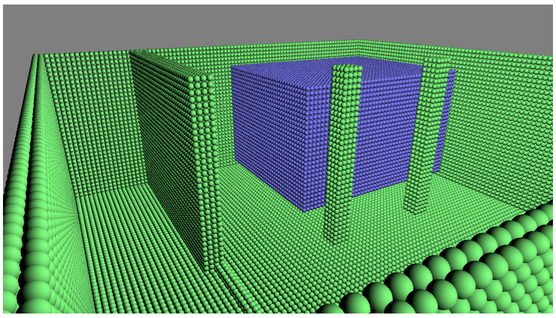

Figure 6.2: *The gallery scene with 51 000 boundary particles (green) and 48 000 fluid particles (blue).*

6.3 Memory usage

To measure the total memory allocated by the CUDA-solution, the CUDA allocation function was wrapped in a new function which then was used instead when allocating new buffers. In the function a variable was increased by the allocated size every time a new buffer was allocated. The memory footprint for the constant variables seen in Figure 5.1 is neglected in the tests since the memory usage for these variables are equal between the implementations.

Since the CPU-version utilizes C++ `std::vector` it is troublesome to get an accurate number of the total amount of memory allocated due to the dynamic nature of the container. This is only a problem for the `std::vector` containing the particles' neighbors since the number of neighbors differ between frames. As such an average number of neighbors where calculated during each simulation. To get how much memory a struct was allocating the C++ function `sizeof` was used. To get the total allocated size for the CPU-implementation, the size of each struct was multiplied with its corresponding count e.g. the number of particles multiplied with the size of the particle struct.

6.4 Physical precision comparison

The parallel GPGPU-implementation should not alter the visual quality of IISPH; thus a verification of this was necessary to be performed. Two common techniques to measure errors and noise in images are the peak signal-to-noise ratio (PSNR) and structured similarity index measure (SSIM) [18]. Both methods are sensitive for different image degradations due to lossy compressions. SSIM has been criticized for giving an unreliable result [19].

The PSNR was chosen as the metric for comparing the simulations. An acceptable value for images with bit depths of 8-bit ranges between 30 – 40 dB. The higher the value, the better the result is.

The comparison images were saved as portable network graphics (PNG), which is a lossless format, to avoid issues due to image degradation.

Chapter 7

Results

To ease further reading this section will only present a compilation of processed data. All collected data can be reviewed in Appendix A. Multiple test runs were executed. The data presented is from a single run but has been cross-validated with the others to ensure the values being reasonable.

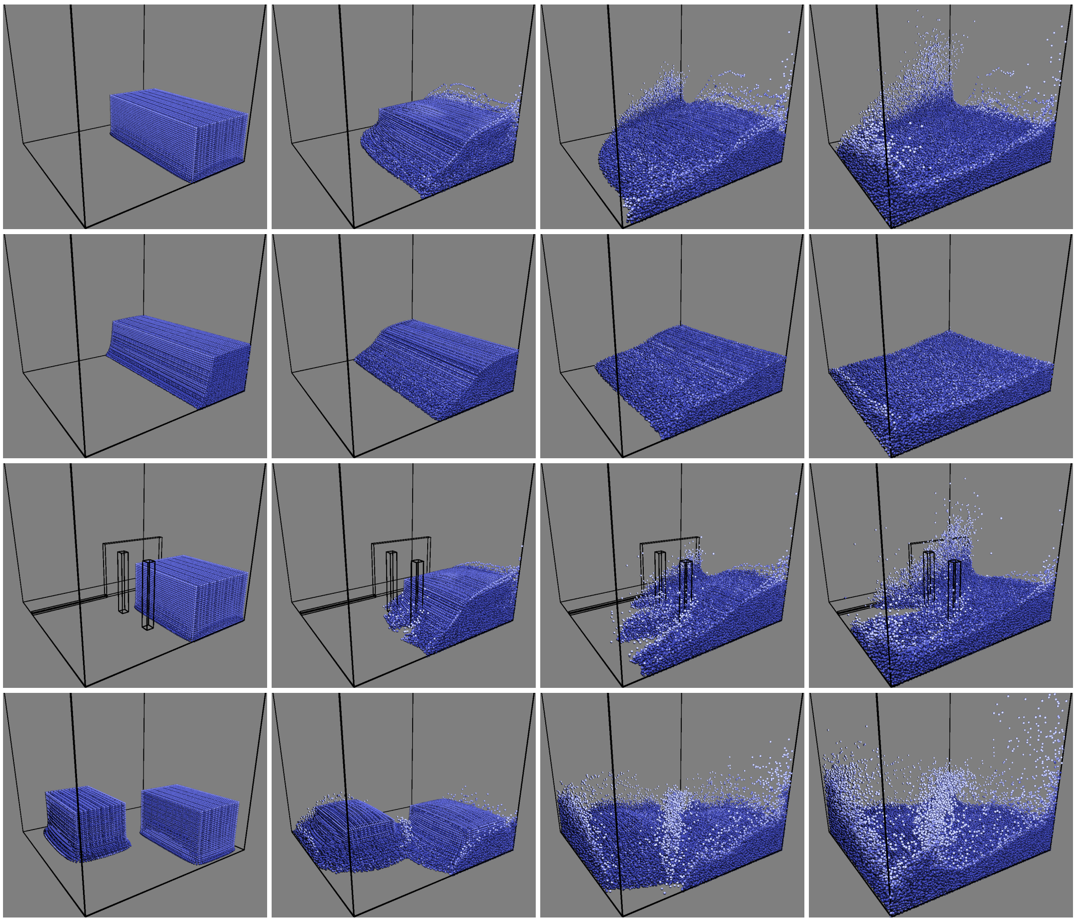

Figure 7.1: *All four scenes visualized at the same time interval. From top to bottom: simple-scene (121 000 particles), breaking dam (114 000 particles), gallery (99 000 particles) and two blocks (116 000 particles). In this case a time-step of 3.5 ms and particle spacing of 0.05 m was used.*

In all measured scenarios a fixed time-step of 3.5 ms and particle spacing of 0.09 m was used. All kernels were launched with blocks of a maximum of 256 threads per block. To increase or decrease the total number of particles in a scene the dimension was changed. Each of the scenes used the same set of dimensions, hence the number of particles varied between scenes. To ensure accuracy of the time measurements the average frame-time was computed from test scenarios running for 300 and 1 000 frames. The deviation between these two cases was considered negligible and thus only data from the 1 000 frames scenarios are being presented in this section.

A time-lapse for the scenes can be seen in Figure 7.1.

7.1 Time usage

Note that the lines for the graphs comparing the scenes in this section ends at different x-values. This is due to the amount of particles varying between scenes and all scenes were running with the same dimension changes to scale them up (resulting in increased amount of particles) during the experiments.

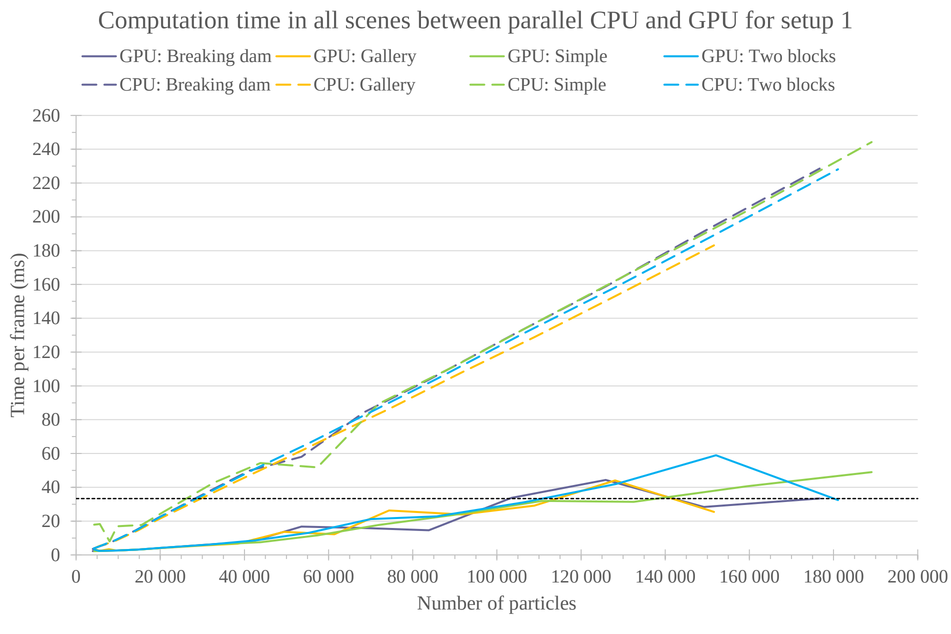

Figure 7.2: *The graph visualizes the computation time of the algorithm per frame on setup 1 for the CUDA-solution compared to OpenMP. All four tests scenes used a time-step of 3.5 ms and a particles spacing of 0.09 m. Measurements was taken over 1 000 frames. The black horizontal dashed line denotes the 30 FPS mark.*

The CUDA-solution is able to achieve higher performance than an OpenMP-implementation on the CPU according to Figures 7.2 and 7.3. The CPU has a steady time per frame regardless of scene while the CUDA times fluctuate ever so slightly on computer setup 2, but on setup 1 the fluctuation is greater. The cause for the fluctuation in setup 1 is unknown.

Although the scenes differ slightly from each other, they still follow a linear growth when the amount of particles increases. As an example, on setup 1 the *gallery*-scene with *approximately* 150 000 particles required 25.4 ms per frame to compute the physics calculations using the proposed CUDA-implementation. In the same scenario using OpenMP, the time required was 183.1 ms. On setup 2 the corresponding values were 58.1 ms with CUDA and 124.6 ms with OpenMP.

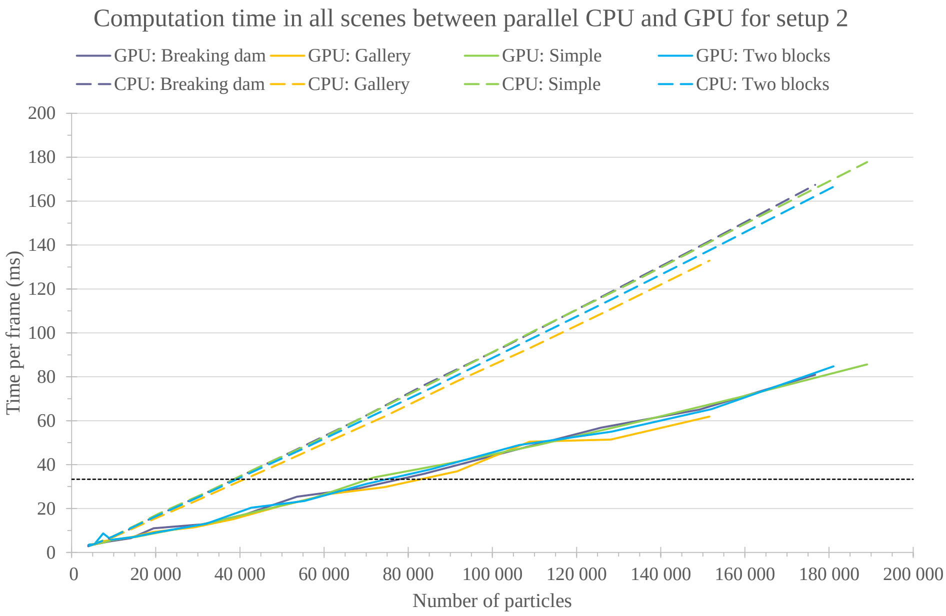

Figure 7.3: *The graph visualizes the computation time of the algorithm per frame on setup 2 for the CUDA-solution compared to OpenMP. All four tests scenes used a time-step of 3.5 ms and a particles spacing of 0.09 m. Measurements was taken over 1 000 frames. The black horizontal dashed line denotes the 30 FPS mark.*

As seen in Figure 7.4 the sequential CPU-solution also follows a linear growth rate and the times compared with the CUDA-solution is significantly slower. The same fluctuations can be observed for the CUDA-implementation as in Figure 7.2. Computer setup 2 was tested as well but the result was not as good as on setup 1 and is thus not included in this chapter.

When disabling OpenMP, the CPU-version required for the *gallery*-scene with *approximately* 150 000 particles 582.6 ms to calculate the physics each frame.

Figure 7.5 show a linear growth for time spent in the majority of the kernels in the implementation except for `FindGridCellStartEnd` and `DensityError` which seems to be more or less constant during the experiment. All measurements can be directly mapped to the kernels in Algorithm 2. Note that these are the total execution times for a single frame, meaning that the kernels in the pressure solve procedure are the total execution time per frame and not per iteration.

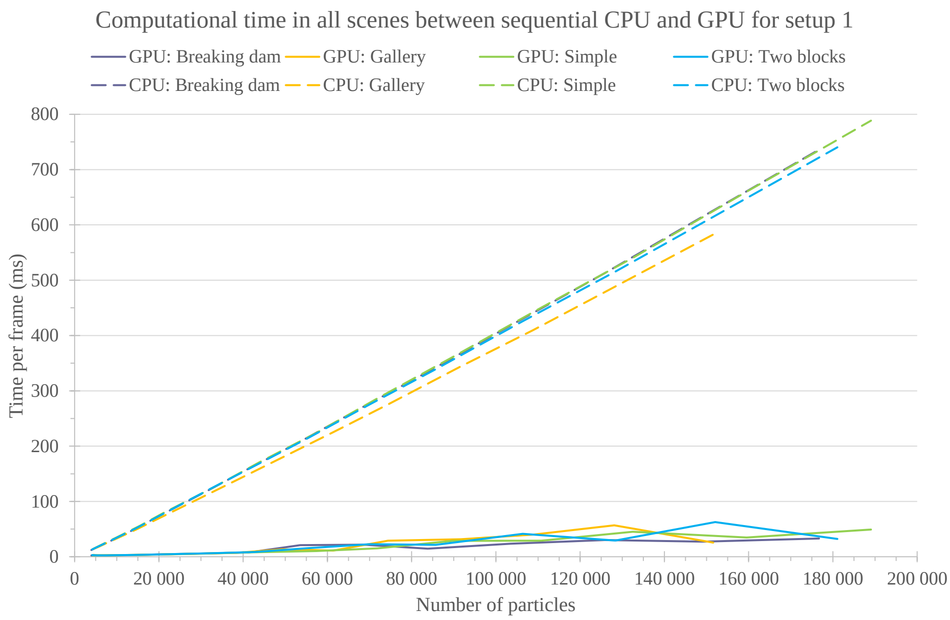

Figure 7.4: *The graph visualizes the computation time of the algorithm per frame on setup 1 for the CUDA-solution compared to the sequential CPU-implementation. All four tests scenes used a time-step of 3.5 ms and a particles spacing of 0.09 m. Measurements was taken over 1 000 frames.*

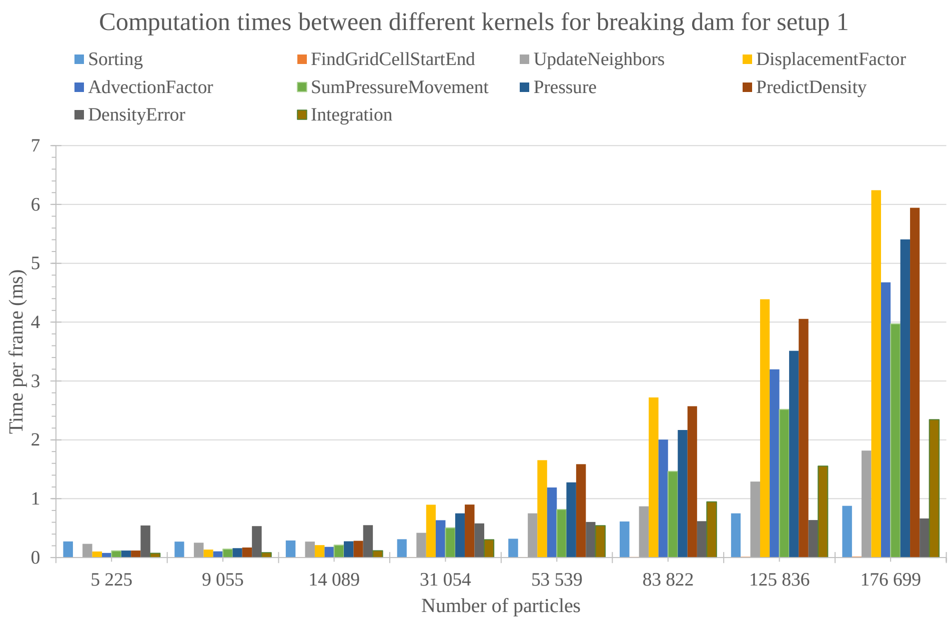

Figure 7.5: *Visualizing the computation time for each kernel in each frame in the breaking dam scene. A time-step of 3.5 ms and particle spacing of 0.09 m was used. Measurements was taken over 300 frames on computer setup 1.*

As such it is clear that the execution time for the advection prediction and pressure solve procedures are the majority of the total execution time each frame. Where pressure solve is the slower one of the two.

To show the cost of incompressibility the fine-grained approach presented was applied to SPH and measured with both setups, the results can be seen in Table 7.1. The results for both setups in Table 7.1 follow a linear growth rate just as IISPH does where setup 2 are slightly steeper than setup 1.

SPH CUDA	**Setup 1**		**Setup 2**	
Particles	time (ms)	FPS	time (ms)	FPS
7 600	0.79	1268	1.42	707
20 000	1.15	874	3.89	277
54 000	3.19	329	6.90	146
103 000	7.42	173	14.04	76
175 000	16.55	64	22.19	46

Table 7.1: *Table showing the results of a SPH implementation according the fine-grained approach presented in this thesis.*

7.2 Memory usage

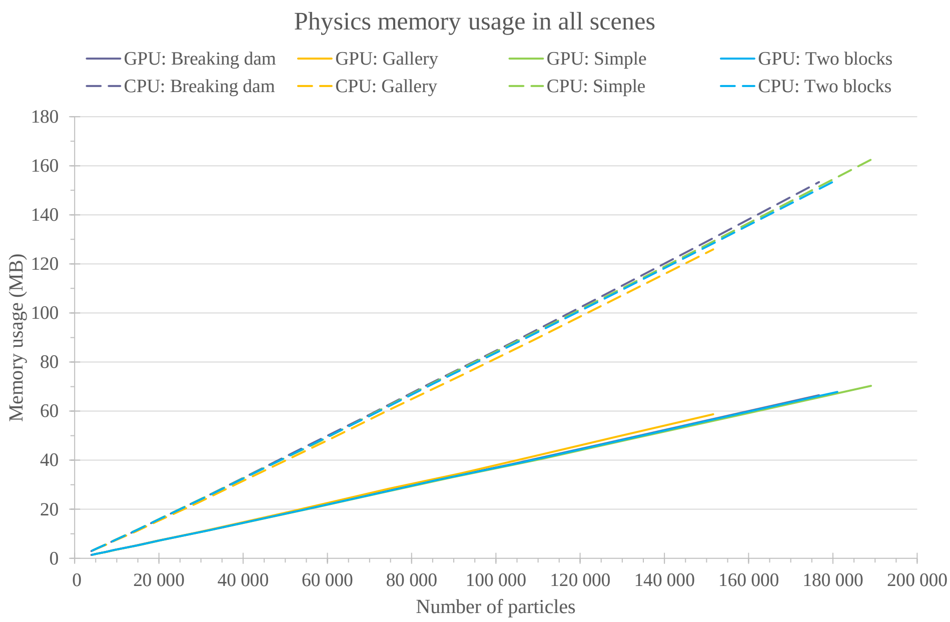

Figure 7.6: *The memory usage in all scenes is growing linearly with the number of particles.*

A similar linear growth in memory usage was detected in both the CPU- and the GPU-implementation for all test scenes when the number of particles increased, see Figure 7.6. As an example the *gallery*-scene noted for *approximately*

150 000 particles a usage of 126 MB on the CPU-versions and 59 MB on the GPU-version whereas the result in the *breaking dam* scene was 128 MB and 56 MB respectively.

7.3 Physical precision

The *simple*-scene was used when doing the image comparison. The program took a screen capture every 50 frame for each solution, sequential, OpenMP and CUDA. The boundary particles are not rendered while doing the comparisons since they remain constant throughout the simulation and they are not relevant for rendering, because they are only used during physics computations to get a better wall collision. For all the screen captures the camera's position and view direction are identical.

The results can be seen in Figures 7.7 and 7.8. There is no difference between the sequential and the OpenM- version as can be seen in the bottom row in Figure 7.7. There is however differences between the sequential and CUDA-version, see the bottom row in Figure 7.8, the most noticeable is the back section in frame 150 is quite scattered compared to the sequential and slight differences in the front.

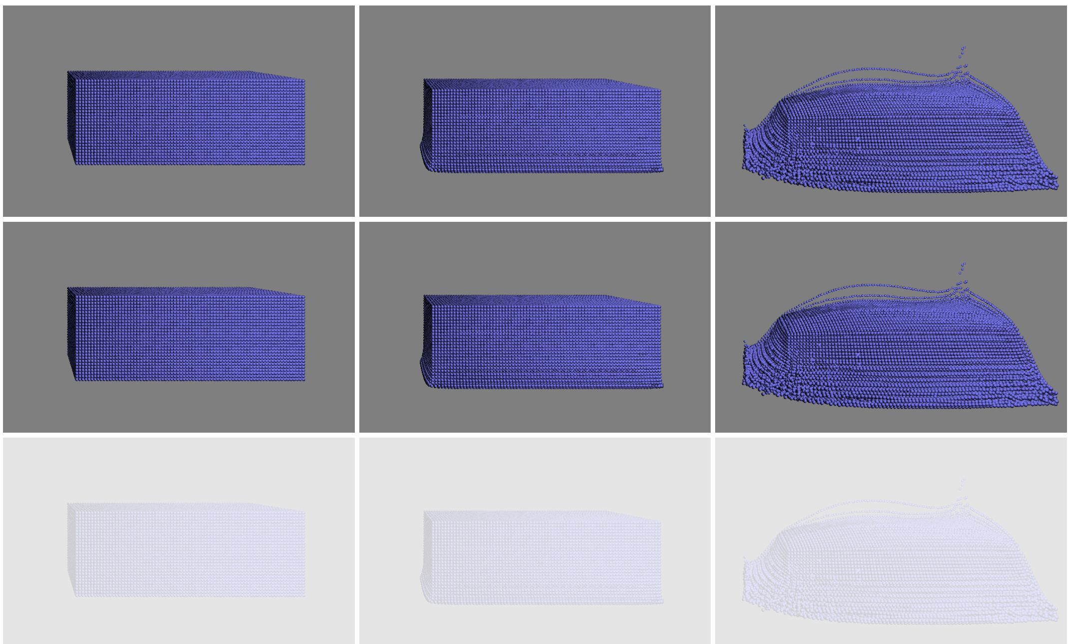

Figure 7.7: *Screen comparison of simple scene every 50 frame with 43 000 fluid particles. This is frame 50 – 150. Top row: sequential. Middle row: OpenMP. Bottom row: PSNR difference between top and middle. No difference was measured between the two. The fading of the bottom row is a result from the tool used and does not imply anything.*

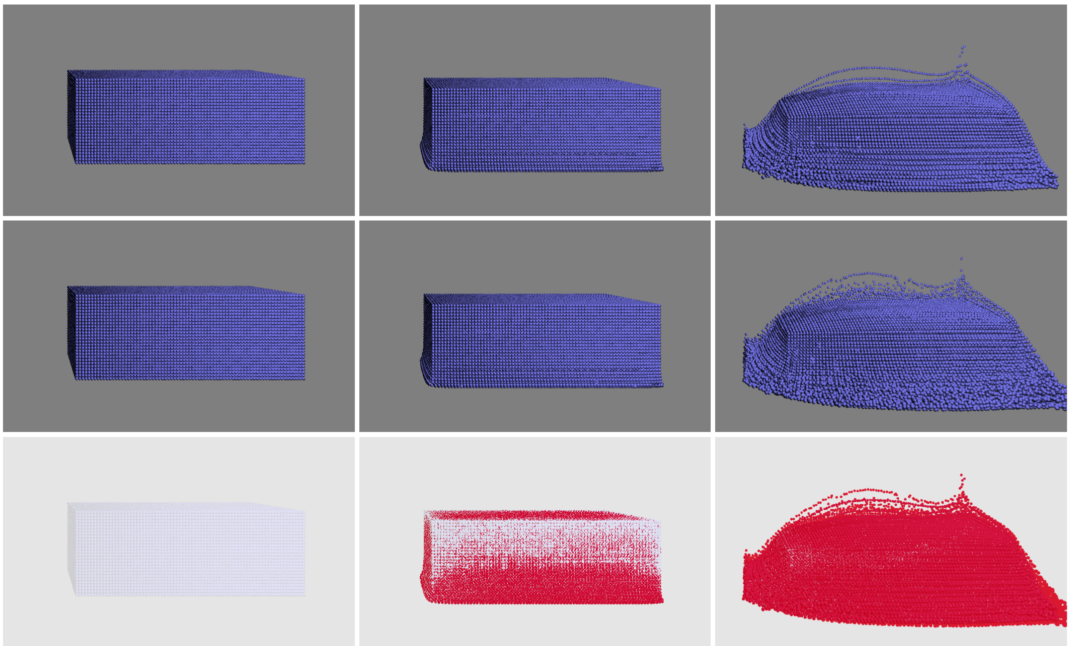

Figure 7.8: *Screen comparison of simple scene every 50 frame with 43 000 fluid particles. This is frame 50 – 150. Top row: sequential. Middle row: CUDA. Bottom row: PSNR difference between top and middle, note that red indicates difference, however, not how much it differs. The fading of the bottom row is a result from the tool used and does not imply anything.*

7.4 Shared memory

The numbers in Table 7.2 are from an early development stage where a shared memory approach was explored for the first section of advection prediction. The difference between approaches is the kernel structure and how much preparations is needed. For the global approach an additional 94.6 ms is required for preparations once each frame, for such as finding neighbors. This is not needed in the shared memory approach. However similar differences between the other kernels was observed.

	Dimension				
Approach	Grid	Block	Duration (ms)	Occupancy (%)	Block limit reason
Global	{817,1,1}	{256,1,1}	14.1	75	Registers
Shared	{17251,1,1}	{27,1,1}	129.8	25	Blocks

Table 7.2: *Performance for first part of advection prediction with 220 000 particles. Nsight's profiler was used for the numbers for the different approaches. The global approach is the proposed implementation and shared is the alternative approach.*

Chapter 8

Analysis and Discussion

Based on the results certain conclusions can be made regarding the performance of the proposed implementation as well as the qualitative result.

8.1 Time usage

By averaging the measured time results, the usage of an average scene for a certain amount of particles can be compiled. For setup 1 this can be seen in Table 8.1 and for setup 2 in Table 8.2. Setup 1 achieved a speed-up of the parallel GPU-solution compared to the parallel CPU-version with a factor of ∼6. In setup 2 the speed-up was lower than that of setup 2 with a speed-up factor of ∼2.

Setup 1	Physics - GPU		Physics - CPU (OpenMP)		
Particles	Time (ms)	FPS	Time (ms)	FPS	Speedup
7 600	2.58	388	6.89	146	2.67
20 000	4.06	247	22.18	45	5.47
54 000	10.21	100	64.20	16	6.29
103 000	21.07	49	126.80	8	6.02
175 000	39.18	28	221.16	5	5.64

Table 8.1: *The average time measured on computer setup 1 for each scene with a calculated speed-up between the GPU and CPU, OpenMP was activated.*

Tables 8.1 and 8.2 show that depending on hardware setup the CUDA solution can compute 54 000 – 103 000 particles while maintaining ∼45 FPS. While the OpenMP solution on the CPU can only handle 20 000 – 37 000 particles.

Calculations from Tables 8.1 and 8.2 show that real-time performance was achieved with CUDA for almost 154 000 particles with 30 FPS on setup 1. The corresponding values for OpenMP on setup 2 (which had a better CPU) was approximately 39 000 particles. The proposed solution can support up to about 4 times the amount of particles while running in real-time compared to the OpenMP-solution.

Setup 2	Physics - GPU		Physics - CPU (OpenMP)		
Particles	Time (ms)	FPS	Time (ms)	FPS	Speedup
7 600	4.58	219	5.34	188	1.17
20 000	9.45	106	16.23	62	1.72
54 000	23.18	43	46.23	22	1.99
103 000	47.82	21	92.29	11	1.93
175 000	83.06	12	161.06	6	1.94

Table 8.2: *The average time measured on computer setup 2 for each scene with a calculated speed-up between the GPU and CPU, OpenMP was activated.*

Sequential CPU	Setup 1		Setup 2	
Particles	Time (ms)	FPS	Time (ms)	FPS
7 600	25.46	39	28.29	35
20 000	72.91	14	86.24	12
54 000	206.43	5	242.17	4
103 000	409.60	2	478.97	2
175 000	711.90	1	830.92	1

Table 8.3: *The average time measured on both computer setups for the sequential CPU implementation.*

As can be seen by comparing Tables 8.3 and 8.1 the performance is improved by a magnitude with CUDA compared to the sequential CPU-version. For real-time with 30 FPS the sequential CPU could at best handle approximately 9 400 particles (calculated from Table 8.3 with setup 1). This is fewer than the CUDA-implementation with a factor of ∼19.

The CPU time differences for OpenMP between the computer setups are probably mostly due to the CPU in setup 2 has two more cores than the CPU in setup 1, but also the additional 50% available cache memory. However, times measured on sequential CPU are faster on setup 1 because of the higher CPU frequency, even though it has fewer cores. This is because without parallelism (OpenMP), the program runs on one core which makes extra cores unnecessary. The differences between the GPUs however depend on numerous properties. For example the GPU in setup 1 can support twice the amount of CUDA blocks per multiprocessor which increases the possible parallelism accordingly.

The complexity of the scene had a marginal impact on the time measurements. While there exist minor deviations of computational time between scenes, though the particle count is identical, this can be caused by several factors. It may be that the allocated resources from the OS varied during the experiment. It is possible that the *gallery*-scene (which is more complex than the *breaking dam*) measured lower computation time due to it being constructed with a higher amount of static boundary particles. While these particles are still included in computing the advection and pressure, they are being ignored in the integration process which results in fewer computations.

To get better times it is assumed that by excluding inactive boundary particles (boundary particles without an active fluid particle within its global support radius) from the computations the execution time is also reduced. It would not require any significant restructuring of the code-base to add the extra code needed. The saved time would most likely be more or less equal for each implementation which would not affect the overall speed-up gotten from the experiments.

With an increase of the particle count, it is the physics-kernels (*DisplacementFactor*, *PredictDensity*, *Pressure*, *AdvectionFactor* and *SumPressureMovement* in Figure 7.5) that are affected the most in terms of execution time. A trend is established for approximately more than 30 000 particles where these kernels become dominant in execution time. This can be expected due to these kernels being heavily influenced of looping the neighbors of each particle and thus, adding a particle to the simulation will cause these kernels to increase their execution time.

There are two interesting paths to explore to increase the performance even more. First a shared memory approach which is capable of reaching 100% occupancy if the memory access to global memory turns out to be the major bottleneck. Secondly, to improve the proposed solution the number of registers used must be decreased as seen in Table 7.2 to reach a higher occupancy. Alternatively, if it is not a memory problem nor a register problem, a different algorithm structure or approach which includes the advection prediction and pressure solve procedures would be desired. This due to they are the procedures which takes up the majority of the execution time, as seen in Figure 7.5.

Compared with the numbers in Tables 8.1 and 8.2 it is obvious that there is a significant cost that comes with incompressibility, SPH runs on average 3.18 times as fast as compared with IISPH for setup 1 and 3.33 times for setup 2. However, since SPH would require smaller time-steps it would (for a given time-span) also need to process more frames than IISPH. IISPH can support a time-step up to 0.02 ms [3] while SPH can support to 0.0015 ms [20], which is a factor of up to approximately 13 time more frames required to simulate any given time-span. This would yield with e.g. 20 000 particles a total execution time that would be 3.78 times as long for SPH than IISPH.

8.2 Memory usage

There are distinct similarities in the linear growth of the memory allocated between scenes in the CPU- and CUDA-versions respectively seen in Figure 7.6. This was expected due to allocations only depending on the amount of particles and the partitioning of the simulation domain into a uniform grid. An average memory usage for a scene depending on the number of particles can be compiled and seen in Table 8.4.

Particles	GPU (MB)	CPU (MB)
7 600	2.7	5.7
20 000	7.3	15.6
54 000	19.8	43.9
103 000	38.1	86.2
175 000	65.8	149.4

Table 8.4: *Average memory usage for each scene depending on the total number of particles. In the test cases the CPU-versions utilized a little more than twice the amount of memory than that of the GPU-version.*

As can be seen the CPU-versions utilize approximately twice the amount of memory than that of the CUDA-version. This can be explained by specific differences in how and what data is stored in the implementations.

In the CPU-versions padding occurs on the particle struct with 3 bytes per instance, which yields an additional usage of 0.5 MB for 175 000 particles due to using AoS. This padding does not occur on the CUDA-version because of it following SoA. However it is only a fraction of the difference in memory usage between the implementations. Further, the CUDA-version allocates unused memory because of the need to use `float4` instead of `float3` to utilize the CUDA texture loads from global memory. While it was possible to merge certain variables to be stored in the same array to eliminate these empty allocations in the CUDA-version, some padding of this type could not be avoided.

The major variety however is caused by another difference in the implementations. The CPU-versions store certain neighbor data (such as distance between particle and neighbor, as well as kernel values) for each particle to reduce unnecessary repeated calculations. This data accounts for an additional 24 bytes of usage per neighbor of a particle. In the CUDA-implementation it was however considered that adding some extra computations in order to reduce memory bandwidth usage would be preferable. Averaging a particle to have 23 neighbors at any time this yields a decreased memory usage of 92.1 MB for 175 000 particles.

It is assumed that the implementations would have similar memory usage if this neighbor data was not stored in the CPU-versions, however they would be required to perform more computations which would affect performance negatively. Further analysis would be necessary to confirm this however.

8.3 Physical precision

As can be seen in Figure 7.8 and in Table 8.5 the CUDA-solution deviate considerably from the sequential implementation while OpenMP does not deviate at all, see Figure 7.7. Note that the PSNR does not tell how large the total physical

	PSNR (dB)	
Frame	Bottom, Figure 7.7	Bottom, Figure 7.8
50	∞	∞
100	∞	35.2
150	∞	20.5

Table 8.5: *The PSNR values from the comparisons between top and middle rows of Figures 7.7 and 7.8. In frame 100 the CUDA solution is still within the acceptable range. In frame 150 the value is below the threshold, this difference is visible when comparing the middle rows of Figures 7.7 and 7.8 for frame 150.*

error is, rather it conveys if there is a difference in the particles' positions relative to the camera.

The reason to the images being identical in the first 50 frames is because the particles are falling due to gravity. Since they are evenly spaced out when created they influence each other equally resulting in a standstill until the first particles hit the ground.

It is arguable if PSNR was a useful metric for the precision. As it tells if a pixel in an image differs from the original image but not to what extent, it only indicates if a deviation has occurred. Thus no conclusions can be made whether the error is within an acceptable range or not. It is also deemed to be subjective whether the differences seen between the middle rows of Figures 7.7 and 7.8 are acceptable or not.

8.4 Shared memory

There is a large difference in duration between the global and shared memory approach, approximately 9.2 times as much. Although there is a more expensive preparation stage for the global approach than the shared memory approach, the significant increase in duration between each kernel makes it less important for the overall time. As seen in Table 7.2 the occupancy for the kernel is 3 times as much for the global approach and the limiting factor is the registers used, whereas the shared memory approach implemented only had an occupancy of 25% and was limited by the number of blocks. The reason for the slower times could very well be because of mistakes in the implementation, however, it could also be that the shared memory described by Goswami *et al.* [13] no longer is a feasible solution for newer hardware where it seems to be worth more to run more particles in parallel than utilize memory efficiently for few particles at a time.

Chapter 9

Conclusions and Future Work

In summation, this thesis confirms that IISPH can be implemented as a parallel solution on the GPU using CUDA. As a proof-of-concept a global memory approach using SoA for data storage was proposed and implemented. An experimental approach using shared memory was investigated but the result from it was inferior to the global memory approach.

The proposed implementation achieved a positive speed-up compared to the sequential CPU-version as well as the parallel one. By using CUDA, a real-time performance was achieved for approximately 175 000 particles. This was up to 4 times the amount of particles than that of using OpenMP and 19 times that of the sequential CPU-implementation.

We can conclude that the pressure solve procedure is the most time consuming section of the three procedures in the algorithm. The three most computational expensive kernels were the `computeDisplacementFactor`, `predictDensity` and `computePressure`, the latter two are found in the pressure solve procedure.

While the proposed solution did achieve a speed-up it however did alter the physical accuracy according to the PSNR. Why the visual outcome of the GPU-version was differing from the CPU-versions is not known. Even though the individual movements of the particles have been altered in the CUDA-solution, the overall behavior of the fluid is maintained. Due to this, the proposed solution is believed to maintain the nature of IISPH.

The memory usage differs, however, if the extra memory used to store precomputed values in the CPU-implementation is neglected, the deviation is minimal. As such our hypothesis stating that the memory usage is similar between implementations is supported.

We believe that the implementation can be seen as a stepping stone for future GPGPU-implementations, such as a shared memory implementation. We do not believe that the proposed implementation is sufficient to be fully utilized in video games and virtual reality applications which demand real-time computations. However, with improved implementations and hardware, we believe there could be potential for GPGPU-implementations of IISPH in these types of real-time applications.

9.1 Future work

While the proposed solution did achieve a positive improvement of performance, it accesses spatially closely located particles' data from global memory multiple times. Investigations on an approach to store this data in shared memory with only a single access from global memory would be worthwhile and probably the most viable step to improve the performance.

Further analysis is required to identify to why the CUDA-implementation differs in physical accuracy from the sequential CPU-version, whereas the OpenMP-implementation does not. A data comparison method could be utilized to verify if the difference of the computed values is within an acceptable range. Alternatively, it may be viable to investigate whether the differences are acceptable or not. If the deviation is deemed to be acceptable then it may not be necessary to make any further investigations.

The implementation was limited to compute capability 3.0. A viable step would be to improve by taking advantage of latest feature-sets to allow for better kernel launches and optimizations. This is however likely to be easy modifications which would only yield minor performance improvements.

It would also be viable to exclude the inactive boundary particles from the computations. Since there is no need to compute pressure values for boundary particles which will not affect fluid particles in a given frame, it would be viable to exclude them completely from the simulation of that frame.

Bibliography

Books and Journals

[1] J. J. Monaghan, *Smoothed particle hydrodynamics*, English, 2005.

[2] K. T. Claypool and M. Claypool, “On frame rate and player performance in first person shooter games,” English, *Multimedia Systems*, vol. 13, no. 1, pp. 3–17, 2007.

[3] M. Ihmsen, J. Cornelis, B. Solenthaler, C. Horvath, and M. Teschner, “Implicit incompressible SPH,” English, *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 3, p. 435, 2014.

[4] S. Premože, T. Tasdizen, J. Bigler, A. Lefohn, and R. T. Whitaker, “Particle-Based Simulation of Fluids,” English, *Computer Graphics Forum*, vol. 22, no. 3, p. 410, 2003.

[5] F. Losasso, J. O. Talton, N. Kwatra, and R. Fedkiw, “Two-Way Coupled SPH and Particle Level Set Fluid Simulation,” English, *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 4, p. 804, 2008.

[6] B. Solenthaler and R. Pajarola, “Predictive-corrective incompressible SPH,” English, *ACM Transactions on Graphics*, vol. 28, no. 3, p. 6, 2009.

[7] M. Becker and M. Teschner, “Weakly compressible SPH for free surface flows,” English, ser. SCA ’07, Eurographics Association, 2007, p. 217.

[8] D. Enright, S. Marschner, and R. Fedkiw, “Animation and rendering of complex water surfaces,” English, ser. SIGGRAPH ’02, vol. 21, ACM, 2002, ch. 3, pp. 736–744, ISBN: 0730-0301.

[9] S. J. Cummins and M. Rudman, “An SPH Projection Method,” English, *Journal of Computational Physics*, vol. 152, no. 2, pp. 584–607, 1999.

[10] S. Shao, “Incompressible SPH simulation of wave breaking and overtopping with turbulence modelling,” English, *International Journal for Numerical Methods in Fluids*, vol. 50, no. 5, pp. 597–621, 2006.

[11] X. He, N. Liu, S. Li, H. Wang, and G. Wang, “Local Poisson SPH For Viscous Incompressible Fluids,” English, *Computer Graphics Forum*, vol. 31, no. 6, pp. 1948–1958, 2012.

[12] J. M. Domínguez, A. J. Crespo, and M. Gómez-Gesteira, "Optimization strategies for CPU and GPU implementations of a smoothed particle hydrodynamics method," *Computer Physics Communications*, vol. 184, no. 3, pp. 617–627, 2013, ISSN: 0010-4655. DOI: `http://dx.doi.org/10.1016/j.cpc.2012.10.015`.

[13] P. Goswami, P. Schlegel, B. Solenthaler, and R. Pajarola, "Interactive SPH Simulation and Rendering on the GPU," in *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA '10, Madrid, Spain: Eurographics Association, 2010, pp. 55–64.

[14] F. Thaler, B. Solenthaler, and M. Gross, "A Parallel Architecture for IISPH Fluids," in *Workshop on Virtual Reality Interaction and Physical Simulation*, J. Bender, C. Duriez, F. Jaillet, and G. Zachmann, Eds., The Eurographics Association, 2014, ISBN: 978-3-905674-71-2. DOI: `10.2312/vriphys.20141230`.

[15] X. Nie, L. Chen, and T. Xiang, "Real-Time Incompressible Fluid Simulation on the GPU," English, *International Journal of Computer Games Technology*, vol. 2015, pp. 1–12, 2015.

[16] D. A. Fulk and D. W. Quinn, "An Analysis of 1-D Smoothed Particle Hydrodynamics Kernels," English, *Journal of Computational Physics*, vol. 126, no. 1, pp. 165–180, 1996.

[17] H. Anton and C. Rorres, *Elementary Linear Algebra with Supplemental Applications*. Wiley, 2011, ISBN: 9780470561577.

[18] A. Horé and D. Ziou, "Is there a relationship between peak-signal-to-noise ratio and structural similarity index measure?" English, *IET Image Processing*, vol. 7, no. 1, pp. 12–24, 2013.

[19] R. Dosselmann and X. D. Yang, "A comprehensive assessment of the structural similarity index," English, *Signal, Image and Video Processing*, vol. 5, no. 1, pp. 81–91, 2011.

[20] P. Goswami and R. Pajarola, "Time adaptive approximate sph," English, in. 2011, pp. 19–28, ISBN: 3905673878; 9783905673876.

Web Resources

[21] Parallel Programming and Computing Platform, NVIDIA® Corporation, [Online]. Available: `http://www.nvidia.com/object/cuda_home_new.html` (visited on Apr. 14, 2015).

[22] J. Hoberock and N. Bell. Thrust, [Online]. Available: `https://github.com/thrust/thrust` (visited on Apr. 14, 2015).

[23] Compute capabilities, NVIDIA® Corporation, [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities (visited on Apr. 14, 2015).

[24] Particle demo, NVIDIA® Corporation, [Online]. Available: http://docs.nvidia.com/cuda/cuda-samples/#particles (visited on Jun. 4, 2015).

[25] Thread hierarchy, NVIDIA® Corporation, [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy (visited on Apr. 14, 2015).

[26] CUDA Library Graphics Interoperability, NVIDIA® Corporation, [Online]. Available: http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/group__CUDART__INTEROP.html (visited on Apr. 14, 2015).

[27] Acquiring high-resolution time stamps, Microsoft®, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/dn553408(v=vs.85).aspx (visited on Apr. 14, 2015).

[28] CUDA Library cudaEventElapsedTime, NVIDIA® Corporation, [Online]. Available: http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/group__CUDART__EVENT_g14c387cc57ce2e328f6669854e6020a5.html (visited on Apr. 14, 2015).

[29] ID3D11Query interface (windows), Microsoft®, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476578(v=vs.85).aspx (visited on Apr. 14, 2015).

[30] D3D11_QUERY enumeration (windows), Microsoft®, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476191(v=vs.85).aspx (visited on Apr. 14, 2015).

Appendix A

Additional Experiments Data

Following is the partial relevant data that was gathered during the experiments.

Table A1 (Setup 1 [CUDA & OpenMP]) contains data of execution times on computer setup 1 for physics and rendering using the proposed CUDA-solution and OpenMP.

Table A2 (Setup 2 [CUDA & OpenMP]) contains data of execution times on computer setup 2 for physics and rendering using the proposed CUDA-solution and OpenMP.

Table A3 (Setup 1 & 2 [single-thread CPU]) contains data of execution times on computer setups 1 and 2 for physics and rendering using the sequential CPU-implementation.

Table A4 (Memory usage [VRAM & RAM]) contains data of memory usage on computer setup 1 for physics and rendering using the proposed CUDA-solution and the CPU-implementations.

Setup 1 (CUDA & OpenMP)								
Scene	Frames	Particles	Spacing	Time-step	Time: physics [GPU] (ms)	Time: rendering [GPU] (ms)	Time: physics [CPU] (ms)	Time: rendering [CPU] (ms)
SIMPLE	300	4288	0.09	0.0035	2.248935	13.469247	4.268302	12.184612
SIMPLE	300	5650	0.09	0.0035	2.472281	13.179368	5.780956	10.669256
SIMPLE	300	7948	0.09	0.0035	2.593857	13.063757	9.214053	7.375593
SIMPLE	300	9776	0.09	0.0035	2.523607	13.182771	11.220481	5.857633
SIMPLE	300	15554	0.09	0.0035	3.344905	12.248573	17.094330	2.300213
SIMPLE	300	20972	0.09	0.0035	4.198549	11.455010	25.037664	3.386557
SIMPLE	300	32568	0.09	0.0035	6.276952	9.340478	39.546911	5.170774
SIMPLE	300	43825	0.09	0.0035	8.004032	7.670774	54.568949	7.689797
SIMPLE	300	57362	0.09	0.0035	10.264237	5.764541	64.831025	11.989330
SIMPLE	300	71890	0.09	0.0035	12.850800	3.763618	91.233588	13.812187
SIMPLE	300	89723	0.09	0.0035	17.900671	1.806938	116.450923	18.156967
SIMPLE	300	110754	0.09	0.0035	21.177983	2.177495	147.114191	23.174895
SIMPLE	300	132478	0.09	0.0035	28.837132	1.714149	175.204491	28.679991
SIMPLE	300	159523	0.09	0.0035	29.745489	1.306056	207.868697	35.992826
SIMPLE	300	189042	0.09	0.0035	36.415105	0.150745	249.727931	44.783583
SIMPLE	1000	4288	0.09	0.0035	2.242683	14.163314	17.889968	10.578171
SIMPLE	1000	5650	0.09	0.0035	2.341580	13.416295	18.269835	10.220225
SIMPLE	1000	7948	0.09	0.0035	2.483381	13.259445	7.986275	8.677942
SIMPLE	1000	9776	0.09	0.0035	2.570378	13.163953	16.954982	6.653696
SIMPLE	1000	15554	0.09	0.0035	3.276622	12.437155	17.662175	2.195612
SIMPLE	1000	20972	0.09	0.0035	4.287971	11.423180	25.764662	3.378650
SIMPLE	1000	32568	0.09	0.0035	6.204666	9.505436	42.499844	5.213715
SIMPLE	1000	43825	0.09	0.0035	7.509002	8.230111	54.358247	7.637247
SIMPLE	1000	57362	0.09	0.0035	11.551832	8.258636	51.805731	14.263311
SIMPLE	1000	71890	0.09	0.0035	17.656177	7.340355	89.420650	13.410000
SIMPLE	1000	89723	0.09	0.0035	23.635443	5.345737	111.417195	17.794919
SIMPLE	1000	110754	0.09	0.0035	31.996238	5.541827	139.359253	22.875106
SIMPLE	1000	132478	0.09	0.0035	31.418370	3.348619	167.868256	28.186616
SIMPLE	1000	159523	0.09	0.0035	40.680696	1.777909	203.634635	35.487840
SIMPLE	1000	189042	0.09	0.0035	48.974512	0.154148	244.242066	43.716657
BREAKING_DAM	300	3965	0.09	0.0035	2.201143	17.238369	3.454120	12.961759
BREAKING_DAM	300	5225	0.09	0.0035	2.341406	13.357940	4.623344	11.816158
BREAKING_DAM	300	7173	0.09	0.0035	2.438271	13.251337	6.425576	10.019831
BREAKING_DAM	300	9055	0.09	0.0035	2.548236	13.126017	8.365140	8.141726
BREAKING_DAM	300	14089	0.09	0.0035	3.051888	12.614225	14.487726	2.544145
BREAKING_DAM	300	19487	0.09	0.0035	3.967958	11.678625	21.993941	2.827219
BREAKING_DAM	300	31054	0.09	0.0035	5.944228	9.734457	37.289243	4.882897
BREAKING_DAM	300	41259	0.09	0.0035	7.389714	8.295574	50.292241	6.931067
BREAKING_DAM	300	53539	0.09	0.0035	9.471145	6.488437	65.908352	9.329675
BREAKING_DAM	300	68416	0.09	0.0035	12.837727	4.020064	84.932637	12.495220
BREAKING_DAM	300	83822	0.09	0.0035	15.575995	1.244797	105.082205	16.127214
BREAKING_DAM	300	103463	0.09	0.0035	20.223377	0.194287	131.268777	20.987790
BREAKING_DAM	300	125836	0.09	0.0035	22.630269	0.142737	160.728293	26.476872
BREAKING_DAM	300	149168	0.09	0.0035	29.653967	0.730201	192.511343	32.838191
BREAKING_DAM	300	176699	0.09	0.0035	32.961099	0.153147	231.291419	39.692685
BREAKING_DAM	1000	3965	0.09	0.0035	2.254822	14.080434	3.519283	12.983380
BREAKING_DAM	1000	5225	0.09	0.0035	2.352432	13.409789	4.853065	11.767811
BREAKING_DAM	1000	7173	0.09	0.0035	2.449202	13.300184	6.371324	10.127535
BREAKING_DAM	1000	9055	0.09	0.0035	2.562108	13.182371	8.318996	8.210492
BREAKING_DAM	1000	14089	0.09	0.0035	3.059666	12.670379	14.459799	2.520923
BREAKING_DAM	1000	19487	0.09	0.0035	3.987563	11.722968	21.949598	2.817410
BREAKING_DAM	1000	31054	0.09	0.0035	6.044994	9.671596	37.011075	4.737858
BREAKING_DAM	1000	41259	0.09	0.0035	7.679974	8.098284	49.870636	6.861600
BREAKING_DAM	1000	53539	0.09	0.0035	16.728078	10.981454	58.012977	10.930911
BREAKING_DAM	1000	68416	0.09	0.0035	15.875887	10.716800	84.366994	12.476402
BREAKING_DAM	1000	83822	0.09	0.0035	14.543119	2.880770	103.688565	16.059449
BREAKING_DAM	1000	103463	0.09	0.0035	33.800248	4.346278	129.807072	20.716530
BREAKING_DAM	1000	125836	0.09	0.0035	44.351176	3.840591	158.790029	26.317518
BREAKING_DAM	1000	149168	0.09	0.0035	28.351539	1.042401	191.421595	31.887577
BREAKING_DAM	1000	176699	0.09	0.0035	33.355676	0.254645	228.588220	39.077294
TWO_BLOCKS	300	4162	0.09	0.0035	2.230546	15.062879	3.686343	12.730236
TWO_BLOCKS	300	5418	0.09	0.0035	2.383969	13.295379	4.870081	11.641590
TWO_BLOCKS	300	7498	0.09	0.0035	2.462652	13.202590	6.692833	9.728651
TWO_BLOCKS	300	9456	0.09	0.0035	2.759705	12.894594	8.622288	7.812309
TWO_BLOCKS	300	14666	0.09	0.0035	3.133483	12.504219	14.657088	2.505509

TWO_BLOCKS	300	20272	0.09	0.0035	4.120137	11.507461	22.418148	3.061745
TWO_BLOCKS	300	31922	0.09	0.0035	6.024593	9.589918	37.091853	4.900414
TWO_BLOCKS	300	42704	0.09	0.0035	8.201786	7.629633	50.539779	7.160187
TWO_BLOCKS	300	55304	0.09	0.0035	13.969603	11.345505	65.918661	9.835862
TWO_BLOCKS	300	70004	0.09	0.0035	18.661212	14.277523	83.877223	12.630450
TWO_BLOCKS	300	85698	0.09	0.0035	15.121357	8.600767	103.232327	16.302583
TWO_BLOCKS	300	106380	0.09	0.0035	18.145589	0.140635	130.130482	21.161258
TWO_BLOCKS	300	128360	0.09	0.0035	33.812315	3.479748	157.958730	26.807490
TWO_BLOCKS	300	152052	0.09	0.0035	36.209479	3.109489	189.378329	32.483950
TWO_BLOCKS	300	181056	0.09	0.0035	56.472420	0.471453	227.167754	40.070950
TWO_BLOCKS	1000	4162	0.09	0.0035	3.355358	13.683052	3.795848	12.727834
TWO_BLOCKS	1000	5418	0.09	0.0035	2.395146	13.343426	4.940949	11.598348
TWO_BLOCKS	1000	7498	0.09	0.0035	2.487922	13.243630	6.990219	9.645171
TWO_BLOCKS	1000	9456	0.09	0.0035	2.608105	13.116808	8.848005	7.694396
TWO_BLOCKS	1000	14666	0.09	0.0035	3.167942	12.546159	15.061477	2.348658
TWO_BLOCKS	1000	20272	0.09	0.0035	4.186237	11.513967	22.909420	3.087069
TWO_BLOCKS	1000	31922	0.09	0.0035	6.209261	9.493725	37.559602	4.949861
TWO_BLOCKS	1000	42704	0.09	0.0035	8.626104	7.318134	51.148864	7.283706
TWO_BLOCKS	1000	55304	0.09	0.0035	13.069616	11.173338	66.035673	9.788416
TWO_BLOCKS	1000	70004	0.09	0.0035	21.177088	10.779363	84.567887	12.844857
TWO_BLOCKS	1000	85698	0.09	0.0035	22.782838	9.022175	104.078841	16.449924
TWO_BLOCKS	1000	106380	0.09	0.0035	31.129338	8.721983	131.176487	21.667845
TWO_BLOCKS	1000	128360	0.09	0.0035	42.003998	3.551714	158.621367	26.786769
TWO_BLOCKS	1000	152052	0.09	0.0035	58.951295	2.833925	189.778515	32.612674
TWO_BLOCKS	1000	181056	0.09	0.0035	32.476677	0.354641	228.138787	40.482345
GALLERY	300	4450	0.09	0.0035	3.190760	15.052569	3.922270	12.522937
GALLERY	300	5626	0.09	0.0035	2.398090	13.294379	4.846358	11.611861
GALLERY	300	7802	0.09	0.0035	3.205151	12.481297	6.857592	9.594022
GALLERY	300	9592	0.09	0.0035	2.605574	13.067560	8.572540	7.973664
GALLERY	300	14598	0.09	0.0035	3.129832	12.534748	14.188038	2.589689
GALLERY	300	19376	0.09	0.0035	3.882021	11.768512	20.312624	2.427435
GALLERY	300	29170	0.09	0.0035	5.364282	10.316116	32.689521	3.752810
GALLERY	300	38416	0.09	0.0035	6.675506	8.977029	43.945540	5.161565
GALLERY	300	49410	0.09	0.0035	8.308464	7.505314	56.819815	7.147275
GALLERY	300	61388	0.09	0.0035	11.924198	7.673377	70.931680	8.821386
GALLERY	300	74385	0.09	0.0035	13.199854	5.118321	87.070392	11.517479
GALLERY	300	91618	0.09	0.0035	19.848427	3.764120	108.530720	14.783822
GALLERY	300	108848	0.09	0.0035	18.541126	2.280794	129.645316	18.108120
GALLERY	300	128099	0.09	0.0035	39.742483	4.216252	153.990116	22.286539
GALLERY	300	151586	0.09	0.0035	25.749033	2.771464	184.459802	27.649299
GALLERY	1000	4450	0.09	0.0035	3.350924	12.418036	4.004549	12.497312
GALLERY	1000	5626	0.09	0.0035	2.418512	13.334817	5.311905	11.457813
GALLERY	1000	7802	0.09	0.0035	3.303592	12.442659	7.037364	9.583311
GALLERY	1000	9592	0.09	0.0035	2.629064	13.105397	8.605672	7.958850
GALLERY	1000	14598	0.09	0.0035	3.140309	12.585597	14.391533	2.465570
GALLERY	1000	19376	0.09	0.0035	3.943328	11.771815	20.784378	2.314927
GALLERY	1000	29170	0.09	0.0035	5.460632	10.266568	32.760790	3.558623
GALLERY	1000	38416	0.09	0.0035	6.684185	9.070218	43.859358	5.002512
GALLERY	1000	49410	0.09	0.0035	13.673903	7.258576	56.677478	6.847186
GALLERY	1000	61388	0.09	0.0035	12.176595	6.633175	71.022267	8.706575
GALLERY	1000	74385	0.09	0.0035	26.266393	9.161805	86.401850	11.259331
GALLERY	1000	91618	0.09	0.0035	23.890609	8.039227	107.876491	14.612156
GALLERY	1000	108848	0.09	0.0035	29.158459	6.739578	128.798301	17.872693
GALLERY	1000	128099	0.09	0.0035	44.064187	5.161362	153.058320	22.214370
GALLERY	1000	151586	0.09	0.0035	25.406326	1.101259	183.138934	27.225391

Setup 2 (CUDA & OpenMP)								
Scene	Frames	Particles	Spacing	Time-step	Time: physics [GPU] (ms)	Time: rendering [GPU] (ms)	Time: physics [CPU] (ms)	Time: rendering [CPU] (ms)
SIMPLE	300	4288	0.09	0.0035	3.357967	12.05651	3.026798	13.099016
SIMPLE	300	5650	0.09	0.0035	3.621356	11.791873	3.990074	12.288427
SIMPLE	300	7948	0.09	0.0035	4.662431	10.572622	5.751108	10.554658
SIMPLE	300	9776	0.09	0.0035	5.634961	9.739258	7.204202	9.136849
SIMPLE	300	15554	0.09	0.0035	7.111207	8.319203	12.411283	3.931052
SIMPLE	300	20972	0.09	0.0035	9.106135	6.458088	17.441939	5.107641
SIMPLE	300	32568	0.09	0.0035	13.50496	2.042992	27.863477	8.521931
SIMPLE	300	43825	0.09	0.0035	18.672306	0.328791	38.249731	12.513929
SIMPLE	300	57362	0.09	0.0035	24.312019	0.389096	50.023962	17.401201
SIMPLE	300	71890	0.09	0.0035	33.483269	0.475704	63.999325	22.328248
SIMPLE	300	89723	0.09	0.0035	40.686855	0.520612	80.91167	29.820824
SIMPLE	300	110754	0.09	0.0035	52.713871	0.550123	101.234777	38.091911
SIMPLE	300	132478	0.09	0.0035	60.075089	0.523178	122.227333	46.754661
SIMPLE	300	159523	0.09	0.0035	73.461044	0.88565	149.104439	58.666823
SIMPLE	300	189042	0.09	0.0035	94.957253	1.001127	177.580268	71.636569
SIMPLE	1000	4288	0.09	0.0035	3.311753	12.188667	3.098009	12.997973
SIMPLE	1000	5650	0.09	0.0035	3.728399	11.7614	4.074437	12.294201
SIMPLE	1000	7948	0.09	0.0035	4.651156	10.816087	5.811413	10.554979
SIMPLE	1000	9776	0.09	0.0035	5.800931	9.688897	7.314227	9.061789
SIMPLE	1000	15554	0.09	0.0035	7.147046	8.412227	12.656993	3.625357
SIMPLE	1000	20972	0.09	0.0035	9.159836	6.337478	17.893264	5.090961
SIMPLE	1000	32568	0.09	0.0035	13.316824	2.285175	27.752811	8.35513
SIMPLE	1000	43825	0.09	0.0035	18.580275	0.340338	38.233051	12.432774
SIMPLE	1000	57362	0.09	0.0035	24.739269	0.390699	49.980657	17.488451
SIMPLE	1000	71890	0.09	0.0035	34.157181	0.425343	64.081122	22.157277
SIMPLE	1000	89723	0.09	0.0035	40.550907	0.50746	81.104454	29.435898
SIMPLE	1000	110754	0.09	0.0035	48.952339	0.535367	101.724916	37.814765
SIMPLE	1000	132478	0.09	0.0035	58.485191	0.556217	122.308809	46.285693
SIMPLE	1000	159523	0.09	0.0035	71.10099	0.961352	149.214784	58.184383
SIMPLE	1000	189042	0.09	0.0035	85.640785	0.98573	177.775939	71.30874
BREAKING_DAM	300	3965	0.09	0.0035	3.313157	12.217216	2.845883	13.226363
BREAKING_DAM	300	5225	0.09	0.0035	3.648968	11.810157	3.725117	12.542478
BREAKING_DAM	300	7173	0.09	0.0035	4.251536	11.215127	4.945652	11.399891
BREAKING_DAM	300	9055	0.09	0.0035	5.042572	10.361233	6.531545	9.850245
BREAKING_DAM	300	14089	0.09	0.0035	6.521958	8.944066	10.986417	5.337634
BREAKING_DAM	300	19487	0.09	0.0035	10.128682	6.536677	16.129022	4.418624
BREAKING_DAM	300	31054	0.09	0.0035	12.904405	2.676836	26.561787	8.395226
BREAKING_DAM	300	41259	0.09	0.0035	17.486607	0.338414	36.245552	11.335737
BREAKING_DAM	300	53539	0.09	0.0035	25.360052	0.397115	47.30318	15.363662
BREAKING_DAM	300	68416	0.09	0.0035	29.985325	0.400964	61.416795	20.633934
BREAKING_DAM	300	83822	0.09	0.0035	36.886059	0.506498	76.400985	26.813593
BREAKING_DAM	300	103463	0.09	0.0035	52.243568	0.546915	95.240524	34.395664
BREAKING_DAM	300	125836	0.09	0.0035	59.358273	0.567444	117.161713	43.31086
BREAKING_DAM	300	149168	0.09	0.0035	70.370201	0.962314	140.224848	53.171049
BREAKING_DAM	300	176699	0.09	0.0035	92.228828	1.003693	168.025454	65.092834
BREAKING_DAM	1000	3965	0.09	0.0035	3.307386	12.253784	2.79905	13.326122
BREAKING_DAM	1000	5225	0.09	0.0035	3.603721	11.909596	3.698172	12.689391
BREAKING_DAM	1000	7173	0.09	0.0035	4.309541	11.242071	4.937632	11.473989
BREAKING_DAM	1000	9055	0.09	0.0035	5.075775	10.463559	6.508449	9.903814
BREAKING_DAM	1000	14089	0.09	0.0035	6.49544	9.100923	10.881845	5.475887
BREAKING_DAM	1000	19487	0.09	0.0035	11.00409	6.370839	16.110417	4.39168
BREAKING_DAM	1000	31054	0.09	0.0035	12.8311	2.795522	26.351682	7.715191
BREAKING_DAM	1000	41259	0.09	0.0035	17.307695	0.327828	35.78717	11.201013
BREAKING_DAM	1000	53539	0.09	0.0035	25.41254	0.384284	46.938783	15.174407
BREAKING_DAM	1000	68416	0.09	0.0035	29.218731	0.397756	60.802198	20.402979
BREAKING_DAM	1000	83822	0.09	0.0035	35.877808	0.51163	76.21173	26.463952
BREAKING_DAM	1000	103463	0.09	0.0035	45.702606	0.54467	94.351025	33.917714
BREAKING_DAM	1000	125836	0.09	0.0035	56.870388	0.543066	116.323538	42.793777
BREAKING_DAM	1000	149168	0.09	0.0035	65.036209	0.693828	139.409768	52.423652
BREAKING_DAM	1000	176699	0.09	0.0035	80.91629	0.881479	167.42497	64.2383
TWO_BLOCKS	300	4162	0.09	0.0035	3.370628	12.104946	2.94853	13.107677
TWO_BLOCKS	300	5418	0.09	0.0035	3.697771	11.724511	3.779969	12.486985
TWO_BLOCKS	300	7498	0.09	0.0035	4.65298	10.792671	5.156398	11.152576
TWO_BLOCKS	300	9456	0.09	0.0035	5.419012	10.043991	6.776935	9.563796
TWO_BLOCKS	300	14666	0.09	0.0035	6.924841	8.637088	11.144557	5.124321

TWO_BLOCKS	300	20272	0.09	0.0035	8.846799	6.653117	16.337202	4.672996
TWO_BLOCKS	300	31922	0.09	0.0035	12.969267	2.564566	26.296189	8.013508
TWO_BLOCKS	300	42704	0.09	0.0035	19.464207	0.326545	35.902006	11.696925
TWO_BLOCKS	300	55304	0.09	0.0035	22.398964	0.387812	47.07447	15.693415
TWO_BLOCKS	300	70004	0.09	0.0035	33.236233	0.397756	60.299549	20.820302
TWO_BLOCKS	300	85698	0.09	0.0035	38.129269	0.505215	74.569701	26.803649
TWO_BLOCKS	300	106380	0.09	0.0035	48.756798	0.536971	94.241963	34.76423
TWO_BLOCKS	300	128360	0.09	0.0035	54.971565	0.559425	114.619921	43.125134
TWO_BLOCKS	300	152052	0.09	0.0035	67.040535	0.895914	137.482895	53.168162
TWO_BLOCKS	300	181056	0.09	0.0035	80.524124	0.993749	165.434584	65.285297
TWO_BLOCKS	1000	4162	0.09	0.0035	3.604253	11.881689	2.97772	13.136867
TWO_BLOCKS	1000	5418	0.09	0.0035	3.750336	11.76974	3.83867	12.540874
TWO_BLOCKS	1000	7498	0.09	0.0035	8.640626	6.961058	5.320313	11.076553
TWO_BLOCKS	1000	9456	0.09	0.0035	5.755675	9.751768	6.899149	9.844792
TWO_BLOCKS	1000	14666	0.09	0.0035	6.968998	8.615596	11.567333	4.764416
TWO_BLOCKS	1000	20272	0.09	0.0035	9.169113	6.463541	16.654445	4.729452
TWO_BLOCKS	1000	31922	0.09	0.0035	13.066149	2.539867	26.663793	8.10589
TWO_BLOCKS	1000	42704	0.09	0.0035	20.338459	0.345471	36.454053	11.819139
TWO_BLOCKS	1000	55304	0.09	0.0035	23.398626	0.400964	47.457791	15.929181
TWO_BLOCKS	1000	70004	0.09	0.0035	31.196497	0.406096	60.917675	21.072749
TWO_BLOCKS	1000	85698	0.09	0.0035	38.008766	0.510668	75.099295	27.123137
TWO_BLOCKS	1000	106380	0.09	0.0035	48.919518	0.542424	94.657683	35.16359
TWO_BLOCKS	1000	128360	0.09	0.0035	55.035233	0.558463	115.345185	43.730429
TWO_BLOCKS	1000	152052	0.09	0.0035	65.253983	0.826628	138.05066	53.491178
TWO_BLOCKS	1000	181056	0.09	0.0035	84.76133	0.99407	166.550868	66.093962
GALLERY	300	4450	0.09	0.0035	3.60334	11.948089	3.103783	12.990916
GALLERY	300	5626	0.09	0.0035	3.758291	11.756909	3.809159	12.473192
GALLERY	300	7802	0.09	0.0035	4.736365	10.699647	5.493208	10.876713
GALLERY	300	9592	0.09	0.0035	5.418283	10.01031	6.617191	9.749202
GALLERY	300	14598	0.09	0.0035	7.054163	8.705412	10.853297	5.49385
GALLERY	300	19376	0.09	0.0035	9.724667	6.971964	15.003435	3.491916
GALLERY	300	29170	0.09	0.0035	11.584375	3.956393	23.052549	5.67701
GALLERY	300	38416	0.09	0.0035	15.101686	0.522216	31.196612	8.14727
GALLERY	300	49410	0.09	0.0035	20.64867	0.349961	40.515338	11.084252
GALLERY	300	61388	0.09	0.0035	26.868971	0.416361	50.959972	14.364138
GALLERY	300	74385	0.09	0.0035	30.046797	0.408983	62.452566	18.517485
GALLERY	300	91618	0.09	0.0035	37.557682	0.508743	78.457129	24.18006
GALLERY	300	108848	0.09	0.0035	49.207813	0.553972	93.47147	29.471824
GALLERY	300	128099	0.09	0.0035	53.260437	0.556538	111.110042	36.285969
GALLERY	300	151586	0.09	0.0035	64.533104	0.763756	133.219203	44.943586
GALLERY	1000	4450	0.09	0.0035	3.651572	11.97343	3.090311	13.070788
GALLERY	1000	5626	0.09	0.0035	3.784943	11.743437	3.813329	12.60984
GALLERY	1000	7802	0.09	0.0035	5.264343	10.295154	5.366183	11.074629
GALLERY	1000	9592	0.09	0.0035	5.628757	9.945835	6.711498	9.728352
GALLERY	1000	14598	0.09	0.0035	7.14989	8.709261	10.910073	5.487755
GALLERY	1000	19376	0.09	0.0035	9.30679	6.922244	15.079779	3.470104
GALLERY	1000	29170	0.09	0.0035	11.426493	4.064814	23.105156	5.644612
GALLERY	1000	38416	0.09	0.0035	15.123842	0.584766	31.115136	8.063869
GALLERY	1000	49410	0.09	0.0035	21.011576	0.347075	40.325442	10.993474
GALLERY	1000	61388	0.09	0.0035	26.650871	0.401606	50.705921	14.120994
GALLERY	1000	74385	0.09	0.0035	29.755447	0.40353	61.942219	18.311229
GALLERY	1000	91618	0.09	0.0035	36.924942	0.511309	77.976934	23.868271
GALLERY	1000	108848	0.09	0.0035	50.434681	0.546915	92.99288	29.046482
GALLERY	1000	128099	0.09	0.0035	51.35955	0.55301	110.741155	35.754451
GALLERY	1000	151586	0.09	0.0035	61.874756	0.654373	132.908055	44.353688

Setup 1 & 2 (single-thread CPU)								
Scene	Frames	Particles	Spacing	Time-step	Time: physics [Setup 1] (ms)	Time: rendering [Setup 1] (ms)	Time: physics [Setup 2] (ms)	Time: rendering [Setup 2] (ms)
SIMPLE	300	4288	0.09	0.0035	13.421374	2.979458	15.032616	1.308425
SIMPLE	300	5650	0.09	0.0035	18.467714	0.848914	20.470007	1.145794
SIMPLE	300	7948	0.09	0.0035	26.844249	1.145899	30.990657	1.572741
SIMPLE	300	9776	0.09	0.0035	33.444580	1.455196	39.038161	2.005461
SIMPLE	300	15554	0.09	0.0035	55.441580	2.204715	66.621267	3.491593
SIMPLE	300	20972	0.09	0.0035	77.390233	3.203172	92.171646	4.736185
SIMPLE	300	32568	0.09	0.0035	123.553212	5.164654	145.830553	6.831141
SIMPLE	300	43825	0.09	0.0035	169.728504	7.564556	200.180402	9.580470
SIMPLE	300	57362	0.09	0.0035	223.950013	10.440414	265.578248	12.914885
SIMPLE	300	71890	0.09	0.0035	286.233956	13.410864	335.268331	16.215620
SIMPLE	300	89723	0.09	0.0035	362.147972	17.786661	424.643174	21.276104
SIMPLE	300	110754	0.09	0.0035	451.788155	22.797868	528.956006	27.170915
SIMPLE	300	132478	0.09	0.0035	543.700918	28.045401	638.646243	32.965003
SIMPLE	300	159523	0.09	0.0035	663.541069	35.481434	778.285594	41.332958
SIMPLE	300	189042	0.09	0.0035	793.472399	43.150490	928.716325	50.135879
SIMPLE	1000	4288	0.09	0.0035	13.779317	2.677969	15.468865	0.963917
SIMPLE	1000	5650	0.09	0.0035	19.018743	0.846512	21.106096	1.158946
SIMPLE	1000	7948	0.09	0.0035	27.592867	1.143797	31.067321	1.622139
SIMPLE	1000	9776	0.09	0.0035	34.315115	1.389433	39.622286	1.994555
SIMPLE	1000	15554	0.09	0.0035	56.173382	2.180091	67.508199	3.410117
SIMPLE	1000	20972	0.09	0.0035	78.546942	3.273740	93.281193	4.657917
SIMPLE	1000	32568	0.09	0.0035	123.668523	5.059853	145.912671	6.792969
SIMPLE	1000	43825	0.09	0.0035	170.643481	7.503297	200.097963	9.476540
SIMPLE	1000	57362	0.09	0.0035	223.583061	10.451525	262.206944	12.893394
SIMPLE	1000	71890	0.09	0.0035	286.182306	13.269128	334.519010	16.048177
SIMPLE	1000	89723	0.09	0.0035	361.078046	17.568051	422.498820	21.001845
SIMPLE	1000	110754	0.09	0.0035	450.547065	22.673548	526.265057	26.755516
SIMPLE	1000	132478	0.09	0.0035	540.431982	27.674846	630.921755	32.556661
SIMPLE	1000	159523	0.09	0.0035	660.158624	34.747530	769.646908	40.861104
SIMPLE	1000	189042	0.09	0.0035	788.554882	42.677236	919.122062	49.484072
BREAKING_DAM	300	3965	0.09	0.0035	12.034343	4.354177	13.591712	2.728158
BREAKING_DAM	300	5225	0.09	0.0035	16.638660	0.748918	18.490849	0.976427
BREAKING_DAM	300	7173	0.09	0.0035	23.436881	0.990350	25.887830	1.310350
BREAKING_DAM	300	9055	0.09	0.0035	30.397858	1.214365	34.890912	1.803375
BREAKING_DAM	300	14089	0.09	0.0035	48.972474	1.849174	58.442567	2.538262
BREAKING_DAM	300	19487	0.09	0.0035	71.196792	2.829014	84.928314	4.122550
BREAKING_DAM	300	31054	0.09	0.0035	118.092174	4.737344	138.752098	6.378212
BREAKING_DAM	300	41259	0.09	0.0035	160.305465	6.842663	188.691985	8.727219
BREAKING_DAM	300	53539	0.09	0.0035	210.057187	9.247670	246.538238	11.497398
BREAKING_DAM	300	68416	0.09	0.0035	273.574413	12.389284	320.311417	15.015294
BREAKING_DAM	300	83822	0.09	0.0035	338.122527	15.937487	396.485247	19.134957
BREAKING_DAM	300	103463	0.09	0.0035	422.917863	20.525588	495.162451	24.509477
BREAKING_DAM	300	125836	0.09	0.0035	518.772206	25.914858	606.823186	30.590334
BREAKING_DAM	300	149168	0.09	0.0035	620.897465	31.885184	725.726931	37.489479
BREAKING_DAM	300	176699	0.09	0.0035	743.496662	39.019427	868.640072	45.434337
BREAKING_DAM	1000	3965	0.09	0.0035	12.130736	4.322947	13.633092	2.739064
BREAKING_DAM	1000	5225	0.09	0.0035	16.778894	0.748318	18.611780	0.964559
BREAKING_DAM	1000	7173	0.09	0.0035	23.583121	0.974735	26.163052	1.297519
BREAKING_DAM	1000	9055	0.09	0.0035	30.522477	1.210761	35.143680	1.786695
BREAKING_DAM	1000	14089	0.09	0.0035	49.107304	1.839565	58.457001	2.534412
BREAKING_DAM	1000	19487	0.09	0.0035	71.346636	2.802588	84.812195	4.107794
BREAKING_DAM	1000	31054	0.09	0.0035	117.226744	4.667677	137.935415	6.297378
BREAKING_DAM	1000	41259	0.09	0.0035	158.982396	6.767291	187.900964	8.639328
BREAKING_DAM	1000	53539	0.09	0.0035	208.205911	9.147074	244.409281	11.367806
BREAKING_DAM	1000	68416	0.09	0.0035	270.840891	12.203105	316.887507	14.834700
BREAKING_DAM	1000	83822	0.09	0.0035	334.127195	15.692152	390.771352	18.840490
BREAKING_DAM	1000	103463	0.09	0.0035	418.001247	20.200376	488.557775	24.073870
BREAKING_DAM	1000	125836	0.09	0.0035	512.669753	25.463525	598.562047	30.118480
BREAKING_DAM	1000	149168	0.09	0.0035	615.374868	31.309232	718.014953	36.773197
BREAKING_DAM	1000	176699	0.09	0.0035	736.086754	38.460591	859.836509	44.752378
TWO_BLOCKS	300	4162	0.09	0.0035	12.709391	3.692642	14.275275	1.951571
TWO_BLOCKS	300	5418	0.09	0.0035	17.123325	0.784352	19.061822	1.028713
TWO_BLOCKS	300	7498	0.09	0.0035	24.355161	1.024583	27.013416	1.398883
TWO_BLOCKS	300	9456	0.09	0.0035	31.401420	1.299647	36.445369	1.895757
TWO_BLOCKS	300	14666	0.09	0.0035	50.047506	1.973193	59.726613	2.657909

TWO_BLOCKS	300	20272	0.09	0.0035	72.303053	2.971650	86.360878	4.298974
TWO_BLOCKS	300	31922	0.09	0.0035	117.936625	4.873375	138.114726	6.511974
TWO_BLOCKS	300	42704	0.09	0.0035	160.552602	7.048661	189.107063	8.957212
TWO_BLOCKS	300	55304	0.09	0.0035	209.975208	9.479192	246.304717	11.756902
TWO_BLOCKS	300	70004	0.09	0.0035	270.231006	12.447840	316.672590	15.112167
TWO_BLOCKS	300	85698	0.09	0.0035	334.038910	15.977726	390.776163	19.230547
TWO_BLOCKS	300	106380	0.09	0.0035	420.793025	20.697653	491.905021	24.710600
TWO_BLOCKS	300	128360	0.09	0.0035	511.516947	25.736486	597.620905	30.403966
TWO_BLOCKS	300	152052	0.09	0.0035	612.311029	31.656665	715.265303	37.141121
TWO_BLOCKS	300	181056	0.09	0.0035	737.142567	39.192393	860.644853	45.465452
TWO_BLOCKS	1000	4162	0.09	0.0035	13.104570	3.348212	14.682976	1.662236
TWO_BLOCKS	1000	5418	0.09	0.0035	17.702881	0.792760	19.573451	1.024543
TWO_BLOCKS	1000	7498	0.09	0.0035	25.171645	1.033892	27.987598	1.404015
TWO_BLOCKS	1000	9456	0.09	0.0035	32.529302	1.294542	37.598541	1.917249
TWO_BLOCKS	1000	14666	0.09	0.0035	51.741130	1.983402	61.738169	2.884694
TWO_BLOCKS	1000	20272	0.09	0.0035	74.629385	2.999277	88.709564	4.318862
TWO_BLOCKS	1000	31922	0.09	0.0035	121.695030	4.952951	141.708003	6.570675
TWO_BLOCKS	1000	42704	0.09	0.0035	164.208309	7.115025	192.939957	9.037084
TWO_BLOCKS	1000	55304	0.09	0.0035	214.218879	9.574384	250.888536	11.864360
TWO_BLOCKS	1000	70004	0.09	0.0035	275.241612	12.570658	321.986805	15.279289
TWO_BLOCKS	1000	85698	0.09	0.0035	339.034202	16.150992	396.563194	19.362705
TWO_BLOCKS	1000	106380	0.09	0.0035	425.905128	20.942388	497.783151	24.935782
TWO_BLOCKS	1000	128360	0.09	0.0035	515.648010	26.032570	603.170245	30.790816
TWO_BLOCKS	1000	152052	0.09	0.0035	616.671511	31.901400	720.014961	37.416343
TWO_BLOCKS	1000	181056	0.09	0.0035	740.340334	39.544631	864.818726	45.988629
GALLERY	300	4450	0.09	0.0035	13.777215	2.639532	15.379690	0.946595
GALLERY	300	5626	0.09	0.0035	17.540725	0.707178	19.450596	0.923500
GALLERY	300	7802	0.09	0.0035	25.045524	0.910774	27.960332	1.256460
GALLERY	300	9592	0.09	0.0035	30.990926	1.123678	35.972552	1.661273
GALLERY	300	14598	0.09	0.0035	48.341569	1.638672	57.921314	2.257266
GALLERY	300	19376	0.09	0.0035	66.193993	2.303810	79.107602	3.511481
GALLERY	300	29170	0.09	0.0035	102.751659	3.616069	121.161332	4.981575
GALLERY	300	38416	0.09	0.0035	138.336693	4.998595	162.519310	6.634829
GALLERY	300	49410	0.09	0.0035	179.777943	6.774798	212.041233	8.670763
GALLERY	300	61388	0.09	0.0035	225.516916	8.662409	266.094368	10.835006
GALLERY	300	74385	0.09	0.0035	276.957458	11.186030	325.040225	13.668377
GALLERY	300	91618	0.09	0.0035	345.975059	14.505113	406.007016	17.465665
GALLERY	300	108848	0.09	0.0035	413.530258	17.698977	484.957119	21.284444
GALLERY	300	128099	0.09	0.0035	492.360072	21.830039	577.376400	25.949418
GALLERY	300	151586	0.09	0.0035	589.539887	26.987486	692.038590	31.839417
GALLERY	1000	4450	0.09	0.0035	14.045072	2.425126	15.622193	0.851647
GALLERY	1000	5626	0.09	0.0035	17.877048	0.706578	19.792859	0.919651
GALLERY	1000	7802	0.09	0.0035	25.506766	0.899062	28.641008	1.254536
GALLERY	1000	9592	0.09	0.0035	31.530244	1.108063	36.592282	1.668010
GALLERY	1000	14598	0.09	0.0035	49.149945	1.637471	58.507362	2.233208
GALLERY	1000	19376	0.09	0.0035	67.099662	2.288495	80.225168	3.494801
GALLERY	1000	29170	0.09	0.0035	103.685255	3.573428	121.392288	4.950460
GALLERY	1000	38416	0.09	0.0035	138.412065	4.907307	164.520922	6.572279
GALLERY	1000	49410	0.09	0.0035	179.722089	6.679907	212.038346	8.547266
GALLERY	1000	61388	0.09	0.0035	225.184797	8.557008	266.687474	10.682960
GALLERY	1000	74385	0.09	0.0035	275.318786	11.023874	323.044387	13.486179
GALLERY	1000	91618	0.09	0.0035	343.955022	14.308124	403.547664	17.229577
GALLERY	1000	108848	0.09	0.0035	409.642429	17.454242	479.304171	20.927747
GALLERY	1000	128099	0.09	0.0035	487.594801	21.507230	571.025133	25.558719
GALLERY	1000	151586	0.09	0.0035	582.602032	26.600415	681.756915	31.397074

Memory usage (VRAM & RAM)								
Scene	Frames	Particles	Spacing	Time-step	VRAM: physics [GPU] (MB)	VRAM: rendering [GPU] (MB)	RAM: physics [CPU] (MB)	VRAM: rendering [CPU] (MB)
SIMPLE	300	4288	0.09	0.0035	1.477272	0.147217	3.163944	0.114502
SIMPLE	300	5650	0.09	0.0035	1.997223	0.193977	4.242290	0.150871
SIMPLE	300	7948	0.09	0.0035	2.769882	0.272873	5.988449	0.212234
SIMPLE	300	9776	0.09	0.0035	3.489441	0.335632	7.482788	0.261047
SIMPLE	300	15554	0.09	0.0035	5.473457	0.534004	12.027813	0.415337
SIMPLE	300	20972	0.09	0.0035	7.539986	0.720016	16.529243	0.560013
SIMPLE	300	32568	0.09	0.0035	11.645233	1.118134	26.009525	0.869659
SIMPLE	300	43825	0.09	0.0035	15.794048	1.504612	35.627861	1.170254
SIMPLE	300	57362	0.09	0.0035	20.814217	1.969368	46.955414	1.531731
SIMPLE	300	71890	0.09	0.0035	26.325989	2.468147	59.716366	1.919670
SIMPLE	300	89723	0.09	0.0035	33.062908	3.080395	75.415771	2.395863
SIMPLE	300	110754	0.09	0.0035	40.501534	3.802437	93.352249	2.957451
SIMPLE	300	132478	0.09	0.0035	48.821823	4.548271	112.362701	3.537544
SIMPLE	300	159523	0.09	0.0035	59.039623	5.476788	136.379303	4.259724
SIMPLE	300	189042	0.09	0.0035	70.293777	6.490242	162.654770	5.047966
SIMPLE	1000	4288	0.09	0.0035	1.477272	0.147217	3.244560	0.114502
SIMPLE	1000	5650	0.09	0.0035	1.997223	0.193977	4.345043	0.150871
SIMPLE	1000	7948	0.09	0.0035	2.769882	0.272873	6.127918	0.212234
SIMPLE	1000	9776	0.09	0.0035	3.489441	0.335632	7.645008	0.261047
SIMPLE	1000	15554	0.09	0.0035	5.473457	0.534004	12.235241	0.415337
SIMPLE	1000	20972	0.09	0.0035	7.539986	0.720016	16.776646	0.560013
SIMPLE	1000	32568	0.09	0.0035	11.645233	1.118134	26.260185	0.869659
SIMPLE	1000	43825	0.09	0.0035	15.794048	1.504612	35.885357	1.170254
SIMPLE	1000	57362	0.09	0.0035	20.814217	1.969368	46.955414	1.531731
SIMPLE	1000	71890	0.09	0.0035	26.325989	2.468147	59.859921	1.919670
SIMPLE	1000	89723	0.09	0.0035	33.062908	3.080395	75.511337	2.395863
SIMPLE	1000	110754	0.09	0.0035	40.501534	3.802437	93.377914	2.957451
SIMPLE	1000	132478	0.09	0.0035	48.821823	4.548271	112.232925	3.537544
SIMPLE	1000	159523	0.09	0.0035	59.039623	5.476788	136.192047	4.259724
SIMPLE	1000	189042	0.09	0.0035	70.293777	6.490242	162.492172	5.047966
BREAKING_DAM	300	3965	0.09	0.0035	1.377468	0.136127	2.917358	0.105877
BREAKING_DAM	300	5225	0.09	0.0035	1.865902	0.179386	3.923534	0.139523
BREAKING_DAM	300	7173	0.09	0.0035	2.530415	0.246265	5.374973	0.191540
BREAKING_DAM	300	9055	0.09	0.0035	3.266659	0.310879	6.959625	0.241795
BREAKING_DAM	300	14089	0.09	0.0035	5.020786	0.483707	10.881569	0.376217
BREAKING_DAM	300	19487	0.09	0.0035	7.081135	0.669033	15.476978	0.520359
BREAKING_DAM	300	31054	0.09	0.0035	11.177422	1.066154	25.109829	0.829231
BREAKING_DAM	300	41259	0.09	0.0035	15.001179	1.416515	34.042011	1.101734
BREAKING_DAM	300	53539	0.09	0.0035	19.632946	1.838116	44.563118	1.429646
BREAKING_DAM	300	68416	0.09	0.0035	25.252556	2.348877	57.635815	1.826904
BREAKING_DAM	300	83822	0.09	0.0035	31.239555	2.877800	71.256691	2.238289
BREAKING_DAM	300	103463	0.09	0.0035	38.248684	3.552120	88.254761	2.762760
BREAKING_DAM	300	125836	0.09	0.0035	46.769508	4.320236	108.197334	3.360184
BREAKING_DAM	300	149168	0.09	0.0035	55.840027	5.121277	128.911102	3.983215
BREAKING_DAM	300	176699	0.09	0.0035	66.479904	6.066479	153.878418	4.718372
BREAKING_DAM	1000	3965	0.09	0.0035	1.377468	0.136127	2.947292	0.105877
BREAKING_DAM	1000	5225	0.09	0.0035	1.865902	0.179386	3.960438	0.139523
BREAKING_DAM	1000	7173	0.09	0.0035	2.530415	0.246265	5.416817	0.191540
BREAKING_DAM	1000	9055	0.09	0.0035	3.266659	0.310879	7.000561	0.241795
BREAKING_DAM	1000	14089	0.09	0.0035	5.020786	0.483707	10.918152	0.376217
BREAKING_DAM	1000	19487	0.09	0.0035	7.081135	0.669033	15.495964	0.520359
BREAKING_DAM	1000	31054	0.09	0.0035	11.177422	1.066154	25.048519	0.829231
BREAKING_DAM	1000	41259	0.09	0.0035	15.001179	1.416515	33.815331	1.101734
BREAKING_DAM	1000	53539	0.09	0.0035	19.632946	1.838116	44.420872	1.429646
BREAKING_DAM	1000	68416	0.09	0.0035	25.252556	2.348877	57.176338	1.826904
BREAKING_DAM	1000	83822	0.09	0.0035	31.239555	2.877800	70.748245	2.238289
BREAKING_DAM	1000	103463	0.09	0.0035	38.248684	3.552120	87.573914	2.762760
BREAKING_DAM	1000	125836	0.09	0.0035	46.769508	4.320236	107.426582	3.360184
BREAKING_DAM	1000	149168	0.09	0.0035	55.840027	5.121277	128.325485	3.983215
BREAKING_DAM	1000	176699	0.09	0.0035	66.479904	6.066479	153.346039	4.718372
TWO_BLOCKS	300	4162	0.09	0.0035	1.438339	0.142891	3.055344	0.111137
TWO_BLOCKS	300	5418	0.09	0.0035	1.925537	0.186012	4.026253	0.144676
TWO_BLOCKS	300	7498	0.09	0.0035	2.630836	0.257423	5.554531	0.200218
TWO_BLOCKS	300	9456	0.09	0.0035	3.390564	0.324646	7.149052	0.252502
TWO_BLOCKS	300	14666	0.09	0.0035	5.199074	0.503517	11.103291	0.391624

TWO_BLOCKS	300	20272	0.09	0.0035	7.323692	0.695984	15.704231	0.541321
TWO_BLOCKS	300	31922	0.09	0.0035	11.445625	1.095955	25.057598	0.852409
TWO_BLOCKS	300	42704	0.09	0.0035	15.447670	1.466125	34.101059	1.140320
TWO_BLOCKS	300	55304	0.09	0.0035	20.178314	1.898712	44.658298	1.476776
TWO_BLOCKS	300	70004	0.09	0.0035	25.743233	2.403397	57.135658	1.869308
TWO_BLOCKS	300	85698	0.09	0.0035	31.819221	2.942207	70.733582	2.288383
TWO_BLOCKS	300	106380	0.09	0.0035	39.150009	3.652267	88.243362	2.840652
TWO_BLOCKS	300	128360	0.09	0.0035	47.549400	4.406891	107.312767	3.427582
TWO_BLOCKS	300	152052	0.09	0.0035	56.731155	5.220291	128.165131	4.060226
TWO_BLOCKS	300	181056	0.09	0.0035	67.826180	6.216064	153.904114	4.834717
TWO_BLOCKS	1000	4162	0.09	0.0035	1.438339	0.142891	3.128616	0.111137
TWO_BLOCKS	1000	5418	0.09	0.0035	1.925537	0.186012	4.127617	0.144676
TWO_BLOCKS	1000	7498	0.09	0.0035	2.630836	0.257423	5.700142	0.200218
TWO_BLOCKS	1000	9456	0.09	0.0035	3.390564	0.324646	7.347347	0.252502
TWO_BLOCKS	1000	14666	0.09	0.0035	5.199074	0.503517	11.400040	0.391624
TWO_BLOCKS	1000	20272	0.09	0.0035	7.323692	0.695984	16.115936	0.541321
TWO_BLOCKS	1000	31922	0.09	0.0035	11.445625	1.095955	25.648907	0.852409
TWO_BLOCKS	1000	42704	0.09	0.0035	15.447670	1.466125	34.829514	1.140320
TWO_BLOCKS	1000	55304	0.09	0.0035	20.178314	1.898712	45.482456	1.476776
TWO_BLOCKS	1000	70004	0.09	0.0035	25.743233	2.403397	58.008976	1.869308
TWO_BLOCKS	1000	85698	0.09	0.0035	31.819221	2.942207	71.651093	2.288383
TWO_BLOCKS	1000	106380	0.09	0.0035	39.150009	3.652267	89.116806	2.840652
TWO_BLOCKS	1000	128360	0.09	0.0035	47.549400	4.406891	108.051849	3.427582
TWO_BLOCKS	1000	152052	0.09	0.0035	56.731155	5.220291	128.819916	4.060226
TWO_BLOCKS	1000	181056	0.09	0.0035	67.826180	6.216064	154.365829	4.834717
GALLERY	300	4450	0.09	0.0035	1.527328	0.152779	3.235279	0.118828
GALLERY	300	5626	0.09	0.0035	1.989807	0.193153	4.088100	0.150230
GALLERY	300	7802	0.09	0.0035	2.724770	0.267860	5.685009	0.208336
GALLERY	300	9592	0.09	0.0035	3.432587	0.329315	7.098858	0.256134
GALLERY	300	14598	0.09	0.0035	5.178062	0.501183	10.817299	0.389809
GALLERY	300	19376	0.09	0.0035	7.046837	0.665222	14.650669	0.517395
GALLERY	300	29170	0.09	0.0035	10.595284	1.001472	22.365108	0.778923
GALLERY	300	38416	0.09	0.0035	14.122719	1.318909	30.079132	1.025818
GALLERY	300	49410	0.09	0.0035	18.357124	1.696358	39.215424	1.319389
GALLERY	300	61388	0.09	0.0035	23.080971	2.107590	49.252178	1.639236
GALLERY	300	74385	0.09	0.0035	28.323612	2.553806	60.487488	1.986294
GALLERY	300	91618	0.09	0.0035	34.588692	3.145454	74.845917	2.446465
GALLERY	300	108848	0.09	0.0035	41.520378	3.737000	89.591202	2.906555
GALLERY	300	128099	0.09	0.0035	49.329906	4.397930	106.517624	3.420612
GALLERY	300	151586	0.09	0.0035	58.720230	5.204292	127.191910	4.047783
GALLERY	1000	4450	0.09	0.0035	1.527328	0.152779	3.305641	0.118828
GALLERY	1000	5626	0.09	0.0035	1.989807	0.193153	4.176140	0.150230
GALLERY	1000	7802	0.09	0.0035	2.724770	0.267860	5.798283	0.208336
GALLERY	1000	9592	0.09	0.0035	3.432587	0.329315	7.235817	0.256134
GALLERY	1000	14598	0.09	0.0035	5.178062	0.501183	10.992577	0.389809
GALLERY	1000	19376	0.09	0.0035	7.046837	0.665222	14.855721	0.517395
GALLERY	1000	29170	0.09	0.0035	10.595284	1.001472	22.549973	0.778923
GALLERY	1000	38416	0.09	0.0035	14.122719	1.318909	30.231258	1.025818
GALLERY	1000	49410	0.09	0.0035	18.357124	1.696358	39.282608	1.319389
GALLERY	1000	61388	0.09	0.0035	23.080971	2.107590	49.190067	1.639236
GALLERY	1000	74385	0.09	0.0035	28.323612	2.553806	60.244865	1.986294
GALLERY	1000	91618	0.09	0.0035	34.588692	3.145454	74.410103	2.446465
GALLERY	1000	108848	0.09	0.0035	41.520378	3.737000	88.891457	2.906555
GALLERY	1000	128099	0.09	0.0035	49.329906	4.397930	105.511383	3.420612
GALLERY	1000	151586	0.09	0.0035	58.720230	5.204292	125.914871	4.047783