Lattice Boltzmann Liquid Simulations on Graphics Hardware

by

Duncan Clough

Thesis presented for the degree of

Master of Science

In the Department of Computer Science

University of Cape Town

February 2014

Supervised by

Assoc Professor J. E. Gain

Assoc Professor M. M. Kuttel

Plagiarism Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the ACM Convention for citation and referencing. Each contribution to, and quotation in, this thesis from the work(s) of other people has been attributed, and has been cited and referenced.
3. This thesis is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.
5. I acknowledge that copying someone else's work, or part of it, is wrong. I know the meaning of plagiarism and declare that all of the work in the thesis, save for that which is properly acknowledged, is my own.

Signature _____

Funding Acknowledgements

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

Scholarship-holder:

_____

Witnesses:

(1) _____ (2) _____

Abstract

Fluid simulation is widely used in the visual effects industry. The high level of detail required to produce realistic visual effects requires significant computation. Usually, expensive computer clusters are used in order to reduce the time required. However, general-purpose Graphics Processing Unit (GPU) computing has potential as a relatively inexpensive way to reduce these simulation times. In recent years, GPUs have been used to achieve enormous speedups via their massively parallel architectures. Within the field of fluid simulation, the Lattice Boltzmann Method (LBM) stands out as a candidate for GPU execution because its grid-based structure is a natural fit for GPU parallelism.

This thesis describes the design and implementation of a GPU-based free-surface LBM fluid simulation. Broadly, our approach is to ensure that the steps that perform most of the work in the LBM (the stream and collide steps) make efficient use of GPU resources. We achieve this by removing complexity from the core stream and collide steps and handling interactions with obstacles and tracking of the fluid interface in separate GPU kernels.

To determine the efficiency of our design, we perform separate, detailed analyses of the performance of the kernels associated with the stream and collide steps of the LBM. We demonstrate that these kernels make efficient use of GPU resources and achieve speedups of 29.6$\times$ and 223.7$\times$, respectively. Our analysis of the overall performance of all kernels shows that significant time is spent performing obstacle adjustment and interface movement as a result of limitations associated with GPU memory accesses. Lastly, we compare our GPU LBM implementation with a single-core CPU LBM implementation. Our results show speedups of up to 81.6$\times$ with no significant differences in output from the simulations on both platforms.

We conclude that order of magnitude speedups are possible using GPUs to perform free-surface LBM fluid simulations, and that GPUs can, therefore, significantly reduce the cost of performing high-detail fluid simulations for visual effects.

Notation

Vectors are denoted in boldface and will refer to 2-dimensional and 3-dimensional coordinate vectors [$\mathbf{a} = (a_1, a_2)$ or $\mathbf{b} = (b_1, b_2, b_3)$]. Vector functions are also denoted in bold face [$\mathbf{f}(\dots) = \mathbf{a}$]. All vectors and vector functions in a single equation are assumed to have the same dimensionality.

For ease of notation the functions of $\mathbf{x}$ and t the parameters may not be explicitly included as parameters to functions, but it is implied that the parameters have the same values for all functions appearing in the equation [e.g. $f_i(\mathbf{x}, t) = f_i(\mathbf{x}) = f_i(t) = f_i$]. Since variables and functions are used consistently and do not overlap this will not diminish the clarity of any equations.

Symbols

$\mathbf{x}$	Coordinates of the current cell in the lattice.
Δt	Change in physical time between each time step.
Δx	Width of a cell in the lattice.
t	Current simulation time step, not simulation time. The next time step is $t + 1$ and total simulation time is $t \times \Delta t$.
i	An arbitrary index used to link functions together by direction.
$\mathbf{e}_i$	The i^{th} direction vector. For example $\mathbf{e}_1 = \langle 1, 0, 0 \rangle$.
$\tilde{\imath}$	Index such that $e_{\tilde{\imath}} = -e_i$.
Q	A constant representing the number of lattice directions.
v	Lattice viscosity.
η	Dynamic viscosity of the fluid [Pa s].
$\mathbf{g}$	Lattice gravity.
$\mathbf{g}_\mathrm{f}$	Gravity [$\mathrm{m/s^2}$].
g_c	Stability constant used during calculation of time step.
C	Smagorinsky Constant.
$\mathbf{u}(\mathbf{x}, t)$	Velocity of the cell at position $\mathbf{x}$ at time step t.
$\overline{\mathbf{u}}(\mathbf{x}, t)$	The average velocity of neighbouring fluid cells to $\mathbf{x}$ at time t.
$\mathbf{u}_0$	Initial lattice velocity of the simulation.
$\mathbf{u}_{0\mathrm{f}}$	Initial fluid velocity of the simulation [m/s].

$\mathbf{u}_\mathrm{s}$	Lattice Velocity with which source cells are initialised.
$\mathbf{u}_\mathrm{sf}$	Velocity with which source cells are initialised [m/s].
$\rho\left(\mathbf{x}, t\right)$	Pressure of the cell at position $\mathbf{x}$ and time step t.
$\overline{\rho}\left(\mathbf{x}, t\right)$	Average pressure of neighbouring fluid cells to $\mathbf{x}$ and time step t.
$m\left(\mathbf{x}, t\right)$	Mass of the cell at position $\mathbf{x}$ and time step t.
$\Delta m_i\left(\mathbf{x}, t\right)$	Change in mass of the cell at position $\mathbf{x}$ at time step t due to streaming with the cell at position $\mathbf{x} + \mathbf{e}_i$.
$m_i^\mathrm{x}\left(\mathbf{x}, t\right)$	Excess mass to be distributed from a converted cell at position $\mathbf{x}$ to the neighbouring cell at position $\mathbf{x} + \mathbf{e}_i$ at time step t.
$f_i\left(\mathbf{x}, t\right)$	Distribution function in the direction $\mathbf{e}_i$ of the cell at position $\mathbf{x}$ at time t.
$f_i^{'}\left(\mathbf{x}, t\right)$	Interim distribution function produced after streaming and used during the collision calculations.
f_i^0	The distribution function in the direction $\mathbf{e}_i$ based on $\mathbf{u}_0$ that is used for initialisation at the start of the simulation.
f_i^s	The distribution function in the direction $\mathbf{e}_i$ based on $\mathbf{u}_s$ that is used for resetting the distribution functions of source cells.
$f_i^e\left(\mathbf{x}, t\right)$	Equilibrium distribution function in the direction of $\mathbf{e}_i$ of the cell at position $\mathbf{x}$ at time t.
$\Omega_i\left(\mathbf{x}, t\right)$	Collision operator used in the generalised Lattice Boltzmann Equation.
$\varepsilon\left(\mathbf{x}, t\right)$	Fill-fraction of a cell at position $\mathbf{x}$ at time t.
$\mathbf{n}\left(\mathbf{x}, t\right)$	Fluid surface normal of an interface cell at position $\mathbf{x}$ at time t. This function is only defined for interface cells.
$n_\mathrm{G}\left(\mathbf{x}, t\right)$	Number of neighbouring gas cells at position $\mathbf{x}$ at time t.
$n_\mathrm{F}\left(\mathbf{x}, t\right)$	Number of neighbouring fluid cells at position $\mathbf{x}$ at time t.
t_x	The x-coordinate of a given thread relative to its thread block and grid.
t_y	The y-coordinate of a given thread relative to its thread block and grid.
t_z	The z-coordinate of a given thread relative to its thread block and grid.
t_w	The width of a thread block.
t_h	The height of a thread block.
t_d	The depth of a thread block.

Abbreviations

ALU	Arithmetic Logic Unit
API	Application Programming Interface
CLSVOF	Combined Level Set and Volume of Fluid
CPU	Central Processing Unit
CTM	Close To Metal
CUDA	Compute Unified Device Architecture
GPGPU	General-Purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
HPC	High-Performance Computing
LBGK Model	Lattice Bhatnagar-Gross-Krook Model
LB	Lattice Boltzmann
LBE	Lattice Boltzmann Equation
LBM	Lattice Boltzmann Method
LGA	Lattice Gas Automata
LS	Level Set(s)
RAM	Random Access Memory
SIMD	Single Instruction Multiple Data
SPH	Smoothed Particle Hydrodynamics
VOF	Volume of Fluid

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Fluids are commonly used for visual effects in films, television shows and advertisements, with effects ranging from water rushing into a sinking ship in *Poseidon* [64], to the billowing smoke of the "Smoke Monster" in *Lost* [1], to swirling liquid in *Coca-Cola* advertisements [67], to a fiery skeleton in *Ghost Rider* [37] (Figure 1.1). Although the term "fluid" is often only used to refer to liquids such as water, these examples indicate that "fluid" is actually a generic term that can be used to refer to both liquids and gases, and can be more generally defined as a substance that has no fixed shape and yields easily to external pressure [61]. These fluid visual effects are generated using fluid simulations — programs that computationally generate fluid motion by using mathematical and physical models for fluid behaviour.

Fluid simulations require massive computation in order to reach the levels of realism audiences now expect, and this computation can be very time consuming. One way of increasing the realism of simulated fluids is to increase the level of detail of the simulation. This increase in detail results in longer simulation times because there is more data for the simulation to process. Long simulation times translate directly into expensive production costs as time spent performing the simulation is time that computer hardware cannot be used to perform other tasks. Therefore, an inexpensive means of reducing simulation time will allow the level of details in simulations to be increased, ultimately resulting in higher quality visual effects. Furthermore, lowering the cost of realistic fluid simulation will make it available for use in low-budget productions.

Graphics Processing Units (GPUs) are specialised computer hardware for rendering 2D and 3D graphics, primarily for modern computer games. In recent years, GPUs have demonstrated, through their multi-threaded multi-processor architecture, great potential for speeding up many simulations and computationally intensive operations [16]. This potential was first exploited in the early 2000s when the introduction of programmable shaders allowed programmers to hook arbitrary code into specific parts of the graphics pipeline. Although computation and data accesses were limited by the structure of the graphics pipeline, researchers were able to use shaders

Figure 1.1: Examples of the use of fluid simulations for visual effects. The ocean floods a ship in *Poseidon* [72], liquid swirls around a *Coca-Cola* bottle in an advertisement [67], a skeleton is enveloped in flames in *Ghost Rider* [71], and the "Smoke Monster" is confronted in *Lost* [91].

to perform fluid simulations [45]. The limitations imposed by shaders were removed with the introduction of frameworks such as NVIDIA's CUDA, ATI's Close-to-Metal, and OpenCL which provide an interface for arbitrary code execution on GPUs, freeing programmers from the structure of the graphics pipeline.

Although arbitrary computation and arbitrary memory accesses are now allowed on GPUs, only certain algorithms are suitable for efficient, parallel GPU execution. The structure of computation and memory accesses are important considerations that can have significant impact on performance [15]. Graphics cards are designed to perform the same operation in parallel on adjacent memory locations, and, consequently, having this structure significantly improves an algorithm's potential to benefit from GPU execution. Within the field of fluid simulation, the Lattice Boltzmann Method (LBM) stands out as a candidate with the appropriate structure. It is a linear approximation of the Navier-Stokes equations that takes a grid-based approach to fluid simulation by representing the fluid with a fixed lattice and tracking fluid as it moves through the lattice. This contrasts the particle-based approach of Smoothed Particle Hydrodynamics (SPH) [53] — a popular method of fluid simulation which represents the fluid as particles whose movement and interactions determine the motion of the fluid. Unlike the mobile particles of SPH, the fixed lattice of the LBM is a natural fit for the fixed layout of GPU memory. Furthermore, the LBM requires that the same operations be applied to all positions in its lattice using information local to each lattice position, which is a natural fit for GPU execution. Lastly, the LBM's linearity

reduces memory requirements since it only requires information from the previous time step — higher order algorithms require information from multiple time steps to be stored. This reduced memory footprint is an important consideration given that the memory available to the GPU is limited. These characteristics of the LBM are all strong reasons to investigate the benefits of implementing the LBM for GPU execution.

1.1 Objectives and Approach

The aim of this thesis is to present a GPU implementation of the LBM for use in animation. The LBM can simulate a variety of effects (examples include airflow [88], smoke [45], flames [90] and liquids [81]), but, for the purposes of reducing scope, we will limit ourselves to liquid simulations with a free-surface. The free-surface is the visible boundary between two fluids, typically between a liquid and a gas (e.g. water and air), but also between two liquids (e.g. oil and water). Without a free-surface, we would be limited to visualizing the fluid internals and could not show details such as splashing. Therefore, our simulation includes a free-surface, but only between a liquid and a gas. We also include interactions with stationary obstacles because their inclusion allows for more interesting fluid effects.

An important part of any GPU implementation is to decide on which GPU framework to use. The two leading GPU frameworks are OpenCL and NVIDIA's CUDA. OpenCL is intended for use across a wide variety of computing platforms, whereas CUDA is specific to NVIDIA's GPUs. CUDA is a very popular platform for NVIDIA GPUs that has mature support libraries and development tools, and is still seeing active development from NVIDIA. On the other hand, OpenCL's platform portability makes it a desirable choice given that there are no significant performance differences between it and CUDA [23]. However, when this research began OpenCL was still in its infancy, and CUDA was relatively mature. Since we were interested in GPU performance and both frameworks could produce equivalent results, we chose to use CUDA because, at that time, it was more mature.

Broadly, our implementation of the LBM can be divided three steps: stream, collide and interface movement. Our approach to implementing the LBM in CUDA is to ensure that the efficiency of the core LBM calculations are not affected by the complexity associated with interface movement. We, therefore, separate the implementation of the LBM into multiple kernels that are specific to each step of the LBM. We focus on ensuring the efficiency of the stream and collide steps since these steps apply across the entire fluid and thus perform the bulk of the LBM calculations. This contrasts with the interface movement kernels which only apply at the free-surface.

With these design considerations in mind, the key objectives of this thesis are to provide:

- an efficient design for a free-surface LBM simulation that handles arbitrarily shaped, stationary obstacles using CUDA;

- a detailed analysis of the performance of our CUDA implementation of the stream and collide steps of the LBM; and
- a comparative analysis of the performance of our complete CUDA LBM free-surface fluid simulation against that of a single-core CPU implementation.

We aim to show that using graphics hardware as a platform for free-surface LBM simulations provides at least an order-of-magnitude performance improvement over single-core CPU implementations.

1.2 Thesis Structure

We begin by providing a history of techniques used to simulate fluids in Chapter 2, which is followed by a detailed look at the LBM and the techniques we use for interacting with obstacles and tracking the fluid surface in Chapter 3. In Chapter 4 we provide an overview of the brief history of General-Purpose GPU computing (GPGPU) and an overview of the CUDA architecture. This is followed by a summary of existing GPU LBM implementations and the presentation of our implementation of a free-surface LBM fluid simulation using CUDA in Chapter 5. We then present an in-depth analysis of the performance of our streaming and collide kernels and look at the performance of our kernels in the context of fluid simulation in Chapter 6. We continue our analysis in Chapter 7, where we look at our LBM simulation as a whole, evaluating the visual results, comparing the performance to that of a single-core CPU implementation and discussing some of the limitations of our implementation. Lastly, we present our conclusions in Chapter 8.

Chapter 2

Fluid Simulation

Before we present our implementation of the Lattice Boltzmann method (LBM) on graphics hardware, we first provide an overview of fluid simulation. We discuss the broad approaches used to model internal fluid flows, as well as techniques used to track the fluid surface. This provides some context for our choice of fluid simulation, the LBM, and surface tracking, Volume of Fluid (VOF), methods.

Since it is difficult to discussion fluid simulation without an understanding of some basic fluid terminology, we begin this chapter with an overview of some important fluid properties.

2.1 Properties of Fluids

Fluids have several unique properties that are used to classify them and describe their physical properties. It is difficult to discuss fluid simulation without a basic understanding of these properties, therefore it is important to define them here. A more detailed discussion on the properties of fluids can be found in Bachelor [5].

For the purposes of simulating fluids, the most important difference between gases and liquids is that a liquid has a *free-surface*. A free-surface is the boundary between two fluids, commonly perceived by the human eye as a boundary between a liquid and a gas such as water and air.

As with all matter, when a fluid moves, the quantity of the fluid does not change. This is known as the law of *conservation of mass*: without any external influence, a fluid in a closed system should maintain the same mass from one time step to the next. This is of vital importance for simulation accuracy, especially for simulations with a free-surface, and therefore a requirement for fluid simulations for scientific or engineering applications. However, in simulations for animation very small differences in mass between time steps do not make a noticeable difference in the visual output [82].

The *density* of a fluid is the mass of a fluid per unit of volume. In most fluids density can vary according to the temperature and pressure applied to the fluid; these fluids are classified

as *compressible*. Some fluids, liquids in particular, show negligible differences in density due to changes in pressure and temperature; these fluids are referred to as *incompressible* and are assumed to have a constant density.

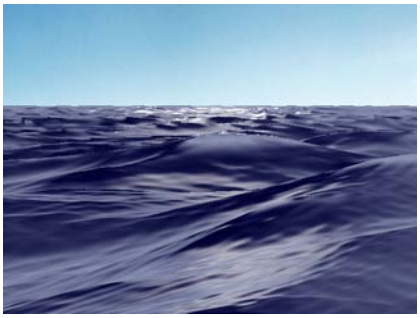
Fluids can be further classified into *Newtonian* and *Non-Newtonian* fluids. Newtonian fluids continue to behave as one would expect a fluid to behave regardless of the stress applied to the fluid, whereas the properties of Non-Newtonian fluids can vary depending on the stress applied. For example, the viscosity of common Non-Newtonian fluids varies depending on the speed at which objects are moved through the fluid [3]. The formal definition of a Newtonian fluid is not relevant here, but it should be noted that models for incompressible Newtonian fluids can have constant viscosity throughout the fluid. This assumption of constant viscosity reduces the complexity of the fluid model. We use a model for incompressible Newtonian fluids because simpler fluid models require less computation and have algorithms that are simpler to parallelize. Furthermore, for the purposes of animation, incompressible Newtonian fluids behave in a way people expect fluids to behave.

2.2 Modelling Fluids

There are two basic approaches to modelling fluids. The first approach is to artificially model the behaviour of the surface fluids using mathematical functions. The second approach is to model the underlying physical interactions taking place within a fluid. Approximating only the fluid surface usually requires far less computation, as the 'invisible' fluid internals do not need to be calculated, but this is done at the cost of limiting the complexity of fluid surface movement. In this section we provide a brief overview of approaches to modelling fluids. For a more general overview of the mathematics behind fluid simulation from a computer graphics perspective, refer to Bridson's work [9].

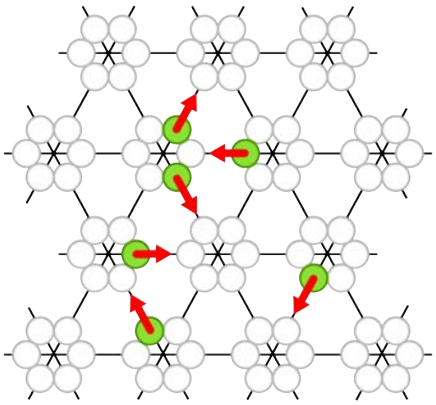
Popular examples of approximating fluid surface behaviour include modelling the surface of the ocean [33] (Figure 2.1a) and using the shallow water equations [86] to model ripple effects in water. The shallow water equations have even been successfully extended to handle water moving over obstacles [40]. These methods can be extremely effective for simple animated scenes, but are unable to effectively model turbulent fluid surfaces, such as splashes. However, it is possible to overcome this limitation by coupling the model with more detailed fluid simulations [84]. In these hybrid models, the large-scale wave effects are handled by computationally less expensive mathematical approximations and the small scale detailed turbulent effects are handled by more computationally intensive methods [82]. Complex, realistic and physically accurate simulations require the underlying fluid dynamics to be modelled throughout the entire scene, which requires finding a solution, or an approximation of a solution, to the Navier-Stokes (NS) equations.

The NS equations are a set of partial differential equations that govern the motion of fluids over time. Detailed fluid simulations must directly or indirectly solve the NS equations in or-



(a) Surface Modelling

(b) Particle Based

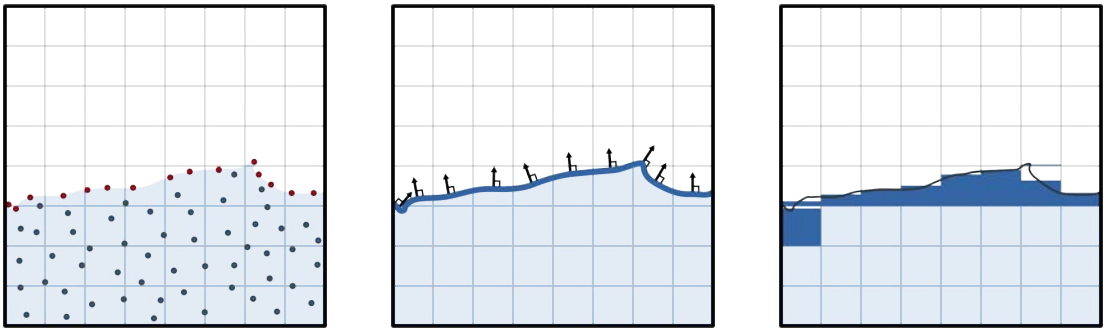


(c) Grid Based

Figure 2.1: There are many different approaches to modelling fluids. In Figure 2.1a, shallow water equations have been used to approximate the fluid surface [33]. Figure 2.1b illustrates how particles can be used to represent a fluid [56]. Figure 2.1c depicts the transfer of information along a rigid 2D grid as part of a grid-based simulation method[69].

der to obtain the information necessary to describe the flow within a fluid at a specific point in time. Since the NS equations are non-linear, solving them directly is computationally very expensive, making computation time a significant limiting factor for fluid simulations. Indirect NS solvers are sometimes able to avoid solving non-linear equations[92], but these also require significant computation. Before the introduction of compute clusters and programmable graphics hardware, it was computationally infeasible for large scenes to even approximate solutions to the NS equations. Modern high-performance computing (HPC) platforms have significantly reduced the computation time required, thus making these simulations possible.

The two most widely used methods for simulating the underlying fluid dynamics are particle- and grid-based methods [69]. Particle-based methods (also known as Lagrangian methods), shown in Figure 2.1b, track particles and their properties as they interact during the fluid simulation — Smoothed Particle Hydrodynamics [52] (SPH) is a popular particle-based method that is used in a number of industry fluid animation implementations such as Houdini [76], Maya [4], Realflow [68] and Blender [24]. Grid-based methods (also known as Eulerian methods), shown in Figure 2.1c, store information about the fluid in a static grid and track the motion of the fluid as is



(a) Marker Particles **(b)** Level Sets **(c)** Volume of Fluid

Figure 2.2: This figure illustrates three different methods for tracking the fluid surface. The Marker Particle method reconstructs the free-surface based on the positions of tracking particles that flow throughout the fluid. Level Sets store a continuous curve, which represents the free-surface that is advected each time step according to local fluid velocities. The Volume of Fluid method discretises the fluid surface on the simulation grid, approximating how much fluid is present at each grid position at the fluid surface.

passes through the grid. In our work, we use an indirect NS solver called the Lattice Boltzmann Method (LBM). It is based on a hybrid model (Semi-Lagrangian) that tracks the flow of particles through a fixed grid. As will be discussed in Chapter 4, our motivation for using the LBM is that the structure of its lattice is naturally aligned with the structure of GPU memory. In Chapter 3 we outline the implementation of the LBM, but Wolf-Gladrow [92] provides a in-depth discussion on the development of the Lattice Boltzmann method as a NS solver.

2.3 Tracking the Fluid Surface

In this work we use a grid-based model for fluid simulation, so we focus on the associated fluid surface models. Surface tracking is simpler in particle-based simulations than in grid-based simulations, as the movement of the fluid particles determines the movement of the fluid surface. Simple methods such as point-splatting [56] (directly rendering the fluid surface from simulation particles) are often sufficient, but complex techniques that provide better models for surface tension can provide greater accuracy [41].

Grid-based fluid simulations track the behaviour of the fluid as it moves throughout the simulation grid, with neighbouring points interacting and exchanging information about the local fluid. When a free-surface is introduced to a fluid simulation, two assumptions no longer hold: grid-neighbours do not always contain fluid and mass is no longer constant throughout the simulation grid. Thus, interactions of a fluid at its free-surface must be managed to ensure that fluid exchanges remain balanced, that the mass of the fluid remains constant and that the free-surface moves correctly through the simulation grid.

A widely used method is to track marker particles as they flow though a fluid and to plot

them on top of the simulation grid [30, 34], shown in Figure 2.2a. Cells that have neighbours that do not contain any of these tracking particles form the fluid surface. The drawback of this approach is that extra management is required to move the marker particles through the grid and ensure that fluid interactions at the fluid surface remain balanced.

Level sets, shown in Figure 2.2b, have been successfully used to model free-surfaces [60, 80]. A number of extensions to the basic algorithm have also been developed to improve its ability to represent complex behaviours as well as address some of level sets' weaknesses [82]. The basic idea is to define a continuous function that represents the fluid surface and then to advect[1] that function according to local fluid velocities. This approach has mathematical advantages that make it relatively simple to model behaviours that have the potential to add significant complexity [59]. For example, level sets allow a fluid surface to break apart and coalesce without additional tracking computations [60]. The disadvantage of using level sets is that they are not effective at conserving mass, particularly when the simulation grid is coarse [87].

The *Volume of Fluid* (VOF) method, shown in Figure 2.2c, uses a simple scheme of tracking a fractional value to represent the percentage of each cell at the fluid surface that is filled with fluid [34]. Unlike level sets and marker particles, which represent a continuous boundary for the fluid surface, the VOF method uses the simulation grid to form a discretised fluid surface boundary. In order to model sub-grid fluid effects at the fluid surface, the VOF method restricts the fluid transfers between cells at the fluid surface. This increases the complexity of fluid surface behaviours that can be modelled using the VOF method. A significant benefit for the purposes of this research is that, by aligning the fluid surface with the simulation grid, the VOF method has a reduced memory footprint, compared to other methods [34]. This is vitally important when working with the constricted memory requirements of graphics hardware.

Since 1981, when the VOF Method was first proposed, various improvements have been developed, particularly for improved modelling of the fluid surface tension [27, 70, 50] and for *Combined Level Sets and Volume of Fluid* (CLSVOF) models [79, 78, 87]. This research uses a VOF implementation that has previously been used for parallel fluid simulations [82, 69]. The standard VOF method for the LBM is adjusted to encourage fluid to flow out of cells that are emptying and encourage fluid to flow into cells that are filling. This improves the visual appearance of the fluid simulation [82] and avoids extra passes over the simulation grid that would otherwise be required to maintain stability in the fluid surface.

— ∼ —

We have provided a brief background to the different approaches to fluid simulation and fluid surface tracking. In the following chapter, we explain the mathematics behind our chosen fluid simulation technique, the Lattice Boltzmann Method.

[1] Definition: move as a result of fluid flow [61].

Chapter 3

The Lattice Boltzmann Method

The Lattice Boltzmann Method (LBM) is a derivative of the Lattice Gas Automata (LGA) [11]. Like the LGA, the LBM is a grid-based method that simulates the flow of a fluid over a static lattice — one that keeps the same shape and position throughout a simulation. The traditional algorithm is divided into the stream step, where distribution functions are propagated throughout the grid, and a collide step, where the distribution functions interact locally.

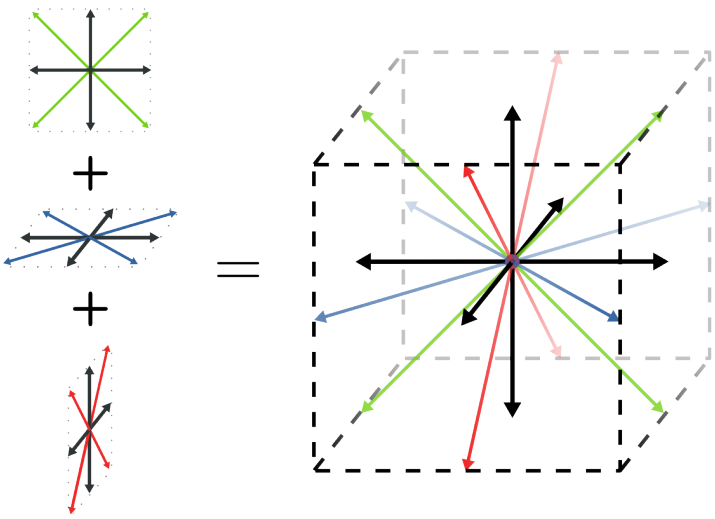
There are various models of the LBM, catering for 2D and 3D simulations at different levels of accuracy. These different models are classified by Qian et al. [66] such that a model with d dimensions and b direction vectors would be the DdQb model (e.g. D3Q15 is a 3D model that streams distribution functions in 15 directions). This thesis uses the D2Q9[1] and D3Q19 models. The D2Q9 model is a popular 2D model [12, 66, 92] that improves on the accuracy of the D2Q4 and D2Q5 models, while avoiding the complexities of models with higher Q values [92]. The D3Q19 model was chosen as it provides sufficient accuracy, while avoiding the extra computation of models such as the D3Q27 model, and done not suffer from visual artefacts, such as checkerboard effects [38], of the D3Q15 model. Both the D2Q9 and D3Q19 are also well suited to parallelization, and have been successfully parallelized on various architectures [69, 83, 65, 45, 63, 85, 54].

This chapter outlines the algorithm for simulating fluid with a free-surface using the LBM. A full derivation of the LBM and proof of physical correctness is not included here, but can be found in works by Succi [77] and Wolf-Gladrow [92].

3.1 Simulating Fluids with the Lattice Boltzmann Method

Underlying Lattice Boltzmann (LB) fluid simulation is a structured regular lattice. In the 2D case a square lattice is usually used, although there are models for a hexagonal lattice. In the 3D case a simple Cartesian cubic lattice is used. Each square, hexagon or cube that is part of the lattice is referred to as a *cell*, with each cell containing local information about the fluid.

[1] The D2Q9 model was used during prototyping and is used to simplify explanations in this chapter. All reported simulations use the D3Q19 model



D2Q9

$\mathbf{e}_0$	$= \langle\ 0,\ 0\ \rangle$
$\mathbf{e}_{1,5}$	$= \langle \pm 1,\ 0\ \rangle$
$\mathbf{e}_{3,7}$	$= \langle\ 0, \pm 1\ \rangle$
$\mathbf{e}_{2,4,6,8}$	$= \langle \pm 1, \pm 1\ \rangle$

D3Q19

$\mathbf{e}_0$	$= \langle\ 0,\ 0,\ 0\ \rangle$
$\mathbf{e}_{1–2}$	$= \langle \pm 1,\ 0,\ 0\ \rangle$
$\mathbf{e}_{3–4}$	$= \langle\ 0, \pm 1,\ 0\ \rangle$
$\mathbf{e}_{5–6}$	$= \langle\ 0,\ 0, \pm 1\ \rangle$
$\mathbf{e}_{7–10}$	$= \langle \pm 1,\ 0, \pm 1\ \rangle$
$\mathbf{e}_{11–14}$	$= \langle \pm 1, \pm 1,\ 0\ \rangle$
$\mathbf{e}_{15–18}$	$= \langle\ 0, \pm 1, \pm 1\ \rangle$

Figure 3.1: The direction vectors for the D2Q9 and D3Q19 Models. Different implementations may use different index orderings. In the 2D model streaming occurs between all adjacent cells, and in the 3D model streaming only occurs between cells that share a common edge (i.e. at least two vertices). In the 3D model distribution functions stream along each of the three axes, and the diagonals of the xy-plane (green), xz-plane (blue) and yz-plane (red). The 2D model is the equivalent of the xy-plane. Both models include a distribution function for rest particles (purple circle in the centre).

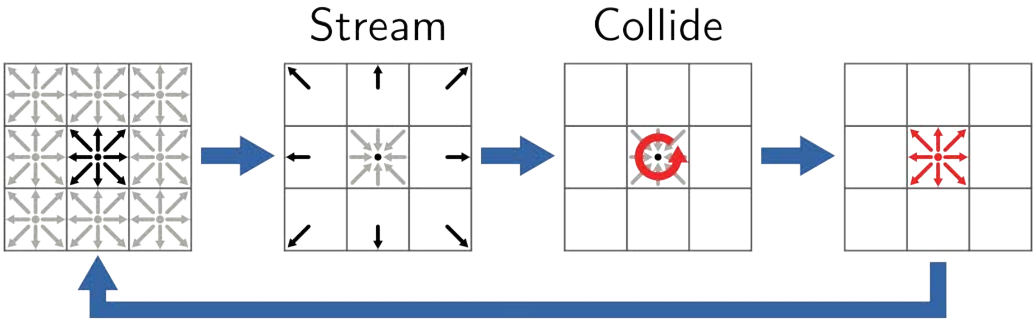


Figure 3.2: The key stages of the LBM. The stream step simulates the flow of fluid throughout the lattice. The collide step simulates the interactions between particles in the fluid. After the collide step, the system is ready to start the next stream step.

The most important information contained in each cell are the *distribution functions* (f_i), each associated with a specific direction vector ($\mathbf{e}_i$, Figure 3.1). Each $f_i(\mathbf{x}, t)$ can be thought of as representing the probability that a particle at position $\mathbf{x}$ at time t will travel in the direction $\mathbf{e}_i$ ($\sum_i f_i$ is not always equal to 1, so the f_i do not create a discrete probability distribution). The interactions between these distribution functions drive the fluid simulation. The calculation of interactions is divided into two steps: the stream step and collide step (Figure 3.2).

During each stream step, a cell's distribution functions are streamed to adjacent cells in each direction. This simulates the flow of the fluid through the lattice. Mathematically, streaming is

defined by:

$$f_i^* (\mathbf{x} + \mathbf{e}_i, t) = f_i (\mathbf{x}, t) \tag{3.1}$$

where $\mathbf{x}$ is the lattice-coordinate, t is the current time step and f_i^* is the interim distribution function used during the collide step.

The collide step simulates the collisions of particles as they move within the fluid. The generalised form of the collide step is the Lattice Boltzmann Equation (LBE):

$$f_i (\mathbf{x} + \mathbf{e}_i, t + 1) - f_i^* (\mathbf{x}, t) = \Omega_i (\mathbf{x}, t) \tag{3.2}$$

where Ω_i is the collision operator.

In this work, we use the Lattice Bhatnagar-Gross-Krook (LBGK) model [66]. The LBGK collision operator is defined as:

$$\Omega_i (\mathbf{x}, t) = \frac{1}{\tau} \Big(f_i^e (\mathbf{x}, t) - f_i^* (\mathbf{x}, t) \Big) \tag{3.3}$$

The interim distribution functions are used to create *equilibrium distribution functions* (f_i^e), which are then combined with each f_i^* during a relaxation sub-step that maintains fluid stability. The equilibrium distribution functions are calculated as follows:

$$f_i^e (\rho, \mathbf{u}, t) = \rho w_i \left(1 - \frac{3}{2} \mathbf{u}^2 + 3 \mathbf{u} \cdot \mathbf{e}_i + \frac{9}{2} (\mathbf{u} \cdot \mathbf{e}_i)^2 \right) \tag{3.4}$$

where ρ is the lattice density at $\mathbf{x}$, $\mathbf{u}$ is the fluid velocity at $\mathbf{x}$ and the w_i are the equilibrium distributions for $\mathbf{u} = \mathbf{0}$. Lattice density and velocity are calculated using the following:

$$\rho = \sum_i f_i^* \qquad\qquad \rho \mathbf{u} = \sum_i f_i^* \mathbf{e}_i \tag{3.5}$$

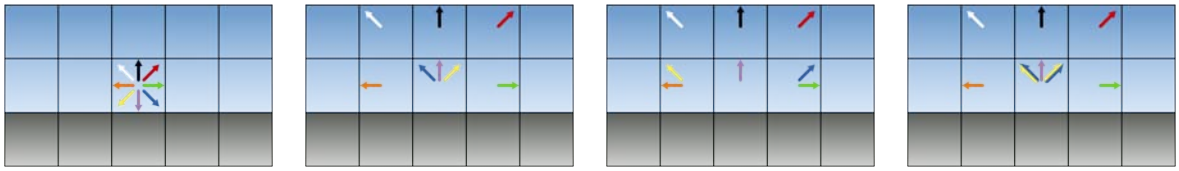
The equilibrium distributions (w_i) are determined such that velocity moments are uniform in all orientations (isotropic) [66]. For the D2Q9 and D3Q19 models their values are

D2Q9		**D3Q19**	
w_0	$= \frac{4}{9}$	w_0	$= \frac{1}{3}$
$w_{1,3,5,7}$	$= \frac{1}{9}$	$w_{1\text{–}6}$	$= \frac{1}{18}$
$w_{2,4,6,8}$	$= \frac{1}{36}$	$w_{7\text{–}18}$	$= \frac{1}{36}$

Substituting the LBGK collision operator (Equation 3.3) into the LBE (Equation 3.2), we get

$$f_i (\mathbf{x}, t + 1) = \frac{1}{\tau} f_i^e (\mathbf{x}, t) + \left(1 - \frac{1}{\tau} \right) f_i^* (\mathbf{x}, t) \tag{3.6}$$

where $\tau > \frac{1}{2}$ is the relaxation parameter (Equation 3.18). The relaxation parameter is used control



(a) Initial f_i **(b)** No-slip **(c)** Free-slip **(d)** Hybrid

Figure 3.3: This figure illustrates three different methods for handling boundary conditions in the D2Q9 model. These methods are applicable to 3D models, and each have their own associated advantages and disadvantages. No-slip boundary conditions reverse fluid flow. Free-slip boundary conditions reflect along the obstacle surface normal. Hybrid boundary conditions involve a mixture of no-slip and free-slip weighted by a user-set parameter. (Figure inspired by Figure 2.4 in [81])

the rate at which the system is brought to equilibrium. Higher values of τ reduce the influence of the collide step on the simulation, increasing the time it takes for the system to reach equilibrium conditions. In the basic model $\tau = 3v + \frac{1}{2}$, where v is the lattice viscosity of the fluid. However, in order to maintain simulation stability, the value of τ is varied for each cell at each time step with local turbulence (discussed in Section 3.4 below).

3.2 Static Obstacles

Complex obstacle boundaries can be included in LB fluid simulations with relative ease and only require an adjustment to the stream step. Each cell that intersects with the obstacle is defined to be an *obstacle cell* and each cell containing only fluid is defined as a *fluid cell* (see Figure 3.4). Whenever distribution functions interact with an obstacle cell, their streaming destination is altered to simulate the effect that an obstacle has on the fluid. The interactions between fluid cells remain unchanged. These rules that define the way in which streaming is altered are referred to as boundary conditions.

No-slip boundary conditions (Figure 3.3b) provide the simplest method of integrating static obstacles into the simulation. Each distribution function that would be streamed into an obstacle cell is reversed and streamed into the opposite distribution function within the same cell. This both creates the effect of the fluid slowing down and the sticking of fluid near obstacles, and ensures that all fluid cells have properly filled distribution functions. Mathematically, no-slip boundary conditions modify Equation 3.1 to be

$$f_i^* (\mathbf{x}, t) = f_{\tilde{i}} (\mathbf{x}, t) \tag{3.7}$$

where $\tilde{i}$ is chosen such that $\mathbf{e}_{\tilde{i}} = -\mathbf{e}_i$.

While no-slip boundary conditions model flows over high-friction surfaces relatively well (e.g. unfinished wood), they are not well suited to smooth surfaces (e.g glass) that exert little friction on passing fluid. For these cases, the *free-slip* boundary conditions (Figure 3.3c) are suggested

where distribution functions are reflected along the local surface normal of obstacle cells[77] — the appropriate direction vector is selected as the normal based on which adjacent cells are obstacles. For free-slip boundary conditions Equation 3.1 is adjusted to be

$$f_i^* (\mathbf{x}, t) = f_r (\mathbf{x} + \mathbf{e}_S, t)$$

where r is the index such that $\mathbf{e}_r$ is the reflection of $\mathbf{e}_i$ along the surface normal, and $\mathbf{e}_S$ is the direction to the cell from which the appropriate distribution function will be sourced.

In order to achieve boundary conditions that lie between those of no-slip and free slip, it is possible to create a hybrid method [69, 82] (Figure 3.3d). To reduce computation time, only the free-slip reflected index (r) is used, since calculating $\mathbf{e}_S$ adds an extra layer of complexity. Using a user-set parameter, $0 \leq \gamma \leq 1$, hybrid boundary conditions adjust Equation 3.1 to be

$$f_i^* (\mathbf{x}, t) = \gamma f_{\bar{i}} (\mathbf{x}, t) + (1 - \gamma) f_r (\mathbf{x}, t)$$

All three of the above methods are unable to accurately represent porous media, smooth curved boundaries [54], or intricate obstacle details [69], especially in low-resolution lattices. Techniques are available to improve the accuracy by storing more information about the obstacle [49] and by modelling porous media [57]. Improvements of these kind generally involve increased complexity and computation.

3.3 Simulating a Fluid Interface

As discussed in Section 2.3, we use the VOF method to track the movement of the fluid surface. In LB simulations without a fluid surface, constant mass is implied and need not be explicitly tracked. However, simulations with a fluid surface involve changes in mass as the fluid surface moves due to the LBM's discretisation of the scene. As a result a mass function ($m (\mathbf{x}, t)$) is introduced to track the variation of mass at the fluid surface, which is then used when calculating the fill fraction ($\varepsilon (\mathbf{x}, t)$) as follows:

$$\varepsilon (\mathbf{x}, t) = \frac{m (\mathbf{x}, t)}{\rho (\mathbf{x}, t)} \tag{3.8}$$

Two new lattice cell types are also required (Figure 3.4). *Gas cells* represent cells that do not contain any fluid, and do not form part of any obstacle. Cells that contain fluid, but are bordered by at least one gas cell are defined to be *interface cells* and are responsible for managing all fluid-gas interactions. Two changes to the traditional LBM are required: streaming has to account for interface cells at the fluid surface; and conversions between gas, interface and fluid cells need to be managed as the fluid surface moves through the lattice.

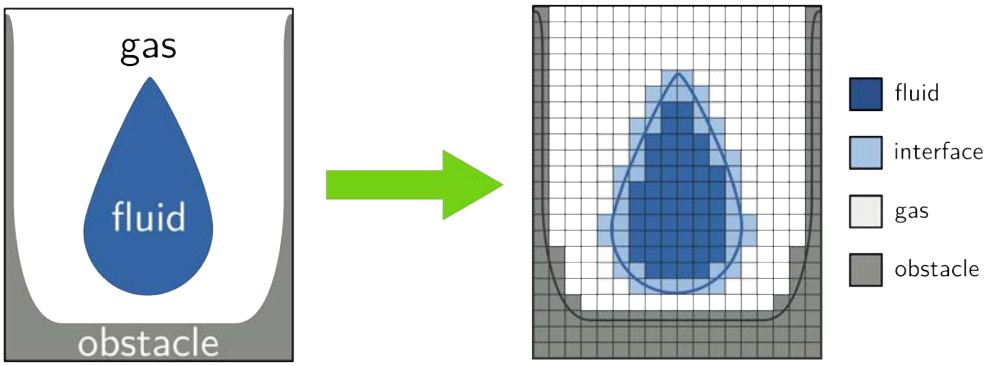


Figure 3.4: *Physical scenes* need to be converted into *lattice scenes*. Each cell in the lattice represents the characteristics of the relative location in the physical scene. Lattice cells that would contain both gas and fluid in the physical scene become *interface* cells (Section 3.3).

3.3.1 Streaming with Interface Cells

Various adjustments to the stream step are required to better simulate the interactions of particles at the fluid surface. The streaming of distribution functions between fluid and interface cells remains unaltered, but an extra step is required to track the transfer of mass. For fluid cells, this mass transfer can be ignored and it is assumed that fluid cells maintain a constant mass, $m(\mathbf{x}) = 1$. The mass transfer between a fluid cell and an interface cell is equivalent to values of the distribution functions streamed into and out of each cell and is defined by

$$m(\mathbf{x}, t+1) = m(\mathbf{x}, t) + \sum_i \Delta m_i(\mathbf{x}, t) \tag{3.9}$$

with

$$\Delta m_i(\mathbf{x}) = f_{\bar{i}}(\mathbf{x} + \mathbf{e}_i) - f_i(\mathbf{x}) \tag{3.10}$$

where the cell at $\mathbf{x}$ is an interface cell and the cell at $\mathbf{x} + \mathbf{e}_i$ is a fluid cell.

Mass transfer between two interface cells needs to be proportional to the contact area between the two partially filled interface cells [82]. The average of two neighbouring cells' fill fractions is used as a rough approximation this contact area: two nearly full cells will have close to 100% contact and a fill fraction average close to 1, while two nearly empty cells will have very little contact and a fill fraction average close to 0. The approximation is then multiplied by Equation 3.10 to give

$$\Delta m_i(\mathbf{x}) = \left(f_{\bar{i}}(\mathbf{x} + \mathbf{e}_i) - f_i(\mathbf{x})\right) \times \frac{\varepsilon(\mathbf{x} + \mathbf{e}_i) + \varepsilon(\mathbf{x})}{2} \tag{3.11}$$

Thürey [82] suggest further adjustments to the mass transfer step to limit the number of visual artefacts arising due to a fluid surface moving faster than interface cells can effectively fill or

		Cell at $\mathbf{x}+\mathbf{e}_i$		
		Fluid cell	$n_\mathrm{F}=0$	$n_\mathrm{G}=0$
Cell at $\mathbf{x}$	Fluid cell	$f_{\bar{i}}(\mathbf{x}+\mathbf{e}_i)-f_i(\mathbf{x})$	$f_{\bar{i}}(\mathbf{x}+\mathbf{e}_i)$	$-f_i(\mathbf{x})$
	$n_\mathrm{F}=0$	$-f_i(\mathbf{x})$	$f_{\bar{i}}(\mathbf{x}+\mathbf{e}_i)-f_i(\mathbf{x})$	$-f_i(\mathbf{x})$
	$n_\mathrm{G}=0$	$f_{\bar{i}}(\mathbf{x}+\mathbf{e}_i)$	$f_{\bar{i}}(\mathbf{x}+\mathbf{e}_i)$	$f_{\bar{i}}(\mathbf{x}+\mathbf{e}_i)-f_i(\mathbf{x})$

Table 3.1: Modifications to the mass transfer step in Equation 3.10 are required to limit the number of visual artefacts resulting from the standard VOF method. In this table, n_F represents the number of neighbouring fluid cells and n_G the represents the number of neighbouring gas cells.

empty. Combining this with the adjustments made in Equation 3.11, we get

$$\Delta m_i(\mathbf{x}) = m_i^*(\mathbf{x}) \times \frac{\varepsilon(\mathbf{x}+\mathbf{e}_i)+\varepsilon(\mathbf{x})}{2} \tag{3.12}$$

where $m_i^*(\mathbf{x})$ should be substituted with the appropriate calculation shown in Table 3.1. Since it is an expensive operation to count neighbouring cell-types at each time step, variables $n_\mathrm{F}(\mathbf{x},t)$ and $n_\mathrm{G}(\mathbf{x},t)$ are introduced to track the number neighbouring fluid and gas cells, respectively. These adjustments are problematic for thin layers of interface cells that are adjacent to obstacles, because they restrict the flow of the interface over the obstacle. To allow the fluid surface to move freely in these situations, we transfer mass in both directions if there are fewer than three neighbouring fluid cells and fewer than three neighbouring empty cells.

3.3.2 Streaming with Gas Cells

During the stream step, cells transfer their distribution functions to their neighbours and receive a full set of new distribution functions from their neighbours. Interface cells' neighbours include gas cells and therefore do not receive a full set of distribution functions after streaming. It is possible to simulate the neighbouring gas behaviours using the LBM, and therefore populating gas cells with their own distribution functions that can be streamed to interface cells, but this can be very computationally expensive. Therefore, we rather approximate the distribution functions of the gas cells. Since the gas has a much lower viscosity than the fluid, we can assume that the gas' motion will follow that of the fluid at the fluid surface [82]. This allows us to use the local fluid velocity ($\mathbf{u}(\mathbf{x})$) and atmospheric pressure ($\rho_\mathrm{A}=1$) to approximate the gas' equilibrium distribution functions (Equation 3.4). These values can be used to approximate the distribution function that needs to be streamed to the interface cell:

$$f_{\bar{i}}^*(\mathbf{x},t) = f_{\bar{i}}^e\Big(\rho_\mathrm{A},\mathbf{u}(\mathbf{x})\Big) + f_i^e\Big(\rho_\mathrm{A},\mathbf{u}(\mathbf{x})\Big) - f_i(\mathbf{x},t) \tag{3.13}$$

where the cell at position $\mathbf{x}$ is the interface cell and the cell at $\mathbf{x}+\mathbf{e}_i$ is a gas cell.

According to Thürey [82], this reconstruction step introduces an imbalance in interface cells'

distribution functions. To correct this, it is also possible to reconstruct f_i if the direction vector, $\mathbf{e}_i$, is pointing into the fluid. Mathematically this is represented

$$f_i(\mathbf{x},t) = f_{\bar{i}}^{e}\Big(\rho_{\mathrm{A}},\mathbf{u}(\mathbf{x})\Big) + f_i^{e}\Big(\rho_{\mathrm{A}},\mathbf{u}(\mathbf{x})\Big) - f_i(\mathbf{x},t) \quad \text{, if } \mathbf{n}\cdot\mathbf{e_i} < 0 \tag{3.14}$$

with the normal defined as

$$\mathbf{n} = \frac{1}{2}\begin{pmatrix} \varepsilon(\mathbf{x}-\mathbf{i}) & - & \varepsilon(\mathbf{x}+\mathbf{i}) \\ \varepsilon(\mathbf{x}-\mathbf{j}) & - & \varepsilon(\mathbf{x}+\mathbf{j}) \\ \varepsilon(\mathbf{x}-\mathbf{k}) & - & \varepsilon(\mathbf{x}+\mathbf{k}) \end{pmatrix}$$

where $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ are the standard unit vectors.

3.3.3 Moving the Fluid Surface

After the collide step, the movement of the fluid surface is simulated by creating new interface, fluid and gas cells. Interface cells that have high mass are converted to fluid cells, and those with negative mass to gas cells:

$$(1+\kappa)\rho(\mathbf{x}) < m(\mathbf{x}) \quad \rightarrow \quad \text{convert to fluid cell} \tag{3.15}$$

$$-\kappa\rho(\mathbf{x}) > m(\mathbf{x}) \quad \rightarrow \quad \text{convert to gas cell} \tag{3.16}$$

where $\kappa > 0$ is a constant. The value of κ affects how aggressive the simulation is in filling or emptying interface cells, with $\kappa = 0.01$ being suitable for most simulations [82]. Note that $m(\mathbf{x})$ can be negative, because it is only an approximation of fluid mass. Equation 3.16 shows that when $m(\mathbf{x})$ drops below 0, thus containing no fluid mass, the cell should be converted to an empty cell.

Reid [69] suggests selecting interface cells for conversion (based on the conditions 3.15 and 3.16) immediately after ρ is calculated during the collide step and storing the selected cells in two queues. One queue stores new fluid (*filled*) cells and the other stores new gas (*emptied*) cells. After the collide step is completed, these queues are processed. The *filled* queue is processed before the *emptied* queue, so that new gas cells that have neighbouring new fluid cells can be removed from the *emptied* queue in order to preserve the layer of interface cells around the fluid. While this method has only minor implications for memory usage, it has the benefit of avoiding an otherwise required pass through the entire lattice, thus reducing simulation time [69].

3.3.4 Converting Interface Cells

The process of emptying and filling interface cells has three steps:

1. Calculate the excess mass of the interface cell.

2. Make sure the layer of interface cells between gas and fluid cells is intact:

   Filled Cell: Neighbouring gas cells are converted to new interface cells.

   Emptied Cell: Neighbouring fluid cells are converted to new interface cells.

3. Distribute the excess mass to neighbouring fluid and interface cells, weighted according to the direction of fluid flow.

During the conversion process, it is likely that new gas cells will have a negative mass, and new fluid cells will have a mass slightly greater than that of a fluid cell. These differences are referred to as the *excess mass* ($m_i^{\mathrm{x}}(\mathbf{x},t)$), which is then distributed to neighbouring fluid and interface cells. For new fluid cells the distribution is weighted to favour cells in the direction of the fluid surface normal ($\mathbf{n}$), and for new gas cells the direction of the negative surface normal is favored. This serves as a rough approximation of fluid direction. As a result, the excess mass to be distributed to neighbouring cells is defined to be

$$m_i^{\mathrm{x}}(\mathbf{x}) = \begin{cases} \left(m(\mathbf{x}) - \rho(\mathbf{x})\right)\frac{\varphi_i(\mathbf{x})}{\varphi_{\mathrm{t}}(\mathbf{x})} & \text{, if } \mathbf{x} \text{ is a filled cell} \\ m(\mathbf{x})\frac{\varphi_i(\mathbf{x})}{\varphi_{\mathrm{t}}(\mathbf{x})} & \text{, if } \mathbf{x} \text{ is an emptied cell} \\ 0 & \text{, otherwise} \end{cases}$$

where the weights are defined as

$$\varphi_i(\mathbf{x}) = \begin{cases} \max\left(0, \quad \mathbf{n}(\mathbf{x}) \cdot \mathbf{e}_i\right) & \text{, if } \mathbf{x} \text{ is a filled cell} \\ \max\left(0, -\mathbf{n}(\mathbf{x}) \cdot \mathbf{e}_i\right) & \text{, if } \mathbf{x} \text{ is an emptied cell} \end{cases} \tag{3.17}$$

$$\varphi_{\mathrm{t}} = \delta + \sum_i \varphi_i$$

with φ_{t} being used to normalise the distributed mass. $\delta > 0$ is chosen as an arbitrarily small number to prevent divide-by-zero errors from arising.

For interface cells neighbouring a filled or emptied interface cell, the mass transfer step (Equation 3.9) is adjusted to be

$$m(\mathbf{x}, t+1) = m(\mathbf{x}, t) + \sum_i \Delta m_i(\mathbf{x}, t) + \sum_i m_i^{\mathrm{x}}(\mathbf{x} + \mathbf{e}_{\bar{i}})$$

where the cell at position $\mathbf{x}$ is the cell being converted to a gas or fluid cell. It is worth noting, that neighbouring fluid cells also have an associated φ_i, which is included in the calculation of φ_{t}. However, these cells can ignore the mass transfer step as it is always assumed that $m(\mathbf{x}) = 1$ for fluid cells.

As mentioned above, some neighbouring cells may need to be converted to interface cells. Converting fluid cells to an interface cell only requires mass to be set to 1 (before the distribution of excess mass) and the fill fraction to be recalculated (Equation 3.8). Converting empty cells to

interface cells requires all the distribution functions to be recalculated. First the average lattice density ($\overline{\rho}$) and lattice velocity ($\overline{\mathbf{u}}$) is calculated for all neighbouring fluid cells. The new distribution functions are then calculated using Equation 3.13 with $\overline{\mathbf{u}}$ and $\overline{\rho}$ being used as the local lattice density and lattice velocity.

3.4 Stability

Although the LBM is a physically-based method with proven physical correctness [77, 92], it is prone to numerical instability when simulation resolutions are too large to effectively model small-scale effects (e.g. turbulence that occurs within a cell) [35]. The Smagorinsky sub-grid turbulence model [75] is used to mitigate problems that arise from simulation instability, by providing high damping in areas with high local turbulence. The Mutliple-Relaxation-Time model is an alternative to the LBGK model with increased numerical stability [21], however, it is not used here because it requires more information to be stored and calculated, thus increasing memory use and computation time.

The Smagorinsky sub-grid turbulence model works by locally altering the relaxation parameter at each time step to control the effect of the equilibrium distribution functions on the simulation. The new relaxation parameter is calculated as follows:

$$\tau = 3\left(v + C^2 S\right) + \frac{1}{2} \tag{3.18}$$

where v is the lattice viscosity, C is the Smagorinsky constant and S is a measure of the local strain defined by

$$S = \frac{1}{6C^2}\left(\sqrt{v^2 + 18C^2|\Pi|} - v\right), \qquad \text{where} \quad \Pi = \sum_i \left[\left(f_i - f_i^e\right)\left(\sum_j \left(\mathbf{e}_i\right)_j\right)^2\right] \tag{3.19}$$

where j is an index for the components of $\mathbf{e}_i$. It is also worth noting that in the implementation, the C^2 in Equation 3.18 cancels with the C^2 in the denominator of S.

Π is referred to as the momentum flux tensor [2] [11]. The Smagorinsky constant affects the turbulence within the simulation with higher values reducing turbulence, thus smoothing the fluid surface. It is usually set within the range $[0.2; 0.4]$ but a value of 0.3 is recommended [69, 82].

While the relaxation parameter is limited to $\tau > \frac{1}{2}$, under certain conditions instability increases as $\tau \to \frac{1}{2}$ [69]. In order to avoid any associated instabilities, the relaxation parameter should be restricted to $\tau > 0.51$ [82]. While the visual results appear unaffected, the effects on simulation accuracy still need to be investigated.

[2] The notation $\Pi_{\alpha\beta} = \sum_i \mathbf{e}_{i\alpha}\mathbf{e}_{i\beta}\left(f_i - f_i^e\right)$ is often used in the literature.

3.5 Parameter and Lattice Initialisation

Lattice Boltzmann simulations do not restrict the physical size of a lattice cell to a single value. The physical cell width (Δx) can be varied, thus altering the physical dimensions of the entire scene. In the D2Q9 and D2Q19 models the speed of fluid within the lattice is limited by Δx and the time step (Δt) because fluid within the lattice should not be able travel further than Δx in a single time step. This gives an upper bound on velocity, $|\mathbf{u}| < u_{\max} = \frac{\Delta x}{\Delta t}$.

Without carefully selecting the time step in relation to the space step, introducing gravity to a simulation can quickly lead to $u_{\max}$ being exceeded, resulting in simulation instabilities. Also, a large time step limits fluid turbulence as $u_{\max}$ becomes restrictively small, while an extremely small time step ($\Delta t < 1$ ms) can lead to a prohibitively long runtimes and floating point inaccuracies.

Taking these limitations into account, the value of the time step is therefore calculated as follows:

$$\Delta t = \sqrt{\frac{g_{\mathrm{c}} \Delta x}{\mathbf{g}_{\mathrm{f}}^2}}$$

where $g_{\mathrm{c}} > 0$ is a constant that is chosen to sufficiently limit the size of the time step in order to maintain stability. For our simulations, we used $g_{\mathrm{c}} \approx 0.001$, but other values in the range $[0.0001; 0.05]$ can improve results, depending on the values of other parameters.

Using Δx and Δt, viscosity (η), gravity ($\mathbf{g}_{\mathrm{f}}$), initial velocity ($\mathbf{u}_{0\mathrm{f}}$) and source velocity ($\mathbf{u}_{\mathrm{sf}}$) are converted to dimensionless lattice units (v, $\mathbf{g}$, $\mathbf{u}_0$ and $\mathbf{u}_{\mathrm{s}}$ respectively):

$$v = \eta \frac{\Delta t}{(\Delta x)^2} \qquad \mathbf{g} = \mathbf{g}_{\mathrm{f}} \frac{(\Delta t)^2}{\Delta x} \qquad \mathbf{u}_0 = \mathbf{u}_{0\mathrm{f}} \frac{\Delta t}{\Delta x} \qquad \mathbf{u}_{\mathrm{s}} = \mathbf{u}_{\mathrm{sf}} \frac{\Delta t}{\Delta x}$$

The initialisation of the distribution functions is the first step to initialise the simulation lattice. For a system at rest (i.e. $\mathbf{u}_0 = \mathbf{0}$), populating all distribution functions with the equilibrium distributions ($f_i^0 = w_i$) is sufficient [69]. For a system in motion, we can initialise the distribution functions using f_i^e (Equation 3.4), substituting the velocity with the desired initial velocity:

$$f_i^0(\mathbf{x}) = f_i^e(\mathbf{x}, \mathbf{u}_0) = \rho w_i \left(1 - \frac{3}{2}\mathbf{u}_0^2 + 3\mathbf{u}_0 \cdot \mathbf{e}_i + \frac{9}{2}(\mathbf{u}_0 \cdot \mathbf{e}_i)^2\right)$$

where the lattice density (ρ) is usually set to 1 for simulations not including gravity. This equation is also used to initialise source distribution functions (f_i^s), but with $\mathbf{u}_0$ replaced by $\mathbf{u}_s$.

For simulations that include gravity, a pressure gradient may need to be initialised [69]. Fluid pressure (P) increases linearly with depth when placed in a container and can be calculated with

$$P = \rho_f |\mathbf{g}| h$$

where ρ_f is the density of the fluid and h is the height of the above fluid column. A common

approximation of fluid pressure for a point in a LB fluid simulation is to use the lattice density, ρ. Although this value is not the same as pressure, adjusting it can approximate adjustments of pressure within the fluid. Fluids that are unbounded by obstacles (e.g. a drop of water in mid-air or a wall of fluid against the side of a container) should not be initialised with a pressure gradient.

— ∼ —

Having described the mathematics behind the LBM, we will, in the next chapter, provide an overview of GPUs and the platform with which we intend to implement the LBM for GPU execution, CUDA. We will then, in Chapter 5 describe our implementation.

Chapter 4

General Purpose GPU Computing

Graphics cards are designed to process and generate images and video in order to free up the CPU to perform other tasks. Their development has largely been driven by ever increasing performance requirements for visual effects in computer games and to a lesser extent engineering and modelling applications. Calculations for graphics often require repeatedly applying the same operations across large sets of data — operations that put significant load on a sequential CPU. In order to reduce this strain on the CPU, graphics cards were developed as separate self-contained hardware specialising in these graphics-related operations. As a result of the parallel nature of graphics calculations, they evolved into effective stream processors that are able to perform Single Instruction Multiple Data (SIMD) operations efficiently. To satisfy the growing demand for increasingly realistic and complicated virtual environments in games, visual effects techniques became more complex. As a consequence, the complexity of operations that could be performed on GPUs increased to the point where arbitrary operations could be applied to arbitrary spaces in memory — ideal for parallel computing. While the complexity and power of GPUs has been increasing for some time, it has only recently become possible to access the computing power of these processors directly.

The ability to program graphics cards was initially introduced through programmable shaders in OpenGL. Shaders were intended to allow the creation of more complex visual effects, but researchers leveraged them to use GPUs to perform general simulations [62]. The introduction of higher-level Application Programming Interfaces (APIs) such as GLSL, Cg and HLSL made programmable shaders more accessible, but arbitrary simulations still had to be algorithmically adjusted to fit within the bounds of the graphics pipeline. Despite these limitations researchers were able to use these shaders for a variety of non-graphical tasks, such as fluid [89] and particle simulations[43]. While significant performance gains using graphics cards were achievable, accessing their full potential as stream processors remained awkward — it was still not possible to perform arbitrary computation outside the confines of the graphics pipeline.

In 2006 NVIDIA released the Compute Unified Device Architecture (CUDA), which allowed

general computation to be performed on GPUs through the *C for CUDA* API. To compete with CUDA, ATI released *Close To Metal* (CTM) in 2006 for GPGPU on its range of graphics cards, which has since been incorporated as part of the *AMD Stream SDK* as the *Compute Abstraction Layer*. Initially, these proprietary APIs were the most common methods for programming GPUs, but OpenCL (Open Computing Language) has recently gained popularity as an alternative. OpenCL is an open standard that has been widely adopted by hardware manufacturers and provides a standard interface for cross-platform programming (specifically for High Performance Computing) for a range of different computing hardware. AMD[1] and NVIDIA have both included support for *OpenCL* , with AMD shifting their focus to OpenCL. Alongside their support for OpenCL, NVIDIA has continued to develop CUDA features and it is still the preferred language for NVIDIA graphics cards, as CUDA has more mature support libraries than the generalised OpenCL standard.

With these high-level languages allowing arbitrary code execution and data access on graphics hardware it is now possible to make full use of their massively parallel architecture. The power of GPUs has been applied to a wide range of fields such as: biomedical applications [31], n-body simulations [29], matrix operations [7], Fourier transforms [26], financial mathematics [73] and fluid simulation [32, 85, 55]. Many of these problems run tens and even hundreds of times faster on GPUs than they do on CPUs.

These orders-of-magnitude improvements fundamentally change the way in which simulations can be used. For example, the 74$\times$ speed-up for Monte Carlo Simulations reported by Singla [73] means that simulations that used to take three days can be completed in less than an hour. As a result there has been significant interest in GPGPU computing because simulations that were previously too time consuming are now practical.

It is worth noting that many GPU-CPU comparisons compare highly-optimized GPU code against unoptimized CPU code and that potential performance gains are sometimes overstated [44]. Despite this, many algorithms are able to achieve at least an order of magnitude improvement when comparing modern GPUs and CPUs, because GPUs are capable of an order of magnitude more memory and instruction throughput. This potential for performance improvements are the motivation for exploring an implementation of the LBM on graphics hardware.

We have chosen to use CUDA in this work because it is still the platform of choice for GPGPU computing on NVIDIA graphics cards, and, when we started our research, OpenCL was still in its infancy, thus limiting access to all the latest NVIDIA GPU features. Writing efficient code for CUDA requires an understanding of the architecture. Therefore, before we present our CUDA implementation, we provide a brief overview of CUDA and the programming concepts particularly pertinent to our design.

[1] ATI was bought by AMD, and AMD have since retired the ATI brand.

4.1 The Compute Unified Device Architecture

Typical modern consumer CPUs have 1–4 Arithmetic Logic Units (ALUs) or cores, each able to process one or two threads simultaneously. In contrast NVIDIA Kepler GPUs have up to 8 Streaming Multiprocessors (SMXs), with each SMX containing up to 192 CUDA cores for performing parallel computation [19]. This means that a high-end GPU is able to perform hundreds more operations in parallel than a CPU and is the reason why GPUs are able to outperform CPUs in SIMD computation.

The CUDA programming model allows programmers to write C functions, called *kernels*, that can be executed on graphics hardware. It is common to refer to the graphics card as the *device*, and the CPU that is controlling the graphics card as the *host*. Code that is running on the host manages the execution of the kernels that run on the device. Since kernel execution requires minimal resources from the host, it is possible to perform computationally intensive work on the CPU in parallel with GPU computation. When kernels are scheduled for execution, threads are grouped into *blocks* and *grids* for parallel execution. Threads typically use their location in a thread block and grid to determine the part of global memory on which to perform operations.

A thread is the smallest logical unit of execution. Although each thread is logically separate and has access to its own local memory space, the GPU groups threads together as *warps*. Warps are the smallest physical unit of execution, with each warp executing the same instructions simultaneously. The current size of a warp is 32 threads. If different threads in the same warp follow different code branches (*divergence*), then the entire warp will execute each code branch with each thread only storing the results when it traverses the relevant code branch for that thread. This divergence effectively results in idle threads, so it is important to take care to ensure all threads within a warp follow the same code paths where possible.

Threads are logically grouped into blocks, and blocks are logically grouped into a grid. Thread blocks and grids can be thought of as a 3D array of threads and blocks, respectively. Different blocks form completely separate execution units, and therefore different blocks are unable to communicate or synchronise with each other directly. While indirect inter-block communication and synchronisation is possible on the GPU [93], it is much slower and less efficient than intra-block communication and synchronisation and can have a significant negative impact on performance if used unnecessarily.

It is worth noting that not all CUDA devices are the same. As NVIDIA has improved and redesigned their device architecture, new features and performance improvements have been added. In order to determine what features are available, each CUDA-enabled device has an associated *compute capability*. The compute capability is represented by a version number, that is divided into major and minor revision numbers. The major revision number refers to the architecture of the device (i.e. Kepler, Fermi or Telsa) and the minor revision number corresponds to an incremental improvement of the device architecture [16]. In this work we use devices with

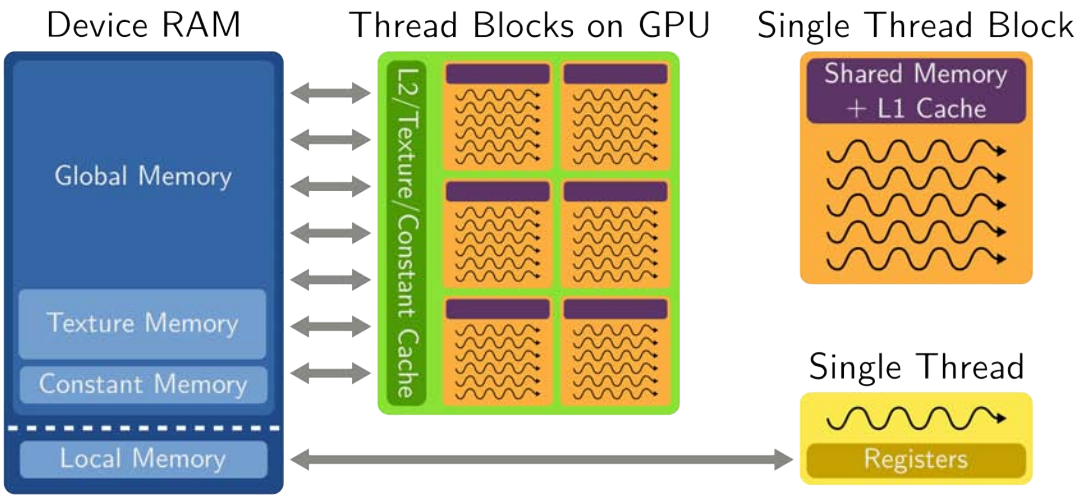


Figure 4.1: Device memory spaces each have their own characteristics. Global, texture and constant memory are all forms of off-chip memory whose transfer rates are enhanced by on-chip caches and are available to all threads. Quick-access shared memory is available to all threads within the same thread block. The memory space for shared memory is also shared by the L1 cache. Threads all have access to thread-specific registers that are not accessible by other threads. Off-chip local memory is used to store thread-specific values when there is no longer enough space in the registers. L1 and L2 caches are not present for GPUs with compute capability < 2.0 which means that reads to global memory are performed directly with no caching. (Aspects of diagram inspired by diagrams in the *CUDA Programming Guide* [16])

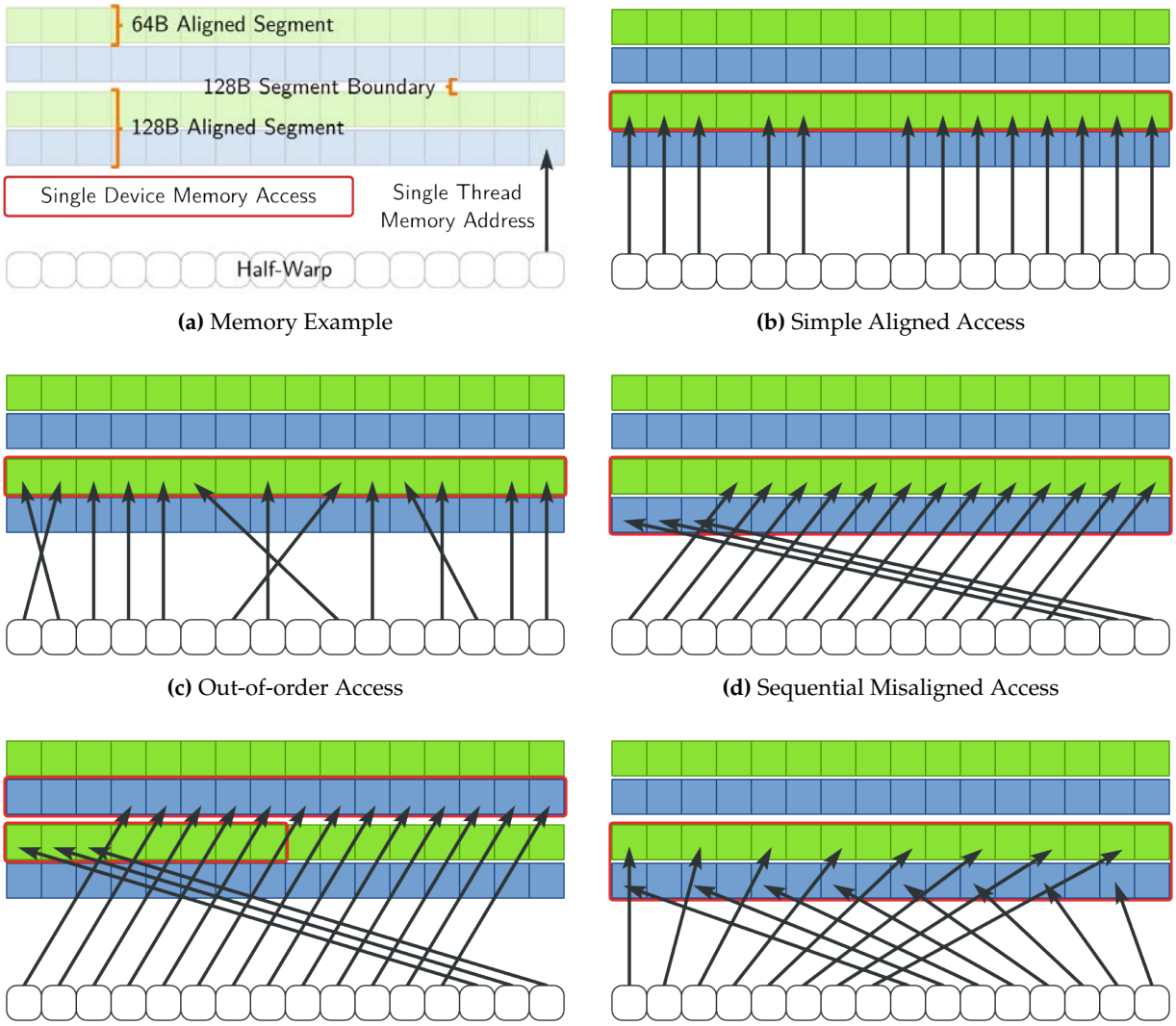
compute capabilities of 2.1 and 3.0.

In our implementation of the LBM using CUDA, the two most important considerations are our use of the various memory spaces on the graphics card and how we make use of latency hiding (performing computation in parallel with time-consuming memory accesses) at various levels of execution. The following sections will discuss these in greater detail.

4.1.1 CUDA Memory Spaces

Using the right type of memory is important when designing algorithms for implementation on the GPU [15]. Algorithms can easily be limited by their memory throughput if they make use of slower memory spaces with poor memory access patterns, or use faster memory spaces incorrectly.

Device memory is located on the graphics card and is physically separate from the CPU's *host memory*. Before data can be used on the device and before results can be obtained, data must be transferred between host and device memory. These transfers are expensive because they are limited by the speed of the bus between the CPU and graphics card — at the time of writing, top-of-the-range GPUs have a theoretical memory bandwidth of up to 336 GB/s [18], while PCI Express has a maximum bandwidth of only 32 GB/s [10]. Therefore, programs should always maximize the number of kernel executions per host-device memory transfer.



(a) Memory Example

(b) Simple Aligned Access

(c) Out-of-order Access

(d) Sequential Misaligned Access

(e) Sequential Misaligned Access, Multiple Reads

(f) Strided Access

Figure 4.2: It is important to ensure that global memory accesses are coalesced to maximize the use of available memory bandwidth. Simple access patterns (b) are aligned to 64 B and 128 B memory blocks and are coalesced into single transactions from global memory. For devices with compute capability ≥ 1.2 this alignment requirement is relaxed and more complex misaligned access patterns can be resolved into one or two reads from global memory (c–f). The *Fermi* architecture allows the same access patterns with coalescing occurring with full warps instead of half-warps. (Figure inspired by diagrams in the *CUDA Best Practices Guide* [15])

To allow for easier host/device memory management, GPUs with compute capability ≥ 2.1 are able to directly access host memory from within a kernel by making use of CUDA's Unified Virtual Addressing. NVIDIA's Kepler Architecture provides further simplification with Unified Memory, where a single pointer can be used to reference both host and device memory and the system automatically migrates data between host and device memory to ensure consistency [17].

Device memory can be divided into *off-chip* and *on-chip* memory. Off-chip memory is stored separately from the GPU in the device RAM. On-chip memory is stored on the GPU and is much faster, but has much less storage space. Figure 4.1 shows the types and scopes of device memory.

Global memory is an off-chip memory that is accessible by all threads. It has the slowest access times of all device memory spaces, but it is also the largest memory space. Structuring memory accesses to ensure all threads in a warp access a single aligned memory segment results in only a single global memory transaction. This behaviour is known as coalescing and is important for maximising memory throughput. Figure 4.2 outlines how memory accesses should be structured to ensure coalescing. On devices with compute capability ≥ 2.0, accesses to global memory are cached in a block-specific L1 cache and a common L2 cache, but it is still recommended that data be manually moved between global memory and faster memory spaces if it is used repeatedly during calculations [15].

Registers provide on-chip storage for individual threads. Registers have extremely fast access times (usually zero extra clock cycles per instruction), but it is not possible to directly share their values between threads. There are only a limited hardware-specific number of registers available to each thread block, and once these are all used the compiler automatically assigns values to off-chip *local memory* — this is known as register pressure. Local memory is as slow as global memory and its use should generally be avoided.

Threads within the same thread block have access to a common *shared memory* that has latency that is typically 100 times less than global and local memory. This memory can be used to share local data between threads from the same thread block, but space is limited and it is also used to store kernel parameters (and the L1 cache in devices with compute capability ≥ 2.0). Using shared memory can be extremely effective for manually caching values from global memory when different threads within the same block require access to the same values.

Since registers and shared memory are limited by the resources available on the GPU, their usage can affect the number of thread blocks being processed simultaneously by each SMX. Thread blocks that use too many SMX resources can result in an SMX being underutilized during kernel execution, because there are too few resources available to process another thread block in parallel. The ratio of warps being processed (resident warps) to the maximum number of resident warps is known as occupancy [16]. Although achieving high occupancy is not a requirement for optimal performance, it can help increase the amount of latency hiding during kernel execution, especially for kernels that require synchronization.

Texture memory and *constant memory* are special read-only memory spaces stored in global memory, but supported by an on-chip cache. On a cache hit, reading from constant memory can be as fast as reading from a register, but a cache miss requires a normal read from global memory. Therefore, constant memory is often used to store repeatedly used constants in order to reduce register pressure. The texture memory cache takes advantage of 2D spacial locality when performing memory reads and caching values. It is designed to perform reads with constant latency so cache hits only help reduce the DRAM bandwidth demand, and do not reduce the global memory fetch latency. The Kepler architecture relaxes the restrictions on the texture pipeline cache,

allowing large segments of read-only data in global memory to be cached without being bound to a texture.

4.1.2 Latency Hiding with CUDA

Global memory accesses on the GPU are expensive and can have latencies of hundreds or even thousands of GPU clock cycles. Waiting for a memory transfer to complete effectively wastes clock cycles that could have been spent on computation. *Latency hiding* in the context of CUDA is the practice of structuring algorithms so that GPU computation occurs while waiting for slow memory accesses to complete. There are three avenues that can explored: overlapping kernel execution and host-device memory transfers, pre-fetching data to faster memory, and ensuring high occupancy.

Overlapping kernel execution with host-device memory transfers is the simplest form of latency hiding to implement. If an algorithm has multiple kernels and one of the kernels does not change the data that needs to be transferred off the graphics card, then it is possible to schedule the execution of that kernel and the transfer of data off the graphics card simultaneously. This hides the cost of the data transfer behind the cost of the kernel execution — the former being an unavoidable cost that is part of the algorithm. This can be extremely beneficial to the overall performance of a GPU implementation because, in some cases, the expensive data transfer can be completely hidden behind the computation.

Accessing global memory on the GPU is the second most expensive memory operation for a GPU algorithm. If portions of global memory are repeatedly accessed, thousands of GPU clock cycles can be wasted waiting for memory accesses to complete. Since lower latency memory spaces are small, it is often not possible to store all the repeatedly accessed memory in these spaces. In order to get around these limitations it is possible to temporarily fetch values that are relevant to specific threads or thread blocks into registers or shared memory, respectively. Once the values are in the lower latency memory, multiple memory accesses can be performed without incurring the cost of a global memory access each time. When the values are no longer needed, they can be written back to global memory. Using this technique, the cost of multiple global read-write pairs can be reduced to a single read-write pair, with the rest being hidden through the use of lower-latency memory. Unfortunately, this approach is sometimes not applicable when it is necessary to ensure global memory is consistent across all thread blocks.

The last latency hiding technique takes place at the warp scheduling level and takes advantage of the way warps are scheduled for execution on the GPU. When threads in a warp attempt to access global memory, they are effectively put into a waiting area on the GPU until their memory accesses complete. At this point a new warp can be scheduled for execution. If there are no data dependencies between operations for each warp, it is possible for calculations to be performed for one warp, while another warp waits for its global memory accesses to complete. Maximizing

occupancy (i.e. concurrently executing thread blocks), increases the number of opportunities for the warp scheduler to perform this kind of latency hiding. It is also possible to structure kernels such that global memory accesses are not interleaved with register computation, thus creating multiple lines of execution to be performed in one warp, while awaiting the completion of memory accesses in another.

— ∼ —

In Chapter 3 we presented the mathematics behind our model for fluid simulation, thus providing an overview of the model we have implemented. In this chapter, we have described the platform that we have used, along with important related design concepts and optimization techniques. Therefore, we are now ready to present our design for a CUDA implementation of free-surface LBM fluid simulation.

Chapter 5

The Lattice Boltzmann Method on the GPU

Fluid simulation on the GPU has been popular since the introduction of programmable shaders. Both particle-based and grid-based methods map well onto SIMD architectures, so it was natural for attention to shift towards graphics hardware to look for performance gains. The three broad methods of simulating fluids all have significant performance benefits when run on graphics hardware (SPH [42, 94, 25], Grid-based methods [28, 14]), but Lattice Boltzmann Methods [88, 54, 85] are particularly well suited to graphics hardware, due to their linear, grid-based and local nature. Researchers had already been experimenting with LB simulations on compute clusters [74] by the time GPUs became a viable option for simulations and it was well established that the LBM's linearity allows for efficient memory usage and its reliance purely on local information means data transfer throughout the simulation can be kept low [39, 2]. These are ideal conditions for GPU execution.

One of the very first GPU LBM implementations [45] was implemented before CUDA and OpenCL were available. This implementation used programmable vertex and pixel shaders to perform the simulation calculations for a basic gas simulation. The 3D simulation information was arranged in 2D texture memory by grouping 2D textures as one would for volume rendering. Since the GPU is designed to operate on textures with four colour channels, different distribution functions were placed in separate color channels for the same texture. This allowed operations on the same texture to be applied to multiple distribution functions simultaneously. This simplified LBM implementation achieved a speedup of 50×. Their implementation was extended to handle dynamic complex obstacle boundaries with achievable speedups of 8× to 15× [46] and multi-resolution simulation grids[95] with achievable speedups of up to 28×.

The D2Q9 LB model has also since been implemented using CUDA [85]. This implementation focussed on taking full advantage of coalesced memory accesses by conforming to the limited op-

timal memory access patterns available to cards with compute capability 1.x[1]. This was achieved by using shared memory to store intermediate results before writing them to global memory. The reported speed up was 22×. This approach was extended to the D3Q19 model [58] and minor improvements were made to memory throughput performance by eliminating misaligned global memory writes through the use of separate arrays for reading and writing distribution functions. However, these improvements come at the cost of doubling device memory usage.

These examples of the LBM for GPUs all focus on simulating the fluid internals and are not able to simulate the complex behaviours of the fluid surface. Our method includes the simulation of the fluid surface on the graphics card, leveraging techniques used by Thürey [82] and Reid [69] in their parallel implementations of the LBM.

Our GPU LBM implementation can be divided into three distinct steps: stream, collide and interface movement. The stream step is responsible for the flow of distribution functions throughout the lattice and the collide step calculates changes based on the interactions between neighbouring distribution functions using the LBGK collision operator, described in Section 3.1. The interface movement step is responsible for simulating the movement of the fluid's free-surface using the VOF method, described in Section 3.3.

Our approach to the implementation of the Lattice Boltzmann is to achieve optimal performance for the stream and collide steps because they perform the bulk of the computation. While the stream and collide steps do most of the work, we cannot forget about the steps that perform the movement of the fluid interface. For each time step, they track the flow of mass throughout the simulation and change position of the fluid interface. These operations are not suited for GPU execution because they require non-contiguous memory accesses, branching and ordered execution. Despite these limitations, fluid interface management has to be executed on the GPU because the data transfer costs, that would result from executing these steps on the CPU, are prohibitively expensive.

Before we delve into the details our CUDA implementation, we first provide an overview of the system used to drive the simulation and the details of how we structure the simulation lattice in device memory.

5.1 Overview

Our LBM implementation can be broken into two parts: fluid simulation and geometry extraction, with the details of the fluid simulation as the focus of this work. Although geometry extraction is required for producing visual results of the underlying fluid simulation, exploring its optimal implementation is beyond the scope of this thesis. Therefore, we have logically separated these two parts of the simulation. This allows our fluid simulation results to be unaffected by our geometry extraction implementation. In our high-level architecture (Figure 5.1), we separate

[1]These restrictions have since been relaxed for cards with compute capability ≥ 2.

Figure 5.1: The broad structure of our implementation. We use two threads on the CPU: one for managing the simulation (Simulation Thread), and the other for converting the simulation output, $\varepsilon(\mathbf{x})$, into 3D geometry to be displayed or stored (Output Thread).

these two processes into different threads of execution and ensure they can run independently. This separation also allows us to easily switch between our CPU and GPU implementations for comparing results.

Although geometry extraction is logically separate, it still relies on the output of the fluid simulation, $\varepsilon(\mathbf{x})$. This means that the geometry extraction could still potentially block the fluid simulation if it needs to access $\varepsilon(\mathbf{x}, t)$, when the simulation is ready to output $\varepsilon(\mathbf{x}, t+1)$. To mitigate this we use double-buffering so that geometry extraction can occur in parallel with host-device data transfer. In order to extract geometry from $\varepsilon(\mathbf{x})$, we use an implementation of *Marching Cubes* [47] by Reid [69], which is based on work by Bourke [8].

The flow of execution of our GPU simulation thread is shown in Figure 5.2. The very first step of the simulation is to initialize both host and device memory so that it can be used by the simulation. The streaming and interface movement steps are divided into multiple kernels, while the collide step is implemented as a single kernel. The designs of each step are discussed in Sections 5.3, 5.4 and 5.5 below. Since the streaming step does not modify $\varepsilon(\mathbf{x})$, we are able to overlap streaming with data transfer. We also limit the host-device memory transfers to those necessary to maintain the desired frame rate. Since we know Δt and we can choose a desired frame rate, r, we only need to transfer data every $\frac{1}{r\Delta t}$ time steps.

Each of our kernels operate on the same global memory structures. We, therefore, describe how we structure our simulation lattice in memory before presenting each kernel design.

5.2 Memory Allocation

As discussed above in Section 4.1.1, CUDA supports multiple memory spaces, each with their own purpose and each suited for specific tasks. Our CUDA LBM implementation makes use of pitched global memory for storing lattice cell information, page-locked host memory for efficient host-GPU memory transfer and constant memory for storing various constants.

Global memory on the GPU is the general data store for large sets of data and we use it to store all the lattice information: cell type, distribution functions, fill fraction, pressure, mass,

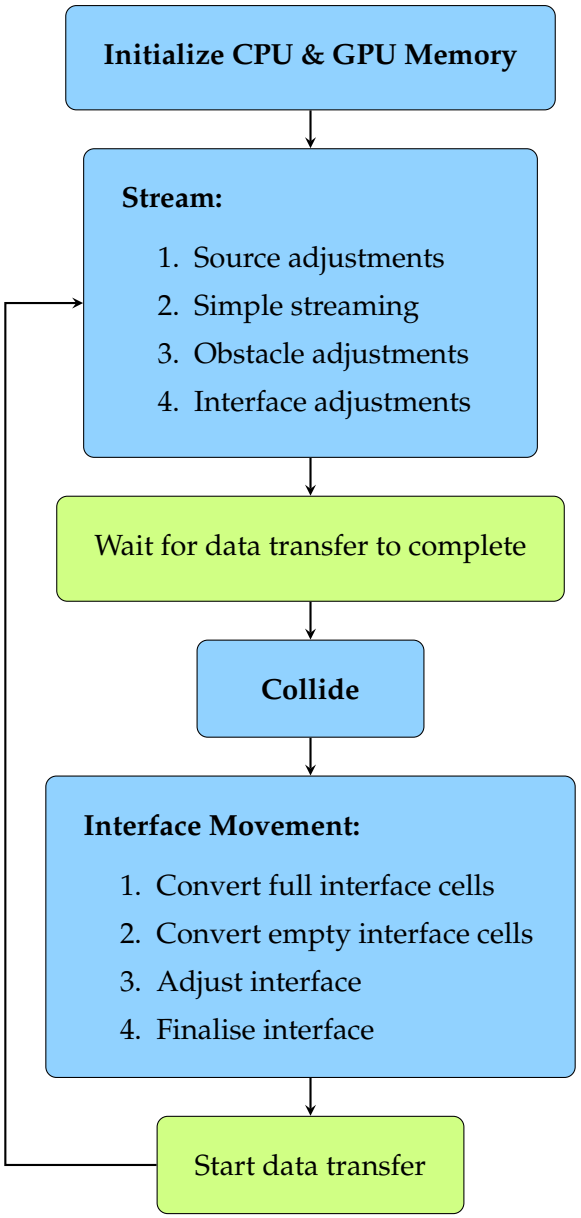


Figure 5.2: The flow of execution for our GPU LBM implementation. The blue nodes represent the steps of fluid simulation and the green nodes highlight the asynchronous data transfer that takes place during the simulation.

neighbour counts and velocity. To keep the structure simple, our memory layout mirrors that of the simulation lattice — we allocate width×height×depth blocks of memory for each of the lattice fields listed above, and lattice position (x, y, z) corresponds to memory position (x, y, z). This straightforward approach allows for optimal coalescing during the streaming steps as adjacent lattice cells use adjacent memory spaces. Given this memory layout, a common initialization operation for GPU kernels is for individual threads to identify those parts of global memory they should work on based on their thread block and grid coordinates, with t_x, t_y and t_z representing

the x-, y- and z-coordinates of a thread, respectively. These coordinates are calculated as follows:

$$\begin{aligned} t_x &= \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x} \\ t_y &= \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y} \\ t_z &= \text{blockIdx.z} \times \text{blockDim.z} + \text{threadIdx.z} \end{aligned}$$

where *threadIdx* provides the coordinates of the thread within the thread block, *blockIdx* provides the coordinates of the thread block in the grid, and *blockDim* provides the dimensions of the thread block. Each of these variables are provided by CUDA and are accessible to all GPU threads.

Unlike our CPU implementation, which is based on the work by Reid [69], we only keep a single copy of the distribution functions in memory because GPU memory is relatively scarce. This change is made possible by restructuring the algorithm for streaming to perform in-place distribution transfers. We also follow the CUDA Best Practices [15] and allocate pitched GPU memory — memory that is specially allocated to ensure proper alignment for coalescing, thus maximizing data access efficiency.

We further reduce the memory footprint of the simulation by using a single integer field to store information about both the empty and fluid neighbour counts. This is possible because the values being stored are bounded and small: $0 \leq n_E, n_F < Q$, where Q is the number of directions used by the LBM model. We store n_E in the first two digits and n_F in the second two digits of each integer. Adjusting the neighbour counts only involves simple addition and subtraction of 1 for n_E and 100 for n_F. The individual values for each count can easily be obtained as follows:

$$n_E = n \,\%\, 100 \qquad n_F = \left\lfloor \frac{n}{100} \right\rfloor \,\%\, 100$$

This technique also takes advantage of the fact that simple calculations are relatively cheap when compared to global memory accesses on the GPU.

The output of our simulation at each time step is the fill fraction for each lattice cell, represented as a field, $\varepsilon(\mathbf{x})$. In order to make sure the transfers from the GPU to host memory are as efficient as possible we use page-locked host memory provided by `cudaMallocHost`. This memory space is locked in RAM and cannot be paged onto disk, which means the memory can be directly accessed by the device, facilitating faster data transfer.

We do not make use of the Unified Memory feature in CUDA 6.0 because it would also reduce simulation performance for three reasons:

1. We extract geometry on the CPU from $\varepsilon(\mathbf{x})$ while the GPU simulates the next time step. Extra synchronization would be required to ensure updates by the GPU simulation do not affect the geometry extraction on the CPU. This would lead to the GPU having to wait for the CPU to complete geometry extraction thus impacting performance.

2. We make use of double buffering to prevent slow CPU operations from holding up GPU data transfers. Implementing double buffering in GPU memory would take up valuable memory space on the GPU, limiting the size of scenes that could be simulated.

3. In our implementation we perform device-to-host data transfers when they can safely overlap with calculations on the GPU. With this approach we are able to completely hide the cost of these data transfers. With Unified Memory, the device-to-host transfers would be managed by CUDA and may not happen at these optimal times.

Our use of pitched GPU memory, and page-locked host memory ensure we maximize the efficiency of GPU memory accesses and device-to-host memory transfers, respectively. Therefore, we do not make use of Unified Memory. However, it is worth noting that in a lot of situations the simplicity and maintainability that comes with using Unified Memory can outweigh the benefit of performance gains from manually managing separate host and device pointers.

Lastly, we store all our simulation constants in constant memory. Since these constants are used by all threads, making use of constant memory allows us to access variables at near register speed without them taking up valuable register space. This plays a significant part in alleviating register pressure in our more complex kernels. Since these constants easily fit within constant memory, we do not need to make use of texture memory or the Kepler read-only data cache. For reference, the complete list of values stored in constant memory is: $\mathbf{e}_i$, $\tilde{\imath}$ for each i, w_i, f_i^s, $\left(\sum_j (\mathbf{e}_i)_j\right)^2$, C^2, $18C^2$, $\frac{1}{6}$, v, v^2, $\mathbf{g}$, κ, $1-\kappa$, ρ_A, Q, lattice width, lattice height, lattice depth and pitch values for accessing pitched memory. These terms are defined in the glossary at the start of this thesis, and some of the compound terms are precomputed values for Equation 3.19.

5.3 Stream Step and Streaming Adjustments

The stream step is responsible for propagating the distribution functions throughout the simulation grid. Various adjustments are performed as part of this step to handle fluid sources, obstacles and the fluid interface. This step needs to:

1. ensure source cells stream predefined source distribution functions;

2. move distribution functions to adjacent fluid and interface cells, along each lattice direction;

3. manage the transfer of distribution functions that interact with static obstacles;

4. reconstruct distribution functions that should have been streamed from empty cells; and

5. track the changes in cell mass as a result of the distribution function streaming.

Our approach to implementing streaming on the GPU is to separate the complexity of making adjustments for source cells, obstacles and the fluid surface from the core work of moving distribution functions. This separation ensures that the bulk of streaming involves simple, identical

tasks that can be performed in parallel — tasks that are perfectly suited to the GPU. This approach works well for streaming because the complexity only applies to a small percentage of cells in the simulation lattice: interface, source and obstacle cells. Typically, these cells account for fewer than 3% of all lattice cells. The complex operations are then performed by separate kernels that can be structured optimally for the operations they perform. A brief overview of these different kernels is provided below, with more details on their design given in the following sections.

The source adjustments need to occur before streaming to ensure that the correct distribution functions are propagated when they are streamed to adjacent cells. This action is performed in a separate kernel (described in Section 5.3.1) that is executed before the distribution functions are moved.

Propagating all distribution functions throughout the lattice is the bulk of the work required for streaming — in a scene without a fluid surface or obstacles this is the only action required. In order to ensure that this propagation happens efficiently the stream step is divided into five kernels that specialise in streaming distribution functions along the x-axis, y-axis, z-axis, xz diagonal and xy diagonal. Streaming along the yz diagonal is performed by streaming the distribution function along the y-axis then the z-axis because movement along the yz diagonal does not allow for efficient coalescing. The details of this approach are discussed in Section 5.3.2.

Interactions with static obstacles are managed using the *no-slip* boundary conditions (Equation 3.7). A separate kernel is executed after core streaming that corrects the movement of distribution functions that were streamed onto obstacle cells. Section 5.3.3 discusses the design of this kernel.

The final interface adjustment actions are necessary for simulating the interactions between a liquid and a gas. Both these actions require reading the type of all cells neighbouring an interface cell. Since this is an expensive operation and both actions do not interfere with one another, they have been grouped into a single *interface adjustment* kernel. Section 5.3.4 discusses the design of this kernel.

5.3.1 Source Adjustments

The source adjustment kernel is the simplest of all the streaming kernels. For each source cell in the scene, the kernel must reset the distribution functions to their source values and ensure that all source neighbours are either interface or fluid cells. The algorithm implemented by the kernel is shown in Pseudocode 1.

In a normal simulation, source cells are a small percentage ($< 1\%$) of all cells. This means that most threads that execute this kernel will only perform a single global memory read to identify the type of the cell. Therefore, we have ensured that this part of the algorithm has coalesced global reads to maximize the memory throughput.

It is likely that only a few threads in a single warp will enter the outer *if* statement (line

Pseudocode 1 The pseudocode for the source adjustment kernel. This pseudocode assumes that x, y and z are within the bounds of the simulation.

```
1:  x = (t_x, t_y, t_z)
2:  if x is a source then
3:      f_0(x) = f_0^s
4:      for i = 1 → Q do
5:          f_i(x) = f_i^s
6:          if x + e_i is empty then
7:              initialise x + e_i as interface cell
8:          end if
9:      end for
10: end if
```

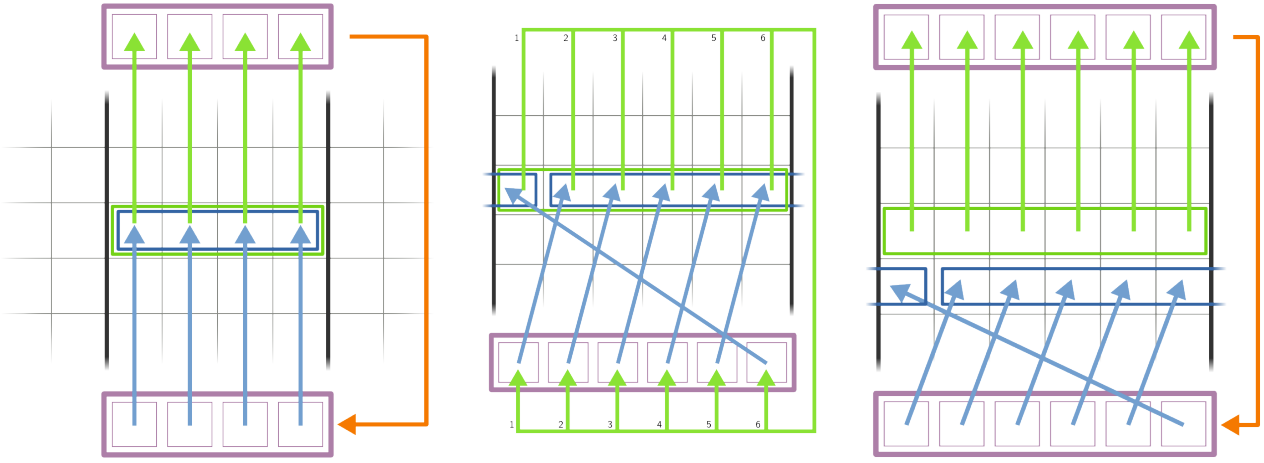
2), and of those even fewer are likely to enter the inner *if* statement (line 6). This means that any global memory operations will be inefficient because coalescing is unlikely. While writing values to global memory (m, ε, and f_j) to initialize a new interface cell is unavoidable, the source distribution function values (f_i^s) are stored in constant memory to ensure rapid access.

The final aspect of this kernel worth mentioning is how it maps to thread blocks and grids. Each thread in a thread block is responsible for a only single cell in the simulation lattice. The thread blocks and grids are then arranged such that each cell is only processed by a single thread, to avoid race conditions and duplication of work. By design, the thread block can be of arbitrary dimensions, although the x dimension of the thread block will typically be a multiple of warp size (32). This allows freedom in choosing launch configurations to best suit the scene being simulated. For these reasons, this model is common for most of our kernels.

5.3.2 Simple Distribution Function Transfers

Moving distribution functions along each non-zero distribution function direction ($\mathbf{e}_1$–$\mathbf{e}_{18}$) is described by a single transfer operation: Equation 3.1. While the operation itself is simple, mapping the operation to work effectively on the GPU for each $\mathbf{e}_i$ requires careful design. Before examining the algorithms used by the five core streaming kernels (x-axis streaming, y-axis streaming, z-axis streaming, xy-diagonal streaming and xz-diagonal streaming), we first discuss some general considerations that influenced the design of each streaming kernel.

The most important restriction in the design of the streaming kernels is that global memory accesses must coalesce as much as possible. By separating the operation of moving distribution functions into a separate kernel and structuring memory to mirror the structure of the lattice, the distributions align naturally for perfect coalescing. In the worst case we have a sequential misaligned memory access across two memory segments, which only results in a single extra coalesced read on devices with compute capability ≥ 1.2 (See Figure 4.2e in Chapter 4). Therefore, at minimum we need to ensure that warps access sequential elements in memory, and, to achieve optimal coalescing, each warp should be aligned to the boundaries of each 128 B memory block.



(a) y/z-axis streaming **(b)** x-axis streaming **(c)** diagonal streaming

Figure 5.3: All LB streaming steps are equivalent to repetitions of one of these three operations. First values are read from global memory (green) into temporary registers (purple), then the values are written from temporary registers to global memory (blue). y/z-axis and diagonal streaming require an extra step (orange) to move the recently read values into the other set of registers in preparation for them to be written to global memory during the next iteration. x-axis streaming does not need to store values across iterations, because it writes the values immediately after they are read. For x-axis and diagonal streaming, the width of the thread block must correspond to the width of the lattice.

In practice, this means that the x-dimension of thread blocks should be a multiple of 32, and that warps should access distribution functions that are adjacent along the x-axis.

We also have to be wary of race conditions along the boundaries between different thread blocks and between different warps from the same thread block. Synchronization between warps within the same thread block is relatively inexpensive, and is used to prevent values from being overwritten before they are read. However, synchronization between different thread blocks is not possible without degrading performance. Coupled with the fact that the device can schedule any thread block at any time, this means the streaming kernels must ensure that the distribution functions accessed by any two thread blocks do not overlap. This limits the axes along which the lattice can be divided for kernel execution, which allows us to make assumptions that simplify the design of the individual core streaming kernels.

Figure 5.3 provides a visual overview of the core algorithm underlying the implementation of each of the streaming kernels. The basic idea behind each kernel is to read distribution functions in the direction being streamed, then write the previously read distribution functions over the memory that was just accessed. Since the distribution functions are laid out in memory in the same logical order as the lattice, this process works slightly differently depending on the axis along which streaming occurs.

A significant aspect of our streaming design is that we blindly stream all distribution functions, regardless of source or destination cell type. There are two reasons for doing this. The first is that in order to make the decision to selectively stream, a global memory read is required

Pseudocode 2 The pseudocode for streaming along the y-axis. d represents a single lattice direction along the y-axis (i.e. $d \in \{3, 4\}$) and n is lattice height. We assume x, y and z are within the bounds of the lattice and that the coordinates of $\mathbf{x}$ are correctly wrapped (e.g. $-1 \rightarrow height - 1$ and $height \rightarrow 0$). z-axis streaming follows the same algorithm, but with $\mathbf{x} = \left(t_x, t_y, -\frac{1}{2}\right) + \frac{1}{2}\mathbf{e}_d$ initially, n as lattice depth and $d \in \{5, 6\}$.

$\mathbf{x} = \left(t_x, -\frac{1}{2}, t_z\right) + \frac{1}{2}\mathbf{e}_d$
$t_1 = f_d\left(\mathbf{x} - \mathbf{e}_d\right)$
for n iterations **do**
    $t_2 = f_d\left(\mathbf{x}\right)$
    `__syncthreads()`
    $f_d\left(\mathbf{x}\right) = t_1$
    $t_1 = t_2$
    $\mathbf{x} += \mathbf{e}_d$
end for
$f_d\left(\mathbf{x}\right) = t_1$

to determine the destination cell type. This would mean an extra global memory operation per fluid cell, to save a single global memory operation per non-fluid cell. Thus, if the scene is mostly fluid cells, selective streaming would actually require more total global memory operations. The second reason is that by streaming blindly, our adjustment kernels are easily able to identify incorrectly streamed distribution functions and perform corrections. If those values were selectively streamed, they would have been overwritten by correctly streamed values, so extra work would have to be done to ensure those values were not lost. To summarise, selective streaming does not provide a guaranteed reduction in global memory operations and it leads to more complex adjustment operations. Therefore, we opted for simplicity and blindly stream all distribution functions. The fact that blind streaming by definition leads to simpler streaming kernels is just an added bonus.

We first look at the y- and z-axis streaming kernels (Pseudocode 2 and Figure 5.3a). The structure of these kernels is almost identical, only the direction of streaming differs. In-place streaming is only possible by having each GPU thread stream its own column of distribution functions in the correct direction to avoid race conditions. Although any start point is possible, we choose to start streaming at the borders of the lattice. With the start point selected, the algorithm simply moves distribution functions one lattice cell at a time, until the last movement overwrites the value of the start point. With this structure, the only limitation for the launch configuration is that the thread block width must be a multiple of warp size (32) to ensure coalesced memory access.

A curious aspect of the design of the y- and z- streaming kernels is our use of the CUDA function `__syncthreads`. This function blocks threads within a thread block until they all reach the synchronization point. It is primarily used to ensure shared memory is ready for use by all threads in a thread block after it has been pre-fetched from global memory. Since all threads in a warp are by definition synchronized, its value in ensuring coalescing is limited. How-

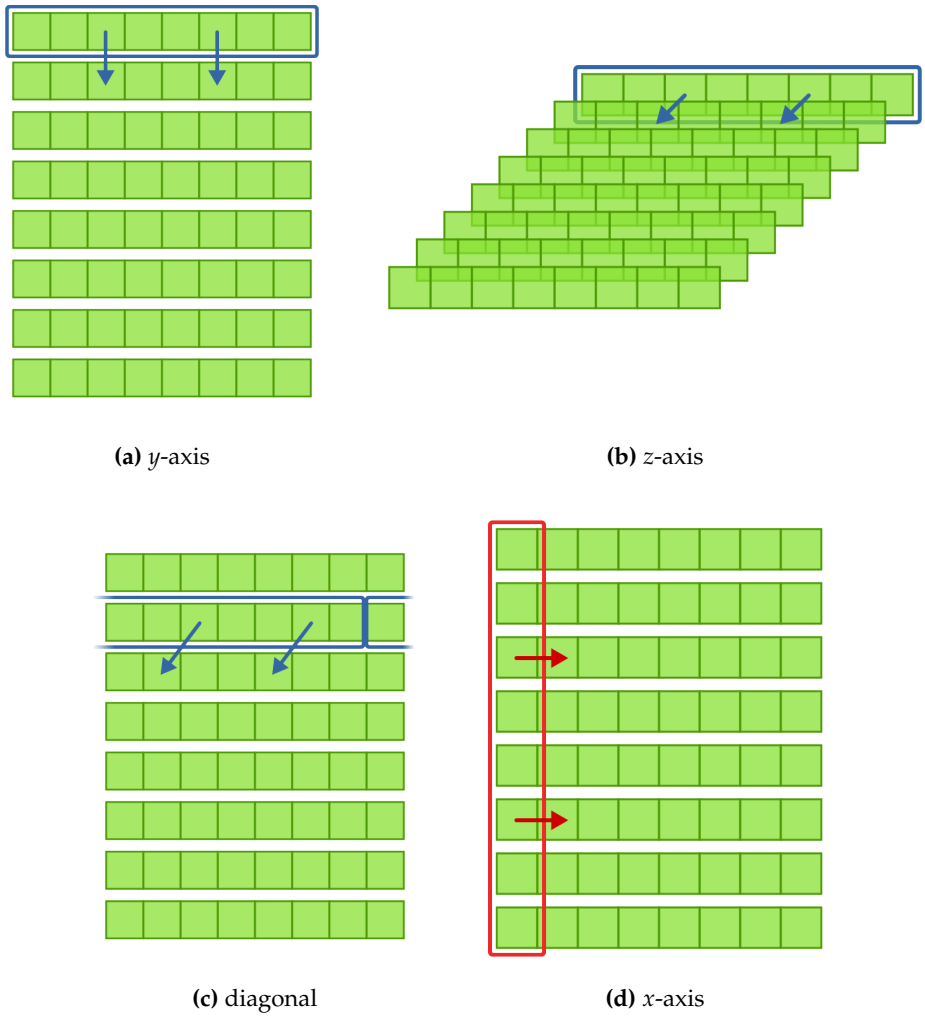
Pseudocode 3 The pseudocode for streaming along the x-axis, where d represents a single lattice direction along the x-axis (i.e. $d \in \{1, 2\}$). We assume that x, y and z_e are within the bounds of the lattice and the coordinates of $\mathbf{x}$ and $\mathbf{x}'$ are correctly wrapped (i.e. $-1 \rightarrow width - 1$ and $width \rightarrow 0$).

$r = \left\lceil \frac{\text{depth}}{\text{gridDim.x}} \right\rceil$
$z_s = r \times \text{blockIdx.x}$
$z_e = z_s + r$
$\mathbf{x} = \left(\text{threadIdx.x}, t_y, z_s\right)$
$\mathbf{x}' = \mathbf{x} + \mathbf{e}_d$
$\mathbf{j} = (0, 1, 0)$
for $z = z_s \rightarrow z_e$ **do**
    $t = f_d\left(\mathbf{x}\right)$
    $\mathbf{x} += \mathbf{j}$
    `__syncthreads()`
    $f_d\left(\mathbf{x}'\right) = t$
    $\mathbf{x}' += \mathbf{j}$
end for

ever, during the design of these kernels we noticed that execution times are slightly faster with `__syncthreads` included between each global memory read-write pair and have, therefore, included it in our final design.

The algorithm for x-axis streaming is given in Pseudocode 3 and Figure 5.3b. While the structure of the previous kernel would also work for x-axis streaming, it would not align for coalescing — the direction of an individual thread's column would be along a row of memory instead of perpendicular to it (see Figure 5.4). Therefore, we instead have all threads in a thread block stream a single row simultaneously: each thread reads a distribution function, then each thread writes that distribution function to the adjacent lattice position. This is only possible because we synchronize all threads in the thread block before each write. The disadvantage of this approach is that, by requiring a single thread block to read an entire lattice row simultaneously, we enforce a limit on lattice width. However, in practice this is not a significant limitation because either we can rotate the scene so that the width is not the widest dimension, or the lattice dimensions will be too large to fit into device memory anyway. The new algorithm also has its advantages, since each row translation is a single self-contained step. This allows us to schedule more thread blocks during kernel execution, because multiple thread blocks can work on the same xy or xz slice simultaneously, which allows for better latency hiding and thus a slight performance improvement. We make use of this freedom with a few extra initialization calculations, where each thread calculates the range for which it operates based on the dimensions of the thread block and grid.

Streaming along the xy- and xz-diagonals has the limitations of all single axis streaming directions, because it needs to happen along the x axis and one of the other axes. We, therefore, combined the algorithms from the single axis streaming kernels to create the diagonal streaming kernels (Pseudocode 4 and Figure 5.3c). As with x-axis streaming, each thread block moves an



(a) y-axis **(b)** z-axis

(c) diagonal **(d)** x-axis

Figure 5.4: The algorithm for streaming along the y-axis, the z-axis and the diagonal allows for contiguous memory accesses (the blue rectangle aligns with the green aligned memory segment). The same algorithm does not apply for x-axis streaming, because contiguous memory access is not possible, given the memory layout that we have used (the red rectangle intersects with multiple green aligned memory segments).

entire row at the same time, but in these kernels the destination address is the row immediately above or below the current row, depending on the direction of the distribution function. This difference means that each thread block must now manage the streaming for an entire slice of the lattice, similarly to how y- and z-axis streaming thread blocks are responsible for an entire column of their lattice slice. We also have to synchronize warps in the thread block to avoid data being overwritten before it is read by other warps. When selecting the dimensions for thread blocks and the grid, only a single thread block can be allocated per lattice slice.

The last distribution functions to be streamed are those on the yz plane. When designing the other streaming kernels, we were able to easily carve the lattice into slices parallel to either xy or xz plane slices that offer contiguous memory accesses for optimal coalescing. Unfortunately, we are unable to do this for streaming the diagonals on the yz plane, because we have to move distri-

Pseudocode 4 The pseudocode for streaming along the xy-diagonals. d represents a single lattice direction along an xy-diagonal (i.e. $d \in \{11, 12, 13, 14\}$) and n is lattice height. We assume x, y and z are within the bounds of the lattice and that the coordinates of $\mathbf{x}$ are correctly wrapped (e.g. $-1 \rightarrow height - 1$ and $height \rightarrow 0$). xz-diagonal streaming follows the same algorithm, but with $\mathbf{x} = \left(t_x, t_y, \frac{1}{2}w - \frac{1}{2}\right)$ initially, n as lattice depth and $d \in \{7, 8, 9, 10\}$.

$(u, v, w) = \mathbf{e}_d$
$\mathbf{i} = (0, v, 0)$
$\mathbf{x} = \left(t_x, \frac{1}{2}v - \frac{1}{2}, t_z\right)$
$\mathbf{x}' = \mathbf{x} + (u, 0, 0)$
$t_1 = f_d\left(\mathbf{x} - \mathbf{i}\right)$
for n iterations **do**
    $t_2 = f_d(\mathbf{x})$
    $\mathbf{x} -= \mathbf{i}$
    `__syncthreads()`
    $f_d\left(\mathbf{x}'\right) = t_1$
    $\mathbf{x}' -= \mathbf{i}$
    $t_1 = t_2$
end for
$f_d\left(\mathbf{x}'\right) = t_1$

Pseudocode 5 The algorithm used for managing streaming with obstacles. t is used as register storage for $f_i(\mathbf{x})$ to avoid reading the same value from global memory twice.

$\mathbf{x} = (t_x, t_y, t_z)$
if $\mathbf{x}$ is a boundary cell **then**
    for $i = 1 \rightarrow Q$ **do**
        $t = f_i(\mathbf{x})$
        if t is not 0 **then**
            $f_i(\mathbf{x}) = 0$
            $f_{\bar{i}}(\mathbf{x} + \mathbf{e}_i) = t$
        end if
    end for
end if

bution functions between adjacent xy and xz planes. We are also unable to slice up memory along the diagonal directions because those slices would have to wrap around the borders of the scene with the distribution functions. Although this would work for scenes with equal dimensions (the wrapping starts and ends at the same place), if a scene has dimensions that are relatively prime to each other the entire lattice becomes a single slice. A single slice would mean only a single thread block could be used, which would be an immense waste of the GPU's resources. Therefore, we divide yz diagonal streaming in two steps: first we stream these distribution functions with y-axis streaming, then we stream them with z-axis streaming. We do this at the same time as we stream the y- and z- axis distribution functions.

5.3.3 Obstacle Adjustments

Since our simple distribution function streaming ignores the structure of the fluid in the scene, the role of the obstacle adjustment kernel is to correct the streaming in the region around obstacles. We use no-slip boundary conditions (Equation 3.7) because they require the least information from global memory — only the distribution functions of each obstacle cell need be read. Free-slip and hybrid boundary conditions require knowledge of the surrounding obstacles in addition to the distribution functions in order to perform obstacle adjustments. This requires extra global memory operations for each obstacle cell and is therefore not well suited to the GPU. Since we use the no-slip boundary conditions, the correction involves taking distribution functions that were streaming into obstacle cells and moving them to the opposite distribution function of their source cell.

Pseudocode 5 outlines the algorithm of the kernel used to perform obstacle adjustments. Since each obstacle adjustment affects a single source and destination distribution function, we do not have race conditions or data dependencies as in the streaming kernels. We can therefore use the same thread block and grid structure as the source adjustment kernel, with a single GPU thread to manage the obstacle adjustments for a single lattice cell.

During each time step, our simple streaming kernels move distribution functions into obstacle cells. Therefore, when we find a non-zero distribution function in an obstacle cell, it implies that the no-slip boundary conditions needs to be applied to a neighbouring fluid cell. Once we apply the obstacle adjustment, we reset the obstacle's distribution function to zero so that the non-zero value does not break simulation correctness. We cache the initial read of the obstacle cell in a register (line 4) to avoid making two global memory reads when only one is required.

We also reduce the number of expensive global memory reads performed by the kernel by pre-computing the obstacle cells that might require obstacle adjustments during memory initialization. If an obstacle cell is completely surrounded by other obstacles, no distribution functions could possibly flow into it and would, therefore, never require obstacle adjustment. This means we only need be concerned with obstacle cells that could be adjacent to fluid cells. To incorporate this, at the start of the simulation we mark obstacle cells that are not completely surrounded by other obstacle cells as boundary cells, and only perform obstacle adjustments on these boundary cells. Since obstacles in scenes are unlikely to be thin planes, but rather shapes with volume, this significantly reduces the amount of global memory read requests and boosts the performance of the kernel.

5.3.4 Interface Adjustments

The final streaming actions, tracking the mass changes of fluid cells and reconstructing distribution functions streamed from empty cells (presented mathematically in Sections 3.3.1 and 3.3.2), introduces the most complexity to the streaming step. A different action needs to be performed

Pseudocode 6 The algorithm used to manage the transfer of mass between interface cells. We cache information about the current cell in registers to prevent multiple global memory reads of the same values while iterating over all neighbouring cells. Δm is kept in register space and only written to global memory once the effect of all cells has been calculated.

```
1:  x = (t_x, t_y, t_z)
2:  if x is an interface cell then
3:      load u, ε, n_F and n_G for x into registers
4:      Δm = 0
5:      for i = 1 → Q do
6:          if x + e_ī is empty then
7:              Reconstruct f_i(x)
8:          else if x + e_ī is fluid, interface or source then
9:              Adjust Δm by mass changes according to Equation 3.12
10:         end if
11:     end for
12:     m(x) += Δm
13: end if
```

depending on the type of neighbour cell and, according to Equation 3.12, certain other properties. For our implementation we incorporate all the discussed interface adjustments required during streaming, except the reconstruction of internal distribution functions as suggested by Thürey (Equation 3.14). This reconstruction relies on the values of neighbouring cells' distribution functions which, in a GPU parallel implementation, results in a race condition along the boundary between two thread blocks when using optimal memory access patterns (a neighbouring distribution cell could be reconstructed before the current cell has a chance to access its previous value for mass transfer calculations). This extra step also requires an extra seven global memory reads per cell, which would have a significant impact on performance. As can be seen in our results in Chapter 7, the omission of this adjustment does not have a noticeable impact on the visual quality of the simulation.

Pseudocode 6 outlines the algorithm used by our interface adjustment kernel. Since reconstructed distribution functions do not affect the mass transfer calculation, we are again able to schedule a single thread to manage the interface adjustments for a single lattice cell. As with previous kernels, this means the grid of thread blocks mirrors the structure of the simulation lattice.

The main difficultly with this kernel is that the interface adjustments require many read requests from global memory. Reconstructing distribution functions requires reading neighbouring distribution functions. The mass transfers require fill fractions, neighbour counts and distribution functions from both source and destination cells. Furthermore, little computation is required for each of these memory reads and, given that the fluid interface is typically a thin layer around the fluid, there is little opportunity for coalescing these global memory operations. In order to reduce the number of such operations, we make use of the fact that interface cells almost always have neighbouring gas and interface cells. This allows us to cache information about the current

Pseudocode 7 The algorithm used to perform the collide calculations on the GPU. We have used references instead of the actual equations in Chapter 3 to improve readability.

1: $\mathbf{x} = (t_x, t_y, t_z)$
2: **if** $\mathbf{x}$ is a fluid or interface cell **then**
3: Using only a single global memory read of each f_i^*, calculate:
4: $\rho, \mathbf{u}$ (Equation 3.5)
5: $\sum_i \left(f_i^* \sum_j (e_i)_j^2 \right)$ — from Π in Equation 3.19
6: $\mathbf{u} += \mathbf{g}$
7: Persist ρ and $\mathbf{u}$ to global memory
8: **if** $\mathbf{x}$ is interface cell **then**
9: Update ε
10: Use Equations 3.15 and 3.16 to mark cells for type conversion
11: **end if**
12: Calculate each f_i^e and store in registers (Equation 3.4)
13: Calculate Equation 3.19 using precomputed value from line 5
14: Calculate $\frac{1}{\tau}$ (Equation 3.18)
15: Calculate each $f_i(\mathbf{x}, t+1)$ and persist to global memory (Equation 3.6)
16: **end if**

cell before iterating through neighbour cells, knowing that such information will be used when calculating at least 1 distribution function and mass transfer. We also cache the change in mass, only writing the result at the end to maximize the opportunities for coalescing.

Lastly the calculation of Δm for Equation 3.12 is structured such that distribution function reads can be coalesced where possible, while ensuring that unnecessary global memory accesses do not occur. This is achieved by flagging which distribution functions are needed, based on Table 3.1, and only performing reads once all threads are no longer executing divergent branches.

5.4 Collide

The collide kernel performs the core computation of the LBM. It recomputes distribution functions to estimate the change in fluid flow as a result of fluid particles interacting with one-another. These calculations are all embarrassingly parallel — they only require information specific to a single lattice cell. This allows the kernel design to focus on maximizing computation and memory throughput without needing to keep track of global information.

Our approach for the collide kernel is outlined in Pseudocode 7. As with other embarrassingly parallel kernels, we structure the grid of thread blocks so that a single thread is responsible for a single lattice cell.

Since the collide kernel performs many calculations and memory accesses, there is opportunity for latency hiding at the warp level (discussed above in Section 4.1.2). In order to take advantage of this latency hiding it is important to place computation between successive global memory accesses. Therefore, we specifically chose to persist $\mathbf{u}$ and ρ as early as possible in our algorithm instead of at the end. This results in a slight decrease in runtime for the collide kernel.

One of the limitations of the collide kernel, is that there are many local variables used in calculating the new distribution functions, but there are only limited registers available per thread block. This register pressure prevents thread blocks with higher numbers of threads, because, if there are too few registers per thread, some local variables get stored in slow local memory. The high number of used registers also limits concurrency as fewer thread blocks are able to execute in parallel. Where possible, local variables are reused (at the cost of code readability) to help reduce register pressure. The equilibrium distribution functions (f_i^e) are the main cause of register pressure since there are 19 values that need to be stored for use in calculating the relaxation parameter and new distribution functions. We experimented with storing each f_i^e in shared memory and with recomputing the values when needed, but the straightforward approach of using registers turns out to be faster.

We also optimized the kernel to cache repeated calculations and reduce global memory reads where possible. Before calculating each f_i^e, we cache the result for $1 - \frac{3}{2}\mathbf{u}^2$ and reuse it for each direction. Also, for each iteration of the f_i^e calculations, we calculate $\mathbf{u} \cdot \mathbf{e}_i$ once and reuse the result. To reduce the number of global memory requests, we calculate part of Equation 3.19 when the distribution functions are initially read to avoid having to fetch the values again later on.

5.5 Fluid Interface Movement

The final step in our implementation is manage the movement of the fluid interface. During this step five tasks are performed:

1. Interface cells that were marked as full during the collide step are converted to fluid cells;
2. Interface cells that were marked as empty during the collide step are converted to gas cells;
3. The fluid and gas neighbour counts for each cell are updated;
4. Excess mass from newly changed interface cells is distributed to neighbouring cells; and
5. The fill fraction for all interface cells is updated.

It is not possible implement these tasks as a single kernel, without synchronization between thread blocks, because of data dependencies. The list of new fluid cells must be processed before the list of empty cells, in case a new fluid cell prevents the conversion of a new empty cell. Since new fluid cells and new empty cells cause new interface cells to be created, those lists must be processed before we can adjust neighbour counts with only a single write to global memory. Lastly, the fill fraction can only be updated once excess mass has been distributed to all cells. Since global synchronization within a kernel is an expensive operation [15], we, therefore, implement four kernels: Convert Full Interface Cells (1), Convert Empty Interface Cells (2), Update Interface (3 and 4) and Finalise Interface (5).

Pseudocode 8 The algorithm used by to convert full interface cells into fluid cells and make resulting changes to surrounding cells. New fluid cells are flagged during the collide step.

```
1:  x = (t_x, t_y, t_z)
2:  if x is a new fluid cell then
3:      for i = 0 → Q do
4:          if x + e_i is an empty cell then
5:              Cache conversion of x + e_i to interface cell
6:              for j = 0 → Q do
7:                  f_j(x + e_j) = f_j^e(ρ̄(x + e_j), ū(x + e_j))
8:              end for
9:          end if
10:         if x + e_i will be converted to an empty cell then
11:             Cache conversion of x + e_i to interface cell
12:         end if
13:         Persist conversion of x + e_i, if required
14:     end for
15: end if
```

5.5.1 Convert Full Interface Cells

The role of this kernel is to maintain a layer of interface cells around the fluid when interface cells are converted into fluid cells. There are two adjustments that need to be made: neighbouring empty cells need to be converted to interface cells, and any existing neighbouring interface cells should be prevented from being converted to empty cells. The algorithm we use for this kernel is outlined in Pseudocode 8. As with previous kernels, the cell type changes (empty to interface and newly empty to interface) are persisted in a branch common to all threads, to increase the chance of coalesced writes.

The conversion of empty cells to interface cells performed by this kernel requires extremely costly global memory operations. The velocity and pressure of all neighbouring cells has to be accessed to calculate $\overline{\rho}$ and $\overline{\mathbf{u}}$. Furthermore we have to initialize m, ρ, $\mathbf{u}$, ε and all 19 f_i for the new interface cell. Unfortunately, these operations are only required for selected interface cells, which means that little coalescing is likely. The only mitigation we have is that each thread is responsible for a single cell. While the GPU is waiting for memory requests from threads following the complex execution path (lines 4–9), it can schedule the calculations to determine the location of type information for other cells. However, the benefits of latency hiding in this case are small, because the calculations are simple and those threads not following the complex execution path also require a read to global memory to determine the cell type.

5.5.2 Convert Empty Interface Cells

This is the simplest of the fluid interface movement kernels. Since conflicting changes are already resolved, this kernel is only responsible for converting the neighbouring fluid cells of a newly empty cell to interface cells (Pseudocode 9). The conversion is as simple as updating the mass of

Pseudocode 9 The algorithm used to convert empty interface cells into empty cells and make resulting changes to surrounding cells. New empty cells are flagged during the collide step.

```
1: x = (t_x, t_y, t_z)
2: if x is a new empty cell then
3:     for i = 1 → Q do
4:         if x + e_i is a fluid cell then
5:             Set flag for x + e_i to interface cell
6:             m(x + e_i) = ρ(x + e_i)
7:         end if
8:     end for
9: end if
```

the fluid cells to be equal to their pressure.

Since there is no computation, only global memory operations, it would have been more efficient to combine this kernel with one of the other interface movement kernels. This would avoid the extra pass over the lattice to read cell types, and allow for the possibility of coalescing reads and writes common to multiple branches. Unfortunately, conflicts need to be fully resolved by the previous kernel before this kernel is run and the subsequent kernel depends on the neighbour adjustments of this kernel. Therefore, without an efficient mechanism for global thread synchronization, the conversion of empty interface cells has to be executed in a separate kernel.

5.5.3 Update Interface

Updating the fluid interface is the most complex of all the fluid movement kernels. This kernel is responsible for updating the neighbour counts of neighbours when a cell has changed type, and distributing excess mass from new fluid and empty cells to neighbouring cells. The algorithm used (Pseudocode 10) differs from the other kernels, which use a single thread to make adjustments to a single cell, based on information local to that cell and its neighbours. Instead, each thread uses information local to a cell and its neighbours to make adjustments to neighbouring cells. This significant difference results in a race condition where two threads could attempt to modify the same value in global memory, potentially resulting in one of the updates being lost.

In order to overcome the potential race condition, we use CUDA's atomic operations. The `atomicAdd` operation allows a thread to read a value from global memory, add to that value, and write the result back to global memory as a single transaction (lines 19 and 21). This prevents other threads from modifying the value in global memory while the addition is being performed. The atomic operations are only able to solve our race condition because the adjustments made to neighbour cells do no affect the information used to update other cells' neighbours. We specifically designed this behaviour into the interface movement kernels by separating tasks that required global synchronisation into separate kernel, so that more complicated race conditions would not arise.

Like the kernel that converts full interface cells to fluid cells this kernel also performs nu-

Pseudocode 10 The algorithm for managing the fluid interface and mass distribution around new fluid and empty cells. The changes in the neighbour counts are cached and persisted later to increase the chance for coalesced writes to global memory by threads from the same warp. The `atomicAdd` operation is required to prevent race conditions from affecting the simulation at the thread block boundaries.

1: $\mathbf{x} = \left(t_x, t_y, t_z\right)$
2: **if** $\mathbf{x}$ is a new interface cell **then**
3: **for** $i = 1 \rightarrow Q$ **do**
4: Adjust neighbour count for $\mathbf{x} + \mathbf{e}_i$ using `atomicAdd`
5: **end for**
6: **end if**
7: **if** $\mathbf{x}$ is a new empty cell or new fluid cell **then**
8: $t = m\left(\mathbf{x}\right)$
9: **if** $\mathbf{x}$ is a new fluid cell **then**
10: $t = t - \rho\left(\mathbf{x}\right)$
11: Cache neighbour count change
12: **end if**
13: **if** $\mathbf{x}$ is a new empty cell **then**
14: $m\left(\mathbf{x}\right) = 0$
15: Cache neighbour count change
16: **end if**
17: Calculate φ_i for $i = 1 \rightarrow Q$ and φ_t
18: **for** $i = 1 \rightarrow Q$ **do**
19: Persist cached neighbour count change for $\mathbf{x} + \mathbf{e}_i$ using `atomicAdd`
20: **if** $\mathbf{x} + \mathbf{e}_i$ is an interface or new interface cell **then**
21: `atomicAdd` $\left(m(\mathbf{x} + \mathbf{e}_i), t\frac{\varphi_i}{\varphi_t}\right)$
22: **end if**
23: **end for**
24: **end if**

merous global memory requests. In particular, calculating φ_i and φ_t (Equation 3.17) requires calculating the fluid interface normal ($\mathbf{n}$) at each neighbouring cell and, therefore, six reads of the fill fraction in global memory per neighbour. These memory requests form the bulk of the work done by this kernel.

5.5.4 Finalise Interface

The final fluid interface movement kernel is responsible for updating the fill fraction for interface cells and finalizing the cell types (Pseudocode 11). These actions are performed separately because they would introduce race conditions if they were included in the *Update Fluid Interface* kernel (Pseudocode 10).

Apart from a divide operation to calculate the new fill fraction for interface cells, there is no computation performed by this kernel. The main work done is the persisting of information to global memory. As with the previous kernels, we schedule a thread per lattice cell and we cache the values determined in different code branches, so a single coalesced memory write can be

Pseudocode 11 The algorithm for finalising the movement of the fluid interface. The changes to global memory are precomputed and cached so that they can be persisted to global memory by all threads in a warp simultaneously.

```
1: x = (t_x, t_y, t_z)
2: Calculate and cache change to fill fraction at x
3: Cache type change for x
4:     new empty → empty
5:     new fluid → fluid
6:     new interface → interface
7: Persist change to fill fraction if necessary
8: Persist type change if necessary
```

performed when all adjustment threads are following the same code branch.

— ∼ —

In this chapter we presented our design for a GPU-based free-surface LBM fluid simulation using CUDA. Since our design involved the implementation of multiple kernels, before investigating the overall performance of our fluid simulation in Chapter 7, we first examine the performance of the individual kernels in the following chapter.

Chapter 6

GPU LBM Kernel Analysis

The performance of an algorithm on any HPC platform is, arguably, the most important aspect of its implementation. The potential reduction of execution time is often the driving factor behind any decision to move from an implementation on a conventional single-core platform to an HPC one. This is no different for our implementation of the LBM on graphics hardware. Before we dive into any overall performance results, it is necessary to examine the individual parts of our algorithm to ensure that they making optimal use of the resources available to them. The purpose of this chapter is to perform this examination.

Our investigation will focus on the performance of the kernels that implement the core LBM operations: the streaming kernels and the collide kernel. Each of these kernels is first analysed in isolation to show how close these kernels can operate to the graphics hardware performance limits. For the streaming kernels, we compare memory throughput and execution time with that of a specially designed benchmark kernel. For the collide kernel, we discuss the effect of latency hiding for performance improvements and show that memory bandwidth is a bottleneck. We then compare performance of these kernels against equivalent CPU code to demonstrate the possible benefits of running a LB simulation on graphics hardware. In addition to the streaming and collide kernels, three streaming adjustment kernels[1] and four fluid interface movement kernels[2] are required for a full free-surface simulation. We discuss why their close relationship to the shape of the fluid interface significantly reduces the value of analysing these kernels in isolation, and discuss the difficulties in measuring their performance in the context of a free-surface fluid simulation. Lastly, we bring together the results and discussion to examine the performance of all the kernels in the context of a fluid simulation to show how well the kernels make use of the available resources during normal fluid simulations. This final analysis is important for showing how well the kernels perform at the tasks for which they were specifically designed, and to explain the breakdown of execution time across all kernels.

All tests in this chapter are performed on the hardware specified in Table 6.1. We used the

[1] Source adjustments , obstacle adjustments and interface adjustments.
[2] Converting full cells, converting empty cells, updating the interface and finalising the interface.

CPU	Intel Core i5-3570K
CPU Cores	4
CPU Memory	8 GiB DDR3
CPU Clock Speed	3.4 GHz
Graphics Card Model	NVIDIA GeForce GTX 560Ti
GPU Generation	GF114 (Fermi)
GPU Clock Rate	822 MHz
GPU Memory	1 GiB GDDR5
GPU Memory Clock Rate	4008 MHz
GPU Cores	384
CUDA Runtime Version	5.0

Table 6.1: The hardware used to perform this analysis.

hardware that was locally available to us to perform these tests. Since the kernel analysis is primarily concerned with efficient resource utilization rather than raw performance, this hardware was acceptible for these purposes.

We begin with the isolated analysis of the streaming and collide kernels.

6.1 Isolated Analysis

Although assessing the performance of kernels in the context of the task they are designed to perform is ideal, it is possible for the peak performance of the kernels to be obscured by the nature of the data being processed. For LB fluid simulations, the shape of the fluid can have a significant effect on the performance as various operations may or may not be performed. However, in an isolated environment without other kernels we can control the simulation data to ensure that it does not significantly impact our performance metrics. The streaming and collide kernels are best suited to this type of analysis because their calculations and branching is not highly dependent on the shape of the fluid interface. Since the code branches in the adjustment and interface movement kernels are highly dependent on the fluid surface, discussing them outside that context is not valuable and therefore they are not included in this section.

We begin by discussing our approach for choosing the lattice dimensions for each isolated analysis. We first investigate the performance of the streaming kernels and end this section with the analysis of the collide kernel.

6.1.1 Lattice Dimensions Limitations

The streaming and collide kernels both require the initialisation of the LB distribution function lattice before they are invoked. The collide kernel similarly requires initialization of data struc-

tures responsible for tracking mass, fill fractions, velocity and pressure, which also align with the distribution function lattice. We therefore need to select appropriate values for the x, y and z dimensions of the distribution function lattice before we can perform any analysis, even in isolation. Using the largest possible lattice will result in the greatest opportunities for parallelism, and therefore the best achievable performance. However, the amount of global memory available on the graphics card limits the size of the simulation lattice used for analysis. Since we have limited resources, we are careful to ensure that the dimensions of the lattice do not impact on simulation performance.

As a result of GPU global memory access patterns, the x-dimension is the most restricted of all the dimensions. Although memory access patterns would suggest limiting the x-dimension to multiples of warp size (32), measuring performance with this restriction showed that the best performance was achieved when the thread block width was a power of 2. Therefore, we restrict the x-dimension to a power of 2 so we can ensure the analysis will cover the cases where the best is achievable.

The use of arbitrary height and depth (y and z dimensions, respectively) for the lattice has little effect on performance, except when the height or depth of the scene is not divisible by the height or depth of the thread block, respectively. Experimenting with launch configurations by varying only the block height/depth showed that a thread block heights/depths of 1, 2, 3 or 4 are viable values, with performance noticeably decreasing for higher values. Therefore we restrict the lattice height and depth to be a multiple of 12 (the lowest common multiple of 1, 2, 3 and 4) to ensure that the height and depth of the scene will always be divisible by the thread block height and depth, respectively.

We also have to choose the sizes of each dimension relative to each other. We attempt to keep the dimensions relatively close in size to each other. This is helpful for being able to compare the performance of different streaming kernels that move data along difference axes, because each streaming kernel will have to move a similar number of cells along its streaming direction. Furthermore, similarly sized dimensions maximize the volume of the scene which reduces the number of operations happening on scene boundaries. Although these boundary effects can be ignored in this isolated analysis, in the context of a fluid simulation they play a significant role in performance.

Lastly, as an example, we look at three different lattice configurations: $239 \times 239 \times 239$, $256 \times 228 \times 228$ and $256 \times 1680 \times 32$. Despite a $239 \times 239 \times 239$ or $256 \times 1680 \times 32$ lattice using more memory than a $256 \times 228 \times 228$ lattice (989.5 MiB or 997.5 MiB versus 964.5 MiB), we will choose the latter for the analysis because it best matches our self imposed restrictions. A width of 256 is a multiple of warp size and a power of 2, which gives us more flexibility when varying the kernel launch parameters, without affecting optimal coalescing configurations. A width and height of 228 (a multiple of 12) ensures that no thread blocks will have unused threads. Choosing

height=228 and depth=228 instead of height=1680 and depth=32 also helps maximize the volume of the scene and means a single thread will stream a similar amount of information along the x, y, and z dimensions.

6.1.2 Streaming Kernels

Five kernels are responsible for moving distribution functions during the streaming step: Three kernels perform transfers along the x, y and z axes and two perform transfers along the diagonals of the xy and xz planes. Transfers along the diagonals of the yz plane are performed by streaming in the y and z directions separately and consecutively because performing these transfers in a single step would require inter-block synchronization. Refer to Section 5.3.2 for more information on the design of these kernels.

Before presenting the results of the analysis we will outline our approach.

Analysis Approach

The focus of the streaming kernels is to transfer distribution functions as quickly and efficiently as possible. Therefore, investigating computation efficiency is unnecessary. In this section we will outline how we measure the kernel performance and how we initialize the simulation lattice.

Since the streaming kernels are focused on data transfer, memory throughput stands out as a measure of kernel performance. However, direct measurements of memory throughput via the CUDA profiler are not an exact representation of algorithm performance, but are only useful for determining how close the code is to the hardware limit [15]. Instead, the CUDA Best Practices Guide [15] suggests calculating *effective bandwidth* as this provides a figure more relevant for measuring performance. It can be calculated as follows:

$$\text{Effective Bandwidth} = \frac{\text{Bytes Read} + \text{Bytes Written}}{\text{Execution Time}} \tag{6.1}$$

Since bytes read and bytes written are identical for all streaming kernels (they all perform a single read/write pair per cell per distribution function being streamed), *execution time* is the only value that will affect our measure of effective bandwidth. Therefore execution time alone is sufficient to represent the performance on the streaming kernels, while memory throughput is used to show that performance is near the practical hardware limits.

To independently verify that we have achieved the best possible performance, we use an optimal *copy-kernel* (Pseudocode 12) as a benchmark when testing these kernels. The copy-kernel has the same data structure as the streaming kernels, but does not alter array positions during data transfer. This ensures the copy-kernel has less computation and maximizes opportunities for coalescing data reads and writes. The copy kernel is also designed to be divided into thread blocks along any axis to allow for better latency hiding. At optimal launch configurations the

Pseudocode 12 The *copy-kernel* used to benchmark the streaming kernels.

```
1:  function COPY-KERNEL(A)
2:      r = ⌈depth / gridDim.z⌉
3:      z_s = blockIdx.z × r
4:      i = t_x + t_y × pitch + z_s × pitch × height
5:      k = pitch × height
6:      z_e = z_s + r
7:      for z_s → z_e do
8:          t = A[i]
9:          __syncthreads()
10:         A[i] = t + 1
11:         i += k
12:     end for
13: end function
```

inclusion of `__synchthreads()` can cause a negligible performance decrease, but its inclusion has a noticeable performance benefit when the launch configurations are restricted to those that are applicable to the streaming kernels. The addition in line 10 is required to prevent compiler optimizations discarding the read/write pair. Due to its optimizations and simplicity, we can therefore consider the performance of the copy-kernel to be the best possible performance that can be achieved by the streaming kernels.

For simplicity, we use only two performance results from the copy-kernel: a *benchmark* value that is the best achievable performance for all kernel launch configurations for all tests, and a second *adjusted benchmark* which represents the best achievable copy-kernel performance with kernel launch configurations that are applicable to the streaming kernel(s) being discussed (the exact restrictions applicable to each kernel are discussed in Section 5.3.2.) For kernels that stream in multiple directions (which require multiple read/write pairs per lattice cell), the benchmarks are multiplied by the number of directions where appropriate.

Unless otherwise specified an array size of $256 \times 228 \times 228$ was used in these tests. This is the maximum array size for a single distribution function array, f_i, such that all 19 distribution functions can fit in GPU memory assuming the restrictions discussed in Section 6.1.1 and the hardware from Table 6.1. The distribution function array was filled with arbitrary floating point values between zero and one; the streaming kernels only transfer data and do not perform calculations based on data values, thus the use of arbitrary data does not affect the validity of the results.

All results were calculated as an average over 100 consecutive kernel invocations. Numerous kernel invocations are grouped together to eliminate minor variances in execution time due to the inaccuracy of timers at the microsecond level. These iterations are all timed after a few "warm-up" iterations so that the time taken for the device to transition from idle to active does not affect results.

	BlockDim	GridDim	T	M	T_C	M_C
copy-kernel	(64,3,1)	(4,76,57)	98.28	108.44	-	-
x-axis streaming	(256,3,1)	(1,76,1)	201.77	105.55	1.42%	98.51%
y-axis streaming	(128,3,1)	(2,76,1)	614.28	103.99	2.93%	97.05%
z-axis streaming	(256,3,1)	(1,76,1)	607.38	105.17	1.77%	98.15%
xy-axis diagonal streaming	(256,3,1)	(1,76,1)	416.48	102.31	4.68%	95.48%
xz-axis diagonal streaming	(256,3,1)	(1,76,1)	414.69	102.75	4.23%	95.90%

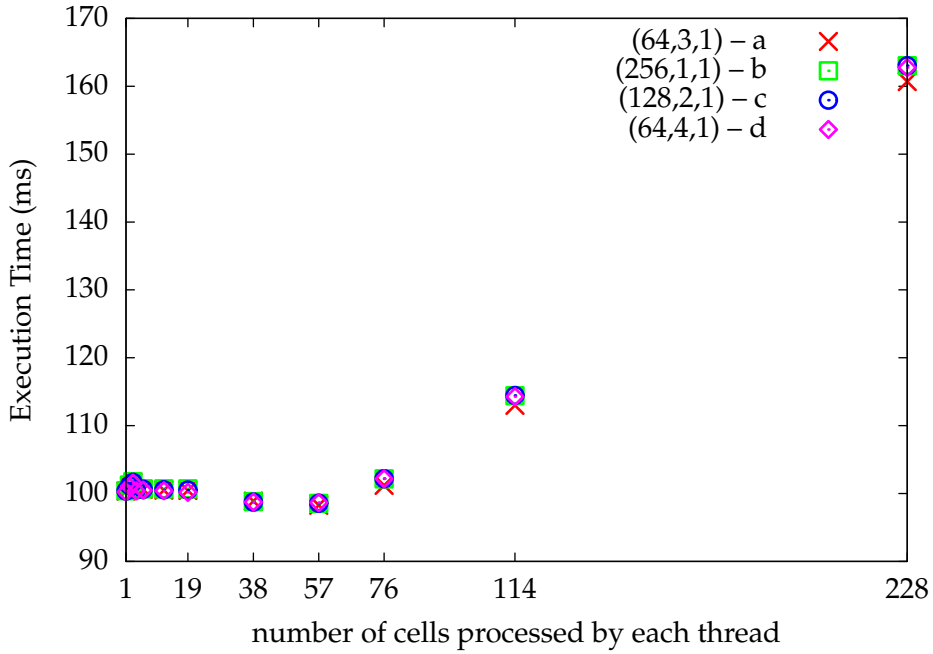
Table 6.2: The optimal launch configurations and performance for all the streaming kernels, showing the execution time (T) and memory throughput (M). We also show the performance of the streaming kernels relative to the performance of the copy kernel when applying each kernel's launch configuration restrictions to the copy kernel. T_C is the percentage that each kernel is slower than the copy kernel and M_C is the percentage of the copy kernel throughput achieved.

Analysis Discussion

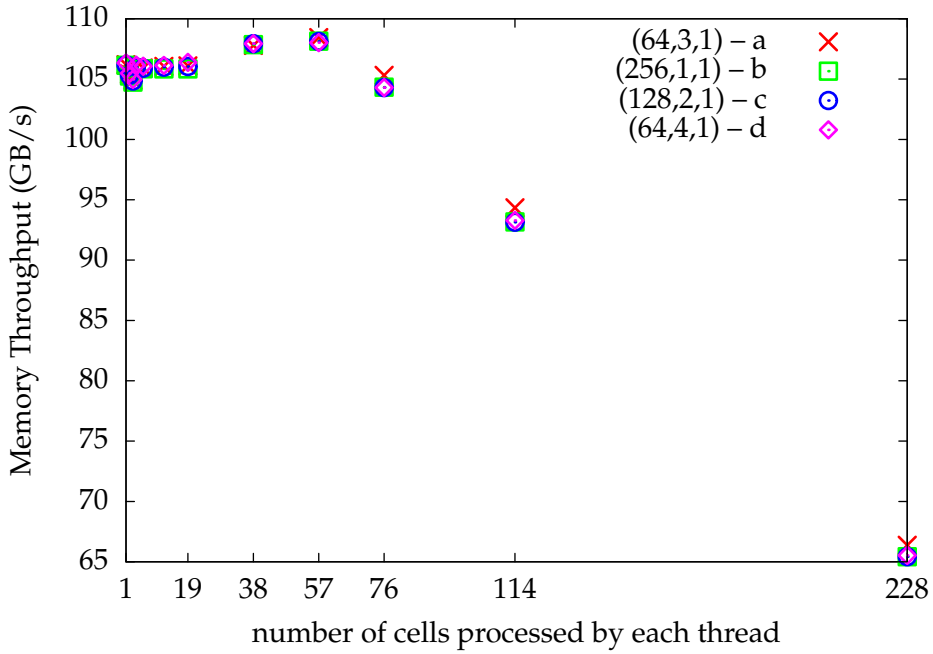
We begin the analysis by measuring performance of the copy-kernel, a simple kernel designed to provide us with streaming performance benchmarks. We then discuss the performance streaming kernels that transfer along the core x, y and z axes, using the benchmarks derived from the analysis of the copy-kernel. The performance of the kernels that stream along the diagonals is discussed, drawing on similarities in performance with the non-diagonal kernels. Lastly, we compare the performance of the GPU to a single CPU core when streaming in all directions.

The copy kernel launch configuration has three variable parameters: thread block width, thread block height, and the number of cells a single thread will process (i.e. the number of thread blocks to allocate along the z-axis). To limit the large search space, we show the top 4 launch configurations (see Figures 6.1a and 6.1b). Almost identical performance is shown by configurations b–d because they have the same number of active threads (256). The optimal benchmark performance for the copy-kernel is measured at 98.3 ms for execution time and 108.4 GB/s for memory throughput. This equates to 84.7% of the theoretical peak performance of our GTX 560Ti (128 GB/s [15]). Bauer [6] reports the maximum practical memory throughput achievable is 85% of a device's theoretical peak performance, which allows us to confirm that our copy-kernel is a realistic benchmark for optimal performance. The memory-write throughput and memory-read throughput were consistently approximately half the total memory throughput — write throughput was consistently 0.4–0.45 GB/s faster than read throughput. Since the results from these individual values are identical to the results of the total memory throughput, we do not report the individual read/write results.

Figures 6.2a and 6.2b compare the best possible performance achieved by each of the streaming kernels (their exact values are listed in Table 6.2). We see that all kernels make efficient use of available memory bandwidth, almost matching their adjusted benchmarks. There is less than a millisecond between the optimal x- and z-axis streaming, with y-axis streaming being a further



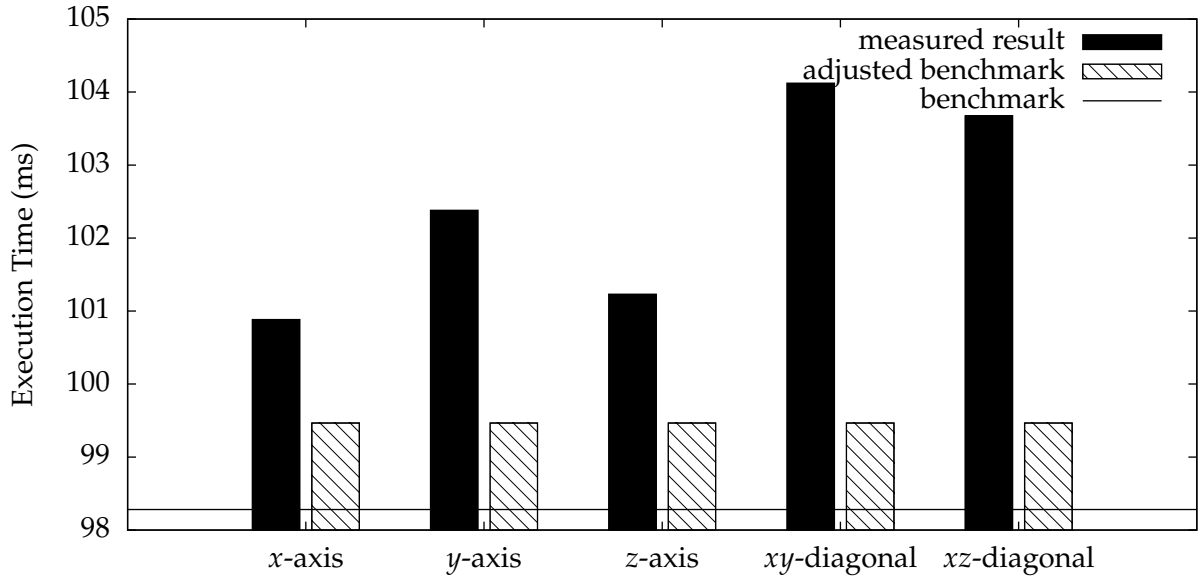
(a) Execution time for selected launch configurations



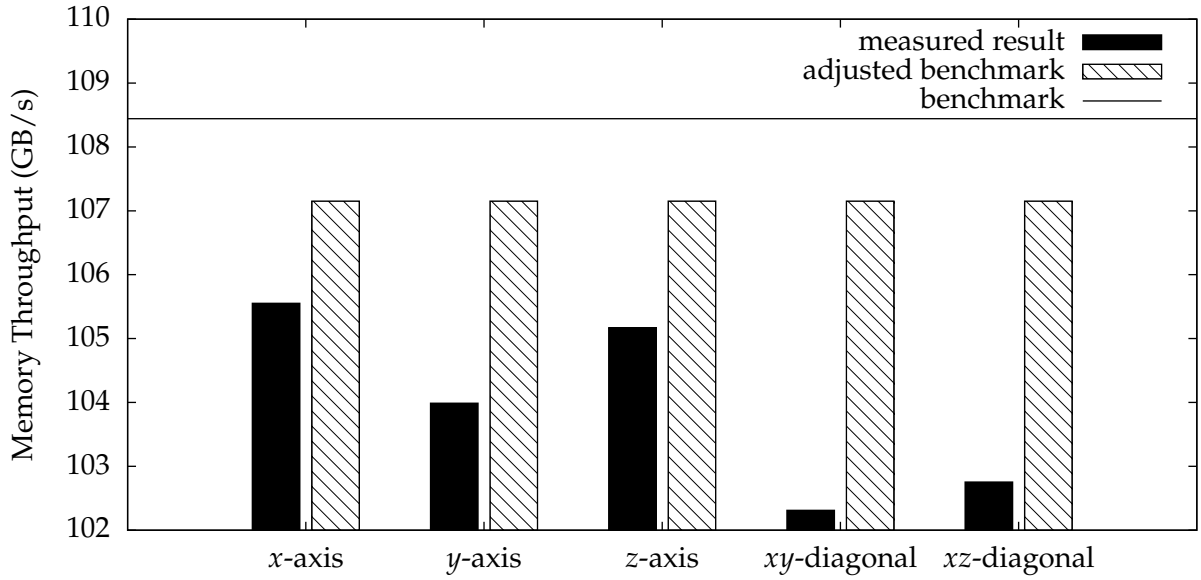
(b) Memory throughput for selected launch configurations

Figure 6.1: The performance of the Copy-Kernel as the number of cells each thread must process (n) is varied for the thread blocks of the top four launch configurations. Only results for the optimal values of n are shown — when $\frac{\text{height}}{n} \in \mathbb{N}$ so that there are no idle threads.

millisecond slower. This is because the orientation of x- and z-axis streaming results in multiple thread blocks accessing contiguous blocks of memory simultaneously (xy-plane slices), whereas different thread blocks are accessing different xy-plane slices simultaneously when streaming along the y-axis. The performance of the diagonal streaming is noticeably worse. Since the di-



(a) Streaming execution time compared to the benchmarks



(b) Streaming memory throughput compared to the benchmarks

Figure 6.2: The performance of the streaming kernels with optimal launch configurations. The benchmark refers to the optimal copy-kernel performance and the adjusted benchmark is the optimal copy-kernel performance given the same launch configuration restrictions as each streaming kernel. Each kernel's execution time has been divided by the number of directions in which distribution functions are streamed so that the performance between different streaming kernels can be compared.

agonal streaming kernels stream along two axes, they are the most restricted in terms of launch configurations and, as a result, are the slowest of all the streaming kernels. The xz-diagonal streams are slightly faster that the xy-diagonal streams, because the xz-diagonal streaming kernel's access pattern is better suited for coalescing reads on xy-slices. Despite the minor differences

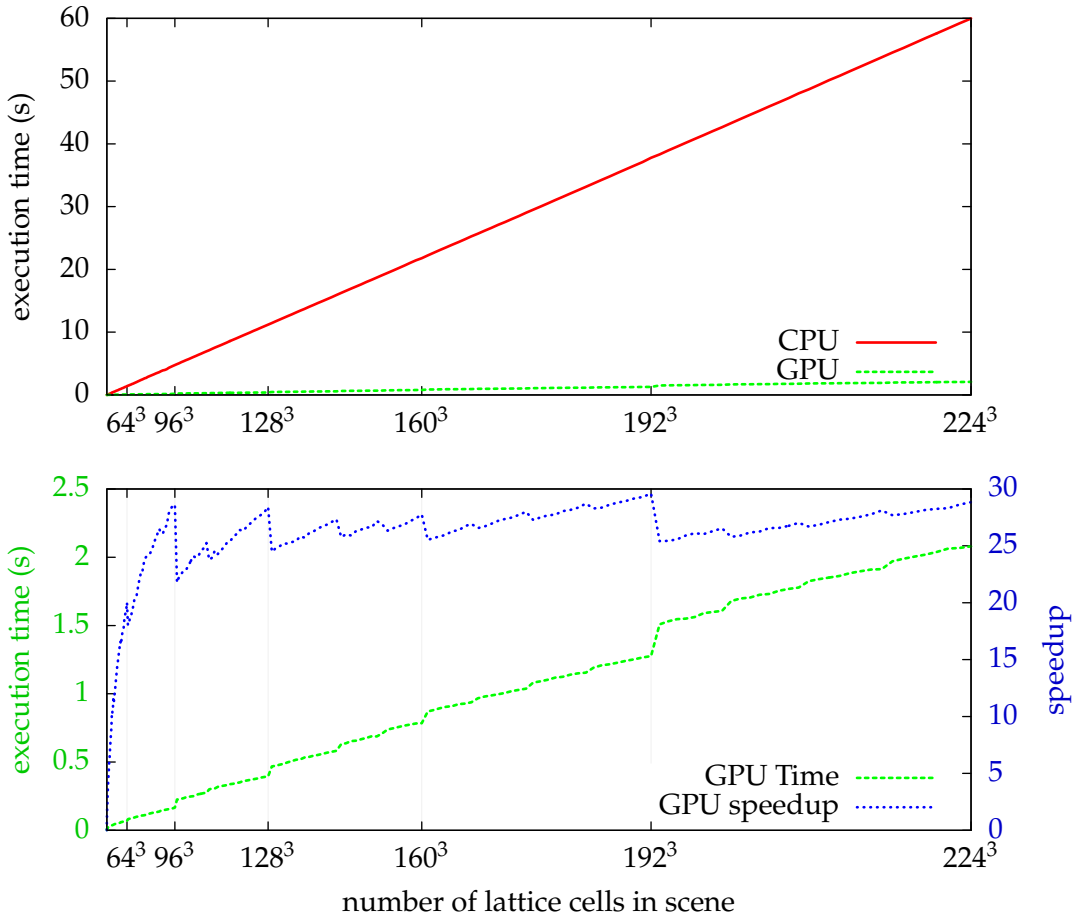


Figure 6.3: These figures show the total execution time and GPU speedup achieved for streaming along all directions for increasing lattice size. Small jumps in the GPU execution time (or spikes in GPU speedup) are seen in the transition from scene sizes that are suited to the GPU and those that are not. For example the big jump in execution time occurs when moving from a lattice size of 192^3 to 193^3 as 192 is divisible by warp size and is therefore better suited for GPU memory operations.

in performance all kernels still perform close to the practical hardware limit.

To test the scalability of the streaming kernels, we combined all the kernels to stream as they would as part of a normal LBM simulation. In our full LBM implementation, the distribution functions are not transferred off the graphics card. Therefore, the total GPU time in these scaling tests includes kernel execution time and only initial data transfer (i.e. the distribution functions are not transferred back to the CPU after each iteration on the GPU). This provides us with a real indication of the performance of the streaming kernels as part of our GPU LBM simulation. The combined streaming kernels were compared against a single-core CPU implementation of pure LBM streaming for increasing lattice sizes (Figure 6.3). In this test we used 3D array sizes with equal dimensions (i.e. $1^3, 2^3, 3^3, \cdots, 224^3$). The maximum array size was limited to 224^3 as this was the largest lattice size that could fit into the graphics card memory and had dimensions that are multiples of warp size. For each array size, the optimal launch configuration for each

streaming kernel was selected automatically and used to determine optimal performance. While the CPU execution time scales linearly with respect to problem size, the GPU version has clear sudden increases in execution time. These steps occur exactly at the boundaries between scene sizes that are suited to GPU memory operations and those that are not. For example, with an array width of 128, four warps are required and all threads within each warp are in use, but with an array width of 129, five warps are required and only 81% of all threads will be used — one warp will have 31 idle threads. These idle threads still require processor time and therefore slow the execution of other warps. As the array width increases, there are fewer idle threads and performance improves until the next warp boundary is reached. This behaviour also results in the spikes seen in the speedup graph. For large array sizes, we see the combined streaming kernels are able to achieve a 25–30$\times$ speed up over the CPU version, peaking near 30$\times$ when the lattice size is optimal for GPU memory operations (i.e. x-dimension is a multiple of warp size).

6.1.3 Collide Kernel

A single kernel is responsible for performing the collision calculations and updating the simulation data for the LBM collide step. The collide kernel is unique in that it performs numerous calculations per memory access and only requires access to data that is local to a single lattice cell.

Before diving into the details of the collide kernel's performance we will first discuss the metrics we focus on and the reasons for choosing them.

Approach

The aim of this analysis is to show that:

1. the memory operations make full use of the available bandwidth;
2. the calculations require less time than the memory operations; and
3. latency hiding is used effectively to ensure calculations happen while waiting for memory operations to complete.

Although Equations 3.3 and 3.19 require a significant number of mathematical operations, the collide kernel remains memory bound: for each cell, at least 43 global memory operations are performed, each requiring multiple clock cycles to complete.

We split the collide kernel into memory-only and calculation-only kernels so that we can measure the computational performance and memory throughput separately. For the memory-only kernel, it is only important to ensure that the memory access patterns are identical to the full kernel. This is achieved by replacing calculations that are not related to memory access with register reads and writes. We do not further separate the memory-only kernel into read-only and

write-only versions as these versions easily achieve high performance because they only perform reads or writes (not both). Replacing global memory operations with register operations is not sufficient for the calculation-only kernel. Calculations with results that do not affect any writes to global memory are automatically removed by the compiler as an optimization (the compiler identifies the code as redundant). Therefore a single memory write of a value that depends on the result of all the computation is required. It is also possible to hide this write behind a conditional that will always be false[3], thus ensuring that the write does not affect the performance of the calculation-only kernel.

Another important difference between the collide kernel and the streaming kernels is that the collide kernel only operates on fluid and interface cells. Therefore, under normal simulation conditions, a significant number of cells may not require any computation since empty cells will occupy significant portions of the scene. In such scenes, it is difficult to reason about kernel performance, as the position of the fluid in the scene affects the efficiency of global memory operations. In order to simplify the performance analysis, we use a scene containing only fluid cells — the equivalent of a simulation that models the internal behaviour of a fluid.

The collide kernel has more data dependencies than the streaming kernels. Therefore, we use a smaller grid size of $256 \times 192 \times 180$, unless otherwise specified. These are the maximum dimensions under our analysis restrictions, such that all the dependent arrays can be stored in GPU global memory. As a result of our design for the collide kernel, it is possible to divide the lattice intro arbitrarily sized thread blocks. Therefore, in searching for the optimal launch configuration, it is only the x-dimension of the thread block that has limited values. Since computation and memory operations are scheduled as warps, thread blocks with an x-dimension that is a multiple of warp size are the only launch configurations that will produce optimal GPU performance. We are also able to populate the simulation data structures with random data without affecting the performance of the kernel, because the calculations are not affected by the values of the data and each cell does not rely on any data from neighbouring cells.

Unless otherwise specified, all results were calculated as an average over 500 consecutive kernel invocations. Numerous kernel invocations are grouped together to eliminate minor variances in execution time due to the inaccuracy of timers at the microsecond level. These iterations are all timed after a few "warm-up" iterations to eliminate the time-cost associated with the device transitioning from idle to active.

Analysis Discussion

In order to understand the performance of the kernel as a whole, we need to look at its memory throughput and computational performance separately. Doing this allows us to reason about the interactions between calculations and memory operations and whether the kernel is making

[3]The conditional should depend on a value determined at runtime so that the compiler is not able to identify the branch as redundant.

	BlockDim	GridDim	*ExecutionTime*	*MemoryThroughput*
memory-only kernel	(32,3,1)	(8,64,192)	1853 ms	104.14 GB/s
calculation-only kernel	(64,3,1)	(4,64,192)	824 ms	– GB/s
full collide kernel	(128,2,1)	(2,96,192)	2086 ms	103.0 GB/s

Table 6.3: The optimal launch configurations and performance for the variations of the collide kernel, showing the execution time and memory throughput where appropriate. The execution time of the full kernel is less than the sum of the execution time of its parts as a result of latency hiding.

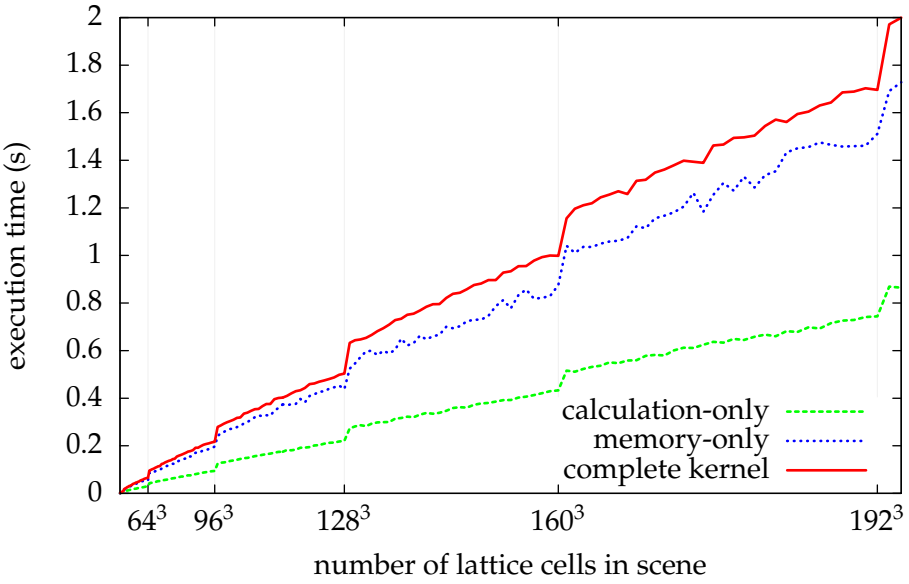


Figure 6.4: This figure compares the execution time of the full collide kernel with the kernels that perform only the memory operations (*memory-only*) and only the calculations (*calculation-only*). In this comparison we perform 100 iterations of each kernel. We see that the full collide kernel takes less time to run than the sum of the execution times of its parts because of latency hiding — computation occurs while waiting for memory operations to complete. Since GPU execution is best suited to scene dimensions that are a power of two, we see steps up in execution time as we increase from the most efficient scene size to a less efficient scene size.

efficient use of computational and memory resources. Therefore, our analysis begins by profiling memory throughput and computation separately before discussing how the computation and memory transfers perform together. This section ends with a comparison of GPU performance to CPU performance.

The memory-only kernel achieves a memory throughput of 81.4% (104.14 GB/s) of the theoretical maximum memory throughput. While there is still room for some improvement (given a practical performance limit of 85% [6]), it is fast enough to ensure that there are no significant gains in memory throughput to be achieved by further modifications to the kernel.

We also consider the computation performance of the collide kernel. Each fluid cell requires the following operations: 575 multiplication and addition operations (some of which the compiler reduces to multiply-add operations), five division operations, one reciprocal square root and one square root. For GPU's with compute capability < 2.0, the performance figures outlined

in the CUDA Programming Guide [16] suggest a theoretical peak instruction throughput just below 1.5 instructions per clock cycle per multiprocessor. While our code achieves an instruction throughput of 1.49, measuring active warps per clock-cycle is more relevant for graphics cards with compute capability $\geq$ 2.0. The profiler shows that our calculation-only kernel has 3 active warps per clock cycle when using optimal launch configurations. However, it turns out that the raw computational efficiency of the collide kernel is not the most important aspect. The fact that the computation requires less time that the memory operations means there is a perfect environment for latency hiding. As shown in Figure 6.4, the collide kernel exhibits excellent latency hiding characteristics. Taking an average of the values in Figure 6.4, we see that the full kernel runs for only 75.5% of the sum of the execution times for the computation- and memory-only kernels. This means that 70.7% of the collide calculations happen while waiting for memory operations to complete. Therefore, any possible inefficiencies of the collide calculations are not significant, because the kernel in its current form is able to complete most calculations while waiting for memory operations to complete.

The profiling of the full kernel shows a slight drop in performance for memory throughput. This is expected, as this kernel also performs the collide calculations. Furthermore, the memory-only and calculation-only kernels have different optimal launch configurations and, for the full kernel, a compromise between the two is required for optimal performance. It is unnecessary to measure the performance in terms of computation, because the kernel is clearly limited by the time taken to complete memory operations.

Based on the discussion above, we can conclude that, for the collide kernel, the memory layout is optimal and the calculations are sufficiently efficient to allow for excellent latency hiding. As part of the design (see Section 5.4) we also minimized the number of memory accesses and mathematical operations, while avoiding frequent use of expensive operators. We can therefore conclude that we have a near optimal GPU implementation for the collide step of the LBM.

Figure 6.5 shows the performance of the GPU collide kernel compared to the equivalent CPU code. It is worth noting the spikes in relative GPU performance for scene dimensions that are powers of 2 (e.g. $32 \times 32 \times 32$, $64 \times 64 \times 64$, etc.). Although scenes of these dimensions are best suited to GPU performance, the spikes are the also the result of an unexplained decrease in CPU performance. Brief experimentation with the CPU algorithm was unable to eliminate or explain the decrease in CPU performance for scenes of these sizes. Therefore, consider these spikes outliers and we leave the investigation to future work. There are two important factors that contribute to these speedup values of up to 223.7$\times$. The first factor is that the collide kernel does not require any transfers between device memory and host memory — costly operations that can have a significant effect on the efficiency of a GPU implementation. The second factor is that we only compare the GPU against a single core of a multi-core CPU. Since the collide step is embarrassingly parallel, it will be possible to parallelize the code on a multi-core CPU without

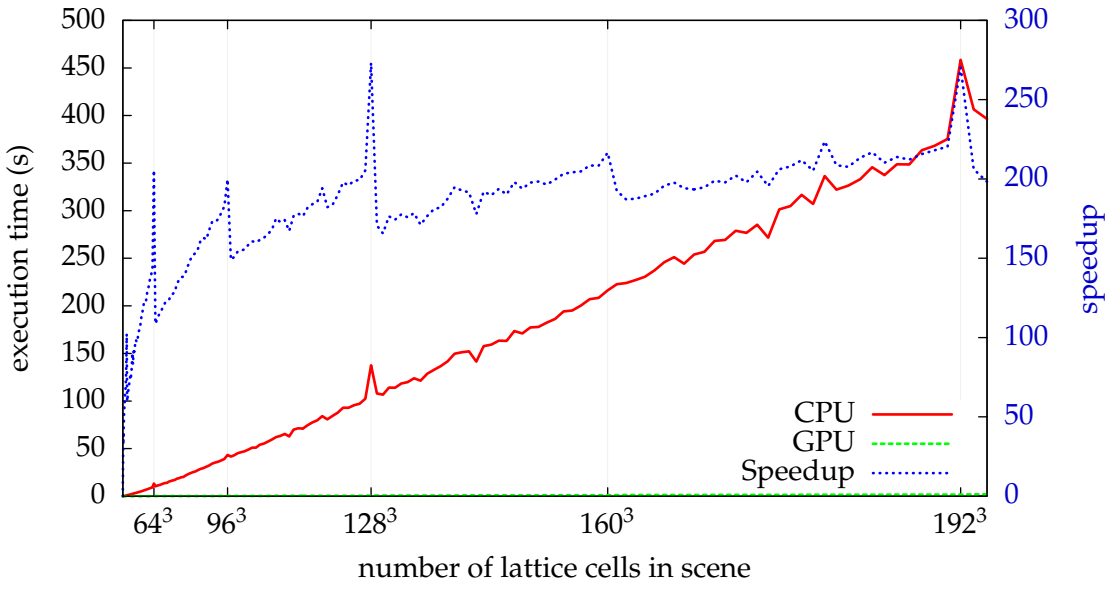


Figure 6.5: We see the speedup of our GPU collide kernel when compared to equivalent single-core CPU code. The spikes in speedup are the result of a combination of scene sizes that are optimal for GPU execution and unexplained sudden decreases in CPU performance for specific lattice sizes. The execution time was recorder after 100 collide iterations, and include the cost of initializing memory before the first run. For each scene size, the GPU kernel launch configuration that resulted in the fastest execution was used.

any significant inefficiencies. This means that on our quad core CPU, we would expect the CPU code to be almost 4 times faster when fully optimized and parallelized. Therefore, we are likely to have speedups that are a quarter of our current speedups (i.e. up to 55×) when comparing our GPU implementation against fully optimised CPU implementation.

6.2 Adjustment and Interface Movement Kernels

Three kernels are responsible for the streaming adjustments (source adjustments, obstacle adjustments and interface adjustments) and four kernels are responsible for interface movement (convert full interface cells, convert empty interface cells, update interface and finalise interface). There are two important differences between these kernels and the streaming and collide kernels: these kernels operate on a small percentage of the entire lattice, and these kernels are unavoidably divergent. With the exception of the source adjustment kernel, the cells on which these kernels operate is dependent on the shapes of both the fluid and obstacles in the scene. The fluid shape is dynamic and changes throughout the simulation, and the positions of obstacles are dependent of the layout of the scene. As a result, these cells will seldom align themselves with warps, which results in inefficient memory access and threads within the same warp following different code branches. Therefore, we expect these kernels to make inefficient use of GPU resources. In this section we will discuss the behaviour of these kernels with regards to their inefficiencies.

6.2.1 Source Adjustments

The source adjustment kernel is responsible for maintaining a fluid interface around all source cells, and adjusting the distribution functions of the source cells to create the effect of fluid flowing into the scene. This kernel only operates on source cells and their neighbours and does not need to be executed if source cells are not present in the simulation. Usually there will only be a few source cells ($< 1\%$ of all cells, if they are present) and these cells will be clustered together. While some coalescing is possible, if the source is parallel to the x-axis, this will not always be possible. The fact that fewer than 1% of all warps are required to perform source adjustments for larger scenes, is more significant. This means that the cost of reseting distribution functions and maintaining a fluid interface around the source cells will be dwarfed by the cost of reading the cell type of all cells in the scene — a cost that will be constant throughout the simulation. We thus expect consistent performance from the source adjustments throughout the simulation that is almost equivalent to performing a single global read for each cell in the simulation lattice.

6.2.2 Obstacle Adjustments

The obstacle adjustments operate on obstacle-fluid boundaries. In a typical large scene, boundary cells around the edges of the scene amount to 3–5% of all lattice cells. While the boundaries on the xy and xz planes align perfectly with warps, the boundaries of the yz planes are particularly inefficient with only a single thread per warp performing changes to the scene. We can calculate the percentage of warps performing obstacle adjustments on the scene boundaries for a lattice with dimensions (x, y, z) as follows:

$$\begin{aligned} & 1 - \frac{\text{number of non-obstacle warps}}{\text{total number of warps}} \\ = & 1 - \frac{(w-2)(y-2)(z-2)}{wyz} \end{aligned}$$

where $w = \left\lceil \frac{x}{32} \right\rceil$ is the number of warps along the x-dimension of the lattice. For large scenes this results in between 20% and 40% of all warps being tied up performing obstacle adjustments. This percentage is lower for larger scenes with fewer obstacles and higher for smaller scenes and more obstacles.

The operations required also result in varied performance. For each obstacle cell, an extra 18 global memory reads are required (one for each neighbour) to decide if there are any neighbouring fluid cells that require adjustments. Then for each of those 18 neighbours that is a fluid cell, an extra two global memory operations are required. This means that the performance of this kernel is dependent on the fluid-obstacle surface area at each time step. Since the shape of the fluid is dynamic, the performance of the obstacle adjustments will vary slightly between time steps.

6.2.3 Interface Adjustments and Interface Movement

The interface adjustment kernel is responsible for managing the flow of mass throughout the lattice as a result of the streaming distribution functions, and the interface movement kernels are responsible for controlling the movement of the fluid interface as a result of changes in mass at the fluid interface. This means the interface adjustment kernel and interface movement kernels operate only on the fluid interface cells and have similar inefficiencies to each other.

For larger, turbulent simulations (i.e. simulations with larger fluid surface areas) the fluid interface is usually represented by fewer than 3% of all lattice cells and these interface cells intersect with approximately 30% of all running warps. In scenes with little turbulence or scenes in which turbulence subsides, the fluid interface can be represented by fewer than 1.5% of all lattice cells, which intersect with fewer than 20% of all warps. These proportions vary throughout the simulation as the fluid interface changes shape, causing different levels of warp-efficiency as the simulation progresses. The intersection of interface cells with warps is important, because warps that do not intersect with interface cells do not need to perform any calculations or memory operations and complete instantly, whereas those that do intersect with interface cells have a number of inefficient memory accesses and calculations to perform.

The fact that the interface cells are unlikely to align themselves with warps for efficient memory accesses is particularly problematic for the interface adjustment and update interface kernels. On average, each warp for these kernels that intersects with the fluid interface has only three threads performing operations, which means, the memory bandwidth from an average of 29 other threads is wasted. Compounding the issue, these kernels need to perform numerous global memory accesses to obtain information from neighbouring cells. Therefore, these kernels are expected to perform poorly relative to the other kernels.

- ∼ -

With the limitations of the kernel discussed above in mind, we are now able to look at how the kernels perform together when producing a full fluid simulation.

6.3 Kernel Performance in a Fluid Simulation

In this section we bring together the results and discussions from previous sections to look at the performance of each kernel in the context of a full fluid simulation. We present the percentage of total execution time spent on each kernel and provide reasons why the percentages for each kernel are to be expected.

For this discussion we use a $192 \times 192 \times 192$ modified "dam break" simulation as our test case. The initial conditions are shown in Figure 6.6 and a selection of stills from the simulation are shown in Figure 6.7. Equal dimensions ensure that no streaming direction has better performance

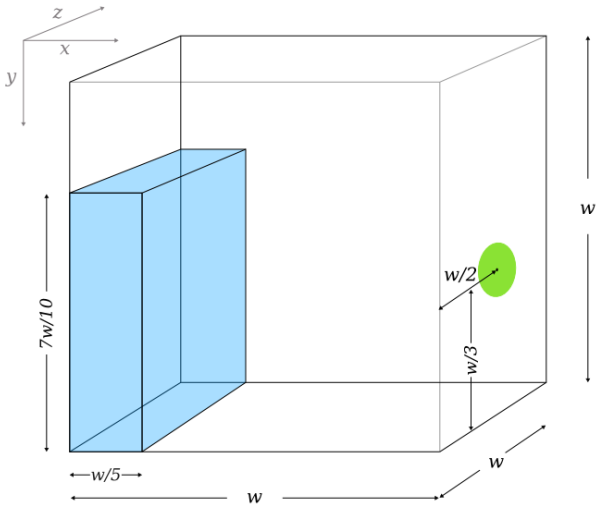


Figure 6.6: The scene used in this analysis. The green circle on the right represents the fluid source, which has a radius of $\sqrt{\frac{w}{3}}$.

due to the data dimensions. The scene is bounded by obstacle cells that contain the fluid. There is a wall of fluid against the left yz-face of the bounding box. The simulation models the fluid as it flows down the left yz-face and begins to slosh from one side of the scene to the other. In order to include the performance of the source kernel, we include a circular fluid source on the right yz-face.

Using a single scene for this discussion is sufficient because the difference in performance profiles between different simulation scenes is not significant enough to affect this analysis. We will therefore leave the discussion of the effect of different scenes to Chapter 7.

Figure 6.8 and Table 6.4 show the percentage of time spent executing each of the kernels throughout the simulation. The most counter-intuitive result is that the core LBM steps (streaming and collide), which perform the bulk of the work for the simulation, only require 47.3% of execution time, while a large portion of the time (38.6%) is spent executing kernels that manage the fluid interface (the interface adjustment kernel and the interface movement kernels). In the discussion below, we explain this result by showing that the streaming and collide kernels are able to make efficient use of the resources available to them, whereas the rest of the kernels do not make efficient use of the available memory bandwidth because of the limitations discussed in Section 6.2.

All of the LBM kernels are limited by memory bandwidth. This means that the efficiency with which individual warps access memory has a significant impact on the overall kernel performance. Warps that have only a few active threads will take almost the same amount of time as warps with all active threads to perform simultaneous aligned global memory operations. Warps with only a few active threads are thus wasting the resources associated with each idle thread. For the adjustment and interface movement kernels this warp inefficiency is unavoidable, but

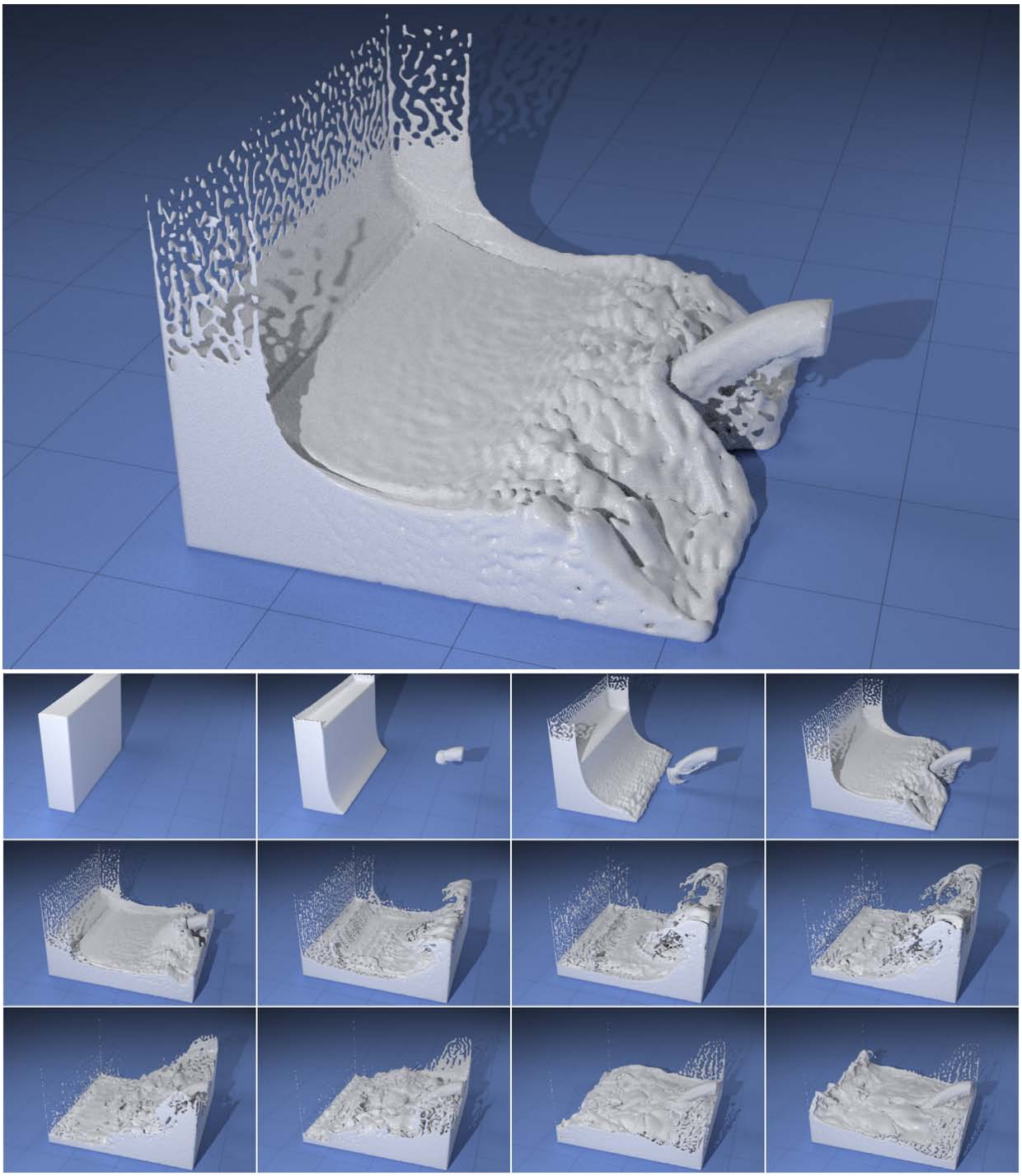


Figure 6.7: Visuals from our test case simulation, rendered using LuxRender [48], with lattice dimensions of $192 \times 192 \times 192$. The large frame is from the 1120th time step. The smaller frames are from time step 0 to 2000 with 181 time steps between each frame.

for the streaming and collide kernels we need to ensure that their performance within the fluid simulation is in line with expectations based on the isolated analyses.

In Section 6.1.2 we showed that the streaming made efficient use of the available memory bandwidth. Profiling confirms that this is still true within a fluid simulation. For each global memory access performed, 32 threads issued global memory instructions. This implies that all

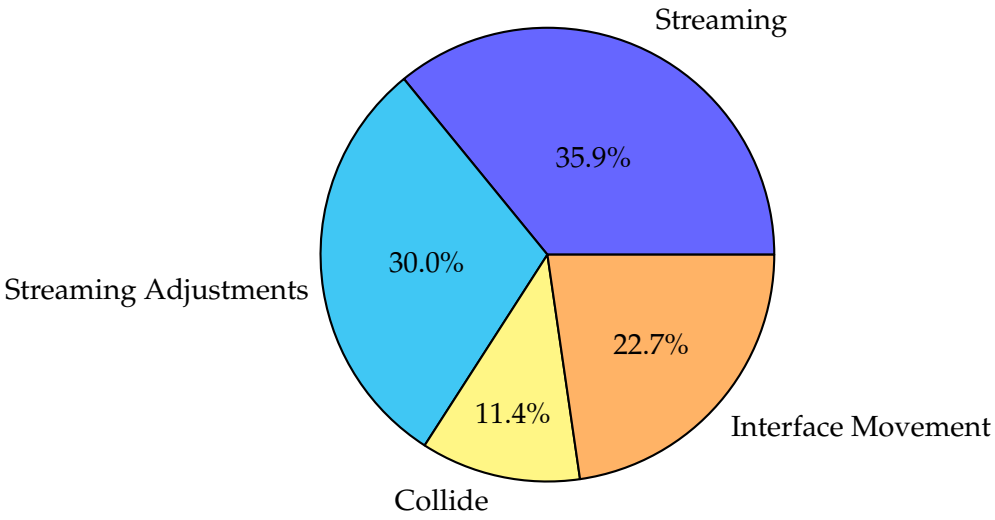


Figure 6.8: A brief overview of how much time is spent performing operations during each of the high-level stages stages of our GPU implementation. See Table 6.4 for per-kernel information.

Stage	**Kernel**	**% of execution time**
Streaming		**35.9**%
	y-axis streaming	9.9%
	z-axis streaming	9.7%
	xy-diagonal streaming	6.6%
	xz-diagonal streaming	6.5%
	x-axis streaming	3.2%
Streaming Adjustments		**30.0**%
	Interface Adjustments	15.9%
	Obstacle Adjustments	12.0%
	Source Adjustments	2.1%
Collide		**11.4**%
Interface Movement		**22.7**%
	Update Fluid Interface	9.2%
	Convert Full Interface Cells	8.2%
	Convert Empty Interface Cells	3.0%
	Finalise Fluid Interface	2.3%

Table 6.4: This table shows the percentage of execution time for each kernel during a fluid simulation for a $192 \times 192 \times 192$ lattice with 2000 time steps.

memory operations were fully coalesced. This is an expected result because there is little difference between the isolated test environment and a fluid simulation from the perspective of our streaming algorithms. The streaming kernels' high memory throughput is the main reason why the streaming kernels have relatively low execution times compared to the other kernels, considering the number of memory operations they are required to perform.

While the collide kernel is also shown to make efficient use of the memory bandwidth in Section 6.1.3, that was in the context of a fluid-only simulation where operations are performed on all cells. In a simulation with a fluid interface and obstacles, the collide kernel warps overlap with the fluid interface and result in memory operation inefficiencies. For our test case, 13–15% of all cells require collision calculations, but 20–35% of all warps overlap with these cells. On average, only 17 threads per warp are actively performing collision operations, which allows us to set a target for expected memory throughput as $\frac{17}{32} = 53.1\%$ of the available memory bandwidth ($\approx$ 55 GB/s). The measured result surpasses this target with a memory throughput of 61.00 GB/s. We can therefore conclude that the collide kernel has excellent performance within the context of a fluid simulation.

The performance of the source adjustment kernel is lower than expected. It makes use of only 30.7% of the available memory bandwidth. Unlike the streaming kernels, each thread of the source adjustment kernel only performs a single global memory read before completion. Since there is a single thread for each lattice cell, initialisation costs (for both hardware and thread-local variables) take a non-negligible amount of the total execution time. In order to improve the memory throughput of this kernel it would have to be redesigned so that, like the streaming kernels, each thread performs the source adjustments for multiple lattice cells. This will allow for better intra-warp latency hiding to take place.

Although the obstacle adjustment kernel accesses an average of only six values per memory request, it was still able to achieve a modest throughput of 64.78 GB/s. The interface adjustment kernel accessed an average of seven values per memory request, but only achieved a throughput of 21.21 GB/s. The difference in performance between these two kernels is likely to result from the obstacle adjustment kernel performing half the number of instructions (since it performs no calculations for the LBM) that the interface adjustment kernel performs. Further investigation is required to confirm this.

As expected, the interface movement kernels also all have low memory throughput (15.29 GB/s to 45.27 GB/s). As with the source adjustment kernel, each thread for the interface movement kernels operates on a single lattice cell. For cells where no work is required, a non-negligible amount of initialisation is required. This is likely to play a role in their poor performance, but further work is required to identify the exact reasons for the poor performance and suggest specific areas where improvements are possible.

6.4 Conclusions

In this chapter we presented an in-depth analysis of the streaming and collide kernels. We showed that these kernels, which perform the core LBM operations, were operating efficiently. The streaming kernels were within 95% of the practical hardware performance limits and achieved speedups of up to 29.6$\times$ when compared to equivalent single-core CPU code. The collide kernel

used more than 90% of the practically available memory bandwidth and effectively hid 70.7% of the cost of collide calculations behind the time taken to perform global memory operations. Ignoring outliers, this allowed the collide kernel to achieve speedups of up to 223.7× when compared to equivalent single-core CPU code.

We then provided a discussion of the inefficiencies that are expected of the adjustment and interface movement kernels within the context of a valid fluid simulation. The most significant inefficiency was that these kernels will have idle threads within warps that perform adjustment and movement operations as a result of the memory access patterns being dependent on the layout of the scene and the shape of the fluid interface. This poor use of available memory bandwidth, and the numerous global memory operations required by these kernels, are the main reasons for their poor performance.

Finally we discussed the performance of the streaming, collide, adjustment and interface movement kernels within the context of a valid fluid simulation. We were able to confirm that the streaming and collide kernels maintained their high performance. The source and obstacle adjustment kernels were shown to have adequate performance, but, as expected, the performance of the interface adjustment and interface movement kernels was low as a result of the dynamic structure of the fluid interface not aligning well for efficient global memory accesses on the GPU. Further investigation into improving the performance of these kernels is left to future work.

Chapter 7

Simulation Results

The promise of massive performance improvements is the main incentive behind any GPU implementation. In order to understand whether our GPU LBM implementation realizes these performance benefits we need to measure its performance and compare it to our reference CPU implementation. This is the focus of this chapter. This contrasts with the previous chapter where we analysed the performance and limitations of the individual components that make up our GPU LBM simulation. When measuring performance, we look at both how quickly the simulation runs and how much the geometry generated by our GPU simulation deviates from that of the CPU simulation. We also discuss simulation artefacts and limitations that result from our implementation of fluid surface tracking.

We structure our discussion around three different test cases: *drop into a pond*, *dam break*, and *flow around corner*. Each of these scenes have been chosen to test specific aspects of our fluid simulation. *Drop into a pond* displays the effects of fluid splashing, without the need for moving obstacles in a scene. *Dam break* is an example of *sloshing* behaviour — an important area of research in many fields [36]. *Flow around corner* showcases fluid sources and the interaction of fluid with obstacles in a scene. Using these scenes we measure the speed-up achieved against our CPU implementation so that we can quantify the performance benefits from our GPU implementation. We also validate the results of the GPU simulation by showing that there are no significant geometric differences between the results of the CPU and GPU simulations for all test cases.

7.1 Test Scenes

Selecting the right scenes with which to measure the performance of our simulation is important. Scenes with high fluid content amplify the performance gains from using graphics hardware because more fluid means more parallelized computation on the GPU. However, scenes that produce visual results (large splashes, turbulent fluid gushing through a scene) typically require more empty space through which the fluid surface can move. More empty space means there

Δx	lattice cell width	0.01 m
C	Smagorinsky constant	0.1
η	viscosity	1×10^{-7} Pa s
g_c	stability constant	5×10^{-4}
$\mathbf{g}_f$	gravity	$\langle 0, -10, 0 \rangle$ m/s^2
$\mathbf{u}_{0f}$	initial fluid velocity	$\mathbf{0}$ m/s^2
$\mathbf{u}_{sf}$	source velocity	$\langle 0, 0, 0.25 \rangle$ m/s^2

Table 7.1: The constants used during the simulations in this chapter. The value for gravity was based on gravity on Earth. The value for viscosity was chosen based on water's viscosity. The rest of the values were chosen based on Reid's results [69] and manually adjusting values to achieve the desired visual output.

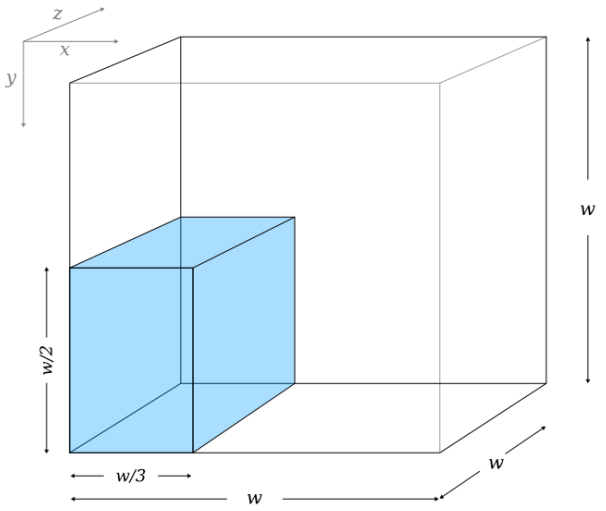


Figure 7.1: The initial conditions for the *dam break* test scene. A wall of fluid is placed against one side of a bounding obstacle box. The width, height and depth are all equal.

are more lattice cells that do not require computation, which means lower expected performance gains from using graphics hardware. In these results we have selected scenes where there is enough free space in which the fluid surface can move, but have ensured that we do not significantly undermine the performance gains of our GPU implementation by minimizing the amount of wasted empty space.

We discuss each test case below. The simulation parameters defined in Table 7.1 are common to all test cases. All visuals are rendered using LuxRender [48], after smoothing the output mesh using MeshLab [51]. An opaque surface is used instead of a transparent one to enhance the visibility of the fluid surface.

7.1.1 Dam break

The *dam break* test scene (Figures 7.1 and 7.2) is a simple scene that showcases fluid interactions with gravity, splashing detail and sloshing motion. The scene is built from a cube as a bounding

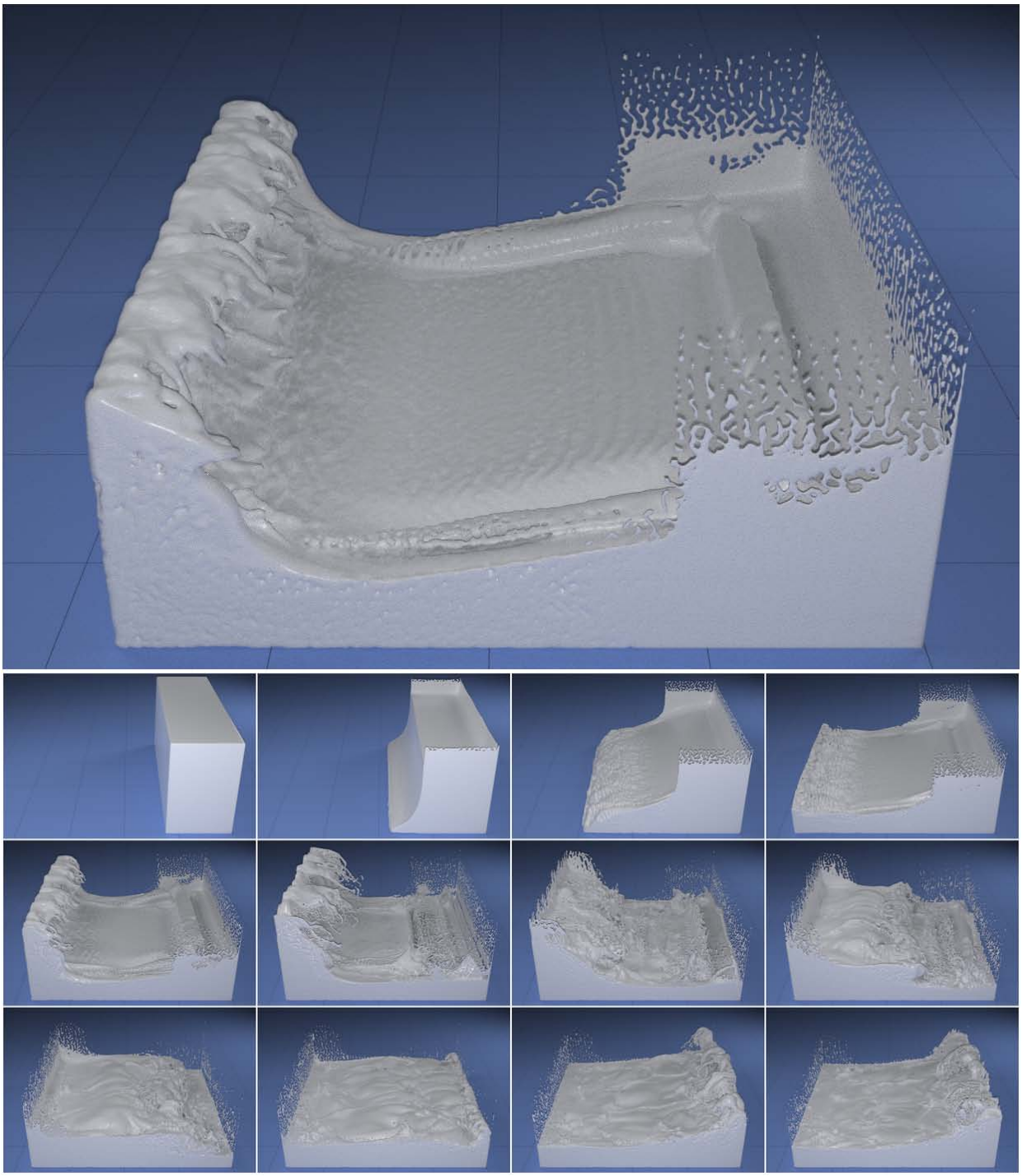


Figure 7.2: Visuals from a *dam break* simulation with lattice dimensions of $320 \times 320 \times 320$. The large frame is from the 1060th time step. The smaller frames are from time step 0 to 3190 with 290 time steps between each frame.

obstacle box, with a wall of fluid against one edge of the scene. Gravitational pull drives the fluid motion. It is possible for the fluid wall to extend to the ceiling of the bounding box for smaller lattices (dimensions $< 192^3$), but, for larger lattices, the greater fall distance results in fluid falling faster than the maximum lattice velocity, thus causing fluid instabilities. Therefore, we set the fluid wall to be half the height of the scene to avoid these instabilities.

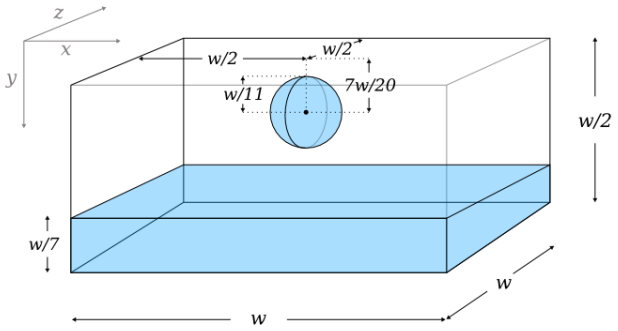


Figure 7.3: The initial conditions for the *drop into pond* test scene. A sphere of fluid is positioned above a stationary pool inside a bounding box. The sphere is positioned so that it will fall into the centre of the pool and cause a splash.

The dam break test case has been inspired by the literature — both Thürey [81, 82] and Reid [69] use variations of this test case to showcase their LBM VOF implementations. It is also a good example of gravity introducing momentum and energy into a scene by driving the fluid across the floor and up the opposite wall. Splashing, sloshing and basic fluid-obstacle interaction can be observed.

There are two noticeable visual artefacts in the visuals from this test scene: the appearance of bubbles where the fluid touches the boundaries of the scene and small drops that are floating against the boundaries of the scene. The first is the result of a simulation artefact that is discussed below in Section 7.4. The second artefact is actually a property of the simulation, and is mirroring similar behaviour from the real world i.e. stationary droplets on vertical surfaces. In the real world, evaporation causes these droplets to eventually disappear, but in our simulation this is not possible. Removing these droplets from the scene would probably be better solved by post-processing geometry and not by modifying the underlying fluid simulation.

7.1.2 Drop into pond

The *drop into pond* test scene (Figures 7.3 and 7.4) demonstrates the behaviour of the fluid simulation with large splashes. The scene consists of a bounding box with stationary fluid at the bottom and a separate large sphere of fluid set above the pool. We rely on gravity to cause the fluid ball to drop and create a splash in the fluid. The width and depth of the scene are twice that of the height to provide a larger surface area on which to see the effects of the splash.

This scene was chosen because of its use in the literature[69, 82] as an example of the visual results of a splash into fluid. The *coronet* pattern that forms after the initial impact is a notable splashing feature that we see exhibited by this simulation. The motionless pool of fluid at the start of the simulation is also evidence of the stability of the fluid simulation.

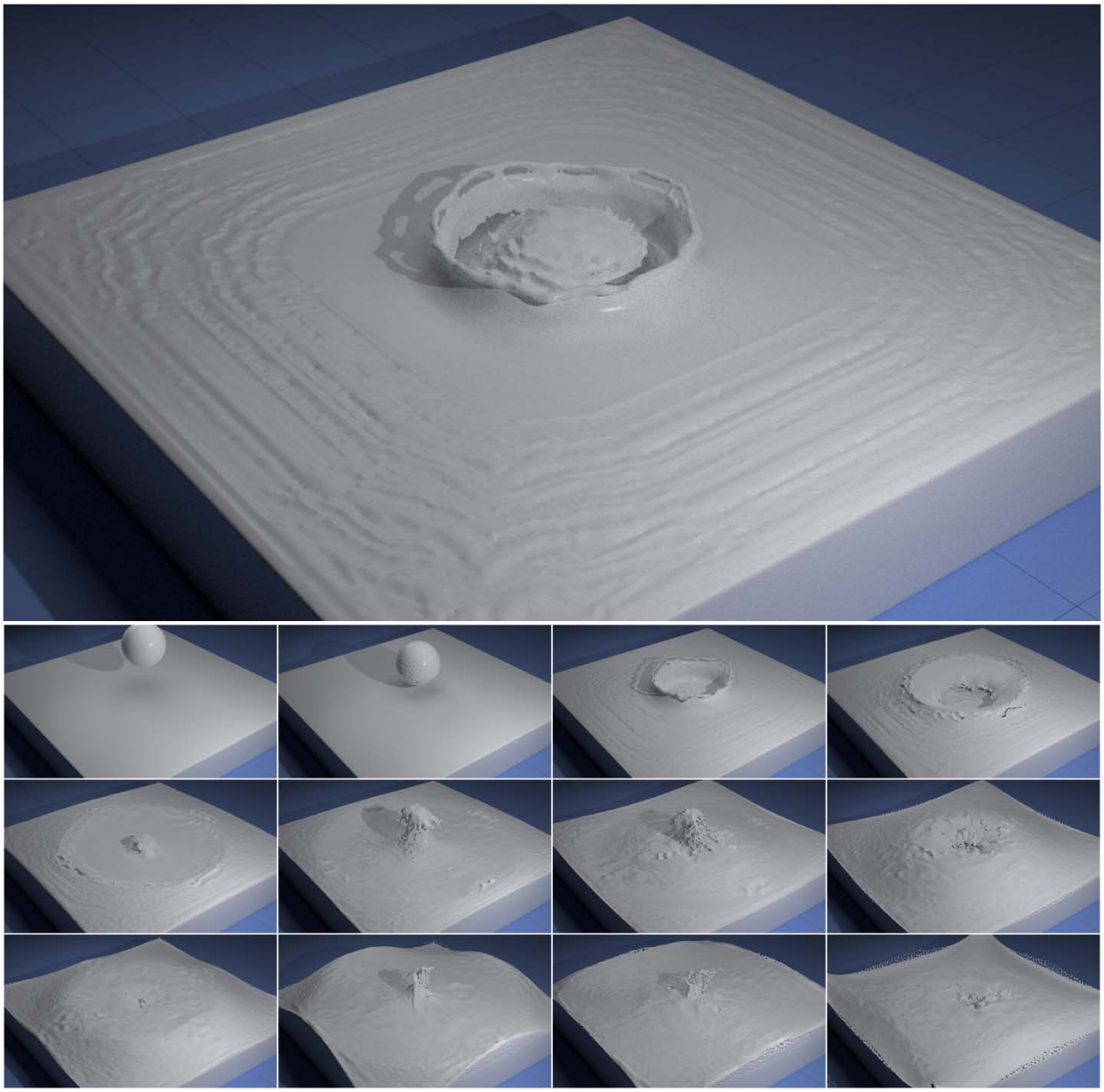


Figure 7.4: Visuals from a *drop into pond* simulation with lattice dimensions of $384 \times 192 \times 384$. The large frame is from the 460th time step. The smaller frames are from time step 0 to 2992 with 272 time steps between each frame.

7.1.3 Flow around corner

The *flow around corner* test scene (Figures 7.5 and 7.6) was chosen to show interaction between fluid and obstacles as the fluid flows through an artificially created passageway and to demonstrate the use of fluid sources. A rectangular wall is inserted into the scene to create a cornered passageway. In this scene the height is much smaller relative to the width and depth than in the other scenes so that the fluid can flow over a greater horizontal distance. We added a wedge on the outer corner of the passage way to force the fluid to change direction more rapidly, resulting in a more dramatic flow to towards the end of the passage. Pillars were added as obstacles with which the fluid can interact as it flows down the passage. A wall of fluid is added over the source

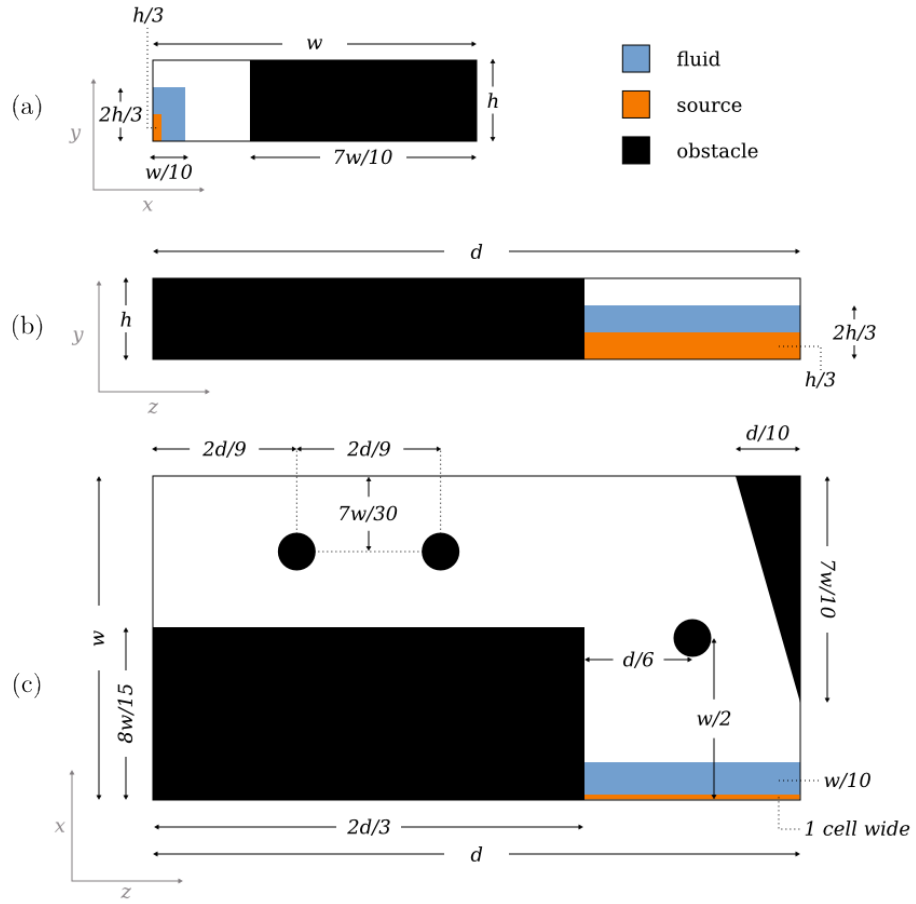


Figure 7.5: The initial conditions for the *flow around corner* test scene are shown via cross sections along the *xy-*, *yz-* and *xz* planes in sub-images (a), (b) and (c) respectively. The radius of the cylinder obstacles is $\frac{w}{15}$ and they extend from the floor to the ceiling. The ratio of width:height:depth is $4:1:8$.

to increase the initial fluid volume in the scene and provide an initial rush of fluid.

This scene was chosen as a good example of fluid interacting with obstacles in a scene. We chose to have the fluid flow around a corner into a narrow passageway so that the fluid will be sloshed across the passageway. This movement across the passageway also adds extra splashing detail. This test case is also intended to be reminiscent of scenes from movies such as *Titanic* and *Deep Blue Sea* where water violently gushes into hallways.

7.2 GPU Performance Compared to CPU

Comparing the performance of the GPU implementation is important for measuring its effectiveness as a whole. By examining the speedup results we are able to quantify the benefit of using a GPU LBM implementation over a CPU version.

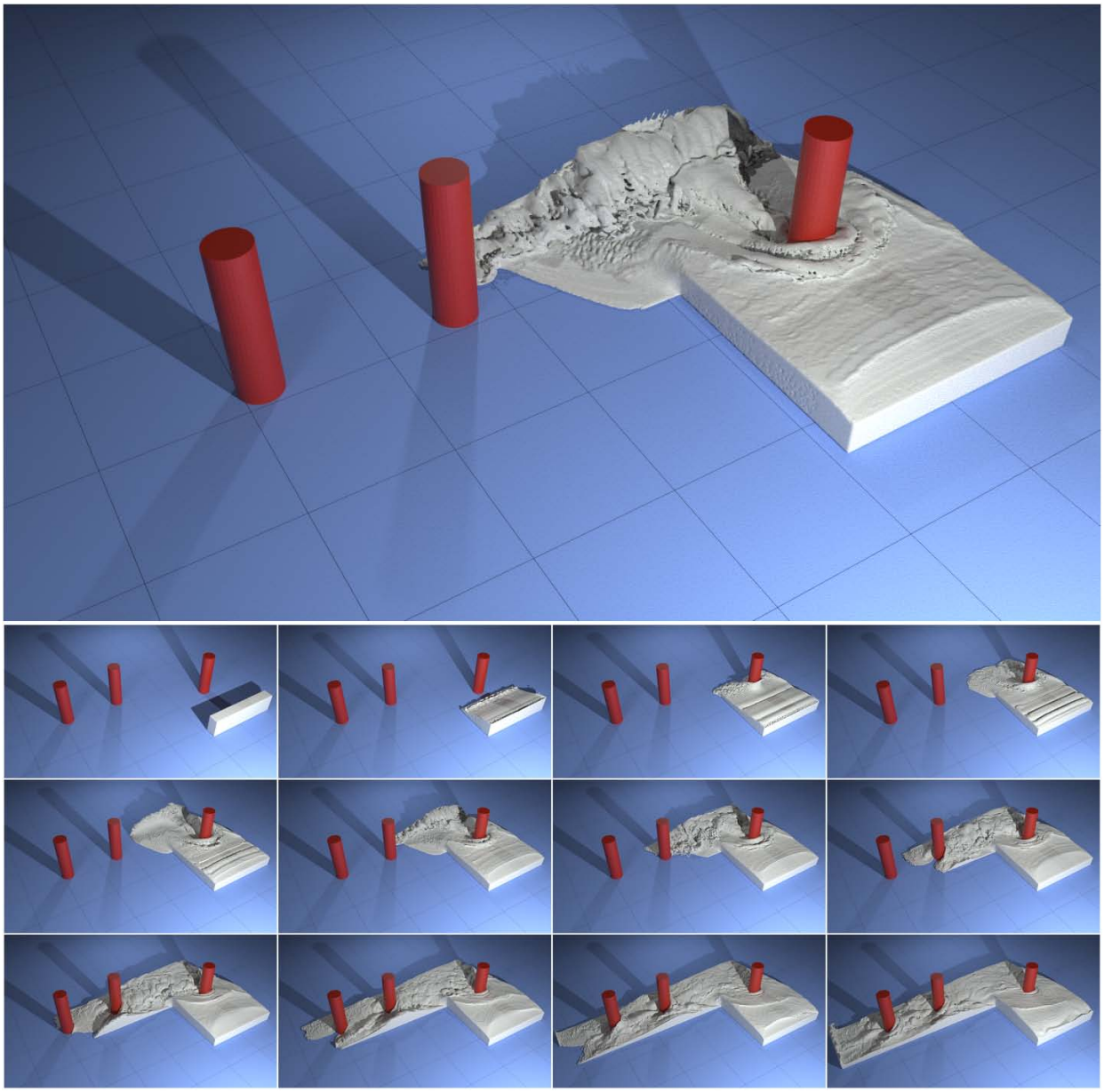


Figure 7.6: Visuals from a *flow around corner* simulation with lattice dimensions of $384 \times 96 \times 768$. The large frame is from the 2118th time step. The smaller frames are from time step 0 to 4499 with 409 time steps between each frame.

7.2.1 Method

Generating speedup data for our GPU implementation requires two measurements — the execution time for both the CPU and the GPU implementations. Since "execution time" is a very broad term, this section defines what we measured and how we measured it. We also outline our method for choosing appropriate kernel launch configurations for the GPU implementation.

For our analysis we used the hardware outlined in Table 7.2. Unlike the analysis in Chapter 6, these results concern themselves with raw performance, so the GPU from Table 6.1 was too out-of-date to be used. Our primary reason for using EC2 instances was that we did not have a new generation GPU at our disposal and EC2 provided relatively inexpensive access to a Kepler GPU

with sufficient memory. We decided to use EC2 instances for both the GPU and CPU simulations to maintain a consistent platform for all tests. We chose the g2.2xlarge for our GPU simulation because it had the most modern GPU and the c3.2xlarge instance for the CPU simulation because it had the fastest available CPU and enough RAM.

We measured the full simulation time for both the CPU and GPU implementations: program startup, data structure initialization, simulation run-time and simulation cleanup. An external wrapper script was used to execute our simulation for both the GPU and CPU simulations, and this wrapper script measured the time it took for the executable to complete.

The output of the profiling simulation is a field from which 3D mesh geometry for the fluid surface can be extracted, i.e. the fill fraction ($\varepsilon(\mathbf{x})$). Therefore, we include the operations to transfer the fill-fraction field from GPU memory to the CPU memory in the profiling simulation. As a further optimization, this data need only be transferred for the frames required to produce the animation. By using the simulation time and a desired video frame-rate our GPU simulation calculates which frames are necessary for transfer from GPU to CPU memory in order to maintain the required video frame-rate.

Since our analysis is focussed on simulation performance, our profiling simulations did not render visuals, write simulation output to disk or generate 3D mesh geometry. These components are unrelated to the LBM simulation internals and could easily be replaced with more efficient implementations or run on different hardware platforms to that of the simulation. Therefore, they were removed so that any inefficiencies in their implementation would not affect the overall results.

Our CPU simulation was run on the system outlined in Table 7.2. All our simulations were done using single-core CPU code, despite the availability of multiple CPU cores. It is possible to parallelize the CPU implementation to take advantage of multi-core CPUs [69], using tools like OpenMP [20]. However, doing so would require understanding another framework, optimizing the CPU code and modifying the algorithm to remove potential race conditions and is, therefore, beyond the scope of this thesis. There were no extra parameters specified for the CPU simulation that were not also specified for the GPU simulation.

Our GPU simulation was run on an NVIDIA GRID K520 (details in Table 7.2), a high-performance device that is available through using Amazon's EC2 cloud computing platform. Unlike the CPU simulation, choosing the number threads to be executed and the logical layout of those threads (the thread block and grid size) for each GPU kernel is an important part of achieving optimal performance. In our testing, this meant finding appropriate dimensions for the thread block and grid for each kernel for each lattice size of each test scene — a process typically used as part of GPU tuning. For our kernels, the dimensions of the thread block determine the dimensions of the grid, so we focused on sampling various thread block dimension combinations. For the purposes of this discussion, we refer to thread block width, height and depth as t_w, t_h and t_d, respectively.

	CPU Simulation	GPU Simulation
Instance Size	c3.2xlarge	g2.2xlarge
Cost	1.68 USD/hour	0.65 USD/hour
CPU	Intel Xeon CPU E5-2680	Intel Xeon CPU E5-2670
CPU Cores	8	8
CPU Clock Speed	2.8 GHz	2.6 GHz
CPU Memory	15 GiB DDR3	15 GiB DDR3
Graphics Card Model		NVIDIA GRID K520
GPU Generation		GK104 (Kelper)
GPU Clock Rate		797 MHz
GPU Memory		8 GiB GDDR5 (4 GiB per GPU)
GPU Memory Clock Rate		2500 MHz
GPU Cores		1536
CUDA Runtime Version		5.5

Table 7.2: The hardware used to perform our speedup simulations. We used EC2 instances to get access to modern GPUs with 4 GiB of RAM. Although both instances have access to 8 cores, we only made use of a single core. The NVIDIA GRID K520 has two GPUs internally, but only a single GPU is available to a g2.2xlarge instance.

For each GPU simulation we picked the near-optimal dimensions by testing the performance of a chosen set of values for t_w, t_h, and t_d for each kernel. For t_w we only considered values that were a multiple of warp size (32) from 32 until one multiple larger than the width of the scene, because threads are batched together as warps by the GPU and these values ensured no partial warps (except for the partial warp that might be present at one lattice boundary). For t_h and t_d we tested values from 1 to 8 because basic manual experimentation showed a large drop off in performance for values greater than 8; this decrease in performance is visible in the results in Chapter 6. We also made sure that the thread-block dimension restrictions described in Section 5 were applied (e.g., horizontal streaming required t_w to be equal to the width of the scene).

Pseudocode 13 outlines the process we used to test each thread block dimension. For each kernel and thread block dimension we measured the total GPU runtime using the CUDA profiler. The thread block dimension with the lowest run-time was selected for the final speedup simulation. This sampling process allowed us to select near-optimal thread block dimensions in a reasonable time because we did not have to run the full simulation for all possible thread block permutations.

Speedup for our simulations was calculated using:

$$\frac{\text{Total CPU Runtime}}{\text{Total GPU Runtime}}$$

In order to get a sense of how this speedup changes with lattice size we iterated through dimen-

Pseudocode 13 The pseudocode describing how we iterated over the possible thread block dimensions to determine the optimal configuration for a given GPU simulation.

```
scene dimensions = (w, h, d)
for z = 1 → 8 do
    for y = 1 → 8 do
        X = array of possible values for x-dimension of thread block
        RUN-SIMULATION(X, y, z)
        Record execution time for each set of kernel launch parameters
    end for
end for

function RUN-SIMULATION(X, y, z)
    N = length [X] × 50
    i = 0
    for n = 0 → N do
        x = X [i]
        CONFIGURE-LAUNCH-PARAMETERS (x, y, z)
        Execute kernels to simulate the next time step
        i = (i + 1) % length [X]
    end for
end function

function CONFIGURE-LAUNCH-PARAMETERS(t_w, t_h, t_d)
    for each kernel k do
        threadBlock [k] = (t_w, t_h, t_d) = restrictions for kernel k applied to (x, y, z)
        grid [k] = ( ⌈w/t_w⌉ , ⌈h/t_h⌉ , ⌈d/t_d⌉ )
    end for
end function
```

Test Scene	width : height : depth	Smallest Lattice	Largest Lattice
Dam break	1 : 1 : 1	$(16, 16, 16)$	$(320, 320, 320)$
Drop into pond	2 : 1 : 2	$(32, 16, 32)$	$(384, 192, 384)$
Flow around corner	4 : 1 : 8	$(64, 16, 128)$	$(384, 96, 768)$

Table 7.3: A summary of the sizes and dimensions of the lattices used by our test scenes.

sions of increasing size, calculating the speed up for each dimension for each test scene. The smallest lattice was chosen for each test scene such that the lowest dimension was 16. Each iteration incremented the lowest dimension, calculating the other dimensions based on their size relative to the lowest dimension for that test scene. For *drop into pond* and *dam break* the lowest dimension was incremented by eight, because it was too time consuming to perform the CPU simulation for the larger lattice sizes ($>$ 6 hours per simulation). The memory requirements for *flow around corner* increase very quickly, which means there are fewer possible lattice sizes that fit within GPU memory. Therefore, we incremented the lowest dimension by four for this test scene. We made exceptions to these increments for small lattice sizes (less than 300 000 lattice cells), where we incremented to lowest dimension by 1, because CPU simulation run-times were

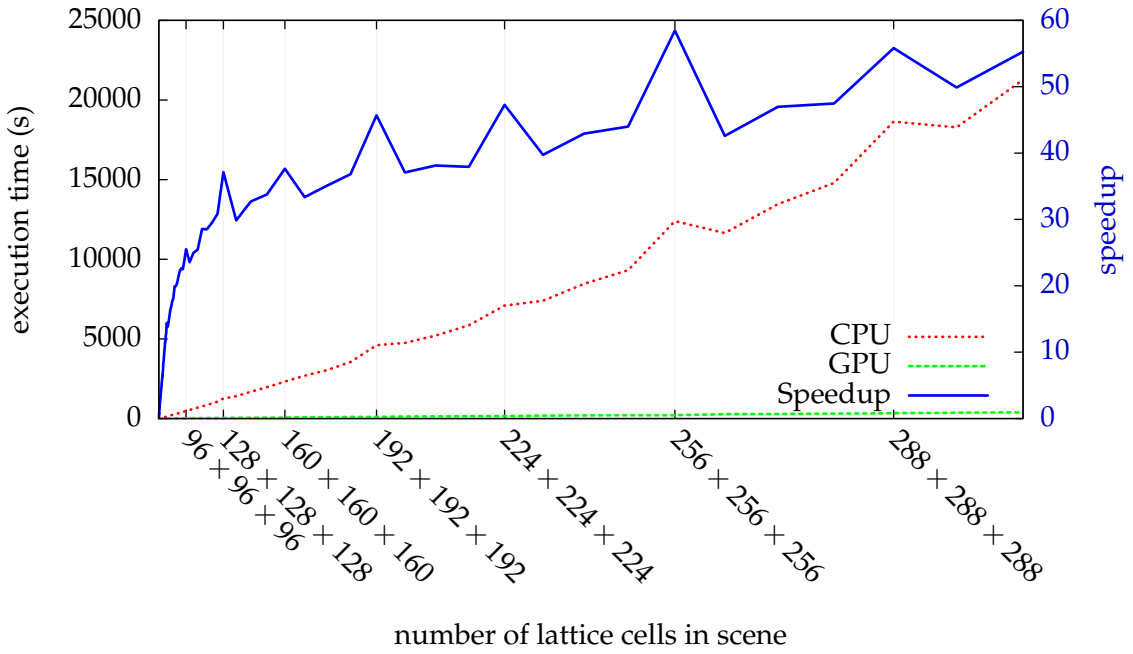


Figure 7.7: The speedup graph for the *dam break* test case. The runtime of the CPU (red) and GPU (green) simulations is measured on the left axis, with the speedup (blue) on the right axis.

not prohibitively long. The largest lattice was chosen as the largest lattice able to fit into GPU memory. See Table 7.3 for more information regarding lattice sizes for our test scenes.

7.2.2 Discussion of Results

Figures 7.7–7.9 and Table 7.4 present the results from our speedup simulations. In this discussion we explain the reasons for:

- the deviations from the linear increase in CPU execution time;
- the logarithmic increase in speedup with respect to the number of lattice cells;
- the variance in speedup for similar dimensions; and
- the differences in performance and speedup between our test scenes;

Aside from explaining the all-important expected overall speedup, the reasons behind these observations help us understand the bottlenecks of our GPU implementation and what properties of scenes will lead to improved GPU simulation performance. We conclude this section with an estimation of the reduction in simulation cost that is implied by our results.

Before comparing the CPU and GPU performance, it is worth noting deviations in CPU execution time, because these deviations affect the calculated speedup. In general there is a clear linear increase in CPU execution time with respect to the number of lattice cells. However, there is a significant deviation from this linear increase for a few lattice sizes, particularly those that

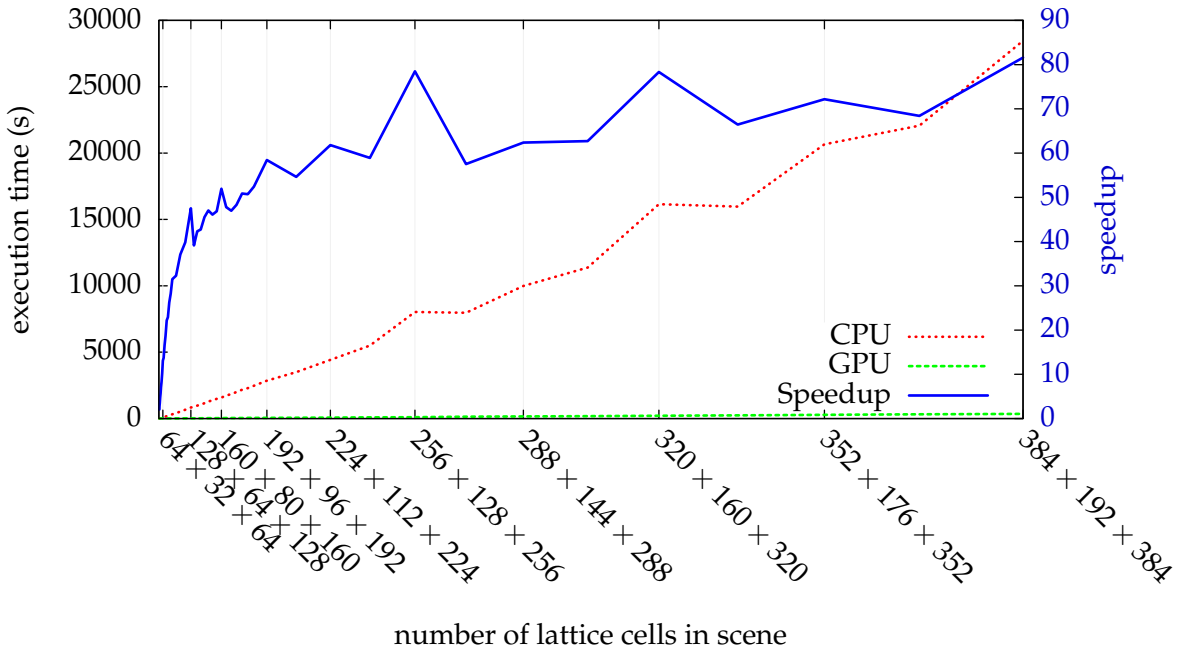


Figure 7.8: The speedup graph for the *drop into pond* test case. The runtime of the CPU (red) and GPU (green) simulations is measured on the left axis, with the speedup (blue) on the right axis.

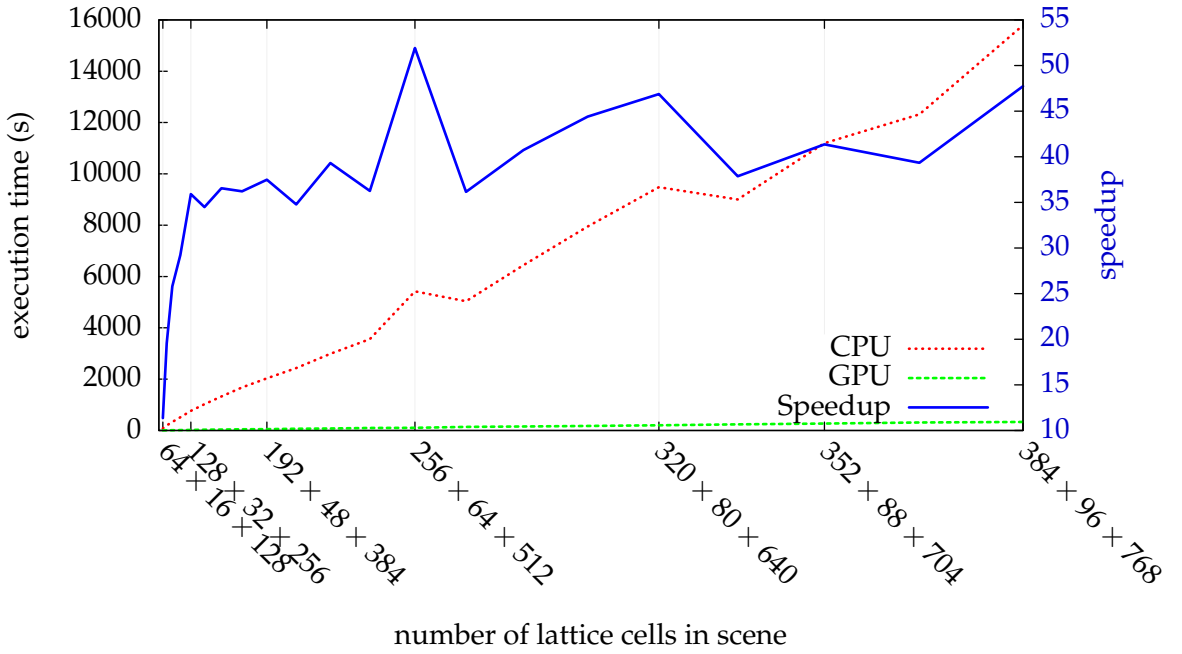


Figure 7.9: The speedup graph for the *flow around corner* test case. The runtime of the CPU (red) and GPU (green) simulations is measured on the left axis, with the speedup (blue) on the right axis.

are of significance for the GPU simulation (i.e. lattices that have dimensions that are multiples of warp size). This is the result of the behaviour noted in Chapter 6 when analysing the performance of the collide kernel. Initial investigation and profiling to find the reason for these anomalies could find no trivial cause. Since this work focuses on the GPU implementation and the surrounding data points give an adequate indication of relative performance we have left the

	Dam break	Drop into pond	Flow around corner
GPU Simulation	896.3 ms	1087.4 ms	680.6 ms
CPU Simulation	68843.2 ms	41278.0 ms	41278.0 ms

Table 7.4: This table shows the average simulation time to generate the simulation data from which each visual frame is rendered. The numbers are averaged for all rendered frames of each test case. On average, 7 LBM steps are performed between each rendered frame.

deeper investigation of these deviations as future work.

All three test cases show a logarithmic increase in speedup with an increase in lattice size. This is caused by both lattice initialization taking a smaller percentage of total runtime in larger lattices and smaller lattices not taking advantage of all the parallelism available on the GPU. For small lattices, the initialization and data transfer costs dominate the execution time for the GPU implementation. Furthermore, small lattices do not produce sufficient simulation data to saturate the available GPU bandwidth during simulation. At these lattice sizes, the fact that the CPU implementation does not perform host-device data transfers and does not waste available parallel bandwidth is significant and means that the GPU implementation's speedup is reduced. For simulations with larger lattices, lattice initialization takes a less significant percentage of simulation runtime because there is more simulation data to process at each time step. The large amount of data to process makes it possible for the GPU bandwidth and parallelism to be used effectively. This results in the performance boost we see for larger lattice sizes. Due to the limited amount of memory available on the GPU we are unable to perform simulations for arbitrarily large scenes and, therefore, unable to measure the trend in speedup beyond what we present in our results. However, the shape of our speedup graphs would suggest that larger lattice sizes would not result in significantly larger speedups.

In addition to the logarithmic speedup increase, there is noticeable variance in speedup for similar lattice sizes. The most significant pattern is the step down in speedup following a lattice with widths that are well-suited for GPU execution (i.e. the width is a multiple of warp size, resulting in all threads being used by the simulation). Examples occur in Figure 7.7 after a lattice of $160 \times 160 \times 160$, in Figure 7.8 after a lattice of $192 \times 96 \times 192$ and in Figure 7.9 after a lattice of $192 \times 48 \times 384$. For lattices with widths slightly larger than the optimal width, extra warps need to be scheduled to handle the extra few cells. This is inefficient because only a few threads of these warps are required to operate on the extra cells, while the rest remain idle. For our test scene widths (16–384 cells wide) this can mean that 10–50% of all warps are idle if the lattice width is slightly larger than a multiple of 32. It is these idle threads that cause the reduction in GPU performance and thus the reduction in overall speedup when compared to slightly smaller lattices that have dimensions that are better suited to GPU execution.

Each test scene also shows a different speedup profile. This is mainly due to differences in the number and position of fluid cells relative to empty and obstacle cells for each scene. The

Test Scene	% Interface Cells	% Fluid and Interface Cells
Drop into pond	0.97%	29.09%
Dam break	1.92%	22.99%
Flow around corner	0.75%	8.07%

Table 7.5: The percentages of all cells that are interface and fluid cells. These values are averaged over the entire simulation.

drop into pond test scene has both the highest percentage of fluid cells of all the test scenes (i.e. the fewest empty/obstacle cells) and relatively few interface cells and, as a consequence, has the highest speedup. On the other hand, the *flow around corner* test case has the lowest percentage of fluid cells (i.e the most empty/obstacle cells) and the highest percentage of interface cells relative to fluid cells and thus has the lowest speedup. Scenes with a higher percentage of empty cells perform relatively poorly because our GPU implementation performs distribution function streaming regardless of cell type, whereas the CPU implementation only streams for fluid cells. Therefore, although the GPU has higher parallelism, it also does a lot more work than the CPU implementation for scenes with a high number of empty and obstacle cells. The percentage of interface cells is also relevant because, as discussed in Chapter 6, the interface management kernels are less efficient than the streaming and collide kernels because of the structure of the fluid interface. This explains why the test scenes with a higher percentage of fluid interface cells are less efficient. Table 7.5 provides an overview of how many interface and fluid cells there are on average for the simulations in each of our test scenes.

Combining our performance results and the hardware information provided earlier in this chapter (Table 7.2, we can derive an estimate of the economic benefit of GPU computing with:

$$\frac{\text{Cost of CPU Hardware}}{\text{Cost of GPU Hardware}} \times \text{GPU speedup}$$

Since our CPU implementation makes use of only a single core, it would be unfair to perform a direct comparison of an instance that has 7 unused CPUs. Therefore, when comparing the cost of our CPU and GPU implementations, using an eighth of the cost of the c3.2xlarge (0.21 USD/hour) results in a more relevant cost comparison. This assumes that the LBM is parallelizable across multiple CPU cores — a safe assumption considering in this work we have parallelized it across orders of magnitude more GPU cores. Taking the above into account we estimate the cost of execution is 17× cheaper for *dam break*, 25× cheaper for *drop into pond*, and 15× cheaper for *flow around corner*. It is worth noting that these estimates are based on EC2 cloud prices, which can be considered a rough approximation of the relative costs of purchasing, maintaining and running the underlying physical hardware.

— ∼ —

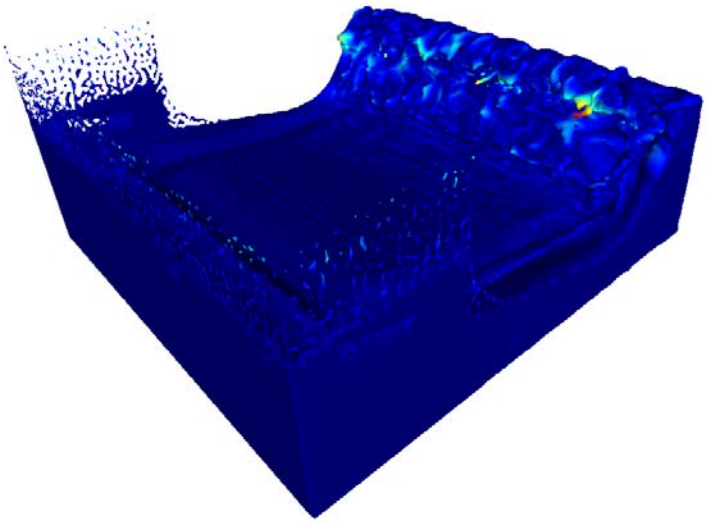
We have shown that our GPU implementation achieves speedups of up to 81.6×, demonstrating that our work obtains the performance benefits expected from GPUs. Furthermore, these speedups can result in reducing the cost of running the simulation by an order of magnitude. We also investigated how certain properties of scenes (i.e. low numbers of empty, obstacle and interface cells) lead to greater GPU performance. Furthermore, we highlighted areas for improvement in the implementation of our streaming and interface management kernels in Chapter 6. Up to this point we have only discussed the computational performance of the simulation, it is still important to show that our results are not achieved at the expense of the validity of the fluid simulation. The next section investigates the geometric differences between the CPU and GPU simulation to determine if there are any accuracy issues.

7.3 Differences in Geometry for CPU and GPU simulations

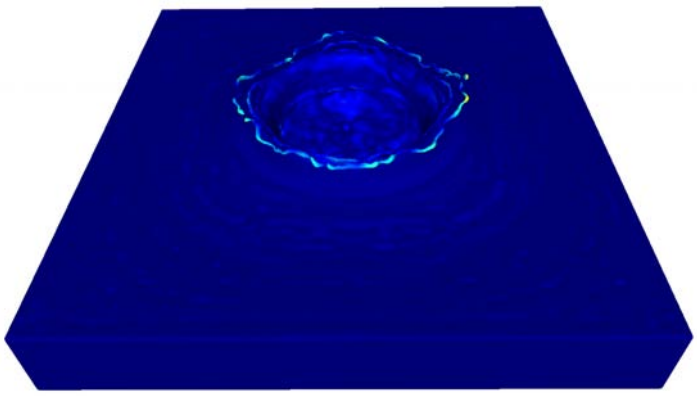
The Lattice Boltzmann method is a well established method of simulating fluid motion [77, 92], and therefore we do not need to show that the algorithm itself leads to a valid fluid simulation. Our CPU implementation follows on from work done by Ried [69], so further validation of the CPU implementation is unnecessary. On the other hand, our GPU implementation has a completely new, different, structure to our CPU implementation, which could introduce unintended differences. The obvious approach of directly comparing distribution functions from each implementation is not trivial because variations in floating point operations between the two platforms cause differences in significant digits after only a few time steps. Instead we compare the primary output of the simulation: the 3D mesh. This has the advantage of being both visually simple to understand, and easy to calculate as it is a common operation applied to 3D geometry. We, therefore, use the deviations between the final geometries produced by each platform as a means for inferring the validity of our GPU implementation from our CPU implementation.

For each of our test scenes, we selected a single time step for which we generated geometry and measured the differences between the CPU- and GPU-generated meshes. We allowed the simulation to run for at least 500 time steps to allow differences between the GPU and CPU implementations to be exaggerated. We also selected frames that captured the details of splashing or turbulence since minor differences in distribution functions of a turbulent fluid surface have a greater effect on the fluid surface geometry than differences near a stable fluid surface. We used a *Hausdorff distance* filter to visualize these differences (via *MeshLab* [51]) — a common measure of geometric distance between 3D meshes [13]. In basic terms, we measure differences in geometry by measuring the distance between every vertex in the GPU-generated mesh and its closest vertex in the CPU-generated mesh. This measured distance is then visualized as a heat-map on the surface of the GPU-generated mesh.

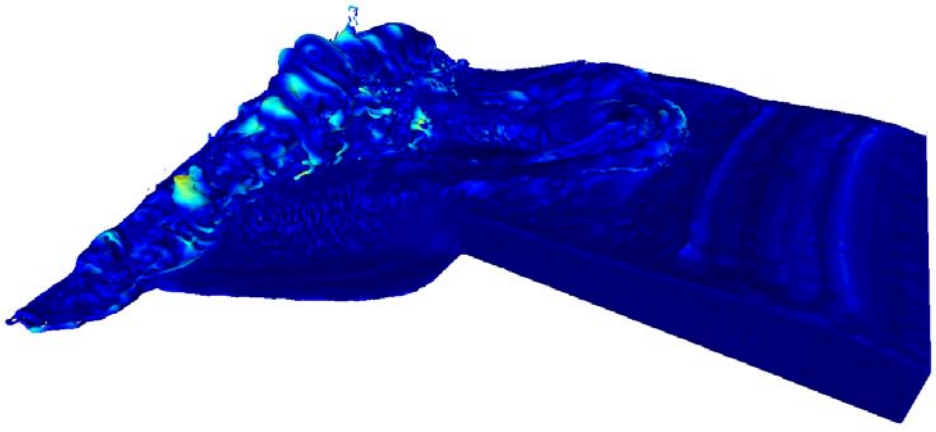
The deviations in geometry between our CPU and GPU simulations cases are shown in Figures 7.10a, 7.10b and 7.10c. We see that there are almost no deviations in the core shape of the



(a) dam break - 1000th time step



(b) drop into pond - 600th time step



(c) flow around corner - 2000th time step

(d) Scale

Figure 7.10: The differences in geometry between the GPU and CPU simulation are visualized as a heat-map on top of the geometry produced by the GPU simulation for each of our test cases. Deviations are calculated using the Hausdorff distance. The colours represent the magnitude of the deviation as a percentage of the length of the bounding box diagonal. Our scale (Figure 7.10d) shows a maximum deviation of 5% of the bounding box diagonal.

fluid, which confirms that there are no significant problems with our GPU LBM implementation. In areas where the fluid surface experiences higher turbulence, the differences between the CPU- and GPU-generated meshes are more pronounced. These minor deviations are due to minor differences in distribution functions between the CPU and GPU simulations, because the different order of operations introduces small differences in the results of floating point operations. Such differences are to be expected and are not symptomatic of any problems with the GPU LBM simulation.

7.4 Simulation Artefacts/Limitations

While our LBM implementation provides an efficient linear algorithm for free-surface fluid simulations, it is not without limitations. Specifically, it is unable to accurately model:

- fluid flowing faster than a maximum speed, defined by the size of lattice cells;
- droplets that are only a few lattice cells in size;
- free surface contact with obstacles; and
- bubbles underneath the fluid surface.

The core structure of the LBM is to divide the simulation scene into a discrete lattice of cells. This discretisation combined with streaming, which only transfers movement information between neighbouring cells, results in an implied limit for the speed at which fluid can move: 1 cell per time step (see Section 3.5 for more information on the maximum lattice velocity). This limit is particularly noticeable in free surface simulations where droplets are accelerated by gravity. Once the maximum fluid velocity is reached, the fluid surface starts to flatten out in the direction it is travelling as the diagonal forward distribution functions move momentum to their horizontal neighbours. There exist methods to mitigate this effect (e.g., using a variable time step that adjusts based on the maximum fluid velocity [82]), but porting these to our GPU implementation is left as future work.

Modelling small droplets is also problematic with a discrete lattice. When droplets only cover a few lattice cells, the transfer of mass between cells is limited because there are only a few neighbours. This prevents proper downward movement due to gravity by obstructing the creation of new downward facing fluid cells. This lack of movement increases in the pressure of fluid cells each time step due to the way gravity is applied to distribution functions. Eventually the pressure is large enough to cause new fluid cells to be created, but also large enough to prevent upward facing interface cells from being converted to empty cells. Visually, this results in small droplets sometimes accelerating slower than gravitational acceleration and gaining volume as

they descend. Our implementation partially mitigates this issue by encouraging interface cells that are emptying to transition to empty cells and those that are filling to transition to fluid cells (see Section 3.3.1 for more details). The effect is that fewer small droplets of water get separated from larger droplets when there are splashing effects. However, this does not solve all cases. Workarounds exist, such as removing droplets that are too small and redistributing the mass throughout the simulation [82], but these techniques require global memory operations that are expensive for a GPU implementation. Investigation into their applicability is also left as future work.

The problem of free surface contact with obstacles is similar to the one mentioned above. Interface cells neighbouring an obstacle will not receive any incoming mass from their neighbouring obstacle cells, which affects the rate at which mass enters and leaves these cells. Ordinarily, the inflow of mass from neighbouring fluid cells will drive fluid cell conversions. However, when an interface cell is pressed against an obstacle, there are fewer neighbouring fluid cells to drive this conversion. Therefore, some interface cells get stuck as interface cells even though they may be at the bottom of a pool of fluid. Visually, this manifests itself as speckles on the fluid obstacle boundary when obstacles are not rendered. These artefacts can be clearly seen in Figure 7.2. Fortunately, this is not a major issue as rendering the fluid transparently makes the artefacts less visible, especially when combined with mesh smoothing. Furthermore, rendering the obstacles in the scene hides these artefacts completely.

Lastly, our implementation of surface tracking only approximates the effect of air pressure on the liquid surface. A constant pressure is assumed to be exerted on the liquid surface by the air, instead of a dynamic pressure derived from a full fluid simulation of air flow. When bubbles form within the liquid, the constant pressure means that when liquid flows into the space occupied by the bubble, there is no corresponding increase in air pressure to push the liquid away. Therefore bubbles in our fluid simulation can be too easily absorbed into the fluid. This is obviously physically incorrect, and techniques exist to solve this issue [82]. Since these techniques require extra data to be stored and processed, solving this issue is left as future work.

7.5 Concluding Remarks

In this chapter we used three test cases to frame a discussion of the performance of our GPU implementation: *dam break*, *drop into pond* and *flow around corner*.

For each test case we measured the speedup of our GPU LBM implementation when compared to our single-core CPU LBM implementation. We showed that a higher percentage of fluid interface cells resulted in lower speedup, because the algorithm for managing the flow of the fluid interface is not as well suited to GPU execution as the algorithms for the stream and collide steps. Scene composition was also shown to play an important role, as scenes with more empty and obstacle cells were relatively less efficient on the GPU as a result of our simple GPU

streaming implementation. Overall, we measured speed-ups of up to 81.6× for scenes with lattice sizes suited to GPU execution, and were unable to determine maximum achievable speedup due to the memory limitations of our graphics card, which prevented us from testing larger simulations. However, we can infer from our speedup results that larger lattice sizes will only lead to minor increases.

We also investigated the geometric differences between our CPU and GPU implementations. No significant deviations were measured between the core fluid shape, but areas of high detail were seen to be slightly more pronounced in the GPU version. Since these differences were minor and our simulation used floats instead of doubles, they were put down the cumulative effect of floating point differences between the CPU and GPU implementations as a result of different orders for some calculations.

Lastly, we discussed the limitations of our current simulation: the fluid has a maximum velocity for a given lattice, the fluid surface tracking is unable to model details that are smaller than a few lattice cells in size, and bubbles inside the fluid are not handled correctly. Solutions exist to mitigate these limitations, but their application to GPU implementations is left as future work.

Chapter 8

Conclusions

High quality fluid simulations require significant computation to model highly detailed fluid behaviour. Since GPUs have potential for inexpensive parallel computation, we have investigated their applicability to fluid simulation, specifically with regards to LBM liquid simulations.

In this chapter, we show that we have achieved our original objectives as defined in Chapter 1. We first summarise our efficient design for a free-surface LBM simulation using CUDA. We then present the key results from our detailed analysis of the performance of the CUDA stream and collide steps. Lastly, we present the key results from our comparative analysis of the GPU and single-core CPU implementations. We conclude with some suggestions for future work.

8.1 Free-surface liquid simulation on GPUs

This thesis presents the design for a CUDA GPU implementation of a free-surface LBM liquid simulation. The LBM has two key stages: a stream step (to simulate the flow of particles through the lattice) and a collide step (to simulate particle interactions). We include a third *interface movement* step to track changes to the fluid's free-surface. Our LBM implementation uses the D3Q19 model for distribution functions, the LBGK collision operator, no-slip obstacle boundary conditions and the VOF method for fluid surface tracking. Our GPU implementation comprises 13 separate CUDA kernels, each with specific responsibilities:

- Five kernels focus on maximizing the use of GPU memory bandwidth for the distribution function transfers required during the stream step.
- Three kernels focus on performing adjustments in the stream step to handle fluid inflows, fluid interaction with stationary obstacles and mass transfers at the fluid interface.
- One kernel performs the collide step of the LBM, while maximizing memory and compute throughput.

- Four kernels manage the movement of the fluid interface using the VOF method for fluid surface tracking.

The execution of these kernels is managed by a separate CPU thread, which takes advantage of simultaneous device-host memory transfer and kernel execution to reduce the wait for data transfers to complete. Geometry extraction and visualization are included in our implementation, but are not the focus of this research.

8.2 Efficient stream and collide GPU kernels

We performed a detailed analysis of our kernels responsible for the stream and collide steps of the LBM. For the streaming kernels, we measured only memory throughput, because the kernels do not perform any significant calculations. The streaming kernels are highly efficient, using 95–98.5% of the practically available GPU memory bandwidth (79.9–82.5% of theoretical memory throughput) and achieving a speedup of up to 29.6× when compared to a single-core CPU implementation. The collide kernel also makes efficient use of GPU resources. It achieves up to 81.4% of theoretical GPU memory throughput when ignoring collision calculations, but still performing memory transfers. The collide kernel also makes effective use of latency hiding to significantly reduce the cost of the collision calculations, with up to 70.7% of the collision calculations taking place while the GPU waits for global memory operations to complete. Our comparison against a single-core CPU collide implementation displayed speedups of up to 223.7×.

Our interface movement and streaming adjustment kernels are comparatively inefficient. This is because the shape of obstacles and the fluid surface do not normally allow coalesced memory accesses on the GPU, as only a few threads per GPU warp usually intersect with obstacles or the fluid interface.

In the context of a fluid simulation, streaming takes the most time (35.9% of total GPU execution time), followed by the streaming adjustments and interface movement kernels (30% and 22.7% of total GPU execution time, respectively) and then the collide step (11.4% of total GPU execution time). The poor performance of the streaming adjustment and interface movement kernels, despite only acting on thin obstacle and fluid interface layers, results from their inability to make efficient use of GPU resources (particularly memory bandwidth) because their actions rely on the shape of obstacles and the fluid interface. The collide step is remarkably efficient, particularly considering that this kernel performs a significant portion of the LBM's calculations, and updates almost all the LBM's data structures (all distribution functions, the pressure, the velocity and fill fraction).

Comparison of the geometry from our GPU and CPU simulations for each test showed negligible differences because of minor differences in floating point values resulting from a slightly different order of operations between the CPU and GPU implementations. Areas of high tur-

bulence and surface detail had more pronounced differences, but the core shape of the fluid remained the same.

8.3 Significant performance improvements from GPU execution

GPU LBM fluid simulation shows at least an order of magnitude speedup against our single-core CPU simulation. We achieve speedups of up to 58.5× for our *dam break* test case, 81.6× for our *drop into pond* test case and 51.9× for our *flow around corner* test case. We attribute the difference in speedup profiles to differences in the types of cells in each simulation since the test cases with more interface, obstacle and empty cells relative to the number of fluid cells have lower speedups. We conclude that the decreased performance of our *dam break* and *flow around corner* test cases follows from the less efficient interface movement and streaming adjustment kernels processing relatively more lattice cells due a higher ratio of interface cells relative to fluid cells than present in the *drop into pond* test case.

The increase in speedup due to the increase in problem size has some deviations from a smooth logarithmic increase. These deviations only occur for lattice sizes where the width is a multiple of GPU warp size, resulting in fewer idle warps.

8.4 Future Work

We have shown that free-surface LBM fluid simulations benefit significantly from GPU execution. However, modern fluid simulations for visual effects include a range of features that we have not implemented, such as moving obstacles, better surface tension models and fluid control. In order to interact with a GPU simulation, these features need access to the simulation data on the GPU. Since host-device memory transfers are expensive GPU operations, accessing simulation data directly and processing it using the CPU during each simulation time step would be too time consuming to make such an approach viable. Therefore, it would be worthwhile to investigate the viability of GPU implementations of these features.

Adaptive time steps [82] and multi-resolution lattices [22] are two significant techniques that can be applied to the LBM to improve simulation performance. Adaptive time steps allow the simulation to take larger time steps, by only reducing the size of the time step when the speed of the fluid requires it. This helps to improve stability by preventing the fluid from moving faster than the lattice speed of sound, but also reduces simulation time by allowing larger time steps to be used. Multi-resolution lattices divide the simulation lattice according the amount of detail required at chosen locations. Areas requiring detail are represented by high-resolution sub-lattices, and areas that do not require detail use low-resolution sub-lattices. As a result, simulations are required to process less data, but can still maintain high levels of detail where it is considered

important. It is worth investigating implementations of both these techniques on GPUs because they could further reduce simulation times of a GPU LBM fluid simulation.

Bibliography

[1] Jeffrey Jacob Abrams, Executive Producer. *Lost*. Television. Buena Vista Television, 2004.

[2] G. Amati, S. Succi, and R. Piva. "Massively parallel lattice-Boltzmann simulation of turbulent channel flow". In: *International Journal of Modern Physics C* 8.4 (1997), pp. 869–877.

[3] Giovanni Astarita and Giuseppe Marrucci. *Principles of non-Newtonian fluid mechanics*. Vol. 28. McGraw-Hill New York, 1974.

[4] Autodesk. *Maya*. 2014. URL: http://www.autodesk.com/products/autodesk-maya/overview (visited on 02/10/2014).

[5] G.K. Batchelor. *An introduction to fluid dynamics*. Cambridge Univ Press, 2001. ISBN: 0 521 66396 2.

[6] Michael Bauer, Henry Cook, and Brucek Khailany. "CudaDMA: optimizing GPU memory bandwidth via warp specialization". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2011, p. 12.

[7] Nathan Bell and Michael Garland. "Implementing sparse matrix-vector multiplication on throughput-oriented processors". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC '09. Portland, Oregon: ACM, 2009, 18:1–18:11. ISBN: 978-1-60558-744-8. DOI: 10.1145/1654059.1654078.

[8] Paul Bourke. *Polygonising a scalar field*. 1194. URL: http://paulbourke.net/geometry/polygonise/ (visited on 02/15/2014).

[9] Robert Bridon. *Fluid Simulation for Computer Graphics*. CRC Press, Sept. 2008.

[10] Ravi Budruk, Don Anderson, and Tom Shanley. *PCI Express System Architecture*. Addison Wesley, Apr. 2008. ISBN: 0-321-15630-7.

[11] Shiyi Chen and Gary D. Doolen. "Lattice Boltzmann Method for Fluid Flows". In: *Annual Review of Fluid Mechanics* 30 (Jan. 1998), pp. 329–364. DOI: 10.1146/annurev.fluid.30.1.329.

[12] S. Chen et al. "A lattice gas model with temperature". In: *Physica D: Nonlinear Phenomena* 37.1-3 (1989), pp. 42–59. ISSN: 0167-2789. DOI: 10.1016/0167-2789(89)90116-4.

[13] Paolo Cignoni, Claudio Rocchini, and Roberto Scopigno. "Metro: measuring error on simplified surfaces". In: *Computer Graphics Forum*. Vol. 17. 2. Wiley Online Library. 1998, pp. 167–174.

[14] Jonathan M. Cohen, Sarah Tariq, and Simon Green. "Interactive fluid-particle simulation using translating Eulerian grids". In: *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. I3D '10. Washington, D.C.: ACM, 2010, pp. 15–22. ISBN: 978-1-60558-939-8. DOI: 10.1145/1730804.1730807.

[15] NVIDIA Corporation. *CUDA Best Practices Guide, version 3.2*. Sept. 2010. URL: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf (visited on 03/02/2011).

[16] NVIDIA Corporation. *CUDA Programming Guide, version 3.2*. Sept. 2010. URL: http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf (visited on 03/02/2011).

[17] NVIDIA Corporation. *CUDA Programming Guide, version 6.0*. Feb. 2014. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (visited on 05/26/2014).

[18] NVIDIA Corporation. *GeForce GTX TITAN Black: Specifications*. 2014. URL: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-black/specifications (visited on 05/26/2014).

[19] NVIDIA Corporation. *NVIDIA GeForce GTX 680*. 2012. URL: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf (visited on 11/27/2012).

[20] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". In: *Computational Science Engineering, IEEE* 5.1 (1998), pp. 46–55. ISSN: 1070-9924. DOI: 10.1109/99.660313.

[21] Dominique d'Humières. "Multiple–relaxation–time lattice Boltzmann models in three dimensions". In: *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 360.1792 (2002), pp. 437–451.

[22] Alexandre Dupuis and Bastien Chopard. "Theory and applications of an alternative lattice Boltzmann grid refinement algorithm". In: *Phys. Rev. E* 67 (6 June 2003), p. 066707. DOI: 10.1103/PhysRevE.67.066707.

[23] Jianbin Fang, A.L. Varbanescu, and H. Sips. "A Comprehensive Performance Comparison of CUDA and OpenCL". In: *Parallel Processing (ICPP), 2011 International Conference on*. Sept. 2011, pp. 216–225. DOI: 10.1109/ICPP.2011.45.

[24] Blender Foundation. *Blender*. 2014. URL: http://www.blender.org (visited on 02/10/2014).

[25] Prashant Goswami et al. “Interactive SPH simulation and rendering on the GPU”. In: *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA ’10. Madrid, Spain: Eurographics Association, 2010, pp. 55–64. URL: http://dl.acm.org/citation.cfm?id=1921427.1921437.

[26] Naga K. Govindaraju et al. “High performance discrete Fourier transforms on graphics processors”. In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. SC ’08. Austin, Texas: IEEE Press, 2008, 2:1–2:12. ISBN: 978-1-4244-2835-9. URL: http://dl.acm.org/citation.cfm?id=1413370.1413373.

[27] Denis Gueyffier et al. “Volume-of-Fluid Interface Tracking with Smoothed Surface Stress Methods for Three-Dimensional Flows”. In: *Journal of Computational Physics* 152.2 (1999), pp. 423–456. ISSN: 0021-9991. DOI: 10.1006/jcph.1998.6168.

[28] Trond Hagen, Knut-Andreas Lie, and Jostein Natvig. “Solving the Euler Equations on Graphics Processing Units”. In: *Computational Science – ICCS 2006*. Ed. by Vassil Alexandrov et al. Vol. 3994. Lecture Notes in Computer Science. 10.1007/11758549_34. Springer Berlin / Heidelberg, 2006, pp. 220–227. ISBN: 978-3-540-34385-1. DOI: 10.1007/11758549_34.

[29] Tsuyoshi Hamada et al. “42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence”. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC ’09. Portland, Oregon: ACM, 2009, 62:1–62:12. ISBN: 978-1-60558-744-8. DOI: 10.1145/1654059.1654123.

[30] F.H. Harlow, J.E. Welch, et al. “Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface”. In: *Physics of fluids* 8.12 (1965), p. 2182. DOI: 10.1063/1.1761178.

[31] Timothy D.R. Hartley et al. “Biomedical image analysis on a cooperative cluster of GPUs and multicores”. In: *Proceedings of the 22nd annual international conference on Supercomputing*. ICS ’08. Island of Kos, Greece: ACM, 2008, pp. 15–25. ISBN: 978-1-60558-158-3. DOI: 10.1145/1375527.1375533.

[32] A. Hérault, G. Bilotta, and R.A. Dalrymple. “SPH on GPU with CUDA”. In: *Journal of Hydraulic Research* 48.S1 (2010), pp. 74–79.

[33] Damien Hinsinger, Fabrice Neyret, and Marie-Paule Cani. “Interactive animation of ocean waves”. In: *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*. SCA ’02. San Antonio, Texas: ACM, 2002, pp. 161–166. ISBN: 1-58113-573-4. DOI: 10.1145/545261.545288.

[34] C.W Hirt and B.D Nichols. “Volume of fluid (VOF) method for the dynamics of free boundaries”. In: *Journal of Computational Physics* 39.1 (1981), pp. 201–225. ISSN: 0021-9991. DOI: 10.1016/0021-9991(81)90145-5.

[35] S. Hou et al. “A lattice Boltzmann subgrid model for high Reynolds number flows”. In: *Pattern formation and lattice gas automata* 6 (1996), p. 149.

[36] Raouf A Ibrahim. *Liquid sloshing dynamics: theory and applications*. Cambridge University Press, 2005.

[37] Mark Steven Johnson, Director. *Ghost Rider*. Film. Columbia Pictures, 2007.

[38] Drona Kandhai. “Large Scale Lattice-Boltzmann Simulation: Computational Methods and Applications”. PhD thesis. Amsterdam, The Netherlands: University of Amsterdam, 1999. ISBN: 9057760371.

[39] D. Kandhai et al. “Lattice-Boltzmann hydrodynamics on parallel systems”. In: *Computer Physics Communications* 111.1-3 (1998), pp. 14–26. ISSN: 0010-4655. DOI: 10.1016/S0010-4655(98)00025-3.

[40] Michael Kass and Gavin Miller. “Rapid, stable fluid dynamics for computer graphics”. In: *SIGGRAPH Comput. Graph.* 24 (4 Sept. 1990), pp. 49–57. ISSN: 0097-8930. DOI: 10.1145/97880.97884.

[41] R. Keiser et al. “A unified Lagrangian approach to solid-fluid animation”. In: *Point-Based Graphics, 2005. Eurographics/IEEE VGTC Symposium Proceedings*. June 2005, pp. 125–148. DOI: 10.1109/PBG.2005.194073.

[42] Peter Kipfer, Mark Segal, and Rüdiger Westermann. “UberFlow: a GPU-based particle engine”. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. HWWS ’04. Grenoble, France: ACM, 2004, pp. 115–122. ISBN: 3-905673-15-0. DOI: 10.1145/1058129.1058146.

[43] A. Kolb, L. Latta, and C. Rezk-Salama. “Hardware-based simulation and collision detection for large particle systems”. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. HWWS ’04. Grenoble, France: ACM, 2004, pp. 123–131. ISBN: 3-905673-15-0. DOI: 10.1145/1058129.1058147.

[44] Victor W. Lee et al. “Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU”. In: *SIGARCH Comput. Archit. News* 38.3 (June 2010), pp. 451–460. ISSN: 0163-5964. DOI: 10.1145/1816038.1816021.

[45] Wei Li, Xiaoming Wei, and Arie Kaufman. “Implementing lattice Boltzmann computation on graphics hardware”. In: *The Visual Computer* 19 (7 2003). 10.1007/s00371-003-0210-6, pp. 444–456. ISSN: 0178-2789. URL: http://dx.doi.org/10.1007/s00371-003-0210-6.

[46] Wei Li et al. “GPU-Based Flow Simluation with Complex Boundaries”. In: *CPU Gems 2*. Ed. by M. Pharr. Addison Wesley, Mar. 2005. Chap. 47, pp. 747–764.

[47] William E. Lorensen and Harvey E. Cline. "Marching cubes: A high resolution 3D surface construction algorithm". In: *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1987, pp. 163–169. ISBN: 0-89791-227-6. DOI: 10.1145/37401.37422.

[48] LuxRender. *LuxRender: GPL Physically Based Renderer*. 2014. URL: http://www.luxrender.net (visited on 02/10/2014).

[49] Renwei Mei et al. "Lattice Boltzmann method for 3-D flows with curved boundary". In: *J. Comput. Phys.* 161.2 (2000), pp. 680–699. ISSN: 0021-9991. DOI: 10.1006/jcph.2000.6522.

[50] Markus Meier, George Yadigaroglu, and Brian L Smith. "A novel technique for including surface tension in PLIC-VOF methods". In: *European Journal of Mechanics - B/Fluids* 21.1 (2002), pp. 61–73. ISSN: 0997-7546. DOI: 10.1016/S0997-7546(01)01161-X.

[51] *Meshlab*. Software developed with the support of 3D-CoForm. Aug. 2012. URL: http://meshlab.sourceforge.net (visited on 12/02/2013).

[52] J J Monaghan. "Smoothed Particle Hydrodynamics". In: *Annual Review of Astronomy and Astrophysics* 30.1 (1992), pp. 543–574. DOI: 10.1146/annurev.aa.30.090192.002551. eprint: http://www.annualreviews.org/doi/pdf/10.1146/annurev.aa.30.090192.002551.

[53] J.J. Monaghan. "Simulating Free Surface Flows with SPH". In: *Journal of Computational Physics* 110.2 (1994), pp. 399–406. ISSN: 0021-9991. DOI: 10.1006/jcph.1994.1034.

[54] Andreas Monitzer. "Fluid simulation on the GPU with complex obstacles using the lattice Boltzmann method". MA thesis. Vienna: Vienna University of Technology, 2008.

[55] Matthias Müller. "Fast and robust tracking of fluid surfaces". In: *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA '09. New Orleans, Louisiana: ACM, 2009, pp. 237–245. ISBN: 978-1-60558-610-6. DOI: 10.1145/1599470.1599501.

[56] Matthias Müller, David Charypar, and Markus Gross. "Particle-based fluid simulation for interactive applications". In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. SCA '03. San Diego, California: Eurographics Association, 2003, pp. 154–159. ISBN: 1-58113-659-5. URL: http://dl.acm.org/citation.cfm?id=846276.846298.

[57] DR Noble and JR Torczynski. "A lattice-Boltzmann method for partially saturated computational cells". In: *International Journal of Modern Physics C-Physics and Computer* 9.8 (1998), pp. 1189–1202. ISSN: 0129-1831.

[58] Christian Obrecht et al. "A new approach to the lattice Boltzmann method for graphics processing units". In: *Computers & Mathematics with Applications* 61.12 (2011), pp. 3628–3638. ISSN: 0898-1221. DOI: 10.1016/j.camwa.2010.01.054.

[59] S. Osher and R.P. Fedkiw. *Level set methods and dynamic implicit surfaces*. Vol. 153. Springer Verlag, New York, 2003.

[60] Stanley Osher and Ronald P. Fedkiw. "Level Set Methods: An Overview and Some Recent Results". In: *Journal of Computational Physics* 169.2 (2001), pp. 463–502. ISSN: 0021-9991. DOI: 10.1006/jcph.2000.6636.

[61] Judy Pearsall et al., eds. *Oxford Dictionary of English*. 2nd ed. Oxford, United Kingdom: Oxford University Press, 2009.

[62] Mark S. Peercy et al. "Interactive multi-pass programmable shading". In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 425–432. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344976.

[63] Liu Peng et al. "Parallel Lattice Boltzmann Flow Simulation on Emerging Multi-core Platforms". In: *Euro-Par 2008 — Parallel Processing*. Ed. by Emilio Luque, Tomàs Margalef, and Domingo Benítez. Vol. 5168. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 763–777. DOI: 10.1007/978-3-540-85451-7_81.

[64] Wolfgang Petersen, Director. *Poseidon*. Film. Warner Bros. Pictures, 2006.

[65] Thomas Pohl et al. "Optimization and Profiling of the Cache Performance of Parallel Lattice Boltzmann Codes in 2D and 3D". In: 13 (4 Dec. 2003), pp. 549–560. DOI: 10.1142/S0129626403001501.

[66] Y. H. Qian, D. D'Humières, and P. Lallemand. "Lattice BGK models for Navier-Stokes equation". In: *Europhysics Letters* 17 (Feb. 1992), pp. 479–484. DOI: 10.1209/0295-5075/17/6/001.

[67] RealFlow™. *RealFlow in TV and Commercials*. 2014. URL: http://www.realflow.com/videos/tv/ (visited on 01/02/2014).

[68] RealFlow™. *RealFlow in TV and Commercials*. 2014. URL: http://www.realflow.com/ (visited on 01/02/2014).

[69] Ashley Reid. "Parallel Fluid Dynamics For the Animation Industry". MA thesis. Cape Town, South Africa: University of Cape Town, May 2009.

[70] Yuriko Renardy and Michael Renardy. "PROST: A Parabolic Reconstruction of Surface Tension for the Volume-of-Fluid Method". In: *Journal of Computational Physics* 183.2 (2002), pp. 400–421. ISSN: 0021-9991. DOI: 10.1006/jcph.2002.7190.

[71] Barbara Robertson. *Burn Cycle*. 2007. URL: http://www.cgsociety.org/index.php/CGSFeatures/CGSFeatureSpecial/burn_cycle (visited on 02/06/2014).

[72] Mike Seymour. *Wipe out: 'Poseidon' Fluid Simulations*. 2006. URL: http://www.fxguide.com/featured/wipe_out_poseidon_fluid_simulations/ (visited on 02/06/2014).

[73] N. Singla et al. "Financial Monte Carlo simulation on architecturally diverse systems". In: *High Performance Computational Finance, 2008. WHPCF 2008. Workshop on*. Nov. 2008, pp. 1–7. DOI: 10.1109/WHPCF.2008.4745401.

[74] P.A. Skordos. "Parallel simulation of subsonic fluid dynamics on a cluster of workstations". In: *High Performance Distributed Computing, 1995., Proceedings of the Fourth IEEE International Symposium on*. Aug. 1995, pp. 6–16. DOI: 10.1109/HPDC.1995.518689.

[75] Joseph Smagorinsky. "GENERAL CIRCULATION EXPERIMENTS WITH THE PRIMITIVE EQUATIONS: I. THE BASIC EXPERIMENT*". In: *Monthly weather review* 91.3 (1963), pp. 99–164.

[76] Side Effects Software. *Houdini: 3D Animation Tools*. 2014. URL: http://www.sidefx.com (visited on 02/10/2014).

[77] S. Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Oxford University Press, USA, 2001. ISBN: 0198503989.

[78] Mark Sussman. "A second order coupled level set and volume-of-fluid method for computing growth and collapse of vapor bubbles". In: *Journal of Computational Physics* 187.1 (2003), pp. 110–136. ISSN: 0021-9991. DOI: 10.1016/S0021-9991(03)00087-1.

[79] Mark Sussman and Elbridge Gerry Puckett. "A Coupled Level Set and Volume-of-Fluid Method for Computing 3D and Axisymmetric Incompressible Two-Phase Flows". In: *Journal of Computational Physics* 162.2 (2000), pp. 301–337. ISSN: 0021-9991. DOI: 10.1006/jcph.2000.6537.

[80] N. Thürey and U. Rüde. "Free surface lattice-Boltzmann fluid simulations with and without level sets". In: *Workshop on vision, modelling, and visualization (VMV Stanford)*. 2004, pp. 199–208.

[81] Nils Thürey. "A single-phase free-surface Lattice Boltzmann Method". MA thesis. Universität Erlangen-Nürnberg, Jan. 2005.

[82] Nils Thürey. "Physically based Animation of Free Surface Flows with the Lattice Boltzmann Method". PhD thesis. Universität Erlangen-Nürnberg, Mar. 2007.

[83] Nils Thürey, T Pohl, and U. Rüde. "Hybrid Parallelization Techniques for Lattice Boltzmann Free Surface Flows". In: *Parallel Computational Fluid Dynamics 2007*. Ed. by Timothy J. Barth et al. Vol. 67. Lecture Notes in Computational Science and Engineering. 10.1007/978-3-540-92744-0_22. Springer Berlin / Heidelberg, 2009, pp. 179–186. ISBN: 978-3-540-92744-0. DOI: 10.1007/978-3-540-92744-0_22.

[84] Nils Thürey, Ulrich Rüde, and Marc Stamminger. "Animation of open water phenomena with coupled shallow water and free surface simulations". In: *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*. SCA '06. Vienna, Austria: Eurographics Association, 2006, pp. 157–164. ISBN: 3-905673-34-7. URL: http://dl.acm.org/citation.cfm?id=1218064.1218086.

[85] Jonas Tölke. "Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA". In: *Computing and Visualization in Science* 13 (1 2010). 10.1007/s00791-008-0120-2, pp. 29–39. ISSN: 1432-9360. DOI: 10.1007/s00791-008-0120-2.

[86] Cornelis Boudewijn Vreugdenhil. *Numerical Methods for Shallow-Water Flow*. Vol. 13. Water Science and Technology Library. Kluwer Academic Publishers, 1995. ISBN: 978-94-015-8354-1.

[87] Z. Wang, J. Yang, and F. Stern. "Comparison of Particle Level Set and CLSVOF Methods for Interfacial Flows". In: *46th AIAA Aerospace Sciences Meeting and Exhibit*. Vol. 3. American Institute of Aeronautics and Astronautics, 1801 Alexander Bell Drive, Suite 500, Reston, VA, 20191-4344, USA, 2008.

[88] Xiaoming Wei et al. "Blowing in the wind". In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. SCA '03. San Diego, California: Eurographics Association, 2003, pp. 75–85. ISBN: 1-58113-659-5. URL: http://dl.acm.org/citation.cfm?id=846276.846287.

[89] Xiaoming Wei et al. "The Lattice-Boltzmann Method for Simulating Gaseous Phenomena". In: *Visualization and Computer Graphics, IEEE Transactions on* 10.2 (Mar. 2004), pp. 164–176. ISSN: 1077-2626. DOI: 10.1109/TVCG.2004.1260768.

[90] X. Wei et al. "Simulating fire with texture splats". In: *Visualization, 2002. VIS 2002. IEEE*. Nov. 2002, pp. 227–234. DOI: 10.1109/VISUAL.2002.1183779.

[91] Wikia. *The Man in Black*. 2014. URL: http://lostpedia.wikia.com/wiki/The_Man_in_Black (visited on 02/06/2014).

[92] Dieter A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models - An Introduction*. Vol. 1725. Lecture Notes in Mathematics. Springer Berlin / Heidelberg, June 2005.

[93] Shucai Xiao and Wu-chun Feng. “Inter-block GPU communication via fast barrier synchronization”. In: *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. Apr. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470477.

[94] He Yan et al. “Real-time fluid simulation with adaptive SPH”. In: *Computer Animation and Virtual Worlds* 20.2-3 (2009), pp. 417–426. ISSN: 1546-427X. DOI: 10.1002/cav.300.

[95] Ye Zhao et al. “Flow Simulation with Locally-refined LBM”. In: *I3D ’07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*. Seattle, Washington: ACM, 2007, pp. 181–188. ISBN: 978-1-59593-628-8. DOI: 10.1145/1230100.1230132.