FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

GPU implementation of Free Surface Lattice Boltzmann code

Wang Zhichao

Studienarbeit

GPU implementation of Free Surface Lattice Boltzmann code

Wang Zhichao

Studienarbeit

Aufgabensteller:	Prof. Dr. U. Rüde
Betreuer:	Dr.-Ing. Stefan Donath
	Dr.-Ing. Harald Köstler
Bearbeitungszeitraum:	15.11.2010 – 1.12.2011

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 12. Dezember 2011 ....

Abstract

Free Surface Lattice Boltzmann, the simulation of fluids with free surfaces algorithms is important for a variety of technical fields. In order to efficiently compute such simulations, there is complex software framework waLBerla. Within the computation sequence, a step called PDF SWEEP is the most expensive one in terms of computational complexity. Nowadays graphics accelerator boards are playing a important role in computer science, not only for the graphical data processing also for the high-performance computing. The graphics cards from Nvidia support the Common Unified Device Architecture (CUDA) which makes porting of existing C code by the use of a special library possible. This thesis focuses on the analyzing, transforming and mapping of parts of the free surface lattice Boltzmann code to GPU with the help of CUDA. With the implementations of CUDA the overall performance of the free surface lattice Boltzmann implementation has been improved. However, the gain is marginal since the transforming of collision and streaming parts in PDF SWEEP leads to additional time cost. This thesis presents results and problems and discusses advantages and disadvantages of those implementations.

Contents

Abstract v

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Contributions . . . 2
- 1.3 Thesis Outline . . . 2

2 Introduction to Free Surface Lattice Boltzmann 3
- 2.1 Free Surface . . . 3
- 2.2 Stream and Collide . . . 3
- 2.3 Mass Exchange . . . 6
- 2.4 Reconstruction of Missing Distribution Funtions . . . 8

3 Implementation 11
- 3.1 The waLBerla Project . . . 11
- 3.2 Routines Analysis . . . 11
- 3.3 Reference Implementation Transformation . . . 14
- 3.4 GPU Implementation with CUDA . . . 17
  - 3.4.1 CUDA Introduction . . . 18
  - 3.4.2 CUDA Implementation . . . 21

4 Results Analysis 27
- 4.1 Simulation Result Analysis . . . 27
- 4.2 SWEEPS parts Analysis . . . 28
- 4.3 PDF SWEEP part Analysis . . . 28

5 Conclusion 33
- 5.1 Summary . . . 33
- 5.2 The future works . . . 34

A Abbreviations 35

List of Figures 37

List of Tables 39

List of Algorithms 41

1 Introduction

1.1 Motivation

In the Lattice Boltzmann Method (LBM) the fluid is represented as cellular automata which describes the fluid particles that move according to their velocity. The implementation of cellular automata in a computer program is usually straightforward. The benefits of LBM are that the simulation does not only simplify the implementation, but it also makes the application of optimization and parallelization techniques known from other numerical algorithms possible, resulting in high-performance implementations [KPR+05]The free surface lattice Boltzmann in project waLBerla is developed at LSS and used for simulation of bubbles and drops that were demonstrated for the 3D models where the involved interface cells are partially filled between gas cells and fluid cells distinctively and coupled with boundary conditions for the free surface. Certainly the much more computing to the extension in free surface lattice Boltzmann fluid simulations is needed additionally which involve much more simulation time with limited CPUs resources. To achieve high performance from the basis of the original project a new technical change or solution is necessary.

Fortunately a new method has been found to solve this heavy computational problem - GPGPU (General Purpose Graphics Processing Units). Because of much faster increase in the performance of GPUs than of CPUs, the graphics accelerator board is playing an increasingly important role in computer science, not only for the graphical data processing also for the high-performance computing. The features of modern graphics card are the multi-core processors and large memories. These GPUs have much more powerful ability to solve the massive data processing problem so that they support their application in the fields of the high-performance computing. Nvidia provides special GPUs for general purpose and suitable programming models. In November 2006 Nvidia published the Common Unified Device Architecture (CUDA), an SDK (Software Development Kit) and API (Application Programming Interface) that provide the C programming language on Geforce 8 series GPUs. The most important feature of GPUs is that GPU cannot only process independent data, but can process all of them in parallel. If in a program there is massive computing, porting code to the GPU is one of the best solutions to efficient speedups. With the GPU data processing ability more and more researchers focus on GPUs to solve their massive computational needs. The usage of GPU computation has been studied for a long time and GPU computing applications have extended in many different fields, for example physical based simulation, signal and image processing, global illumination, geometric computing, databases and data mining [OJL+07]. The first studying of GPU computing was found in [Eng78]. The LBM is implemented efficiently and successfully on GPUs in [TK07] that LBM has been implemented on GPUs. Furthermore, the GPUs programming in the simulation of fluids with free surfaces has been the focus of many researchers for a long time.

The goal of this thesis is to determine if some parts of the code for free surface could be ported to CUDA on a NVidia GPU so that it is possible to get the higher performance than the original program. After that the results of the implementation will be analyzed and evaluated.

1.2 Contributions

In this thesis, the existing parts of free surface lattice Boltzmann code from waLBerla will be analyzed, and some portions of it will be ported to CUDA. First, the routines will be analyzed and transformed into the suitable structure which could be changed into CUDA codes later. And then that transformed part of the routines will be ported to CUDA in order to improve performance. Finally, the CUDA part will be tested to make sure if the efficient results were achieved from the ported CUDA code. The key contributions of this work can be summarized as follows:

- **Routine analysis**
  In free surface LBM of waLBerla there are many parts of computational procedure, which parts we should port on GPU is the first question of this thesis. The execution time from the parts of free surface LBM will be tested in order to determine which parts take the most execution time in the simulation.

- **Porting to CUDA on GPU**
  By considering the features of GPU we use some special methodologies to port the determined CPU implementation of the free surface LBM to CUDA so that it can efficiently run on GPU. The results will be tested and the advantage and disadvantage of the implementation will be discussed.

1.3 Thesis Outline

The outline of this thesis is organized as follows. In Chapter 2 some basic concepts of free surface LBM will be introduced to gain an understanding of how the free surface LBM works in the waLBerla project. Chapter 3 describes the implementation of the thesis and explains how the CPU code is analyzed and transformed for porting to CUDA and the implementation on CUDA. The optimizations of GPU code will be analyzed in Chapter 4 to determine whether these optimizations on GPU could improve performance in contrast to CPU code. Finally, Chapter 5 concludes the work.

2 Introduction to Free Surface Lattice Boltzmann

In this section some basic concepts of the free surface Lattice Boltzmann method will be introduced. For example, the differences between standard lattice Boltzmann and free surface lattice Boltzmann will be described.

2.1 Free Surface

The free surface is the surface of a fluid between gas and liquid. In free surface LBM the free surface is represented by another cell type interface which contains both liquid and gas. Interface cells are between gas and liquid cells and can transform into gas or liquid during the simulation procedure. See Fig. 2.1. Compared to the other cell types, interface cells have additional properties and require extra treatment. A fill level stores the liquid fraction of the volume. For the partial volume, an explicit mass exchange schema computes mass conservation. Gas cells are not taken into account for the flow computation and thus do not store any information except an index that links them similar to interface cells to a data structure that stores the pressure of the closed gas region, see Fig. 2.2.

In free surface LBM the model as 19 velocities is used for three dimensions (commonly as 3 dimension 19 direction (D3Q19) see Fig. 2.3), and this model is a good compromise between computational efficiency and numerical stability in comparison to other models, and that is also the best recommendation for parallelization because of the diagonal structure [DGF+07]. All the 19 directions of cells in the D3Q19 are stored with the distribution functions (DFs). Altogether the new interface cell type and the features of the free surface make up the free surface LBM model.

For further details on the LBM see [Suc01] and on the free surface extension can be found in [Thu03] and [Fei06].

2.2 Stream and Collide

In Fig. 2.3 we know the LBM usually works on an equidistant grid of cells and each of them stores a discrete number of velocities, DFs which represents a number of particles in the fluid that moves along the direction of its velocity direction [PY06]. For each time step t the DF $f_i(x,t)$ for position x is stored for each velocity direction ($i = 1..19$). From these values of DFs and direction vectors $\vec{e}_i$ (see below) the values for density ρ and velocity v can be calculated with the moments.

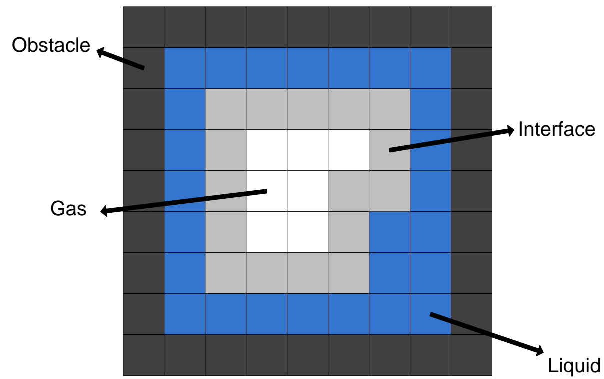

Figure 2.1: The types cells in free surface LBM

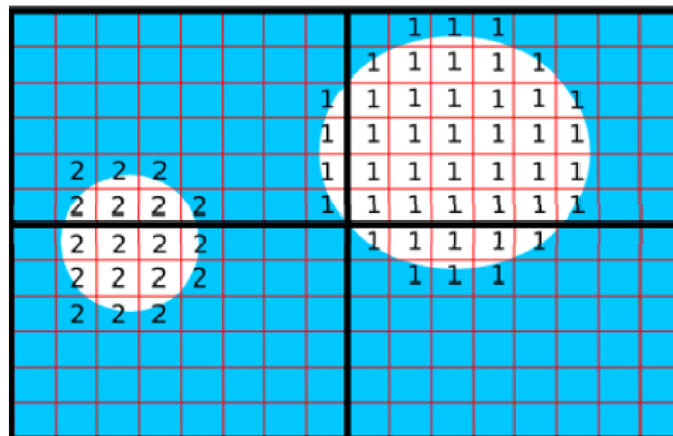

Figure 2.2: Data representation of a bubble in free-surface LBM: Gas and interface cells store an ID of the bubble[DFP+09]. Blue is liquid and white is gas

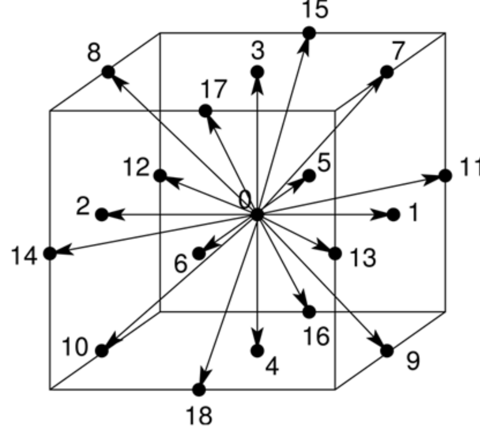

Figure 2.3: Structure of the lattice in D3Q19 model.[DGF+07]

$$\vec{e}_i = \begin{cases} (0,0,0), \ i = 0 \\ (\pm 1, 0, 0)c, (0, \pm 1, 0)c, (0, \pm 1, 0)c, \ i = 2,4,6,8,9,14 \\ (\pm 1, \pm 1, 0)c, (0, \pm 1, \pm 1)c, (\pm 1, 0, \pm 1)c, \ i = 1,3,5,7,10,11,12,13,15,16,17,18,19 \end{cases} \tag{2.1}$$

$$\rho = \sum_{i=1}^{19} f_i \tag{2.2}$$

$$v = \frac{1}{\rho} \sum_{i=1}^{19} f_i \vec{e}_i \tag{2.3}$$

The most important part of LBM is the two processing steps - (1) the streaming and (2) the collision depicted in Fig 2.4. During streaming, the particles are moved according to their corresponding velocities. The particles move to the neighboring cells along their velocity direction. After each streaming step, all cells have a complete set of particle distribution functions. The collision step accounts for the influence of the fluids viscosity on the particles movement. This is finished by the incoming DFs from the streaming step with a set of equilibrium distribution functions f_i^{eq} that can be calculated for each cell with its density and velocity [TR04]. The ω_i and f_i^{eq} are defined with the velocity vector:

$$\omega_i = \begin{cases} \frac{1}{3}, & i = 0 \\ \frac{1}{18}, & i = 2,4,6,8,9,14 \\ \frac{1}{36}, & i = 1,3,5,7,10,11,12,13,15,16,17,18 \end{cases} \tag{2.4}$$

if i= C,

$$f_i^{eq}(\rho, \vec{u}) = \omega_i \rho \left(1 - \frac{3}{2}\vec{u}^2\right); \omega_i = \frac{1}{3} \tag{2.5}$$

if i = N, S, E, W, T, B,

$$f_i^{eq}(\rho, \vec{u}) = \omega_i \rho \left(1 + 3\vec{u}^2 + \frac{9}{2}\vec{e}_i \vec{u}^2 - \frac{3}{2}\vec{u}^2\right); \omega_i = \frac{1}{18} \tag{2.6}$$

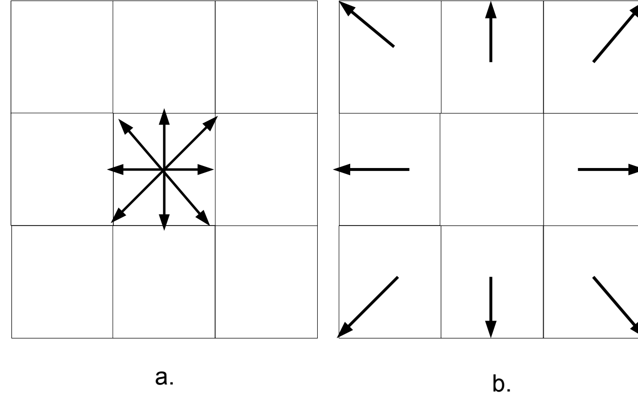

Figure 2.4: Streaming and Collision step. Left a is a cell at time t after collision and right b is the cell at next time step t+1 after streaming.

if i = NE, NW, SE, SW, ET, EB, WT, WB, NT, NB, ST, SB

$$f_i^{eq}(\rho, \vec{u}) = \omega_i \rho \left(1 + 3\vec{u}^2 + \frac{9}{2}\vec{e}_i\vec{u}^2 - \frac{3}{2}\vec{u}^2\right); \omega_i = \frac{1}{36} \tag{2.7}$$

f_i^{eq} fulfills a Maxwell-Boltzmann equilibrium distribution. According to the H-Theorem, the BGK model uses a single relaxation factor $\frac{1}{\tau}$ to relax the distribution towards equilibrium which is as below:

$$\tilde{f}_i(x,t) = f_i(x,t) - \frac{1}{\tau}\left[f_i(x,t) - f_i^{eq}(\rho, \vec{u})\right] \tag{2.8}$$

Above all we can split lattice Boltzmann equation into two parts and the movement of the collision and streaming is showed in Fig 2.4 :

$$f_i(x + e_i\delta t, t + \delta t) = \tilde{f}_i(x,t), \text{Streaming step} \tag{2.9}$$

$$\tilde{f}_i(x,t) = f_i(x,t) - \frac{1}{\tau}\left[f_i(x,t) - f_i^{eq}(\rho, \vec{u})\right], \text{Collision step} \tag{2.10}$$

The extension LBM for free surface involves the interface cell computation, and additional calculations of mass exchange and reconstruction of missing distribution functions must be computed during the streaming and collision steps for each time step if the located cell type is interface.

2.3 Mass Exchange

The movement of the fluid results in the conversion of cell types. Since different cell types have different state flags, the conversion leads to the changing of cell types and results in the changing of state flags which will be set again. The changes depend on the initial state and the conversion rule for the different cell types. The conversion of a cell does not only depend on the fluid fraction,

but also on the conversion of all neighboring cells [TRK05].Therefore, if an interface cells convert to a fluid (gas) cell, neighboring gas (fluid) cells have to be converted to interface cells see Fig. 2.6. And similarly if the interface cells become completely filled or completely empty the mass of zero or equal to the density, at the same time the type of interface cells will change into gas or fluid cells. See [Poh08].

During the conversion mass exchange takes place, from the Fig. 2.6 the fluid cell changes into interface cell if that cell's neighborhood changes to gas cells. At the same time the amount of the fluid mass will be reduced. Similarly the interface cell changes into another interface or fluid cells in next time the amount of the fluid mass will be reduced or increased, if the mass of the interface cells changes into less than 0, the interface will transform into gas cells. This mass changing is called mass exchange among all the fluid and interface cells in the simulation. See Fig. 2.5.

The mass of an interface cell is given by the mass exchanges of all neighboring cells. If its neighboring cells are gas, the mass exchange will be 0. And if neighborhood cells are fluid cells, the mass exchange will be computed as follow:

$$\Delta M_i(x,t) = \tilde{f}_i(x_{e_i},t) - f_i(x,t) \tag{2.11}$$

where $x_{e_i} = x + e_i$, and $m(x,t)$ shows the mass of the cell at position x at time t. $\epsilon(x,t)$ is the fraction of the cell that is filled with fluid. If its neighboring cells are interface:

$$\Delta M_i(x,t) = \frac{\epsilon(x_{e_i},t) + \epsilon(x,t)}{2}\left(\tilde{f}_i(x_{e_i},t) - f_i(x,t)\right) \tag{2.12}$$

For the next time step all directions of a interface cell are computed together:

$$M(x,t+1) = M(x,t) + \sum_{i=1}^{19} \Delta M_i(x,t) \tag{2.13}$$

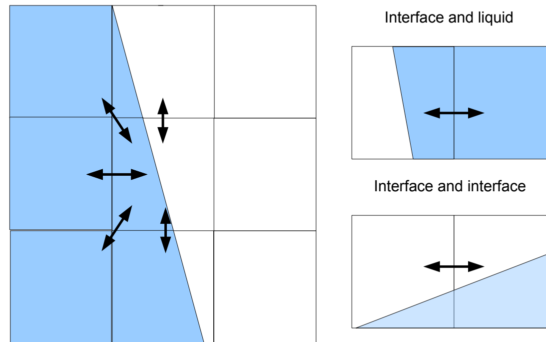

Figure 2.5: The mass exchange takes place among the fluid and interface cells

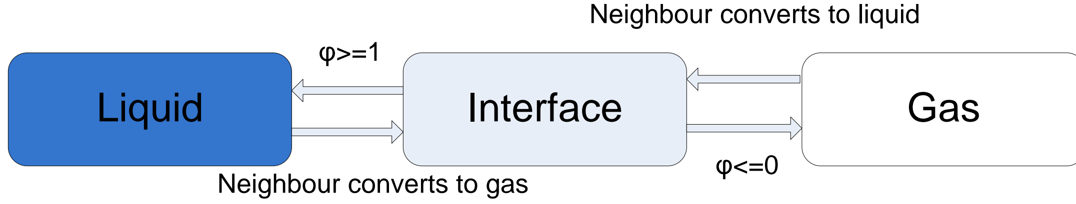

Figure 2.6: Cells type conversation. [Poh08]

2.4 Reconstruction of Missing Distribution Funtions

As described for the free surface there is a transformation between interface, fluid and gas cells. These fluid and gas cells that have neighboring interface cells can change into interface cell in next time step 2.6.

From the Fig. 2.1 the interface cells are between the gas and fluid cells. The gas cells have a special feature that they have no DFs. From LBM, after the streaming the cell must get all the DFs from all 19 directions of the all neighbors, but in free surface model the interface may have the gas neighbor cells with no DFs, therefore, they can get the DFs from interface and fluid neighborhood directly except gas cells. The DFs from gas cells must be reconstructed. In order to satisfy the free surface boundary conditions that the force from the liquid must balance the gas force and unknown incoming distribution functions must be defined [Don11]. All the unknown incoming distribution functions are denoted as $n \cdot e_i >= 0$. As the required force balance is needed for opposite lattice directions. Besides that, the forces exerted by the gas are known and are given by the gas pressure and the velocity at the interface. Above all, with the velocity of interface and the density of gas cells [KTH+05], the reconstructed DFs can be defined as:

$$f_i^{out}(x - e_i, t) = \left(f_i^{eq}(v, \rho^G) + \tilde{f}_i^{eq}(v, \rho^G) \right) - \tilde{f}_i^{out}(x, t), \quad where n \cdot e_i > 0 \tag{2.14}$$

Here $\tilde{f}_i^{out}$ is opposite to f_i^{out} and f_i^{out} is an unknown function that must be reconstructed. v and ρ^G are the velocity and density of the gas cells. The $\rho^G(t) = \frac{1}{c^2}p(t)$ is gas density and $v = v(x,t)$ is the velocity of the interface cell x [Don11]. It is important to ensure that not only the missing distribution functions from the gas cells are reconstructed but all distribution functions with $n \cdot e_i > 0$ that they are sometimes interface cells [KTH+05].

From all above, the resulting of reconstructed DFs can be calculated if the condition $n \cdot e_i >= 0$ is fulfilled. Also it is important that this free surface boundary condition is only fulfilled if the number of reconstructed DFs is the same as not reconstructed DFs from others. With this condition it is known that even the interface cells at $x + e_i$ fulfilled $n \cdot e_i >= 0$ must be reconstructed [Don11].

Finally, if we want to port the free surface code on GPU, the relation among mass exchange, the reconstruction, streaming and collision must be analyzed and we must figure out which parts can be transformed into CUDA code to improve performance. In next section we will analyze and improve the free surface code, and then the improved part of free surface implementation will be ported into CUDA.

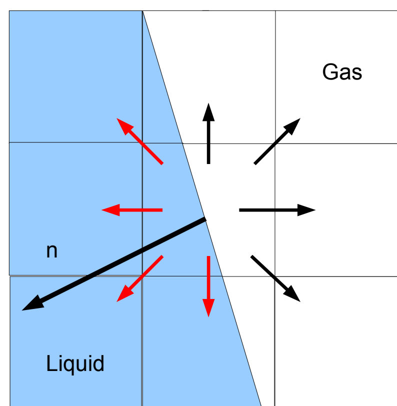

Figure 2.7: The missing distribution functions are reconstructed if the condition $n \cdot e_i > 0$ is fulfilled. The red is the reconstructed distribution functions and n is normal vector of the free surface at point x.

3 Implementation

This section of this thesis is divided into two parts. The first part is the introduction to the reference implementation and waLberla project, and the second part contains the analysis of the routines of free surface part and the implementation of the transformation of free surface code and porting it to CUDA code. In other words, we must determine which parts of free surface lattice Boltzmann implementation can be ported into CUDA.However, before the implementation we must know how the free surface in waLBerla Project works.

3.1 The waLBerla Project

The waLBerla project is built up by department of computer science 10 system simulation in university Erlangen-Nuremberg and this project supplies a framework for massive computation based on lattice Boltzmann simulations. The programmer can develop the application for specific simulation fields with the functionality from this framework. In waLBerla there are some SWEEPs which contains of the work steps and data fields[Don11]. Each work step can compute the data and process them with involved computation procedures. All of the results from a work step computation will be stored in data fields which can be read by other work steps. All SWEEPs work together in order to solve one specific problem, for instance, free surface flow simulation.

One of the most important application field of waLBerla is free surface flows simulation which consists of some SWEEPs, and they are built up and work together in order to process data with involved work steps of free surface algorithm. In this thesis all of SWEEPs about free surface computation will be researched and analyzed so that we can find out which associated SWEEPs or which parts in SWEEPs take the most performance time and these parts could be ported on GPU later.

3.2 Routines Analysis

In this thesis the located routines for free surface are, PDF SWEEP, FS FIND BORDER CROSSING MERGES SWEEP, FS MERGE BUBBLES SWEEP,FS NORMALS SWEEP,FS CLOSE INTERFACE LAYER SWEEP, FS COUNT INTERFACE NGHBR SWEEP, FS UPDATE FILL MASS BUBBLE SWEEP, FS UPDATE BUBBLES SWEEP and FS CHOOSE CONVERSION INTERFACE SWEEP. The functionality of these SWEEPs will be discussed below.

- **PDF SWEEP**
  PDF SWEEP can compute the streaming and collision from LBM, and for free surface in this SWEEP some additional functions are written that are the reconstruction of missing DFs and mass exchanging.

- **FS FIND BORDER CROSSING MERGES SWEEP**
  FS FIND BORDER CROSSING MERGES SWEEP and FS MERGE BUBBLES SWEEP perform in order to detect if the bubbles cross the patch boundaries and merge them together.
- **FS MERGE BUBBLES SWEEP**
  FS MERGE BUBBLES SWEEP can make the bubbles merge together if crossing path boundaries of them happens.
- **FS NORMALS SWEEP**
  In this SWEEP the normal n of interface cells are computed with fill values.
- **FS CLOSE INTERFACE LAYER SWEEP**
  In FS CLOSE INTERFACE LAYER SWEEP the complete interface cells layer is ensured by conversion among the different cells type.
- **FS COUNT INTERFACE NGHBR SWEEP**
  The number of interface cells is ensured so that the right number of cells is kept in conversations.
- **FS UPDATE FILL MASS BUBBLE SWEEP**
  In this routine the conversion of liquid into interface is performed, and fill level and volumes changes of bubbles are calculated.
- **FS UPDATE BUBBLES SWEEP**
  In FS UPDATE BUBBLES SWEEP (FS is Free Surface) the new volume and pressure of bubble are computed, at the same time if the bubbles are coalesced, the old index and volume values will be deleted.
- **FS CHOOSE CONVERSION INTERFACE SWEEP**
  The possible conflicts are resolved if the cell conversion happens from interface to gas or liquid in FS CHOOSE CONVERSION INTERFACE SWEEP.

[Don11]

This thesis focuses on all of SWEEPs above that work together and processes the computation to free surface. In order to porting code into CUDA to gain the better performance, SWEEPs must be located at first which cost the most performance time. Therefore, all SWEEPs time cost should be measured and their results will be compared to locate the most time cost. The implementation of time measuring stays in SWEEPs choosing procedure for one free surface simulation scenario so that each of SWEEPs time cost can be calculated. For each SWEEP the time measuring starts at beginning of each part of SWEEPs and stops at the end of them for each time step, and all the results of the measured SWEEP for one time step is stored temporary and added together until the end of the last steps of simulation, and the time percentage of each SWEEP time cost in the entire simulation time cost will be calculated as results shown in Table 3.1. From the Table 3.1 the most time cost of free surface is in PDF SWEEP and the free surface find border crossing merges sweep, and the SWEEP of bubbles updating and merging cost nearly no time since both parts do not take place and we do not research them in the situation of this thesis. From Chapter 2 we have known the free surface LBM is the extension of standard LBM and contains the mass exchange and the reconstruction of missing distribution functions besides streaming and the collision for interface cells. In other words, in PDF SWEEP the work step contains four kinds of computation procedure, streaming, collision, mass exchange and the reconstruction of missing distribution. If we want to get better performance from porting into CUDA, the most of time consuming parts of CPU code should be located at first and then optimized in

SWEEP Name	Time	Percentage of CPU time
PDF SWEEP:	53.2489	57.4639
FS FIND BORDER CROSSING MERGES SWEEP:	30.8003	33.2384
FS CLOSE INTERFACE LAYER SWEEP:	3.3275	3.5909
FS NORMALS SWEEP:	1.8657	2.0134
FS UPDATE FILL MASS BUBBLE SWEEP:	1.2648	1.3649
FS COUNT INTERFACE NGHBR SWEEP:	0.5289	0.5708
FS CHOOSE CONV INTERFACE SWEEP:	0.4539	0.4899
FS UPDATE BUBBLES SWEEP:	0	0
FS MERGE BUBBLES SWEEP:	0	0

Table 3.1: Computing Time from different SWEEPs for 300 time steps

Part	Time	Percentage of PDF SWEEP
STREAM LIQUID:	10.84543	20.36730
RECONSTRUCTION:	4.68727	8.80257
MASS EXCHANGE:	4.82026	9.05232
COLLIDE:	18.5072	34.75643

Table 3.2: Computing time cost in PDF SWEEP in 300 time steps

order to satisfy computation features of GPU. In the case of PDF SWEEP all the computation can not be ported into CUDA absolutely, since for each cells, for example, the cell type interface can decide whether the computation of the reconstruction of missing distribution will be executed so that we have to use if-else statement which could reduce the CUDA efficiency strongly. Therefore, before the implementation of porting into CUDA some parts must be transformed. But first of all computation procedure in PDF SWEEP should be located which consumes most performance time. From the performance time measuring of 300 time steps the results are shown in Table 3.2.

From the result of the Table 3.2 we found the collision part costs the most time in PDF SWEEP and should be ported into CUDA so that the whole free surface part could be optimized. Before that we must know how CUDA work efficiently. In general GPUs can process independent data in parallel. Furthermore, GPU programs can be especially effective when the programs want to process many independent data at the same time and in the same way. If the data that program processes depends on each other inseparably we must transform the CPU codes and reduce the dependency of data so that the ported parts perform efficiently. From the last section we have known how free surface LBM works, and in PDF SWEEP of waLBerla all the DFs of the whole cells are read from the initial state at first one by one from one time step to the next time step, and for each time step all the new DFs of each cell for next time step are calculated and stored for the next time step calculation. Since massive DFs in PDF SWEEP are calculated and the streaming and collision computations execute repeatedly in PDF SWEEPs for each cell, and from this reason all cells DFs could be set on GPU so that all the computation results are calculated quickly.

3.3 Reference Implementation Transformation

Until now we knew how the CPU code works can decide how the GPU code could be ported. Analyzing how CPU codes of the four parts in PDF SWEEP work together is the most important step. From the Algorithm 1 we can see that these four parts (mass exchange, reconstruction of missing DFs, Streaming and Collision) execute sequentially, and each part calculate the results which supply on the next part computation. Therefore the data (DFs) among mass exchange, the reconstruction of missing DFs, streaming and collision depend on each other closely. If we want to port the collision part alone, the CPU codes of PDF SWEEP must be transformed and customized onto GPU computation.

If the collision part is ported into CUDA alone, the loop for all computations must be divided into two parts, and the streaming, the reconstruction of missing DFs and mass exchanging work at first part and then the collision executes in the other part alone which will be ported into CUDA later in the next section. The implementation of the transformation is like Algorithm 2.

From the Algorithm 1 in the four parts of PDF SWEEP from the loops collision computation needs the result of f_i from the streaming part after the reconstruction of missing DFs and mass exchanging. At first loop the streaming, reconstruction and mass changing execute, and certain values from mass exchange and the reconstruction of missing DFs before the collision and after the streaming have to be stored in arrays if the loop splits into two parts. And these temporary arrays could be supplied to collision computation in next loop.

But we must be care of the DFs from the reconstruction of missing DFs. We knew in free surface lattice Boltzmann that the reconstruction of the missing DFs can generate new DFs for the interface cells streaming from the gas cells and interface cells, and this reconstruction computation is based on the boundary condition that we have discussed in 2.4 that all the missing incoming distribution functions which are fulfilled $n \cdot e_i >= 0$ must be reconstructed. For that reason the reconstructed DFs can come from either gas cells or interface cells. If the collision computation must be separated from another parts and execute alone, the DFs from the neighboring cells must be calculated and got correctly. Therefore, for the fluid cells neighboring the located cells can get DFs from them easily, but for the gas and interface cells neighborhood the located cell can not get DFs from them directly. The implementation of this thesis changed the some parts of the reconstruction so that the reconstructed DFs can be used in separated collision part correctly. For the gas cells neighborhood the solution is that the reconstructed DFs are stored in the gas cells temporary so that the located cells read from gas cells in separated collision without reconstruction computation directly. But for the interface cells which are fulfilled as $n \cdot e_i >= 0$ the reconstructed DFs can not be stored in the interface cells unfortunately, since the reconstructed DFs can overwrite the DFs in interface cells neighborhood which will be calculated as next located cells or neighborhood cells of another located cells so that this overwrites DFs in interface cells lead to the computation errors. For this reason above, only the reconstructed DFs from gas cells are stored in gas cells but the reconstruction from the interface cells fulfilled $n \cdot e_i >= 0$ will be not reconstructed and in separated collision parts the DFs will be read from interface neighborhood cells directly, see Fig 3.1. As stated previously, in separated collision computation the located cells can read from neighborhood cells directly without thinking about the cell type of neighborhood cells and itself. And with temporary stored filled value of mass exchange and the reconstructed DFs in gas cells the collision computation can calculated directly without if-else statement which can be ported into CUDA efficiently.

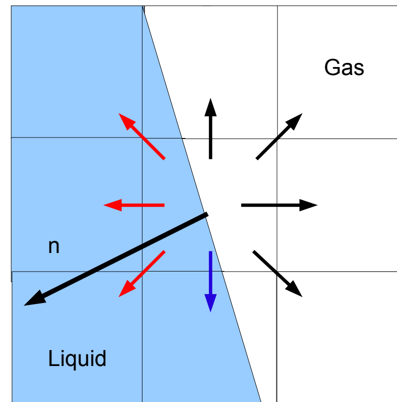

Figure 3.1: The boundary condition is changed for transformed PDF SWEEP. The red arrows are reconstructed DFs that are stored in gas cells, the black arrows are from fluid cells, the blue arrow is reconstructed in original code, since it is fulfilled $n \cdot e_i >= 0$, but for changed boundary condition it is not reconstructed.

Algorithm 1: PDF SWEEP

for $z = 1 \rightarrow zSize$ **do**
  for $y = 1 \rightarrow ySize$ **do**
    for $x = 1 \rightarrow xSize$ **do**
      InitializePDFfromSRC();
      {Initiated f_i from SRC};
      Streaming();
      if $location == INTERFACE$ **then**
        Reconstruct();
        {Reconstructed f_i};
        MassExchange();
      end if
      Collide();
      CopyNewPDFtoDST();
    end for
  end for
end for

Algorithm 2: Transformed PDF SWEEP

```
for z = 1 → zSize do
  for y = 1 → ySize do
    for x = 1 → xSize do
      InitializePDFfromSRC();
      Streaming();
      {Copy f_i};
      if location == INTERFACE then
        Reconstruct();
        {Reconstructed f_i};
        PDFinGASstore();
        MassExchange();
        FillValueStore();
      else
        COPYDF();
        {copy and store f_i} ;
      end if
    end for
  end for
end for
for z = 1 → zSize do
  for y = 1 → ySize do
    for x = 1 → xSize do
      Collide();
      CopyNewPDFtoDST();
    end for
  end for
end for
```

Time steps	Transformed PDF SWEEP	Original PDF SWEEP	Relative time loss
200	11.8882	10.8882	9.18
250	15.0757	13.7572	9.58
300	18.2212	16.8212	8.32

Table 3.3: Time comparison of PDF SWEEPs for 200, 250 and 300 time steps.

In the transformed code with two loops in Algorithm 2 in comparison to the original code the boundary condition is changed so that the correctness of simulation will be influenced, but from the result of simulation with the transformation we have found that the result is nearly the same as the original simulation result (see Fig. 3.2) which is sine-like function, but at peak value of simulated wave, for instance, at about 50μs the peak value of transformed simulated values is a little lower than original simulated result, and the same at around 90μs and 140μs the peak values of transformed simulation are lower and higher than original results. The reason of this inaccuracy is the changed boundary condition which is discussed from last section. But from the translated result the changed result can generate almost the same result and not lead to the false simulation result, either, and that can conclude the correctness of division of loop and changed boundary condition. But at the same time temporary stored filled value is needed and additional reading and writing operations from memory to CPU are needed and all DFs of the cells will be stored temporary before the collision and the transformation of entire PDF SWEEP leads to low performance certainly. With this question the time for 200, 250 and 300 time steps are measured (see Table.3.3) and we found that the transformation of PDF SWEEP takes longer processing time, but the time difference from both of algorithm is not so large. In next section we can find that the porting codes into CUDA on GPU can get much better performance which could compensate this time loss.

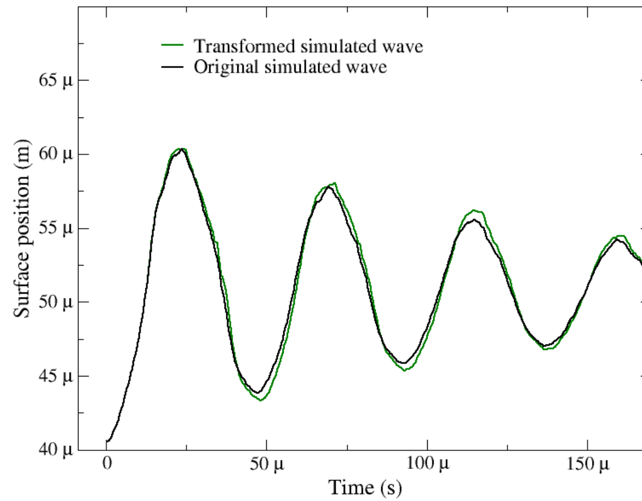

Figure 3.2: The comparison of original simulated and transformed simulated (boundary condition changed) results.

3.4 GPU Implementation with CUDA

In this section at first CUDA will be discussed. From the discussion we can know what is CUDA, why CUDA should be used and how GPU works with CUDA, and then with the features of CUDA the implementation of transformation from last section.

3.4.1 CUDA Introduction

What is CUDA?

CUDA (Compute Unified Device Architecture) is a parallel computing architecture which is developed by Nvidia. CUDA is the computing engine in Nvidia Graphics Processing Units (GPUs) that is accessible to software developers through variants of programming languages. Through CUDA programming it is able to provide an increase in computing performance by harnessing the power of the GPU, and the programmer can also take advantage of the massive parallel computing power of an Nvidia graphics card with CUDA for general purpose computation. Before talking

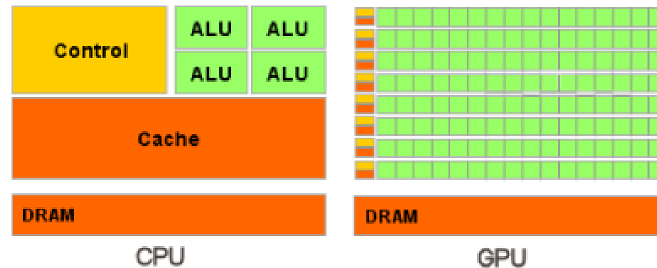

Figure 3.3: Hardware architecture of CPU and GPU. [dif08]

about GPU we have to discuss CPU. As we know CPUs of Intel and AMD with more than one core can do one or two tasks at a time very well and also execute tasks very fast. But GPU has ability to dealing with massive number tasks at the same time and doing those tasks much more quickly. For instance, GPU processes one 3 dimension object which consists of thousands of points and each point has position value, if this object rotates, the new positions of entire points of this object will be calculated at the same time, in order to accomplish this work GPU uses hundreds of ALUs (Arithmetic Logic Units).

Fortunately, the programmer can write program with a tool of Nvidia GPU in order to gain an unprecedented amount of computational power from GPU as we need, and that is CUDA [gpg02].

Why CUDA should be used?

The hardware difference between CPU and GPU is shown in Fig. 3.3. In GPU there are numbers of ALUs much more than CPU. As a result, GPU has much more power on amounts of computation. From the architecture of GPU we can see, in GPUs there are a number of multiprocessors, and each of them executes in parallel with others and each multiprocessor has several groups of processors [gpg02]. There are many of multiprocessor on GPU, and we call each processor in the multiprocessor core, and each core can execute a sequential thread, and the cores execute in NVIDIA called SIMT (Single Instruction, Multiple Thread), and then all cores in the same group execute the same instruction at the same time as classical SIMD (Single instruction, Multiple Data) processors. SIMT handles operations differently than SIMD, though they look much the same, where some cores are disabled for conditional operations [gpg02]. As previously discussed, CUDA enables the developer to programs with the features of GPGPU inside hundreds of processors on the GPU, and its massive data processing ability is much more powerful than in any type of CPU. However, this does put a limit on the types of applications that are well suited to CUDA. In order to get better performance of the applications efficiently on a GPU building up

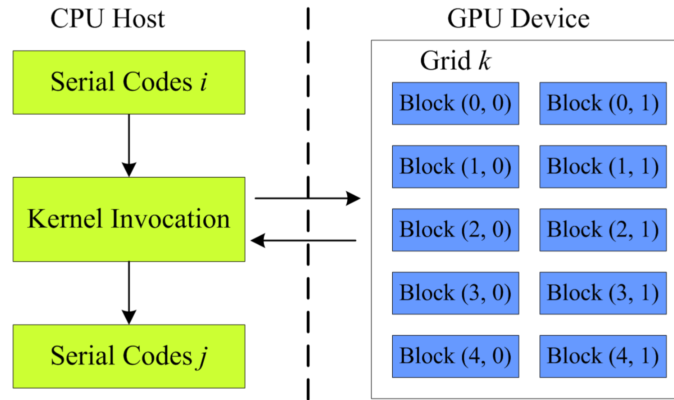

Figure 3.4: The kernel invocation of CPU and GPU[dif08]

hundreds of threads are necessary, and in another word the more threads are built up and the better program performance may be achieved. If the data in an algorithm is processed sequential, then it does make no sense to use CUDA. Moreover, if we cannot break the data into pieces of a thousand threads, then with CUDA porting onto GPU is not the best solution.

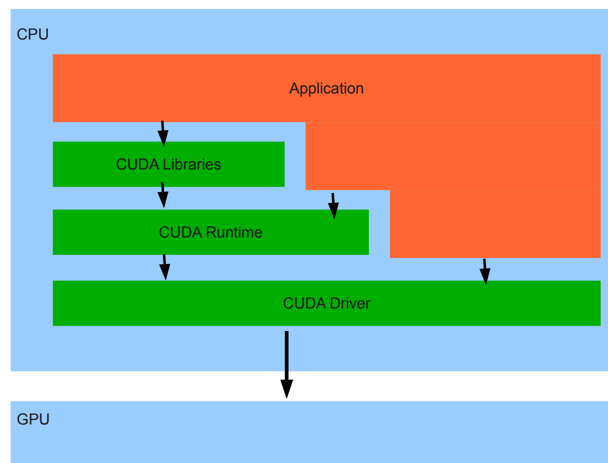

Figure 3.5: CUDA software diagram. [cud]

How CUDA is used?

As we knew that CUDA is a scalable parallel programming model and a software environment for parallel computing. Nvidia provides a standard toolkit for programming CUDA which contains the compiler, debugger, profiler, libraries, and other information. The CUDA also supports standard languages (e.g. C extension) and APIs for GPU computing, such as OpenCL and DirectX

Compute. CUDA programming can be taken full advantage of when writing in C, since C programming language is not strange for the most developers. As we said the main idea of CUDA is making thousands of threads executing in parallel, but one of the most important to remember is that the entire program does not need to be written in CUDA. If the program is very large with the customized interface and many other functions, then most of the codes should be written in C or any other appropriate language. Then, if something extremely computationally intense is needed, the program could call the CUDA kernel function. So the main idea is that CUDA should only be used for the part of the large program in order to process the massive computation problem [gpg02]. Fig. 3.5 shows the CUDA software diagram and from this diagram we can see clearly that the CUDA can be called by any other application if that is needed. Above all CUDA should only be used for the part of the application and be invoked by the code on the CPU with CUDA kernel function in Fig. 3.4.

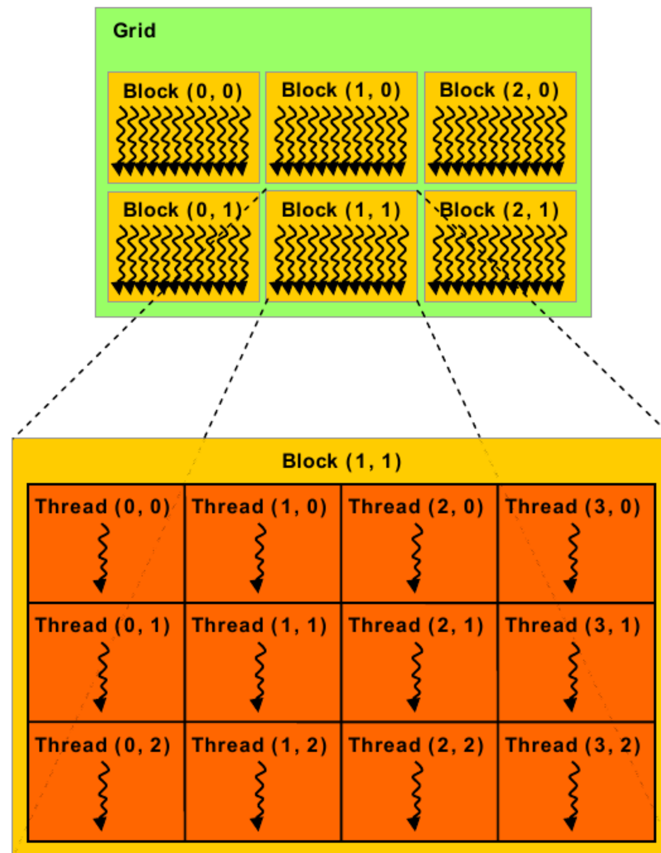

Figure 3.6: The organisation of threads, blocks and grids. [SJ11]

In programming of CUDA, GPUs are programmed as sequence of kernels and all kernels execute before the next begins. The multiprocessors execute in parallel asynchronously, however, GPUs do not support a coherent memory model so that the multiprocessors can synchronize with each other and threads on one multiprocessor can't send or receive results to or from threads on the other multiprocessors. For CUDA the host program on CPU launches a sequence of kernels. A

kernel is organized as a hierarchy of threads [Hab08]. Many threads are grouped into block and many blocks are grouped into a grid. Each thread has an index in the block, and at the same block has an index in grid [Wol08]. Kernels can use these indexes to compute the data (see Fig. 3.6).

3.4.2 CUDA Implementation

As we have discussed in section 3.3 the collision has been separated into another loop and the boundary condition has also been changed to support for porting into CUDA on the GPU. From the transformation in Algorithm 2 we knew that in collision each cell can read DFs from neighborhoods directly without knowing the type itself and reconstructions of missing DFs so that each cell can do the same computation. Until now the collision part can take the advantage of the CUDA feature and be ported into CUDA directly. The Algorithm 3 shows the procedure of CUDA implementation. The implementation of CUDA contains four parts, initiation data for GPU computation, on GPU memory copy, collision on GPU and copying results on CPU back and resource free.

Algorithm 3: CUDA PDF SWEEP

```
for z = 1 → zSize do
  for y = 1 → ySize do
    for x = 1 → xSize do
      InitializePDFfromSRC();
      Initiated f_i from SRC;
      Streaming();
      Copy f_i;
      if location == INTERFACE then
        Reconstruct();{Reconstructed f_i }
        PDFinGASstore();
      end if
      MassExchange();
      fillValueStore();
    end for
  end for
end for
init();
copyOnGPU();{Copy f_i onto GPU }
pdfKernel();
copyOnGPU();{Copy f_i into DST }
```

Initiation for GPU computation

Before the porting implementation something must be prepared. In Algorithm 2 all the missing distribution functions are stored into gas cells so that in next collision step they could be read directly without determining the type itself and reconstructing missing DFs. From the LBM model we know all the DFs for each cell are stored for 19 directions see Fig. 2.3, from the definition

in waLBerla each cell is a data structure which contains all information to this cell, for example, 19 directions DFs, cell type, velocity and so on. In waLBerla there are two different data layout configurations for 19 directions DFs of each cell so that they can satisfy different optimal computations. The first data layout is as Fig. 3.7 that 19 directions DFs are stored for each cell in memory sequentially and all cells have the same sequential order of 19 directions. The other data layout is as Fig. 3.8 that DFs at one direction of all cells are stored sequentially and all 19 directions have the same sequential order of all cells. But the data layout as Fig. 3.7 cannot be used in the CUDA code efficiently since the CPU code calculates each direction for a cell in a loop. If we want to port the code onto GPU, the principal aim is that we separate data into threads as much as possibly and each thread computes them at the same way at one time. Imaging a thread is only for a cell (x,y,z) collision computation and all threads are all the cells. With data layout all DFs at each direction of entire cells are grouped together and for 19 directions there are 19 groups. As above first step we have to configure the data layout as Fig. 3.8 in waLBerla so that all DFs are computed by GPU easily without additional data layout changing cost. As results new DFs after collision GPU computation have the same data layout configuration as what the CPU need.

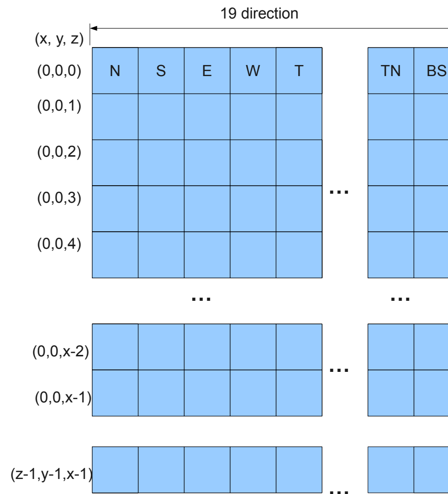

Figure 3.7: DFs data layout configuration on CPU

On GPU memory copy

With the new data layout some space on GPU memory for data copy must be allocated, and then all data is copied on that area. This area is used for DFs and some constant values, for instance, fill values (See Algo. 4) of all interface cells. The memory space of the result of density and velocity must also be allocated at the same time which could be calculated from new DFs later.

According to the architecture of the GPU we set the size of one block as $xSize$ (x size) threads and one grid as size of $ySize * zSize$ (y and z size), since it is 3 dimension simulation for D3Q19 model of LBM, the grid and block model is exact solution for 3 dimension computation, see Algorithms 5. Each cell located in (x,y,z) in D3Q19 model can be located as position (ThreadIndex

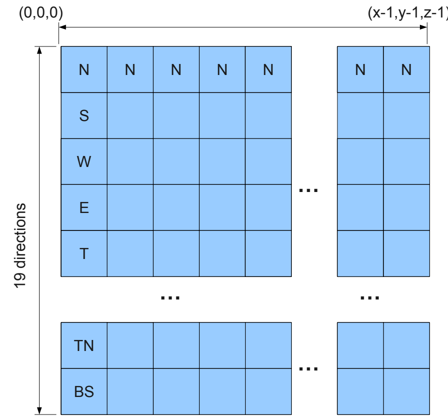

Figure 3.8: DFs data layout configuration for copying on GPU

X, BlockIndex X, BlockIndex Y) on GPU. See Fig. 3.9.

Algorithm 4: DFs and fill values are copied on GPU.

```
cudaMemcpy(d_fills,h_fills,dimA*sizeof(Real), cudaMemcpyHostToDevice);
cudaMemcpy(d_src,h_src,(19*dimA)*sizeof(Real), cudaMemcpyHostToDevice);
```

Algorithm 5: The configuration of block and grid on GPU.

```
dim3 dimblock(xSize,1);
dim3 dimgrid(ySize,zSize);
```

Collision on GPU

Once all the necessary data is copied onto GPU the kernel code that will be called (Algo. 6) and perform the collision completely on GPU. In the kernel we write the code only once and consider on the collision only for one cell. With the data layout see Fig. 3.8 all results of new DFs will be generated only once those have the same layout as Fig. 3.8 and the velocity and density are also calculated after new DFs are generated on GPU for each cell. The collision part on GPU is as in Algorithm 7. From now on the loop of the collision in transformed Algorithm 2 has been ported on GPU completely.

Copy results on CPU back

After computation on GPU the results on GPU will be copied on CPU back for the next time step computation, see Algorithms 8 and the memory resource of collision part on GPU must be

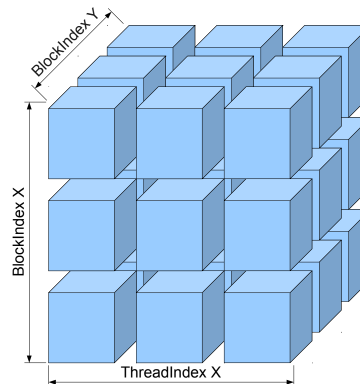

Figure 3.9: Points located on GPU

Algorithm 6: Kernel Invocation

CStreamCollide_gpu<<< dimgrid, dimblock>>>();
cudaThreadSynchronize();
checkCUDAError(”kernel invocation”);

Algorithm 7: The collision on GPU

Initial(); {initial}
getPDFsfromNeighbouring(); {got PDF from neighboring cells}
forceTermCalculate(); {force term to calculate}
generateNewPDFs(); {new PDF of this cell after collision to generate}
generateVelocity(); {velocity of this cell from new PDFs to generate}
generateDensity(); {density of this cell from new PDFs to generate}

released at once. With the same data layout configuration which is defined in waLBerla new DFs of all cells are stored into destination DFs with new velocity and density. Finally all the data resource must be released that have been cached before.

Algorithm 8: The results copied back on CPU and memory is release.

```
cudaMemcpy(h_velocity,d_velocity,sizeof(Real), cudaMemcpyDeviceToHost);
cudaMemcpy(h_dst,d_dst,sizeof(Real), cudaMemcpyDeviceToHost);
cudaMemcpy(h_rho,d_dens,sizeof(Real), cudaMemcpyDeviceToHost);
cudaFree(d_velocity);
cudaFree(d_dst);
cudaFree(d_dens);
```

4 Results Analysis

In this section following two aspects of the result will be analyzed, the correctness of the result and the efficiency of GPU codes.

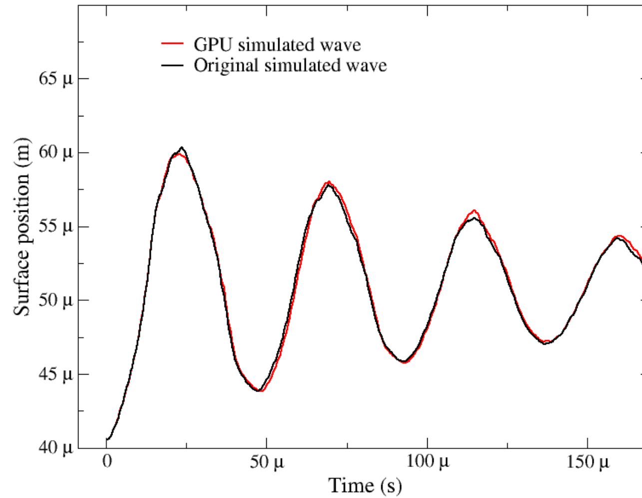

Figure 4.1: GPU simulated result compared to the original simulated result.

4.1 Simulation Result Analysis

Fig. 4.1 shows the results comparison of original CPU and GPU simulation in 40,000 time steps. As shown in Fig. 4.1 GPU simulation result can coincide with original GPU simulation result, and the correctness of the GPU implementation is proved. The simulation result of GPU code is nearly the same as the result of the CPU code, but in the Fig. 4.1 we can see that some parts of the result cannot be 100 percent same as the original results. As shown in Fig. 3.2 the GPU simulated result is also sine-like function as original simulated result, but still at peak value of simulated wave, for example, at about 25μs the peak value of GPU simulated result is lower than original result, and at about 120μs the peak value is much higher, and those deviation at those peak value points exits also in the transformed (Changed boundary condition) simulated result in Fig. 4.1. Therefore, we can conclude that this deviation between the GPU and original simulated results is caused by the changed boundary condition. Although there is deviation in GPU simulations result, it is not so critical. The simulation result of GPU is almost right, since GPU simulation result in Figure 3.2 has the same trends and nearly all result values are the same as original simulation result. Besides that in Figure 4.2 the peak values, for instance, at around 25 μs, 40μs and 120μs, the GPU simulated result is lower and higher than transformed simulate

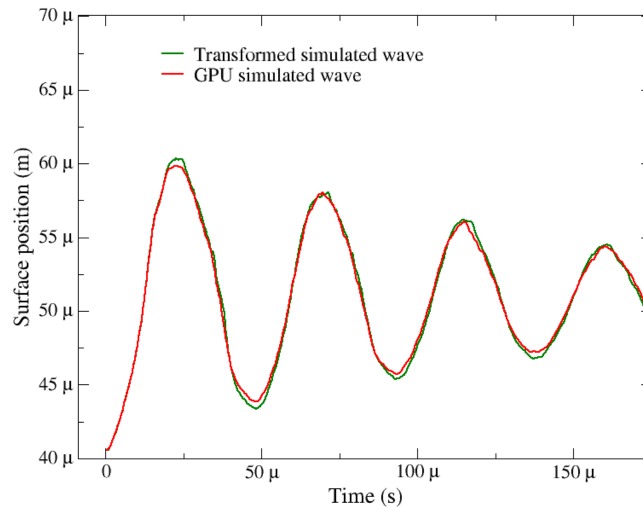

Figure 4.2: GPU simulated result compared to the transformed (changed boundary condition and separated loops) simulated result.

result obviously. This difference is caused by no double precision in GPU implementation which is implemented in original waLBerla code. Both reasons, no double precision and changed boundary condition, make GPU simulated result close to original results at the peak points.

4.2 SWEEPS parts Analysis

Fig. 4.3 shows the performance comparison of CPU, CPU transformed and GPU simulation in 300 time steps. Clearly the GPU code simulated 17.28% less time than another two implementation from all simulation results, and the performance of the total simulation is improved. For the PDF SWEEP performance comparison after 300 time steps the result as shown in Fig. 4.4. For PDF SWEEP part the performance time has been reduced 30.04% in comparison to CPU implementation. For 300 time steps in PDF SWEEP part saving nearly 17 seconds is the reason to the time saving in total simulation. All above, performance comparison result prove that the implementation of porting on GPU can improve the total performance time of the simulation.

4.3 PDF SWEEP part Analysis

In this section PDF SWEEP will be analyzed and the improved result of the GPU implementation will be also discussed.

We measure the time cost of collision parts after 300 time steps and the result will be compared as shown in Fig. 4.5. We can see after 300 time steps simulation CUDA implementation in collision computation saves nearly 17 seconds. After performance time measuring of the time steps from 10 to 300 time steps we found the speedup of collision parts keeps around factor 4 with

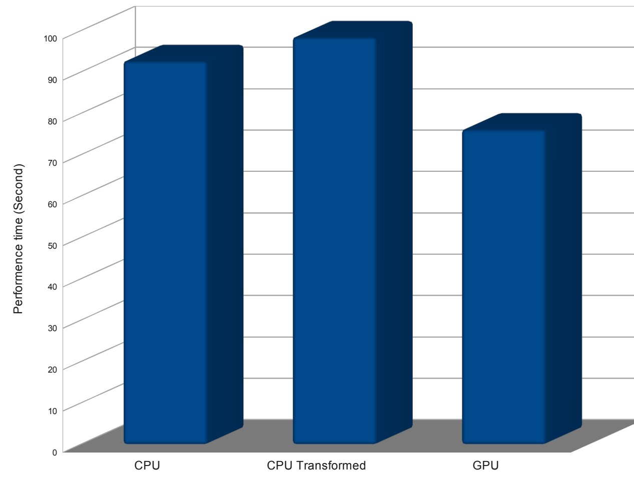

Figure 4.3: The time comparison of three different implementations.

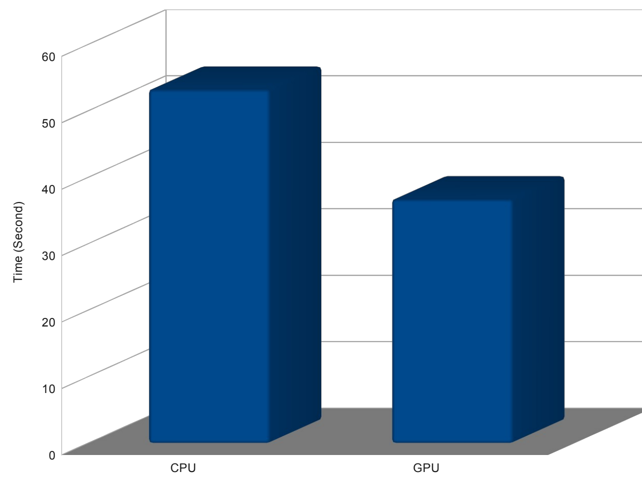

Figure 4.4: The time comparison of SWEEPs from the CPU and GPU.

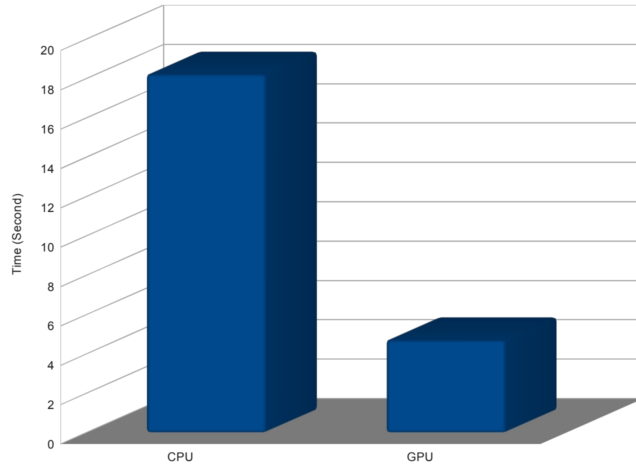

Figure 4.5: The time comparison of collision to CPU and GPU

Part	Time	Percentage of PDF SWEEP
STREAM LIQUID:	10.8114	29.4424
COLLIDE:	4.9133	13.3802
RECONSTRUCTION:	4.6103	12.5551
MASS EXCHANGE:	3.9484	10.7525

Table 4.1: Performance time in PDF SWEEP after ported collision on GPU.

the formula calculation:

$$S = \frac{T_{cpu}}{T_{gpu}} \tag{4.1}$$

In that formula the collision time cost T_{cpu} is the execution time of original CPU code and T_{cpu} is the execution time of collision ported on GPU. The important point is that this result of speedup depends on the right data layout configuration as Chapter 3. With this configuration the GPU code can get efficient performance without additional data layout changing cost. From the result of Table 4.1 we can see, after the collision computation is ported on GPU, the collision need less performance time which takes 11.42553% in all the PDF SWEEP before GPU implementation in 3.2 the collision takes 34.75643% in PDF SWEEP computation. Moreover, the CUDA implementation needs more time cost on data transferring between the CPU and GPU memory, because of this reason the data transferring time cost is measured in order to check out how much time the data transferring takes. Time measuring starts at Algo.8 and Algo.4 parts which contains data copied on GPU and results copied on GPU back, and the measuring results are in Table 4.2. In Table 4.2 data copy between GPU and CPU costs 98.6284% the performance time from collision computation time ported on GPU, and the real collision computation on GPU costs only 1.3716 seconds. The most time of collision porting implementation costs on the data transferring. If we want to gain better performance on collision computation, one optimized method on data transferring between GPU and CPU must be the best solution.

Above all, all comparison of the performance time between CPU and GPU implementation

Part	Time	Percentage of collision (on GPU)
Data copy (GPU and CPU):	4.8458	98.6284
Collision on GPU:	0.0674	1.3716

Table 4.2: The time comparison of data copy between GPU.

prove that collision computation porting on GPU can improve the performance time of simulation efficiently.

5 Conclusion

5.1 Summary

In order to optimize the simulation time of the free surface lattice Boltzmann method some routines of waLBerla could be ported into CUDA on GPU. At first we analyze all associated routines of free surface algorithm and locate the PDF SWEEP costs that require the most time in the simulation procedure of all free surface SWEEPs, furthermore, from the time measuring of the four main computation parts, streaming, collision, mass exchange and the reconstruction of missing DFs in PDF SWEEP we found that the collision consumes the most performance time and it should be ported on GPU. And then the CPU code structure of collision is transformed into another form hat satisfied the feature of CUDA. At the same time the boundary condition is changed so that all the reconstructed DF from gas cell can be stored in gas cells, and the purpose of this changed boundary condition is that in collision computation the located cell can get the DF from neighborhoods without knowing their types. After this, the collision part can be implemented in CUDA easily. With the implementation of CUDA the data layout of all cells DFs has to be changed so that they can be copied into GPU memory as GPU hardware architecture. At last from the same simulation result of ported GPU code and CPU implementation of waLBerla we can see it is possible and beneficial to port CPU code onto GPU with the transformation of some parts of PDF SWEEP. The structure transformation of the CPU code can lead to additional cached cost, for instance, cached DFs of all cells, although for each time step and then the efficiency of performance will be reduced with additional cached cost, the complete porting on GPU can compensate for that time loss with the massive parallel computation.

With the implementation of CUDA the same result can be simulated correctly as original CPU implementation of waLBerla with a clearly reduced time cost of collision, but the changed boundary condition leads to the deviation of the simulation result. Before copying all DFs data of each cell on GPU some data layout must be configured so that GPU can process them efficiently. With this data layout GPU can calculate and return all new DFs that could be used in the next time step computation. DFs will be copied to GPU and then back to CPU continuously, and we could get the factor 4 of speedup with the right data layout configuration in waLBerla which could all DFs are copied into GPU memory directly without data layout changing cost. From the time measuring of copy time between CPU and GPU the copying procedure influence between CPU and GPU is significant. The collision computation on GPU is only 1.37% of performance time in GPU implemetation. The copying data takes almost 98.62% time. Additionally, in this thesis the implementation of CUDA cannot satisfy the double precision which is implemented in CPU code in waLBerla.

From this thesis we can see, that some parts of the free surface LBM are ported on GPU possibly. With the CUDA implementation relative low performance routines efficiency could be improved. GPU can help us with the massive computation in waLBerla not only for traditional LBM, but also for the extension free surface LBM. Although in this thesis collision computation for free surface can get better performance than CPU implementation, there are still other parts

that could be ported into CUDA on GPU, for instance,FIND BORDER CROSSING MERGES SWEEP.

5.2 The future works

As the future works we could improve some aspects of the implementation.

First, the changed boundary condition can lead to inaccuracy of the simulation result, but if we cannot compute all the reconstructed DF in any reconstructed neighborhoods cells directions the inaccuracy result is not avoided. It is possible to cache all the reconstructed DFs so that the correct boundary condition can be fulfilled.

And then, the data transferring between the CPU and GPU memory must be improved. In this thesis the data transferring execute after each time step before the next time step, it is possible, that the result could be cached in the GPU memory and in new time step only the changed DFs of the cells could be copied on GPU memory so that the quantity of the transferred data could be reduced.

At last, from the time measuring of SWEEPs we found there are some other SWEEPs take also much time, for example, FIND BORDER CROSSING MERGES SWEEP, and we could also port this part on GPU in order to the more efficient performance.

A Abbreviations

ALU	Arithmetic Logic Unit
API	Application Programming Interface
BGK model	Bhatnagar Gross Krook model, which is a model for collision processes in gases
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
D3Q19	A three-dimensional Lattice Boltzmann model
DFs	Distribution functions
GPU	Graphics Processing Unit
GPGPUs	General Purpose Graphics Processing Units
H-Theorem	Description of the increase in the entropy of an ideal gas with Boltzmann equation
LBM	Lattice Boltzmann Method
SDK	Software Development Kit
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads

List of Figures

2.1 The types cells in free surface LBM 4
2.2 Data representation of a bubble in free-surface LBM: Gas and interface cells store an ID of the bubble[DFP+09]. Blue is liquid and white is gas 4
2.3 Structure of the lattice in D3Q19 model.[DGF+07] 5
2.4 Streaming and Collision step. Left a is a cell at time t after collision and right b is the cell at next time step t+1 after streaming. 6
2.5 The mass exchange takes place among the fluid and interface cells . 7
2.6 Cells type conversation. [Poh08] 8
2.7 The missing distribution functions are reconstructed if the condition $n \cdot e_i > 0$ is fulfilled. The red is the reconstructed distribution functions and n is normal vector of the free surface at point x. . . . 9

3.1 The boundary condition is changed for transformed PDF SWEEP. The red arrows are reconstructed DFs that are stored in gas cells, the black arrows are from fluid cells, the blue arrow is reconstructed in original code, since it is fulfilled $n \cdot e_i >= 0$, but for changed boundary condition it is not reconstructed. 15
3.2 The comparison of original simulated and transformed simulated (boundary condition changed) results. 17
3.3 Hardware architecture of CPU and GPU. [dif08] 18
3.4 The kernel invocation of CPU and GPU[dif08] 19
3.5 CUDA software diagram. [cud] 19
3.6 The organisation of threads, blocks and grids. [SJ11] 20
3.7 DFs data layout configuration on CPU 22
3.8 DFs data layout configuration for copying on GPU 23
3.9 Points located on GPU 24

4.1 GPU simulated result compared to the original simulated result. . . 27
4.2 GPU simulated result compared to the transformed (changed boundary condition and separated loops) simulated result. 28
4.3 The time comparison of three different implementations. 29
4.4 The time comparison of SWEEPs from the CPU and GPU. 29
4.5 The time comparison of collision to CPU and GPU 30

List of Tables

3.1 Computing Time from different SWEEPs for 300 time steps 13
3.2 Computing time cost in PDF SWEEP in 300 time steps 13
3.3 Time comparison of PDF SWEEPs for 200, 250 and 300 time steps. 16

4.1 Performance time in PDF SWEEP after ported collision on GPU. . 30
4.2 The time comparison of data copy between GPU. 31

List of Algorithms

1 PDF SWEEP 15
2 Transformed PDF SWEEP 16
3 CUDA PDF SWEEP 21
4 DFs and fill values are copied on GPU. 23
5 The configuration of block and grid on GPU. 23
6 Kernel Invocation 24
7 The collision on GPU 24
8 The results copied back on CPU and memory is release. 25

Bibliography

[cud] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide.*

[DFP+09] S. Donath, C. Feichtinger, T. Pohl, J.Götz, and U. Rüde. A parallel free surface lattice boltzmann method for large-scale applications. Technical report, Institute for System-Simulation in University of Erlangen-Nuremberg, 2009.

[DGF+07] S. Donath, J. Götz, C. Feichtinger, K. Iglberger, S. Bergler, and U. Rüde. On the resource requirements of the hyper-scale walberla projec. Technical report, Institute for System-Simulation in University of Erlangen-Nuremberg, 2007.

[dif08] Introduction, gpu and cpu differences, October 2008. http://www.ixbt.com/.

[Don11] Stefan Donath. *Wetting Models for a Parallel High-Performance Free Surface Lattice Boltzmann Method.* PhD thesis, University of Erlangen-Nuremberg, Computer Science 10 System Simulation, 2011.

[Eng78] J. N. England. A system for interactive modeling of physical curved surface objects. *SIGGRAPH Comput. Graph.*, 12(3):336–340, 1978.

[Fei06] Christian Feichtinger. Simulation of moving charged colloids with the lattice boltzmann method. Master's thesis, University of Erlangen-Nuremberg, Computer Science 10 System Simulation, 2006.

[gpg02] What is gpgpu, October 2002. http://gpgpu.org/.

[Hab08] Johannes Habich. Performance evaluation of numeric compute kernels on nvidia gpus. Master's thesis, Friedrich-Alexander-Universitt Erlangen-Nrnberg, 2008.

[KPR+05] C. Körner, T. Pohl, U. Rüde, N. Thürey, and T. Zeiser. *Parallel lattice Boltzmann methods for CFD Applications.* Springer, 2005.

[KTH+05] C. Körner, M. Thies, T. Hofmann, N. Thürey, and U. Rüde. Lattice boltzmann model for free surface flow for modeling foaming. *Journal of Statistical Physics*, 121:179–196, 2005.

[OJL+07] Owens, D. John, Luebke, David, Govindaraju, Naga, Harris, Mark, Kruger, Jens, Lefohn, E. Aaron, Purcell, and J. Timothy. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26:80–113, 2007.

[Poh08] Thomas Pohl. *High Performance Simulation of Free Surface Flows Using the Lattice Boltzmann Method*. PhD thesis, University of Erlangen-Nuremberg, Computer Science 10 System Simulation, 2008.

[PY06] L. Schaefer P. Yuan. Equations of state in a lattice boltzmann model. *Physics of Fluids*, 18, 2006.

[SJ11] Edward Kandrot Sanders Jason. *CUDA by example. An introduction to General-Purpose GPU Programming*. Addison-Wesley, 2011.

[Suc01] Sauro Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Oxford University Press, 2001.

[Thu03] Nils Thuerey. A single-phase free-surface lattice boltzmann method. Master's thesis, Friedrich-Alexander-Universitt Erlangen-Nrnberg, 2003.

[TK07] J. Tölke and M. Krafczyk. Towards three-dimensional teraflop cfd computing on a desktop pc using graphics hardware. In *In Proceedings of International Conference for Mesoscopic Methods in Engineering and Science ICMMES07*, Munich, 2007.

[TR04] Nils Thürey and Ulrich Rüde. Free surface lattice boltzmann fluid simulations with and without level sets. In *VMV 2004*, 2004.

[TRK05] N. Thuerey, U. Rüde, and C. Körne. Interactive free surface fluids with the lattice boltzmann method. Technical report, Institute for System-Simulation, Insititue of Science and Technology of Metals of University of Erlangen-Nuremberg, 2005.

[Wol08] Michael Wolfe. Understanding the cuda data parallel threading model a primer, October 2008. http://www.ixbt.com/.