Stony Brook University

Flow Simulation and Visualization on GPU Clusters

A Dissertation Presented
by
Zhe Fan

to
The Graduate School
in Partial Fulfillment of the
Requirements
for the Degree of
Doctor of Philosophy
in
Computer Science
Stony Brook University

August 2008

Stony Brook University
The Graduate School

Zhe Fan

We, the dissertation committee for the above candidate for the degree of Doctor of Philosophy, hereby recommend acceptance of this dissertation.

Arie Kaufman, Dissertation Advisor
Distinguished Professor, Computer Science Department

Klaus Mueller, Chairperson of Defense
Associate Professor, Computer Science Department

David Xianfeng Gu
Assistant Professor, Computer Science Department

Bengt-Olaf Schneider
PhD, NVIDIA Corporation

This dissertation is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Flow Simulation and Visualization on GPU Clusters

by

Zhe Fan

Doctor of Philosophy

in

Computer Science

Stony Brook University

2008

In recent years, the performance of graphics processing units (GPUs) has been increasing dramatically. Modern GPUs have surpassed CPUs in raw computational power by an order of magnitude. Because of the explicit data parallelism in the graphics pipeline, the GPU can efficiently use hundreds of thread processors to process data in parallel. Moreover, the GPU is becoming more and more flexible and programmable. As a result, accelerating general-purpose computation on the GPU (GPGPU) has become an active area of research.

This dissertation presents efficient ways to use GPUs and GPU clusters for GPGPU in general and flow simulation and visualization in visual applications in particular. We focus on a GPU-friendly method, called the Lattice Boltzmann Model (LBM), a mesoscopic method that applies linear and local operations at discrete lattice sites. We describe an optimized LBM implementation on a single GPU and its applications in real-time modeling of natural phenomena, such as fire, smoke, wind, and heat shimmering. We also present a novel GPU-based adapted unstructured LBM algorithm for simulating flow on arbitrary 3D triangular surfaces. We further extend the LBM implementation from a single GPU to a GPU cluster and describe how to efficiently manage the communication among the multiple GPUs. We also present an application of the GPU cluster simulation in urban dispersion modeling. We further present an LBM implementation of irregular-shaped simulation domain on a GPU cluster and its application for thermal fluid dynamics in a pressurized water reactor of a nuclear power plant. Finally, Zippy, a

general framework for GPU cluster programming, is presented. Zippy abstracts the GPU cluster architecture with two characteristics important to high performance—two-level parallelism hierarchy and non-uniform memory access (NUMA)—and hides other architecture details. It simplifies the programming of a GPU cluster while maintaining high performance. We also present three example applications developed using Zippy and show how simulation and visualization modules can be seamlessly integrated on a GPU cluster.

This dissertation is dedicated to my wife Lei.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to express my gratitude to Distinguished Professor Arie Kaufman for being a great advisor, who has firmly guided me through my PhD years. With his expertise in graphics and visualization and enthusiasm and aspiration for research, Arie has taught me not only knowledge but also career principles. He has made me learn to conquer tasks little by little, to bear a big picture in mind, and to be self-disciplined. Arie has been always attentive to my interests and needs. He has given me the freedom to explore my areas of interest, as well as invaluable support and advice. At times when I was uncertain, Arie's encouragement and suggestions were crucial for me to refocus and finally reach the goal. For many things of this kind I cannot thank Arie enough.

I want to thank the committee members, Professor Klaus Mueller, Professor David Xianfeng Gu, and Doctor Bengt-Olaf Schneider. Klaus has participated in discussions of most part of the work described in this dissertation and has given me many excellent suggestions. He also taught me the first course in visualization that fascinated me. David guided me through an interesting project of using GPUs for real-time ray tracing. His deep understanding of mathematics and geometry has made the discussion with him always inspiring and enjoyable. I thank Dr. Schneider for his detailed comments on my dissertation and for that he spent lots of time out of his busy schedule to serve as the committee member.

I thank Dr. Suzanne (Suzi) Yoakum-Stover. I was very lucky to work with her, a person with a strong background in Physics. She has been both a mentor and friend in my junior PhD years. I think she would be smiling if she sees my dissertation. Also, many thanks go to the coauthors, Wei Li, Xiaoming Wei, Ye Zhao, Feng Qiu, Shengying Li, Fang Xu, and Kaloian Petkov, and the colleagues, Huamin Qu, Nan Zhang, Susan Frank, Wei Hong, Haitao Zhang, Jianning Wang,

Lujin Wang, and Joseph Marino. In particular, Feng has contributed a tremendous amount of time to our discussions. I also thank Bin Zhang for managing our cluster. He took so much trouble to help me and made my research easier. I thank Stella Mannino for her excellent administration work.

Finally, I want to acknowledge my wife, my parents, and my in-laws for their support and for always being there to share my happiness and stress.

Vita, Publications and Field of Study

Education

- **Stony Brook University**
  Ph.D., CS, 2001 - August 2008 (expected)
  M.S. received in 2004
- **Institute of Software**
  Chinese Academy of Sciences
  M.S., CS, 1998 - 2001
- **The Special Class of the Gifted Young,**
  University of Science and Technology of China
  B.E., CS, 1993 - 1998

Field of Study

- Flow Simulation and Visualization on GPUs and GPU Clusters
- Physics-Based Modeling of Natural Phenomena
- Networked graphics
- Visualization
- Real-Time Rendering

Experience

- **Research Assistant**, Jun. 2002 – current
  Visualization Lab, Stony Brook University

- **Intern**, Jun. 2007 – Aug. 2007
  Siemens Corporation Research
- **Teaching Assistant**, Sep. 2001 – May 2002
  Computer Science Department, Stony Brook University
- **Intern**, Nov. 1999 – Apr. 2000
  IBM China
- **Intern**, Aug. 1997 – Jul. 1998
  Lotus Beijing Software Development Center

Honors

- **Outstanding Teaching Assistant**, Stony Brook University, 2001-2002
- **University Fellowship**, Stony Brook University, 2001
- **Outstanding Master Thesis**, Institute of Software at Chinese Academy of Science, 2001
- **Outstanding Graduate**, Anhui Province, China, 1998
- **Outstanding Graduate**, University of Science and Technology of China, 1998
- **University Fellowship**, University of Science and Technology of China, 1993-1997

Publications

1. Z. Fan, Y. Kuo, Y. Zhao, F. Qiu, and A. Kaufman. Visual simulation of thermal fluid dynamics in a pressurized water reactor. Submitted, 2008.
2. K. Petkov, F. Qiu, Z. Fan, A. Kaufman, and K. Mueller. Efficient flow simulation with LBM on face-centered cubic lattices. In preparation, 2008.
3. Z. Fan, F. Qiu, and A. Kaufman. Zippy: A framework for computation and visualization on a GPU cluster. *Computer Graphics Forum (Eurographics)*, 27(2):341-350, 2008.

4. Z. Fan, C. Vetter, C. Guetter, D. Yu, R. Westermann, A. Kaufman, and C. Xu, Optimized GPU implementation of learning-based non-rigid multi-modal registration. *SPIE Medical Imaging*, no. 6914-107, 2008.
5. F. Qiu, F. Xu, Z. Fan, N. Neophytos, A. Kaufman, and K. Mueller, Lattice-based volumetric global illumination. *IEEE Transactions on Visualization and Computer Graphics*, 13(6): 1576–1583, 2007.
6. Y. Zhao, F. Qiu, Z. Fan, and A. Kaufman. Flow simulation with locally-refined LBM. *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 181–188, 2007.
7. Y. Zhao, Y. Han, Z. Fan, F. Qiu, Y. Kuo, A. E. Kaufman, and K. Mueller. Visual simulation of heat shimmering and mirage. *IEEE Transactions on Visualization and Computer Graphics*, 13(1):179– 189, 2007.
8. S. Li, Z. Fan, X. Yin, K. Mueller, A. Kaufman, X. Gu, Real-time reflection using ray tracing with geometry field, *Eurographics Short Papers*, pages 29–32, 2006.
9. Z. Fan, Y. Zhao, A. Kaufman, and Y. He. Adapted unstructured LBM for flow simulation on curved surfaces. *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 245–254, 2005.
10. W. Li, Z. Fan, X. Wei, and A. Kaufman. Flow simulation with complex boundaries, Chapter 47, pages 747–764. in Matt Pharr (ed.), *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, 2005.
11. F. Qiu, Y. Zhao, Z. Fan, X. Wei, H. Lorenz, J. Wang, S. Yoakum-Stover, A. Kaufman, and K. Mueller. Dispersion simulation and visualization for urban security. *IEEE Visualization*, pages 553–560, 2004.
12. Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. *ACM/IEEE Supercomputing Conference*, 12 pages in CD, 2004.
13. Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for scientific computing and large-scale simulation. *ACM Workshop on General-Purpose Computing on Graphics Processors*, pages C–32, 2004.
14. Z. Fan, W. Li, X. Wei, and A. Kaufman. GPU-based voxelization and its application in flow modeling. *ACM Workshop on General-Purpose Computing*

on Graphics Processors, pages C–7, 2004.

15. X. Wei, Y. Zhao, Z. Fan, W. Li, F. Qiu, S. Yoakum-Stover, and A. Kaufman. Lattice-based flow field modeling. *IEEE Transactions on Visualization and Computer Graphics*, 10(6):719–729, 2004.
16. Z. Fan, M. Oliveira, C. Ma, and A. Kaufman, Sketch-based interface for collaborative design, *Eurographics Workshop on Sketch-Based Interfaces and Modeling*, pages 143–150, 2004.
17. Y. Zhao, X.Wei, Z. Fan, A. Kaufman, and H. Qin. Voxels on fire. *IEEE Visualization*, pages 271–278, 2003.
18. Z. Fan, C. Ma, and M. Oliveira, *A sketch-based collaborative design system*, *Brazilian Symposium on Computer Graphics and Image Processing*, pages 125–131, 2003.
19. X. Wei, Y. Zhao, Z. Fan, W. Li, S. Yoakum-Stover, and A. Kaufman. Blowing in the wind. *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 75–85, 2003.

Chapter 1

Introduction

1.1 Motivation

The graphics processing units (GPUs) are the processors equipped on commodity 3D graphics cards for accelerating raster-based rendering. Nowadays, computer games and virtual environments require complicated 3D meshes with huge amount of graphics data to be rendered at real-time speed (no less than 15 frames per second). This has been beyond the power of any software rendering. The GPU is the hardware accelerator specially designed to fulfill this task.

Figure 1.1 shows a comparison of the growth of the computational power of GPUs and CPUs from 2003 to 2008. During this period, the GPU computational power has increased 73 times. On average, it has doubled every 11 months. In comparison, the CPU computational power (doubled every 20 months on average in Figure 1.1) has increased 10 times. The computational power of today's GPUs has surpassed that of CPUs by an order of magnitude. The high GPU computational power and its fast growth are made possible by the explicit data parallelism in raster-based rendering. Contiguous data elements, such as vertices, fragments, and geometry primitives, are subject to the same operations in parallel. These operations are relatively simple and usually have only a small number of branches. Data elements are processed independent of each other: the results computed for one element are not needed by other elements in the same rendering pass. It is therefore relatively easy to predict the memory access and to achieve efficient memory

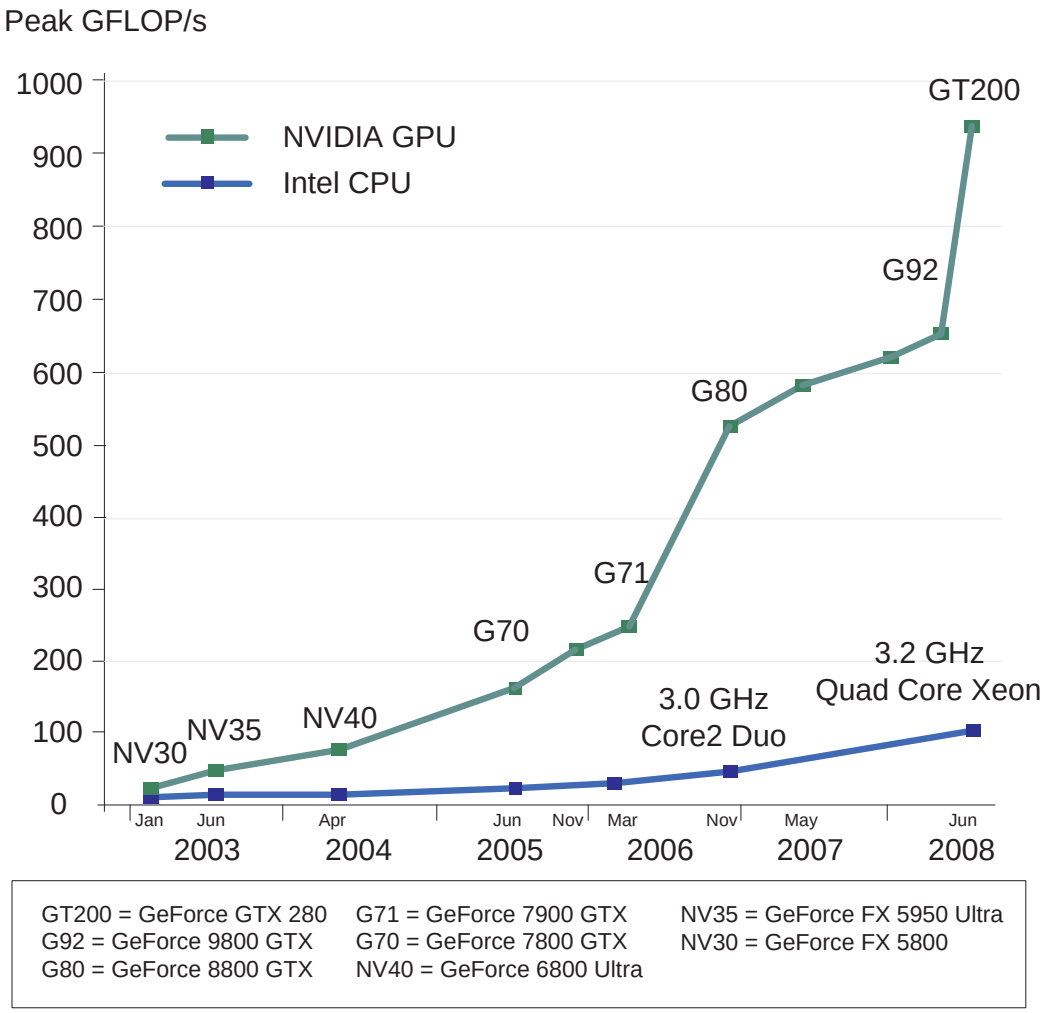


Figure 1.1: A comparison of the growth of the computational power of GPUs (green) and CPUs (blue). (The figure is adapted from Figure 1–2 of NVIDIA CUDA programming guide [114].)

pre-fetching. In addition, the independence eliminates the possibility of read-write hazards.

Because of the data parallelism in raster-based rendering, GPUs can devote a much larger portion of transistors to computation and a much smaller portion to control flow and cache than CPUs [114]. Thus, a modern GPU has smaller and simpler cores than the CPU cores, but it has many more cores than the CPU. For example, an NVIDIA GeForce GTX 280 has 240 thread processors that collectively

deliver a theoretical peak performance of 933 GFLOPS.

To match its high computational power, the GPU is coupled with a very fast on-board texture memory, whose bandwidth is an order of magnitude higher than a PC system memory. For example, a GeForce GTX 280 has a 142 GB/second bandwidth texture memory. Furthermore, the GPU exploits a stream processing model. The data stream through the computation kernels. It is possible to hide the latency of memory access with calculations.

The GPU is also becoming more and more flexible and programmable. High-level shading languages, such as GLSL [123], Cg [97], and HLSL [131], have been incorporated into the graphics pipeline. These languages allow the graphics developers to design customized vertex, geometry, and fragment shaders to replace the previously fixed transformation and lighting calculations. At the same time, the booming market for computer games drives high volume sales of graphics cards, which keeps GPU prices low compared to other specialty hardware accelerators.

Because of the computational power and programmability of modern GPUs, the research of general-purpose computation using GPUs (GPGPU) [4, 115] has become an active area of research. Researchers have implemented a wide range of applications on the GPU for acceleration. Examples include physics-based simulation, linear algebra operations, fast Fourier transform, image/volume processing and reconstruction, registration, volume rendering, flow visualization, database operations, etc. While there have been many GPGPU examples on the single GPU, in this dissertation, we present efficient methods to use GPU clusters for simulation and visualization in visual applications.

By extending GPGPU to GPU clusters, we are able to achieve further acceleration of applications and support an increase in the problem size. Due to the high performance/cost ratio and the fast performance growth of GPUs, we believe GPU clusters are promising for future high performance computing (HPC). Major GPU vendors have started to target the HPC market. NVIDIA has announced the Tesla S870, a 1U rack-mount server with 4 GPUs that are dedicated to computation. AMD has announced the FireStream 9170 which supports double-precision floating point computation. The increasing attention of the major GPU vendors to HPC will only make GPU clusters even more promising. GPU clusters will facilitate the online visualization: because the simulation results reside on the GPU texture

memories, the results can be directly visualized with the GPUs, which will enable immediate visual feedbacks in simulations.

We use the visual simulation of flows as our primary application and our focus is on the lattice Boltzmann method (LBM) [136]. The LBM is a relatively young computational fluid dynamics (CFD) method. Unlike the traditional macroscopic CFD methods that solve the Navier-Stokes differential equations, the LBM models the dynamics of fluid particle distribution functions at a mesoscopic level, a more fundamental level. The LBM is attractive to us for several reasons. First, it is easy to handle the complex-shaped, static or moving boundaries in the simulation. Second, it is also easy to model the interactions of multi-component and multi-phase fluids. These two features make the LBM attractive to graphics applications, where interactions are usually important. Most importantly, the LBM applies a set of local rules to discrete lattice sites. This computation is explicitly data parallel, hence can greatly take advantage of the GPU acceleration.

1.2 Contributions

Figure 1.2 shows the main contributions of this dissertation and their interdependencies. These contributions are described in five chapters. In Chapter 3, we begin with the elementary task: an LBM implementation on a single GPU. Directly mapping the computation from the CPU to the GPU does not necessarily achieve acceleration. Therefore, we present various optimization techniques to help the GPU implementation gain substantial speedups over the CPU implementation. In Chapter 4, we extend our method of regular simulation domain to an adapted unstructured LBM algorithm for flow simulation on 3D triangular surfaces of arbitrary topology and present how this irregular simulation domain is handled on the GPU. In Chapter 5, we extend the LBM implementation to a GPU cluster. On the GPU cluster architecture, the network bandwidth and latency and the overheads for transferring data between GPU texture memories and system memories tend to be the bottlenecks. We present efficient methods to manage the communication among the multiple GPUs. In Chapter 6, we go further to present an LBM implementation of an irregular-shaped simulation domain on the GPU cluster. The method is applied to the visual simulation of thermal fluid dynamics in a pressurized water reactor of

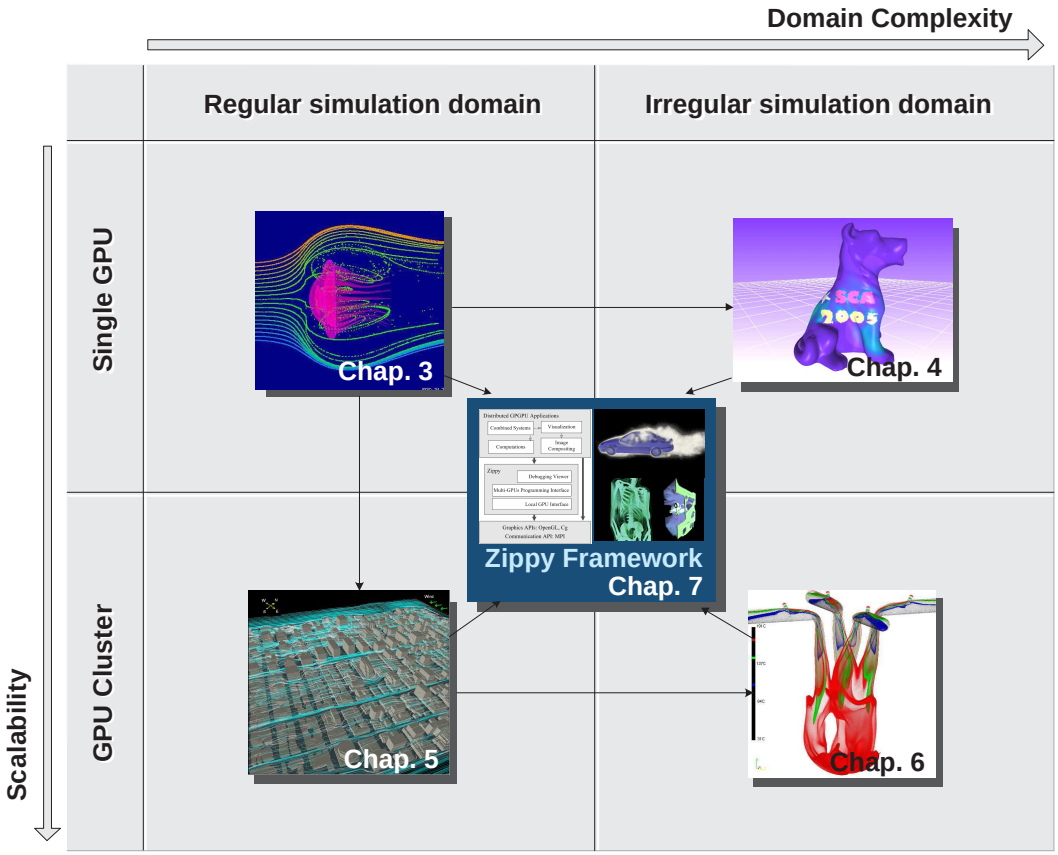


Figure 1.2: The main contributions of this dissertation and their interdependencies. (The arrows show the interdependencies.)

a nuclear power plant. Motivated by the previous implementations, in Chapter 7, we study the programming model of the GPU cluster. Currently, there has not been any mature programming toolkit for GPU clusters. Programming a GPU cluster is low-level and difficult. The complexity of programming also makes the performance optimization difficult. We present Zippy, a general framework to simplify the programming of general-purpose computation and visualization applications on GPU clusters.

The following list describes these contributions in more details.

LBM on GPU: We have developed an optimized LBM implementation with complex boundary conditions on a GPU [87]. We have developed a depth-peeling based voxelization algorithm [28, 87] for on-the-fly boundary generation. We

have also used this implementation to model natural phenomena, such as blowing of bubble and feather in the wind [150, 151], fire [164], smoke [163], and heat shimmering and mirage [162].

Adapted Unstructured LBM on GPU: We have developed a novel adapted unstructured LBM algorithm on the GPU, which can effectively model fluid dynamics on 3D curved surfaces of arbitrary topology [35].

LBM on GPU Cluster: We have proposed to use GPU clusters for high performance computing and have implemented the LBM on a 32-node GPU cluster as an example application [32, 33]. To the best of our knowledge, we are the first to develop a scalable GPU cluster for high performance scientific computing and large-scale simulation. We have used our GPU cluster LBM implementation for modeling of contaminant dispersion in urban environments [121].

LBM of Irregular-Shaped Simulation Domain on GPU Cluster: We have developed a visual simulation system [27] on a GPU cluster for modeling thermal fluid dynamics inside a pressurized water reactor of a nuclear power plant. We classify the irregular-shaped simulation domain into nonempty and empty cells and apply a packing technique to organize the nonempty cells for efficient computation and storage. To our knowledge, this is the first system that combines 3D simulation and visualization for analyzing pressurized thermal shock risk in a pressurized water reactor.

A General Framework for GPU Clusters: We have developed Zippy [31], a general framework for programming parallel visualization, graphics, and computation modules on GPU clusters. We have proposed a Global Array (GA) + Stream processing programming model. It exposes two characteristics important to high performance—non-uniform memory access (NUMA) and two-level parallelism hierarchy—and hides other architecture details. This model combines the best features of shared-memory and message passing. We have developed three example applications using Zippy: sort-last volume rendering, Marching Cubes isosurface extraction and rendering, and LBM flow simulation with online visualization.

Chapter 2

Background

2.1 GPU

2.1.1 Programmable Graphics Pipeline

The raster-based rendering on graphics hardware involves a sequence of processing stages that run in parallel and in a fixed order, known as the graphics pipeline (see Figure 2.1). The first stage of the pipeline is the *vertex processing*. The input to this stage is the 3D polygonal mesh. The 3D world coordinates of each vertex of the mesh are transformed to a 2D screen position. Lighting and texture coordinates associated with each vertex are also evaluated. In the second stage, *rasterization*, the transformed vertices are grouped into rendering primitives, such as triangles. Each primitive is scan-converted, generating a set of fragments in screen space. Each fragment stores the state information needed to update a pixel. In the third stage, called the *fragment processing*, the texture coordinates of each fragment are used to fetch colors of the appropriate texels (texture pixels) from one or more textures. Mathematical operations may also be performed to determine the ultimate color for the fragment. Finally, various raster operations (e.g., depth test, stencil test, and alpha blending) are conducted to determine whether and how the fragment is used to update a pixel in the frame buffer.

Early graphics cards for commodity PCs were fixed-function hardware. In 1999, NVIDIA firstly introduced the term *GPU* with its GeForce 256. In addition to the vertex processing capability, T&L (Transform and Lighting), the GeForce

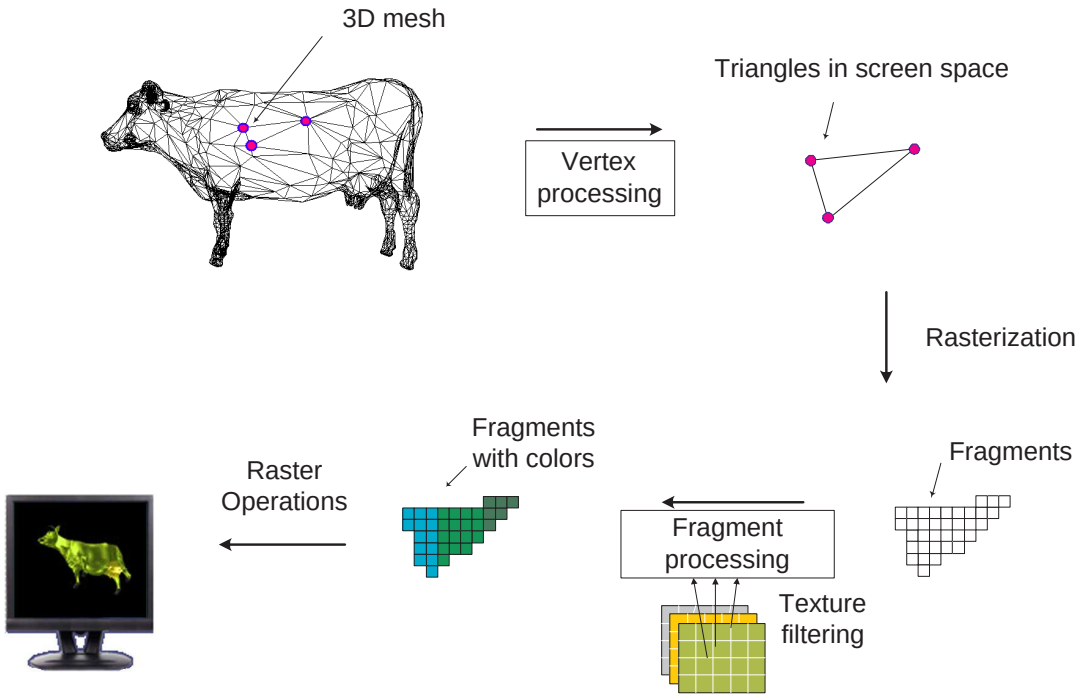


Figure 2.1: A simplified illustration of the graphics pipeline.

256 provides register combiners that allow the developers to configure the functionality of the graphics hardware. In 2000, Microsoft released DirectX 8.0, which allows developers to program *vertex shaders* and *fragment shaders*. The shader is a set of instructions that specifies the customized rendering effects. The introduction of shaders started the history of programmable graphics pipeline. Since then, shader specification has evolved from the early restrictive Shader Mode 1.x to today's much more flexible Shader Model 4.0. High level shading languages, GLSL [123], Cg [97], and HLSL [131], have been available to GPU programmers. Further, a new stage, *geometry shader*, has been added into the graphics pipeline. It provides a new functionality of dynamically generating graphics primitives on the GPU. The readers are referred to Krüger [76] for a history of the programmable graphics pipeline.

To process large graphics data sets at real-time speed, modern GPUs are data driven and emphasize data parallelism. For instance, NVIDIA GeForce 7900 GTX has 8 vertex processors and 24 fragment processors that compute concurrently.

These processors support 4-dimensional vectors (representing the xyzw homogeneous coordinates or the RGBA color channels) and a 4-component vector floating point SIMD instruction set for computation. NVIDIA GeForce 8800 GTX and GeForce 9800 GTX employ a unified shader model. They have 128 scalar processors that are dynamically allocated to vertex, geometry, and fragment processing. NVIDIA GeForce GTX 280 released this year has 240 scalar processors.

2.1.2 GPGPU

This section briefly surveys the research in GPGPU. The interested readers are referred to Owens et al. [115] and the *GPU Gems* books [112, 118] for further surveys. A description of the early GPGPU history and additional information can be found on the GPGPU website [4].

Physics-based visual simulations have been developed on the GPU. Harris et al. [56] have implemented the coupled map lattice (CML), a discrete-time discrete-space method, on the GPU. Researchers have further proposed several GPU-based sparse-matrix solvers for partial differential equations (PDE) and have used them to solve the Navier Stokes equations in fluid dynamics simulations. Krüger and Westermann [79] have implemented a framework for linear algebra operators and a conjugate gradient solver. Boltz et al. [8] and Goodnight et al. [47] have implemented multigrid solvers. Harris et al. [55] have developed the Jacobi and Gauss-Seidel solvers and have simulated and rendered clouds in real-time. Krüger and Westermann [80] have implemented a method that combines physics-based 2D simulation with non-physics based 3D simulation. Crane et al. [18] have implemented a real-time smoke simulation that is based on MacCormack Advection Scheme [125]. In addition to these Navier-Stokes solvers, Kolb et al. [75] and Harada et al. [54] have implemented particle-based fluid simulations on the GPU.

Li et al. [89] in our group were the first to implement the LBM simulation on the GPU. This GPU implementation has used the restrictive register combiners of early GPUs and has been limited to fixed-point computations. We extend it to the programmable GPUs and to the GPU clusters.

Flow visualization has benefited from GPU acceleration. Heidrich et al. [57]

have used the pixel texture to compute the line integral convolution (LIC) for visualizing vector fields. Jobard et al. [69] have implemented the 2D Lagrangian-Eulerian advection using textures. Weiskopf et al. [152, 153] have extended Jobard et al.'s method to 3D flows by using offset and dependent textures. Van Wijk [146] has developed an image based flow visualization (IBFV) algorithm on the GPU. Krüger et al. [77] have proposed a particle system on the GPU for 3D flow visualization.

Cabral et al. [11] have used the texture hardware on SGI graphics workstations for slice-based volume rendering. Westermann and Ertl [154] have used the OpenGL alpha test to skip empty space in volume rendering. Engel et al. [23] have proposed pre-integrated volume rendering algorithm on the GPU, which can achieve high quality images for even low resolution volume data. Krüger and Westermann [78] have proposed a ray-casting method that uses early-Z test to skip occluded voxels in ray-level. Li et al. [88] have proposed sub-volume level empty-space skipping and voxel-level occlusion culling and clipping algorithms. Hong et al. [60] have proposed a hybrid volumetric ray-casting method that uses both the CPU and GPU. Neophytou et al. [110] have proposed a volume splatting rendering algorithm with elliptical radial basis functions (RBFs) on the GPU. More information of the GPU-accelerated volume rendering can be found in Engel et al.'s book [22].

Trendall and Stewart [142] have employed the GPU to accelerate the calculation of refractive caustics. Carr et al. [14] and Purcell et al. [119] have implemented ray-tracing on the GPU. Carr et al. [15] have proposed a GPU-based ray-tracing algorithm using the Geometry Images [51]. Li et al. [86] have proposed a ray-tracing algorithm using geometry field, a 4D lookup table. Horn et al. [63] have proposed the k-D tree implementation on the GPU for ray-tracing. Qiu et al. [120] have presented a lattice-based volumetric global illumination algorithm. Zhou et al. [165] have proposed a compensated ray marching algorithm for real-time smoke rendering.

Many other important applications have been developed on the GPU, such as computed tomographic (CT) volume reconstruction [107, 157], volume segmentation [84], volume registration [34], Voronoi diagram computation [58], image processing [124], collision detection [49], dense matrix multiplication [68], database operations [48], and so on. In addition, Sengupta et al. [127] have proposed a scan

algorithm on the GPU, which allows the computation to generate a variable-sized output for each input data element. Fatahalian et al. [38] have shown that the lack of high bandwidth access to cache data may impair the GPU performance in some applications. Fang and Mueller [158] have discussed that using the build-in hard-wired graphics pipeline components, such as bilinear interpolation in rasterizer, can achieve superior performance than merely using the GPU as a multi-core processor. Göddeke et al. [46] have presented a way to emulate double precision arithmetics with single precision operations on the GPU. Because the emulation increases the operation count more than 10 times, they have further used mixed double and single precision operations for finite element method (FEM) computation.

Programming languages, such as Brook [9], Sh [102], PeakStream, and RapidMind, have been proposed for GPGPU programming. They abstract the GPU as a stream coprocessor. NVIDIA CUDA [114] is a new development environment that gains significant attention in the recent years. Using CUDA, the user does not deal with the low-level details of the graphics pipeline and textures anymore. Instead, the user uses the C language and can conveniently access a linear memory on the GPU. CUDA also exposes to the user several new hardware features for performance optimization, such as hardware-supported scatter and Parallel Data Cache.

2.1.3 Multi-Core Processors

Processor parallelism is becoming more ubiquitous. Multi-core architectures are populating HPC systems. Beside the GPU, there have been the multi-core CPU, Cell BE, and ClearSpeed, which are described below and compared in Table 2.1.

Intel and AMD have both released 4-core CPUs. Many-core CPUs (with 8 or more cores) may appear in 2009. The CPU cores are more complex than the GPU cores. Compared with the GPUs, the CPUs are more flexible and can support a wider range of applications. Especially, some programs that require extremely complicated control flows, such as an operating system and a word processor, have to rely on the CPUs. On the other hand, current GPUs have many more cores than the CPUs and devote a larger percentage of transistors to floating point operations. Therefore, the GPUs provide higher compute parallelism and deliver better performance than the CPUs for certain applications. The LBM computation is an example

Table 2.1: GPUs and other multi-core processors. (Thermal Design Power (TDP) in the last column is the maximum amount of thermal power that the processors need to dissipate when running applications.)

	Number of Cores	**Clock** (GHz)	**Theoretic GFLOPS**		**TDP (Watt)**
			32-bit	**64-bit**	
GeForce GTX 280	240	1.3	933	117	236
GeForce 9800 GTX	128	1.7	653	N/A	156
Quad Core Xeon	4	3.2	102	51	150
Cell BE	8+1	3.2	205	14.6	110
ClearSpeed Advance e710	192	0.25	96	96	25

of this kind, as its computational kernels are local and linear. As both CPUs and GPUs have their advantages and disadvantages, we believe that a hybrid CPU+GPU approach will be promising for future research. Whether the CPU and GPU will be loosely coupled or tightly coupled is arguable and unclear yet.

Cell Broadband Engine Architecture (Cell BE) is a microprocessor jointly developed by Sony, Toshiba, and IBM (STI). It has been used in Sony's PlayStation 3 and IBM's blade servers. The Cell BE is a hybrid multi-processor chip. It currently has one power processing element (PPE) and eight synergistic processing elements (SPEs). The PPE and SPEs are linked together by an internal element interconnect bus (EIB), whose bandwith is 204.8 GB/s. The SPEs are designed for vectorized floating point code execution and handle most of the computation workload. The PPE acts as the controller for the SPEs. The current theoretic peak performance of Cell BE (in 32-bit single precision) is between those of the CPU and the GPU.

ClearSpeed Advance e710 has 192 processing elements running at only 250 MHz. One of its advantages is the low power consumption and heat generation. The thermal design power (TDP) of the ClearSpeed Advance e710 is only 25 Watt, significantly lower than the TDP of other processors. Another advantage is that it delivers high 64-bit floating point computational power. However, compared with

commodity GPU, CPU, and Cell BE, ClearSpeed currently has a lower volume of sales and a higher cost.

2.2 Physics-Based Visual Simulation of Fluids

2.2.1 Navier-Stokes Solvers

Physics-based modeling is the key to achieving realistic simulations of fluids, such as gas and liquids, and fluid driven natural phenomena, such as smoke, fire, and clouds. The fundamental equations for modeling fluid dynamics are the Navier-Stokes equations:

$$\nabla \cdot \mathbf{u} = 0 \tag{2.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + f \tag{2.2}$$

where $\mathbf{u}$ is the velocity field, p is the pressure, ν is the kinematic viscosity, ρ is the density, and f is the external force. The first equation conserves mass and the second equation conserves momentum. These two equations are supplemented with boundary conditions.

In computer graphics, finite difference is the most frequently used method for solving the Navier-Stokes equations. The spatial domain is discretized into small cells to form a volume grid, either regular or irregular. Finite difference operators are evaluated on the grid points. Foster and Metaxas [42, 43] have presented an explicit scheme for solving the 3D Navier-Stokes equations. They have simulated liquid and smoke with realistic swirling motions. A main problem with the explicit solver is that the simulation is unstable when large time steps are used. To address this, Stam [132] has devised an unconditionally stable method, called Stable Fluids. The method is based on the semi-Lagrangian advection and an implicit scheme. The implicit scheme uses a sparse matrix solver, such as conjugate gradient or Jacobi, to solve a set of simultaneous finite difference equations. The disadvantage of the Stable Fluids, however, is that the semi-Lagrangian advection has large numerical dissipation and diffusion. Fedkiw et al. [39] have extended this implicit scheme by introducing *vortex confinement* and high-order interpolation. The former adds to the

simulation plausible small scale rolling features and the latter increases the advection accuracy. Combining Navier-Stokes solvers and the level-set based free surface tracking, researchers have developed realistic visual simulations of liquids [24, 41] and fire [111]. Two new advection schemes, the Back and Forth Error Compensation and Correction (BFECC) [71] and the MacCormack method [125], have been proposed to reduce the advection dissipation and diffusion. Both schemes use error estimation to correct the computed data, but MacCormack uses fewer passes of computation than BFECC.

Carlson et al. [13] have developed a two-way coupling method to simulate the interaction between fluids and rigid solid objects. Guendelman et al. [52] have presented a coupling method to simulate the interaction between fluids and thin objects. Hong and Kim [59] have used a multi-phase fluids method to model discontinuous fluids. Losasso et al. [93] have modeled the interactions of multiple liquids. Losasso et al. [92] have extended a Navier-Stokes solver from regular grids to octree structures. Irving et al. [67] have presented a method that combines 2D and 3D simulations to efficiently model large bodies of waters. Kim et al. [70] have proposed a volume control method to preserve the volumes in the liquid/gas interaction and have used it to simulate bubbles. Unstructured tetrahedra meshes have also been used in visual simulations [40, 74].

2.2.2 Particle-Based Methods

Early graphics methods for modeling fluids were procedural particle systems [122, 134]. Recently, a physics-based particle method, Smooth Particle Hydrodynamics (SPH) [106], has been proposed. Mueller et al. [108] have used the SPH in interactive simulation of liquids. Adams et al. [5] have presented an adaptive sampled particles method for performance optimization. Park and Kim [116] have proposed to use vortex particles to simulate turbulent flows. Yuksel et al. [161] have introduced wave particles for fast and stable simulation of surface waves. Cleary et al. [17] have presented a particle based method for animation of bubbles in fluids.

In particle-based methods, the flow quantities are defined on the discrete particles and can be evaluated anywhere in the simulation domain by using a smoothed kernel function. Compared with traditional Navier-Stokes solvers (Section 2.2.1),

which are macroscopic and grid-based, the particle method is microscopic and grid-less. It has several advantages. First, the advection is implemented by moving the particles along the velocity field. This Lagrangian advection is not only simpler but also more accurate. Second, it is easier to handle the microscopic interactions, such as the solid/fluid interactions and the interactions of multiple fluids [109]. A weakness is that particles are less organized than a grid. The lack of topology information makes the reconstruction of liquid/gas surfaces challenging. Selle et al. [126] have presented a hybrid method to combine the best features of particle-based and grid-based methods.

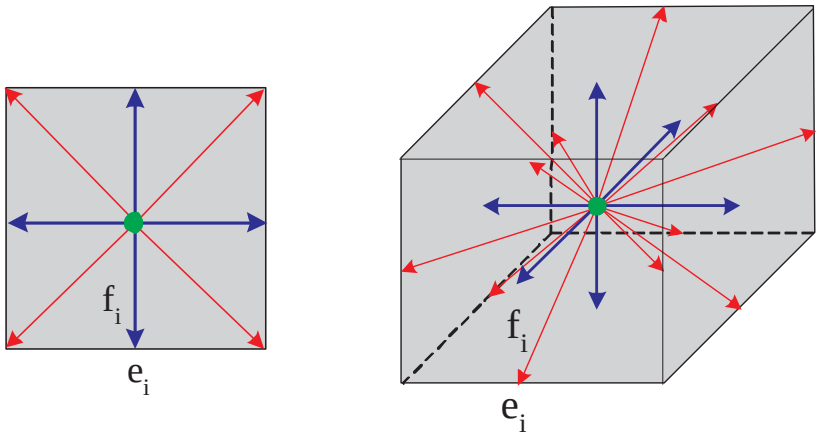
2.2.3 Lattice Boltzmann Method

The LBM [155] is a mesoscopic method. The Boltzmann equation describes how particle distributions move and collide with each other. The particle distribution is not a discrete particle. Instead, it is a probability of the presence of a particle with a given velocity at a given position. Therefore, it is represented by a floating point number. The LBM discretizes the Boltzmann equation on a lattice. The lattice is similar to a grid except that it discretizes not only the domain space but also the angular (directional) space. Each lattice site has a set of links pointing to the directions of neighboring sites. The particle distributions move along the lattice links and collide at the lattice sites. The aggregated motion yields a second-order accurate solution to the Navier-Stokes equations.

Figure 2.2(a) shows a 2-dimensional 9-velocity lattice model, called D2Q9. The 9 velocity vectors include the zero velocity vector, $\vec{e}_0$, and 8 velocity vectors pointing to neighboring lattice sites, $\vec{e}_1$ through $\vec{e}_8$. Associated with each $\vec{e}_i, (i = 0, 1, .., 8)$ is a scalar value, the particle distribution function f_i. Similarly, Figure 2.2(b) shows a 3-dimensional 19-velocity lattice model, D3Q19. The green point at the center represents the zero velocity. The blue arrows represent velocity vectors pointing to nearest neighbors and the red arrows represent velocity vectors pointing to second-nearest neighbors.

The macroscopic fluid density ρ and fluid velocity $\vec{\mathbf{u}}$ are given by

$$\rho(\vec{r},t) = \sum_i f_i(\vec{r},t), \quad \vec{\mathbf{u}}(\vec{r},t) = \frac{1}{\rho(\vec{r},t)} \sum_i f_i(\vec{r},t)\vec{e}_i, \tag{2.3}$$



(a) (b)

Figure 2.2: The LBM lattice cells: (a) a D2Q9 lattice cell, and (b) a D3Q19 lattice cell. (Each velocity vector $\mathbf{e}_i$ is associated with a particle distribution f_i.)

where $\vec{r}$ is the position of the lattice cell and t is the simulation time-step. The LBM explicitly evolves in a two-step process of collision and ballistic streaming,

$$\textbf{Collision} \quad f_i(\vec{r},t^+) = f_i(\vec{r},t) - \tfrac{1}{\tau}(f_i(\vec{r},t) - f_i^{eq}(\vec{r},t)), \tag{2.4}$$

$$\textbf{Streaming} \quad f_i(\vec{r}+\vec{e}_i,t+1) = f_i(\vec{r},t^+), \tag{2.5}$$

where the constant τ represents the relaxation time[1] determined by the kinematic viscosity ν of the fluid ($\tau = 3\nu + \frac{1}{2}$), the notation $f_i(\vec{r},t^+)$ denotes the post-collision particle distribution function, and f_i^{eq} represents the local equilibrium particle distribution function which is given by

$$f_i^{eq}(\rho,\vec{\mathbf{u}}) = \rho(A_q + B_q(\vec{e}_i\cdot\vec{\mathbf{u}}) + C_q(\vec{e}_i\cdot\vec{\mathbf{u}})^2 + D_q(\vec{\mathbf{u}}\cdot\vec{\mathbf{u}})). \tag{2.6}$$

Here, A_q through D_q are constant coefficients specific to the chosen lattice or sub-lattice. The local and linear rules yield the Navier-Stokes equation for an incompressible fluid with second-order accuracy in both time and space.

[1] In this dissertation, unless otherwise specified, fluid properties are described in lattice units. In other words, we focus on dimensionless numbers rather than actual values of physical properties. The interested readers are referred to Wei et al. [149] for rules of converting numbers from lattice units to physical units.

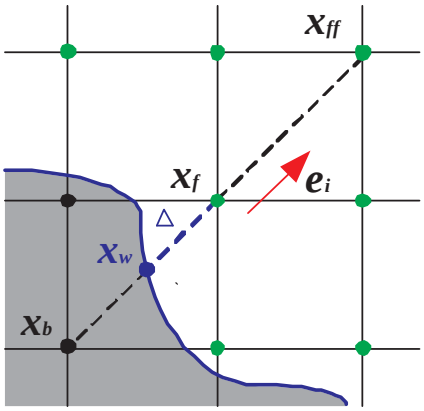


Figure 2.3: Curved boundary condition for the LBM (adapted from [104]).

Mei et al. [104] have developed a curved boundary condition for the LBM. As indicated in Figure 2.3, the particle distribution for a node, $\mathbf{x}_f$, in the fluid adjacent to the boundary, is streamed from its neighbors. Thus, a fictitious particle distribution, $f_i(\mathbf{x}_b,t)$, is defined on the node $\mathbf{x}_b$ which lies just inside the object boundary. The fraction of the link that is intersected by the boundary in the fluid region is denoted by $\Delta = |\mathbf{x}_f - \mathbf{x}_w|/|\mathbf{x}_f - \mathbf{x}_b|$. The post-collision value $f_i(\mathbf{x}_b,t^+)$ is given by:

$$f_i(\mathbf{x}_b,t^+) = (1-\chi)f_i(\mathbf{x}_f,t) + \chi f_i^*(\mathbf{x}_b) + 6A_q\rho\mathbf{e}_i\cdot\mathbf{u}_w \tag{2.7}$$

where,

$$f_i^*(\mathbf{x}_b) = \rho(A_q + B_q\mathbf{e}_i\cdot\mathbf{u}_{bf} + C_q(\mathbf{e}_i\cdot\mathbf{u}_f)^2 - D_q(\mathbf{u}_f)^2) \tag{2.8}$$

and for $\Delta \geq 1/2$,

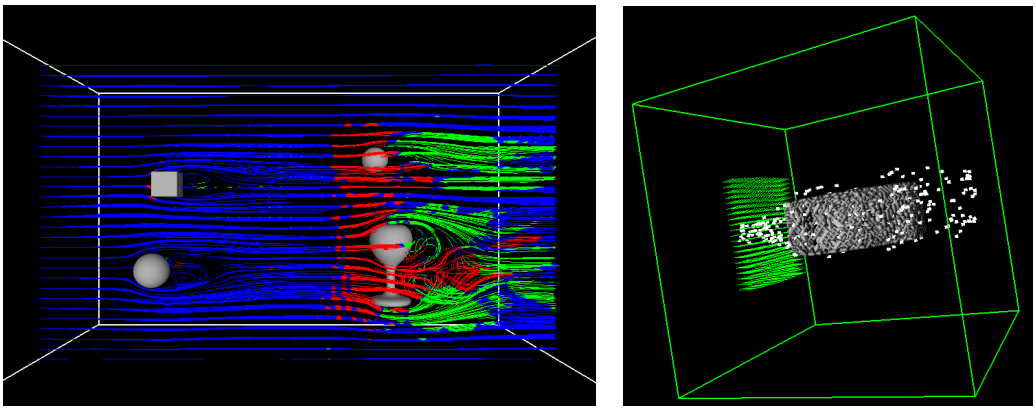
$$\mathbf{u}_{bf} = (1-\frac{3}{2\Delta})\mathbf{u}_f + \frac{3}{2\Delta}\mathbf{u}_w \quad \text{and} \quad \chi = \frac{2\Delta-1}{\tau+1/2} \tag{2.9}$$

while for $\Delta < 1/2$,

$$\mathbf{u}_{bf} = \mathbf{u}_{ff} \quad \text{and} \quad \chi = \frac{2\Delta-1}{\tau-2}. \tag{2.10}$$

Note that $\mathbf{u}_{bf}$ represents the virtual speed of the boundary node $\mathbf{x}_b$ in terms of the fluid velocity, $\mathbf{u}_f$ at node $\mathbf{x}_f$, $\mathbf{u}_{ff}$ at node $\mathbf{x}_{ff}$, and the boundary velocity, $\mathbf{u}_w$ at $\mathbf{x}_w$.

Thus, we see that the rule (which has second order accuracy and good stability) accounts for both arbitrarily shaped and moving boundaries. Figure 2.4 shows two snapshots of our simulation with curved boundaries.



(a) (b)

Figure 2.4: Our implementation of the curved boundary condition in two examples: (a) flow passing four objects, and (b) flow passing a porous object.

The LBM was introduced to the graphics community by Wei et al. [148, 149, 151] in our group for modeling the fire, smoke, and wind. Zhao et al. [162] have used a Hybrid Thermal LBM (HTLBM) [81] for the visual simulations of heat shimmering and mirage. Zhao et al. [163] have used the multi-resolution LBM for the adaptive resolution smoke simulation. Thürey and Ruede [141] have simulated free surface fluids with the LBM. Chu and Tai [16] have used the 2D LBM to simulate ink dispersion in absorbent paper. Zhu et al. [166] have introduced a two-fluid lattice Boltzmann method for simulating mixtures of miscible fluids.

Compared with the grid-based Navier-Stokes solvers, the LBM has several advantages. The advection in the LBM is linear and therefore has less numerical diffusion and dissipation. The handling of complex shaped boundaries is also easier. The discretization of the angular space allows complex shaped boundaries to be accurately represented by the locations of the intersections of boundary surfaces with lattice links. As a mesoscopic method, the LBM can also easily handle the solid/fluid interactions and the interplays of multi-component fluids or multi-phase fluids. All the computation operations are local and linear. The explicit parallelism of the LBM makes it suitable for GPU and GPU cluster implementations.

In contrast, the Navier-Stokes solvers require the sparse matrix solvers, which are less parallelizable and often become the bottleneck. Compared with the particle-based methods, the LBM organizes the data in a regular lattice and hence preserves the advantages of the grid-based methods. A disadvantage of the LBM is that it is unstable when modeling highly turbulent flows. A multi-relaxation-time LBM (MRTLBM) [20] increases the stability but has not completely solved the problem.

Chapter 3

LBM on GPU

In this chapter, we present a GPU implementation of the LBM flow simulation. In the LBM, the computation of each lattice site at every time step depends solely on the properties of the site itself and the neighboring sites at the previous time step. The computation is local, linear, and explicitly parallel. Although mapping the computation to GPU programs is straightforward, to achieve high performance is a demanding task. We present various algorithm-level and machine-level optimization techniques to help the GPU implementation gain substantial speedups over the CPU implementation.

The boundaries in the flow have to be voxelized to discrete lattice sites. For moving and deforming boundaries, this voxelization ought to be repeated at every time step. We propose a fast voxelization algorithm on the GPU and use it to model the interactions between fluids and animated objects.

3.1 GPU-Based LBM Implementation

3.1.1 Algorithm Overview

To compute the LBM equations on the GPU, we divide the LBM lattice and group the particle distributions f_i into arrays according to their velocity vectors. All the particle distributions of the same velocity vector are grouped into the same array, while keeping the neighboring relationship of the original model. For a 2D model, we store the arrays as 2D textures. Figure 3.1 shows the division of a 2D

model D2Q9 according to its velocity directions.

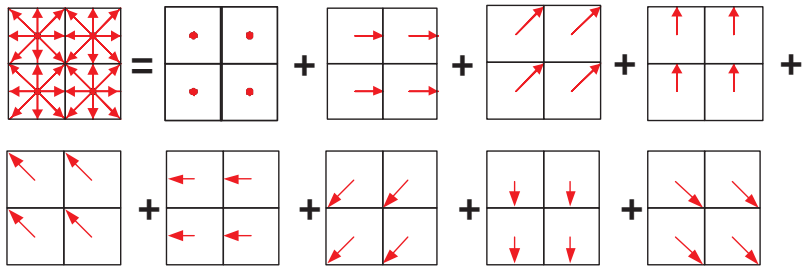


Figure 3.1: Division of the D2Q9 model, according to its velocity directions.

The division of a 3D model is similar. Each of the 19 velocity distributions f_i in D3Q19 LBM, is represented in a volume. As shown in Figure 3.2, we pack every four volumes into one stack of 2D textures (note that a fragment or a texel has 4 color components). Thus, the 19 distribution values are packed into 5 stacks of textures.

All the other variables, the density ρ, the velocity $\mathbf{u}$, and the equilibrium distributions f_i^{eq} are stored similarly in 2D textures. We then render quads mapped with those textures, and use fragment programs to compute the LBM equations.

Figure 3.3 shows the dataflow of the LBM computation on the GPU, where the textures storing lattice properties are represented by the red boxes, while the operations are represented by the blue round boxes. The textures of the particle distributions are the inputs. Density and velocity textures are then dynamically generated from the distribution textures. Next, the equilibrium distribution textures are obtained from the densities and the velocities. According to the collision and the streaming equations, new distributions are computed from the input distributions and the equilibrium distributions. Finally, we apply the boundary and outflow conditions and update the distribution textures. The updated distribution textures are then used as inputs for the next simulation step.

The LBM equations are translated to the operations of the fragment pipeline. With the GPUs becoming more and more programmable, a straightforward translation following Figure 3.3 is not difficult. However, there are usually multiple choices in mapping each equation, and different combinations of the mapping usually result in dramatic difference in performance. In the following sections, we

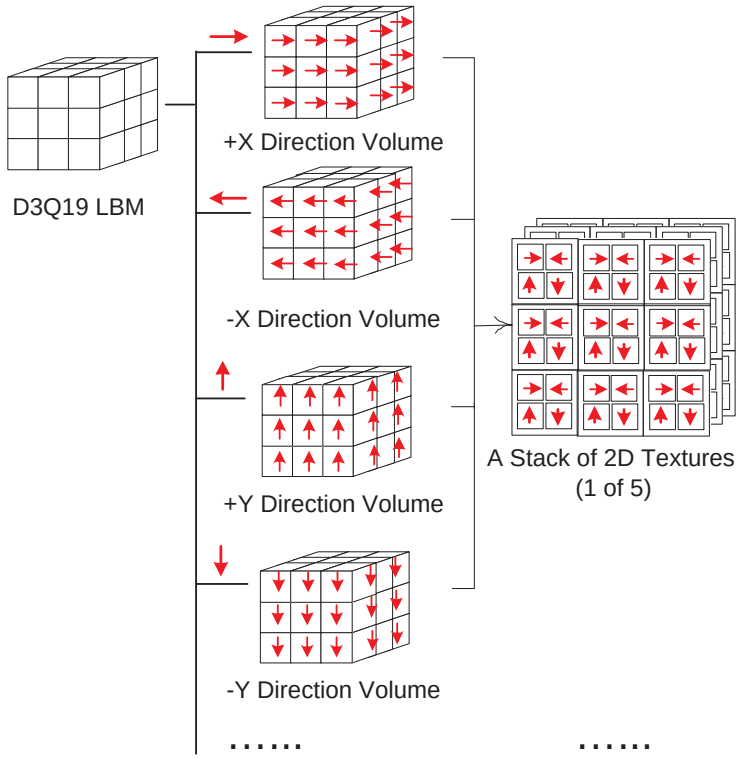


Figure 3.2: Division of the D3Q19 model. Every four direction volumes are packed into one stack of 2D textures. (This figure only shows one of the five stacks of 2D textures.)

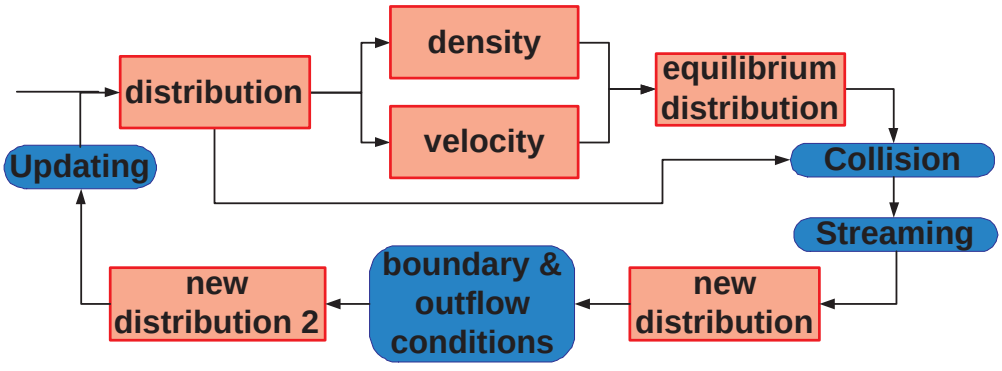


Figure 3.3: Flow chart of the LBM computation on the GPU. (Red boxes are the textures, while blue round boxes are operations.)

describe various optimization strategies that have been applied to our GPU-based LBM.

3.1.2 Packing

During the simulation, textures are updated dynamically at every step by copying from or binding to the frame buffer (or the pixel buffer). In the graphics hardware, it is most efficient to use RGBA textures. Each RGBA texel has four channels, hence can store up to four scalars or a vector with up to four components.

The first optimization we propose is packing different variables into the same texel. For example, we pack four f_is from different directions into a single RGBA texel. In addition, we attempt to pack together those variables that are involved in the same LBM equations, in order to reduce the number of textures to be activated and the number of texture fetches. This also improves the data locality, as well as the cache coherence of the textures.

Table 3.1: Packed LBM variables of the D3Q19 model

Texture	R	G	B	A
uρ	v_x	v_y	v_z	ρ
Tex0	$f_{(\ 1,\ 0,\ 0)}$	$f_{(-1,\ 0,\ 0)}$	$f_{(\ 0,\ 1,\ 0)}$	$f_{(\ 0,-1,\ 0)}$
Tex1	$f_{(\ 1,\ 1,\ 0)}$	$f_{(-1,-1,\ 0)}$	$f_{(\ 1,-1,\ 0)}$	$f_{(-1,\ 1,\ 0)}$
Tex2	$f_{(\ 1,\ 0,\ 1)}$	$f_{(-1,\ 0,-1)}$	$f_{(\ 1,\ 0,-1)}$	$f_{(-1,\ 0,\ 1)}$
Tex3	$f_{(\ 0,\ 1,\ 1)}$	$f_{(\ 0,-1,-1)}$	$f_{(\ 0,\ 1,-1)}$	$f_{(\ 0,-1,\ 1)}$
Tex4	$f_{(\ 0,\ 0,\ 1)}$	$f_{(\ 0,\ 0,-1)}$	$f_{(\ 0,\ 0,\ 0)}$	unused

Table 3.1 lists the contents of the textures packed with the variables of the D3Q19 model, including densities, velocities and particle distributions. In texture **uρ**, v_x, v_y, and v_z are the three components of the velocity stored in the RGB channels, while the density ρ is stored in the alpha channel. The rows in Table 3.1 for textures *Tex0* through *Tex4* show the packing patterns of both the particle distributions f_i and the equilibrium distributions f_i^{eq}. $f_{(x,y,z)}$ is the distribution in the direction of (x,y,z). Note that we pack distributions of the opposite directions in pairs into the same texture. There are two reasons for that. First, when handling complex or moving boundaries, neighboring distributions at opposite directions are needed to evaluate the effects on the same boundary link. Second, when programming the fragment pipeline, we typically need to pass the corresponding velocity

vector $\mathbf{e}_i$ as an argument of the fragment program. When opposite distributions are always neighbors, just two, instead of four, $\mathbf{e}_i$s are needed, while the other two are easily inferred.

3.1.3 Flat Volume

Instead of using a stack of 2D textures to represent a volume, we actually stitch the slices to create a larger 2D texture, that can be considered as a "flat" volume. Similar approaches have been reported in the literature [55, 89]. One advantage of the flat volume is the reduced number of texture switching. It also reduces the number of proxy quads. Specifically, a $W \times H \times D$ volume is stored as a $(W * d1) \times (H * d2)$ texture, where $D = d1 * d2$. We choose $d1$ to be a factor of D that is the closest to $\sqrt{D}$.

More importantly, using the flat volume is also critical in two stages of our algorithms, particle advection (Section 3.3) and voxelization and boundary nodes generation (Section 3.2.1). Without a flat representation, these two will not work. The conversion from the volume coordinates (x, y, z) to the coordinates (u, v) of the flattened texture is as follows:

$$\begin{aligned} u &= (z\%d1) * W + x \\ v &= floor(z/d1) * H + y \end{aligned} \tag{3.1}$$

where W and H are dimensions of each slice in the volume, and every set of $d1$ slices is tiled in a row along the X direction in the flat volume.

3.1.4 Streaming

According to Equation 2.5, each particle distribution having non-zero velocity propagates to a neighboring lattice point at every time step. On current GPUs, the texture fetching unit can obtain a texel at arbitrary position indicated by texture coordinates fully controlled by the fragment program. If a distribution f is propagated along vector $\mathbf{e}$, we simply fetch from the distribution texture at the position of current lattice position minus $\mathbf{e}$. Since the four channels are packed with four distributions with different velocity vectors, four fetches are needed for each fragment.

Figure 3.4 shows as an example the propagation of distribution texture *Tex1*, which is packed with $f_{(1,1,0)}$, $f_{(-1,-1,0)}$, $f_{(1,-1,0)}$, and $f_{(-1,1,0)}$. The fragment program fetches texels by adding the negative values of the velocity directions to the texture coordinates and extracting the proper color component. For example, the blue channel in texture *Tex1* is $f_{(1,-1,0)}$. After propagation, the value of the blue channel will come from a texel at relative position $(-1,1,0)$ to the fragment position. This is equivalent to translating distribution $f_{(1,-1,0)}$ in the direction of (1,-1,0) by 1.

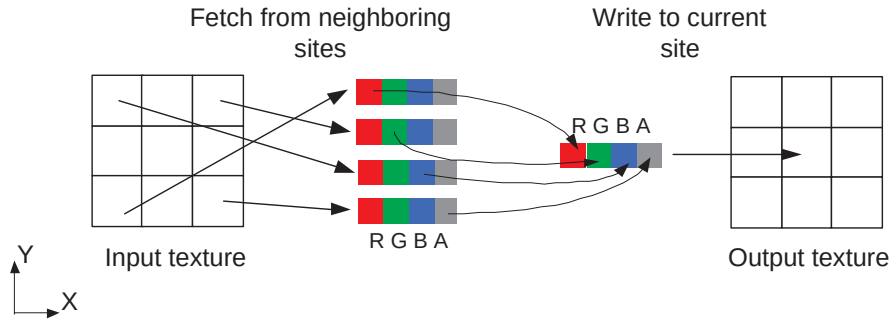


Figure 3.4: Propagation of distribution texture *Tex1*, which is packed with $f_{(1,1,0)}$, $f_{(-1,-1,0)}$, $f_{(1,-1,0)}$, and $f_{(-1,1,0)}$.

To propagate using the flat volume, for each channel, we can add the 3D position of the fragment with the negative of the corresponding $\mathbf{e}_i$, then convert it to the texture coordinate of the flat volume according to Equation 3.1 before fetching. However, this requires Equation 3.1 be executed four times per fragment. One optimization we apply is to push the coordinates conversion to the vertex level, since for each channel inside a slice, the velocity vectors are the same. We can either assign each vertex of the proxy quad with four texture coordinates containing the converted values, or generate the texture coordinates with a vertex program.

3.2 GPU-based Boundary Handling

To handle complex boundary, we need to compute the intersections of the boundary surface with all the LBM lattice links. For a static obstacle, the intersections can be pre-computed. Whereas for either a moving or deformable boundary,

the intersection positions change dynamically. The boundary description can be either continuous, such as a polygonal mesh or a higher-order surface, or discrete, such as a volume. No matter what form a boundary is originated from, the handling of the boundary conditions requires discrete boundary nodes aligned with the LBM Lattices. Even if a boundary is already in a volumetric representation, it has to be revoxelized whenever it moves or deforms.

One solution is to compute the intersection and voxelization on the CPU, and then transfer the computed volumetric boundary information from the main memory to the graphics memory. Unfortunately, both the computation and the data transfer are too time-consuming for interactive applications. Naturally, we want to accelerate the volumetric boundary generation on the GPU as well.

In the following sections, we first propose a general-purpose GPU-based voxelization algorithm that converts an arbitrary model to a Cartesian grid volume. Then, we discuss the handling of three different boundary conditions, while focusing on arbitrary complex boundaries that can move and deform. The generation of the boundary nodes of arbitrary boundaries is performed by extending our general-purpose GPU-based voxelization.

3.2.1 GPU-based Voxelization

An intuitive voxelization approach is the slicing method, which sets the near and far clip planes, so that for each rendering pass, only the geometries falling into the slab between the two clip planes are rendered [36]. This creates one slice of the resulting volume. Then, the clip planes are shifted to generate subsequent slices. Obviously, for this slicing method, the number of rendering passes is the same as the number of the slices in the volume. In most cases, the boundaries are sparse in a volume, in other words, only a small percentage of voxels are intersected by the boundary surfaces. There is no need to voxelize the "empty" space that corresponds to non-boundary voxels.

Our GPU-based voxelization avoids the slicing. Instead, we adapt the idea of depth peeling [26] used for order-independent transparency, in which the depth layers in the scene are stripped away with successive passes (see Figure 3.5). In the first rendering pass, the scene is rendered normally, and the layer of the nearest

fragments, or equivalently, voxels are obtained. From the second rendering pass, each fragment is compared with a depth texture copied from the depth buffer of the previous pass. A fragment reaches the frame buffer only if its depth value is greater than that of the corresponding pixel in the depth texture, while the ordinary depth test is still enabled. Therefore, the second pass generates the second nearest layer, and so on. The process continues until no fragment is further away than the corresponding pixel in the depth texture, which is best determined with the occlusion query extension that returns the pixel count written to the frame buffer. The number of rendering passes of the depth peeling algorithm is the number of layers plus one, which typically is significantly smaller than the number of slices.

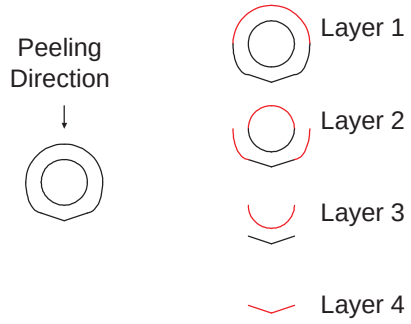


Figure 3.5: Depth peeling for GPU-based voxelization. (Red color presents the layer to be stripped away.)

When rendering order-independent transparent objects, all the layer images are blended in depth order. In contrast, in voxelization, we want the layers to be separated, which is similar to a layered depth image [129]. Assume that the maximum size along any of the major axes of the object being voxelized is D, we allocate a pixel buffer with width and height of maximum number of layers times D and number of attributes times D, respectively. Then, between different rendering passes, we translate the viewport, so that the layer images do not overlap, but be tiled as tight as possible. The pixels of those images are the attributes of the corresponding voxels. The first attribute has to be the 3D positions, while we may need other attributes depending on the application. As shown in the following pseudo-code, the peeling process is applied three times. Each time the image plane is orthogonal to one of the major axes. That is, we perform the peeling from three orthogonal

views to avoid missing voxels. As a result, some of the voxels may be rendered more than once. However, the replication does not affect the final results. The image containing the voxel attributes are then copied to a vertex array whose memory is allocated inside the video memory using extensions, such as Nvidia's *PDR* and *VAR*. Note that different type of voxel attributes are copied to different locations inside the vertex array.

```
1:  for each view direction in X, Y, Z  do
2:     layer = 0
3:     while HasMoreLayer() do
4:        for each attribute do
5:           SetViewportOrigin(attrib_id*D, layer*D)
6:           RenderScene()
7:        end for
8:        if HasMoreLayer() then
9:           UpdateDepthTexture()
10:       end if
11:       layer=layer+1
12:    end while
13: end for
```

The vertex array is essentially an array of voxel positions plus other attributes, that can generate all the boundary voxels for further processing. We may want to convert the vertex array to a flat volume by simply rendering each vertex as a point of size 1. All the vertices of the array pass through a vertex program that translates each voxel properly according to its z value using equations similar to Equation 3.1. The frame buffers for the depth peeling are initialized with some large numbers. If a pixel is not covered by any boundary voxels, then the corresponding vertex created from the pixel falls far away from the view frustum, and is clipped.

3.2.2 Periodic Boundary

In practice, the periodic boundary is actually computed during the propagation. A 2D example is shown in Figure 3.6 and the 3D case is similar. If a periodic boundary face is orthogonal to the X or Y axis, we call it in-slice periodic boundary

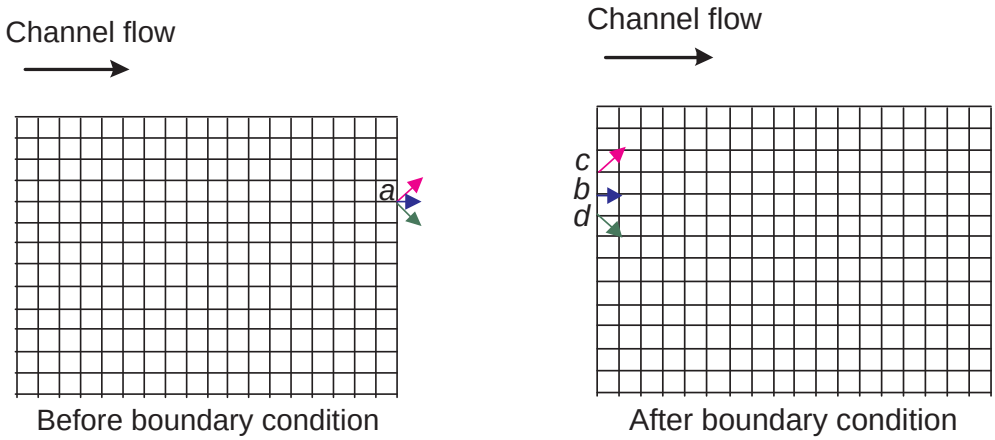


Figure 3.6: Periodic boundary condition example. In this 2D channel flow, the periodic boundary condition is applied to the left-most and right-most lattice points. (For example, the three particle distributions of lattice point *a* propagate to lattice points *b*, *c*, and *d*, respectively.)

face, since a distribution on the face is copied to the opposite side of the lattice but stays inside the same XY slice. For in-slice periodic boundary, we simply apply a modulo operation to the texture coordinates by the width or the height of each slice. Whereas for periodic boundary face perpendicular to the Z axis, which we call out-slice periodic boundary, we need to copy distribution textures of one slice to another.

A naive implementation of the in-slice periodic boundary condition is to apply the modulo operation of the texture coordinates of all the distribution texels. However, this can be very slow. For instance, in Nvidia's fragment program, the modulo is simulated by floating point division. Therefore, one optimization we use first propagates without the periodic boundary condition. Then, we draw stripes of single-texel width that only cover the boundary but with the modulo operation. In this way, the cost for computing is negligible, since the periodic boundary nodes only account for a small percentage of the lattice.

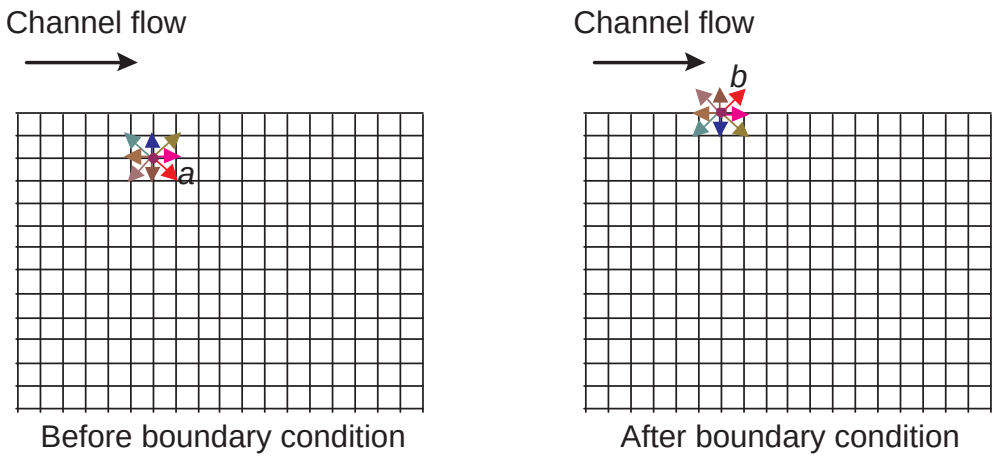


Figure 3.7: Outflow boundary condition example. In this 2D channel flow, the outflow boundary condition is applied to all top-most and bottom-most lattice points. (For example, the particle distributions of lattice point *b* are obtained by copying the particle distributions from lattice point *a* and vertically flipping the particle distributions.)

3.2.3 Outflow Boundary

Similar to periodic boundary, an outflow boundary condition is applied by drawing single-texel-wide stripes. According to Mei et al. [103], nodes at an outflow boundary get their distributions from distributions of internal nodes but with a velocity direction flipped around one of the major axes (see a 2D example in Figure 3.7). Note that when packing the distributions, we guarantee that $f_{(i,j,k)}$ and $f_{(-i,j,k)}$ of the same node are in the same texture, as well as the pairs $f_{(i,j,k)}$ and $f_{(i,-j,k)}$, $f_{(i,j,k)}$ and $f_{(i,j,-k)}$. These can be easily verified from Table 3.1. Therefore, each boundary distribution texture copies from only one distribution texture. To flip the distributions around the major axes, we utilize the swizzle operator to rearrange the order of the color elements.

3.2.4 Complex Boundary

To handle a complex boundary, we adopt Mei et al.'s method [104] (see Figure 2.4 in Section 2.2.3). The boundary does not necessarily lie on the lattice nodes, and can deform during the simulation. Each boundary link is specified by two nodes $\mathbf{x}_f$, $\mathbf{x}_b$, the fraction of the link that is in fluid Δ, and the moving speed of the wall $\mathbf{u}_w$.

We actually deem the regions isolated by the boundary surface as separate fluids. Hence, for each boundary link, the boundary condition affects two distributions, one on each side of the boundary. Besides, the two distributions are in the opposite directions, but co-linear with the link. We refer to them as boundary distributions.

To generate the boundary information, we first create a voxelization of the boundaries using the method described in Section 3.2.1. Besides the position of the voxels, we also need the wall velocity $\mathbf{u}_w$, as well as the coefficients of polygon plane equations which will be used to compute Δ. That is, we need three attributes in total. To preserve accuracy, we treat these attributes as texture coordinates when rendering the vertex array in the next step.

In practice, we don't explicitly generate the flat volume of the boundary voxels, but combine the generation with the computation of the boundary conditions, by rendering the boundary vertex array directly into the pbuffer containing the propagated new distributions, and applying the fragment program for complex boundary conditions. Note that in most cases, for each boundary link, only one node is covered by a voxel from the generated voxel array. However, we need each boundary node to receive a fragment so that the boundary distributions are updated. Therefore, for each packed distribution texture, we render the voxel array three times. In the first pass, they are rendered normally, covering those voxels in the generated voxel array. In the second pass, we first set the color mask, so that only the R and G channels can be modified. Then, we apply a translation to all the voxels using a vertex program. The translated offset is computed according to the following rule:

- $\mathbf{e}_i$: if $flag_1 * flag_2 > 0$
- $-\mathbf{e}_i$: if $flag_1 * flag_2 < 0$
- 0: if $flag_1 * flag_2 = 0$

where $flag_1 = (pos_x, pos_y, pos_z, 1) \bullet (A,B,C,D)$, $flag_2 = (A,B,C) \bullet \mathbf{e}_i$, and $\bullet$ represents a dot product. (pos_x, pos_y, pos_z) is the 3D position of the voxel without translation. The boundary surface inside the voxel is defined by $Ax + By + Cz + D = 0$, where (A, B, C) is a normalized plane normal pointing from the solid region to the liquid region. $\mathbf{e}_i$ is the velocity vector associated with the distribution in the red channel. The third pass is similar to the second pass, except that this time the B and A channels are modified, and $\mathbf{e}_i$ is the velocity vector corresponding to the blue

channel distribution.

In this way, all the boundary nodes are covered by the voxels. We then compute Δ at the beginning of the boundary condition fragment program with the following equations:

$$\Delta' = flag_1/flag_2, \qquad \Delta = 1 - \Delta' \tag{3.2}$$

The meanings of $flag_1$ and $flag_2$ are the same as before. Note that each channel computes its $flag_1$, $flag_2$, and Δ independent of its own $\mathbf{e}_i$. A distribution is a boundary distribution if only, for the corresponding color channel, $1 \geq \Delta' \geq 0$. If it is not a boundary distribution, the fragment program prevents modifying the corresponding color channel by assigning it the old value.

3.3 Visualization

To visualize the simulation, we inject particles into the flow field. The positions of the particles are stored in a texture, and are updated by the current velocity field at every time step. The updating is through a fragment program as well. Similar to the generation of boundary voxels, the updated particle positions are then copied to a vertex array residing in the graphics memory for rendering. The whole simulation and rendering cycle is inside the GPU, hence there is no need to transfer large chunks of data between the main memory and the graphics memory. To better display the flow field, particles are arranged into a regular grid before injection and are colored according to the position where they enter the flow field, as shown in Figure 3.15. Due to the requirement of the vertex array, the total number of particles are constant during the simulation. We use a fragment program to recycle particles that flow out of the border of the field or stay in a zero-velocity-region, and place them back at the inlet. If two particles coincide at exactly the same location at the same time, they will never separate during the advection. Visually, we will see fewer and fewer particles. To avoid this, we add a random offset to each particle when placing it at the inlet.

In the 2D LBM, there is only one velocity slice, while in the 3D LBM, the velocities form a volume. The advection fragment program fetches the velocity indicated by the current position of the particles. Therefore, the fragment program

needs to access either a 3D texture or a 2D texture storing the flat volume. We indeed chose the flat volume storage, which is much faster. Please note that if the velocity is stored as a stack of 2D textures, the advection would be very difficult, if not impossible.

3.4 Performance

We have experimented with our GPU-based LBM implemented using OpenGL and Cg on an Nvidia GeForce FX 5900 Ultra. The graphics board has 256MB 425MHz DDR SDRAM and its core speed is 400MHz. The host computer is a Pentium IV 2.53Ghz with 1GB PC800 RDRAM. All the results related to the GPU are based on 32-bit single precision computation of the fragment pipeline. For comparison, we have also implemented a software version of the LBM simulation using single precision floating point, and measured its performance on the same machine.

Figure 3.8 shows the time in milliseconds per step of the LBM D2Q9 model as a function of the lattice size, running on both the CPU and the GPU. Note that both the X and Y axes are in logarithmic scale. Figure 3.9 compares the two from a different perspective by showing the speedup factor. The times include both simulation and visualization. Note that there is no need to transfer the velocity field or the particle positions between the main memory and the graphics memory. The time spent on advecting and rendering the particles is negligible with respect to the simulation.

For lattice size equals to or greater than 128^2, the simulation on the GPU is about 6 times faster than that on the CPU. The GPU is less effective in acceleration for smaller lattice sizes due to the overhead inside the GPU pipeline, such as switching textures and switching GL context that are independent of the lattice size. As the lattice size increases, the percentage of the overhead among the total time decreases, hence we see better speedups. For large lattice sizes, the overhead is negligible and the simulation times of both the GPU-based LBM and the CPU-based LBM tend to be proportional to the lattice size. Therefore, the speedup factor stays relatively constant.

Figures 3.10 and 3.11 show similar graphs to Figures 3.8 and 3.9, but for the

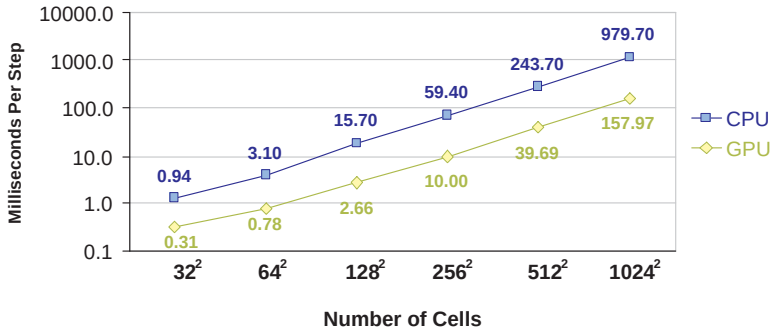


Figure 3.8: Speed (in milliseconds per step) of a D2Q9 LBM simulation.

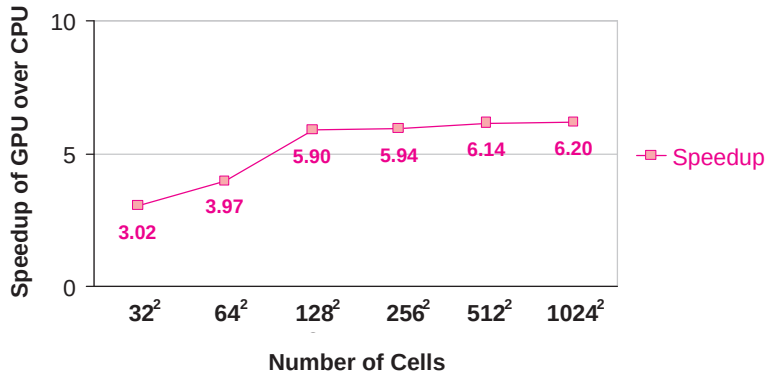


Figure 3.9: Speedup of the LBM on the GPU vs. the software version for the D2Q9 model.

D3Q19 LBM model. Compared with D2Q9, each lattice site requires more computation and more data access operations. Hence, even higher speedup factors have been achieved with the GPU acceleration. The speedup factor varies between 8 and 9 for most of the lattice sizes. When the lattice size approaches 128^3, the speedup is as high as 15.

The step in the GPU timing curve—as well as in the GPU versus CPU speedup—is good evidence showing their different cache behavior. When the lattice size gets bigger and surpasses a certain threshold, cache misses significantly slow down the CPU version. Due to the mostly sequential access patterns of the LBM algorithm and the GPU's optimization for sequential access, the cache-miss rate on the GPU is relatively independent of the lattice size, and we do not see such

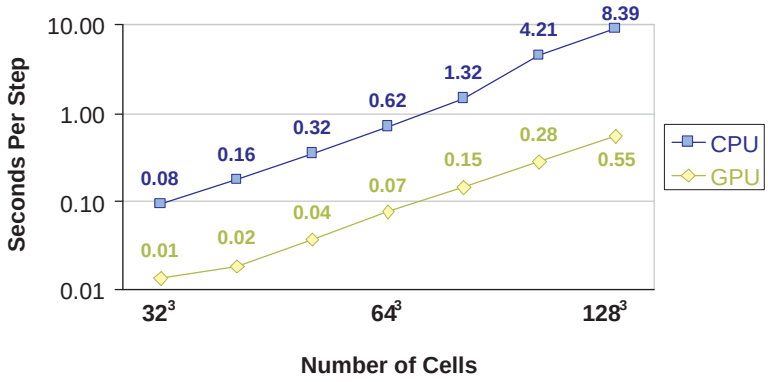


Figure 3.10: Speed (in seconds per step) of a D3Q19 LBM simulation.

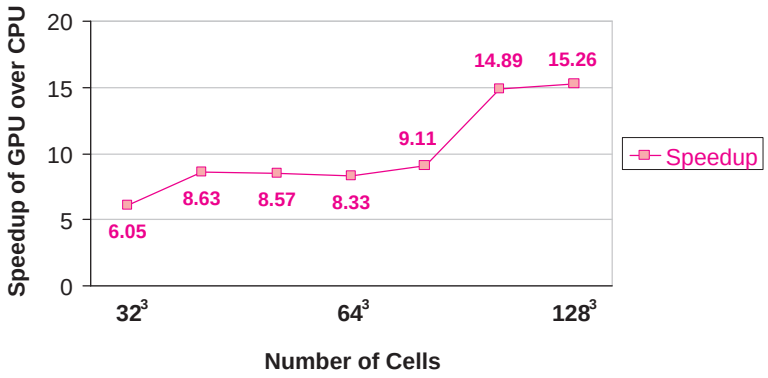


Figure 3.11: Speedup of the LBM on the GPU vs. the software version for the D3Q19 model.

a computation time jump. In an experiment, we modified the software version to reduce the cache-miss rate. The new software version splits large lattices into multiple 64^3 sub-lattices. Figures 3.12 and 3.13 show a new comparison of computation times and a new curve of the speedup factor.

3.5 Results

Figure 3.14 shows 2D flow fields based on the D2Q9 model simulation. In Figure 3.14(a) and Figure 3.14(b), we insert obstacles such as disks and bars (shown in

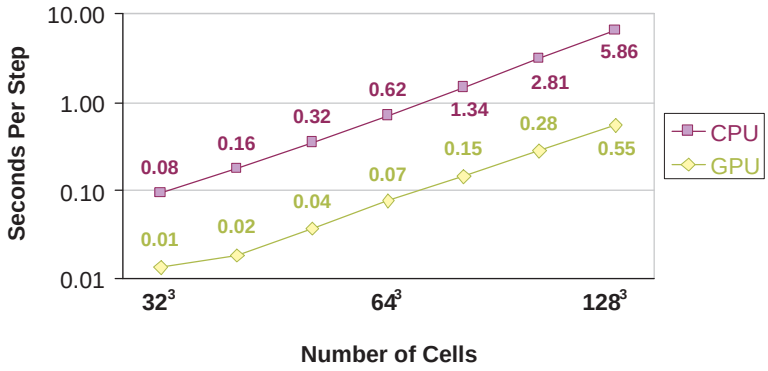


Figure 3.12: Speed (in seconds per step) of a D3Q19 LBM simulation. (Unlike that in Figure 3.10, this software version splits large lattices into multiple 64^3 sub-lattices in order to reduce the cache-miss rate.)

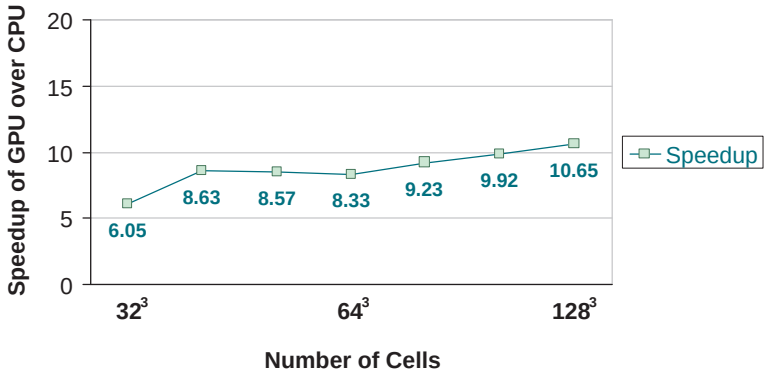
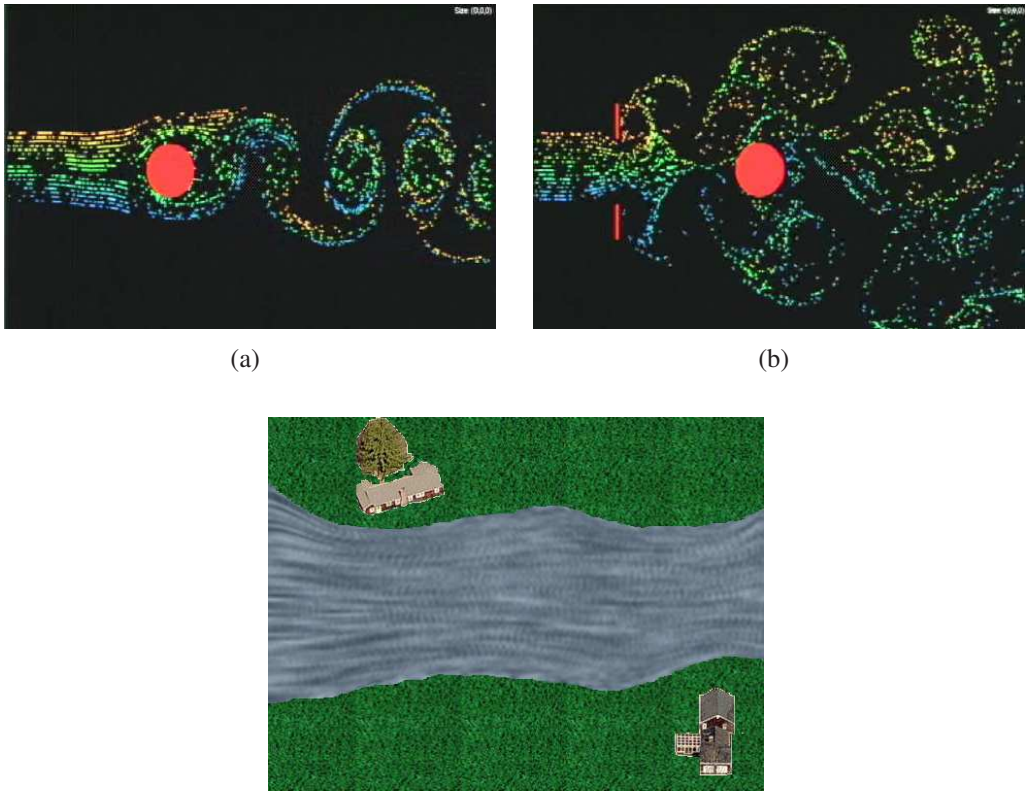


Figure 3.13: Speedup of the LBM on the GPU vs. the software version for the D3Q19 model. (Unlike that in Figure 3.11, this software version splits large lattices into multiple 64^3 sub-lattices in order to reduce the cache-miss rate.)

red) into the simulations. The vortices are generated due to the obstacles. To visualize the flow field, we inject a slice of colored particles and advect them according to the velocity field of the flow. Figure 3.14(c) shows a simulation of a river. We have used a binary bitmap to specify the shape of the river. The simulation is visualized using the image-based flow visualization (IBFV) method [146] in real-time.

Our simulation can handle arbitrarily complex and dynamic boundary, which



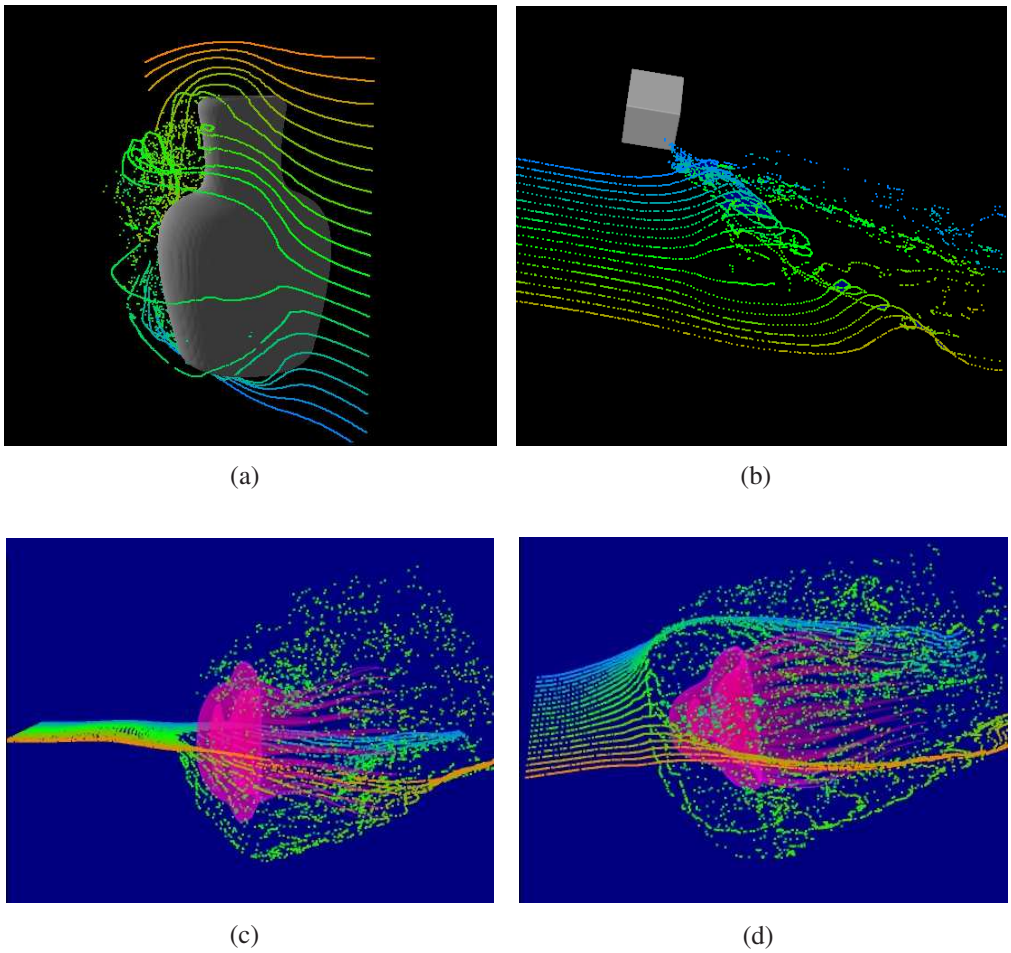
(a) (b)

(c)

Figure 3.14: 2D LBM simulations on the GPU: (a) a (red) disk in the flow, (b) two (red) bars and a (red) disk in the flow, and (c) flow in a river.

usually results in a complex flow field. Figure 3.15 shows 3D LBM simulation results. Colored particles, injected at one end through a slit and advected by the flow, depict the flow field. All the computations—the simulation, the generation of the boundaries, the advection, and the rendering of the particles—are executed on the GPU in real time. Figure 3.15(a) shows the flow over a static vase. Figure 3.15(b) shows the flow interacting with a moving box. Figure 3.15(c) and (d) show a jellyfish swimming from right to left in the flow. The model and the rendering of the jellyfish have been adapted from Nvidia's sample code. The jellyfish deforms its body, and so does the liquid-solid boundary.

We have used our GPU-based LBM implementation in several graphics applications, to model natural phenomena in real-time, such as bubbles and feather in the wind [150, 151], fire [164], smoke [163], and heat shimmering and mirage [162].

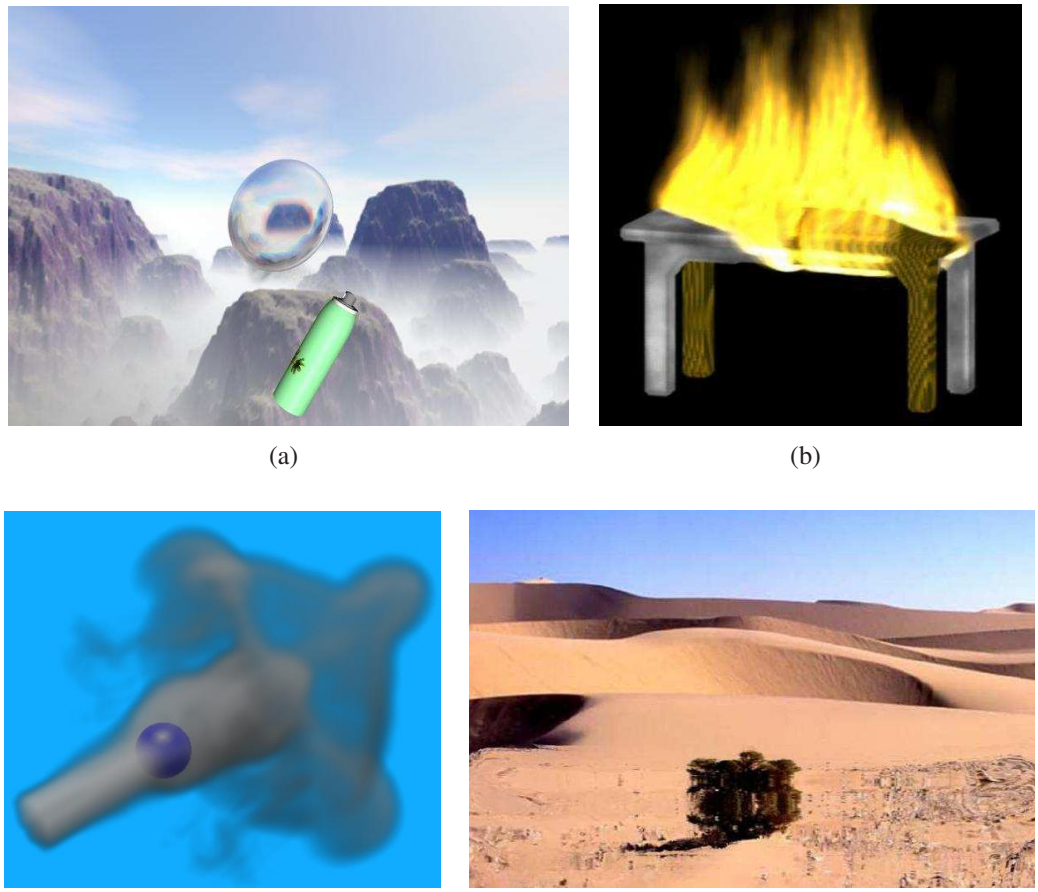


(a) (b)

(c) (d)

Figure 3.15: 3D LBM simulations on the GPU with (a) a static vase, (b) a moving box, and (c) and (d) a jellyfish that swims from right to left.

Figure 3.16 shows the snapshots of our simulations in these applications.



(a) (b)

(c) (d)

Figure 3.16: Visual simulations of natural phenomena using the GPU-based LBM: (a) blowing of a bubble, (b) fire, (c) smoke around a ball, and (d) heat shimmering.

Chapter 4

Adapted Unstructured LBM on GPU

Flow motion on curved surfaces of arbitrary topology is an interesting visual effect. An infinitely thin flow that purely lives on a curved surface would only exist in an imaginary world. However, its visual simulation can create interesting special effects on the 3D surface models and is desirable for computer graphics applications. Furthermore, shallow (thin) fluid flows are actually often seen in the real-world, such as the swirling pattern on soap bubbles, the atmosphere on the earth, and the lava drifting from the peak of the mountain. Reducing their visual simulations from 3D flows to surface flows will simplify the model and make the computation affordable. On the other hand, modeling flows on arbitrary curved surfaces is also a challenging problem, and most previous flow models only solve the flows in 2D planar space or 3D space.

Stam [133] has first proposed a method based on Stable Fluids to simulate fluid flows on curved surfaces. However, this method requires the global surface parameterization that may cause visible distortions in the flow. Although he has further proposed a technique to alleviate the distortions, the distortions are still apparent. Shi and Yu [130] have presented a method that performs the inviscid and incompressible flow simulation directly on surfaces. Their method avoids the global parameterization and directly applies the advection and the pressure solver on triangular surface meshes. However, a sparse linear system derived from the Possion equation for pressure still needs to be solved globally. Note that both methods start from a macroscopic point of view and globally solve the Navier Stokes equations.

In this chapter, we introduce a novel and effective way to model such dynamics. We propose a technique that adapts a recently emerged computational fluid dynamics (CFD) model, unstructured lattice Boltzmann model (Unstructured LBM), from the 2D unstructured meshes to the 3D surface meshes. Unlike the previous methods, our method is based on the mesoscopic kinetic equations for discrete particle distribution functions. No matter how complicated the surface shape and topology are, all computations on the surface mesh only involve the information within local neighborhoods. This model has the following advantages: (i) simplicity and explicit parallelism in computation, which make the method well suited to GPU acceleration; (ii) great capability in handling complex interactions, such as the interactions between flow and boundaries and the interactions of multiple-component fluids; (iii) no need for global surface parameterization that may cause strong distortions; and (iv) capability of being applied to meshes with arbitrary connectivity.

4.1 Unstructured LBM

An unstructured grid is an array of points with their connectivity relationship explicitly stated. Unlike the structured grid, its points have no particular logical order. This allows for more geometrical flexibility. For example, in computer graphics, the triangular mesh has become a dominant method for the representation of 3D surfaces. In modern CFD techniques, the unstructured grid is often used in finite-element or finite-volume computations. These computations have also appeared in visual simulations, such as the modeling of muscle dynamics [139].

The LBM on the unstructured grid was first proposed by Peng et al. [117]. By importing finite-volume techniques within the LBM framework, their model applies to meshes without requiring any special kind of connectivity. Later, Ubertini et al. [143, 144] have improved the 2D unstructured LBM and further incorporated a set of boundary conditions. Because our further development is based on Ubertini et al.'s 2D method, we briefly review their model below.

Figure 4.1 shows a 2D triangular LBM grid. Every grid point has nine velocity vectors, each of which is associated with a particle distribution function. Figure 4.2 shows a closer view of the geometrical 1-ring neighborhood around a grid point P. The neighboring points of P on the grid are denoted as P_k, $k = 1, 2, .., K$. The

green regions in Figure 4.2 are a set of finite-volumes defined around P. (In 2D, the finite-volumes are polygons.) They are denoted as Ω_k, $k = 1,2,..,K$. Each Ω_k is the union of two triangles $\Omega_k^- = [P,E_k,O_k]$ and $\Omega_k^+ = [P,O_k,E_{k+1}]$, where O_k is the center of the triangle $[P,P_k,P_{k+1}]$ and E_k, E_{k+1} are the midpoints of the edges PP_k, PP_{k+1}, respectively.

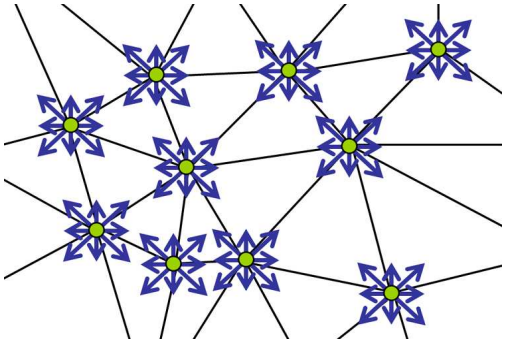


Figure 4.1: The 2D unstructured LBM grid. Every grid point has nine symmetrical velocity vectors (including the zero velocity), each associated with a particle distribution function.

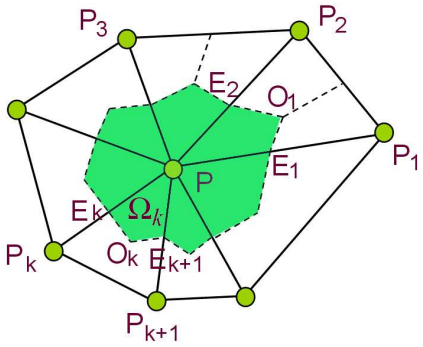


Figure 4.2: The geometrical layout of the 1-ring neighborhood around a grid point P. Points P_k are the neighboring points of P. (The green regions stand for the finite-volumes which are defined around P.)

For the unstructured grid, the Boltzmann equation is written as the following finite-difference equation:

$$f_i(P,t+dt) = f_i(P,t) + dt \sum_{k=1}^{K} (\Phi_{ik} - \Xi_{ik}), \tag{4.1}$$

where Φ_{ik} and Ξ_{ik} denote the streaming and collisional fluxes of the ith particle distribution function f_i coming from the kth finite-volume Ω_k. Applying the linear interpolation rules, the calculation of streaming fluxes is straightforward. The contribution of collisions arises from the integration of the linear interpolated value of the collision term $(f_i - f_i^{eq})/\tau$ over each finite-volume Ω_k. The resulting finite-volume equation takes the following general form:

$$\begin{aligned} f_i(P,t+dt) = \quad & f_i(P,t) + dt \sum_{k=0}^{K} S_{ik} f_i(P_k,t) \\ & - \frac{dt}{\tau} \sum_{k=0}^{K} C_{ik}[f_i(P_k,t) - f_i^{eq}(P_k,t)], \end{aligned} \tag{4.2}$$

where index $k = 0$ denotes the pivotal point P itself. The detailed expressions of the streaming and collision matrices S_{ik} and $C_{ik} = C_k \delta_{ik}$ ($\delta_{ik} = 1$, if $i = k$; and $\delta_{ik} = 0$ if $i \neq k$) are obtained by

$$S_{i0} = 0, \;\; S_{ik} = \vec{e}_i \cdot \vec{N}_k / V_P, \;\; k = 1,2,..,K, \tag{4.3}$$

and

$$C_0 = 1/3, \;\; C_k = \frac{V_{k-1} + V_k}{3V_P}, \;\; k = 1,2,..,K. \tag{4.4}$$

In the above, V_k is the area of Ω_k, while V_P is the area of $\Omega = \bigcup_k \Omega_k$. $\vec{N}_k$ is defined as

$$\vec{N}_k = \left[\frac{5}{12}(\vec{A}_{k-1}^{+} + \vec{A}_k^{-}) + \frac{2}{12}(\vec{A}_{k-1}^{-} + \vec{A}_k^{+}) \right], \;\; k = 1,2,..,K, \tag{4.5}$$

where $\vec{A}_k^{\mp}$ are the vectors normal to the lines $E_k O_k$, $O_k E_{k+1}$, with magnitude equal to the length of these lines. Similarly, $\vec{A}_{k-1}^{\mp}$ associate with lines $E_{k-1}O_{k-1}$ and $O_{k-1}E_k$, respectively.

4.2 Out Method: Unstructured LBM on Curved Surfaces

In this section, we present the basic algorithm of our model. This model is devised by adapting Ubertini et al.'s unstructured LBM from 2D meshes to manifold

surface meshes. To successfully realize this adaptation, the following problems had to be solved: (1) for each mesh point, we need to define its nine velocity vectors; and (2) in order to apply Equation 4.2 in the computation, for each mesh point P, we need to locally flatten P's 1-ring neighborhood.

To solve the first problem, for each mesh point, we firstly define the local frame $\langle \vec{s}, \vec{t} \rangle$, in its tangent space, then define the nine velocity vectors based on this local frame. Note that for most surface meshes (specifically close meshes whose genus is not one), it's impossible to define on them the local frames that are globally continuous. This means that there are unavoidable differences in the orientations of velocity vectors between neighboring mesh points. Fortunately, in Section 4.2.3 we introduce a technique to rotate and align velocity vectors and recompute the corresponding particle distribution functions. With this technique, there is no need for local frames to be globally continuous.

For the second problem, we flatten the 1-ring neighborhoods to the tangent planes in pre-processing. We use the *ghost points*, G_k, on behalf of the neighboring points P_k (see Figure 4.3). In the simulation, for each time step we first update the states of all ghost points based on corresponding neighboring points, and then for each mesh point we execute the streaming and collision computation with information from ghost points, a procedure similar to that in 2D unstructured LBM. Section 4.2.1 through Section 4.2.3 give more details.

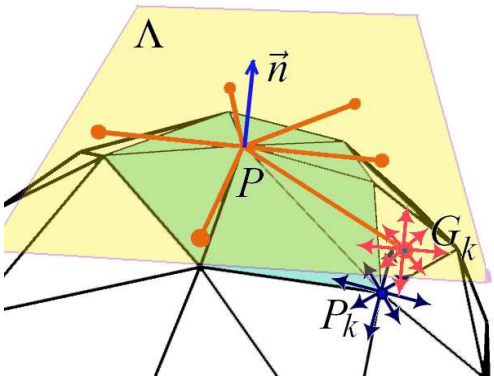


Figure 4.3: The 1-ring neighborhood of P is flattened to its tangent plane Λ. Ghost point G_k is on behalf of neighboring point P_k. The velocity vectors (dark blue) at P_k are transformed into vectors (pink) that lie in Λ.

4.2.1 Define Velocity Vectors for Mesh Points

To define the nine velocity vectors for each mesh point P, we first define the local frame $\langle \vec{s}, \vec{t} \rangle$, where $\vec{s}$ and $\vec{t}$ are two orthogonal unit vectors defining the local tangent space at P. Any definition of local frames can be used for our model and we choose the following simple method. Assuming $\vec{n}$ is the normal vector at mesh point P, we let

$$\vec{s'} = \begin{cases} \vec{n} \times \vec{x} & , \quad if\ |\vec{n} \cdot \vec{x}| \leq \frac{\sqrt{2}}{2} \\ \vec{n} \times \vec{y} & , \quad otherwise, \end{cases} \tag{4.6}$$

$\vec{s} = \vec{s'}/|\vec{s'}|$ and $\vec{t} = \vec{n} \times \vec{s}$. In the above, $\vec{x}$, $\vec{y}$, and $\vec{z}$ are three mutually orthogonal unit vectors, "$\times$" denotes the vector cross product, and "$\cdot$" denotes the vector dot product. It can be seen that in above definition the local frame is uniquely determined by $\vec{n}$.

Similarly to the D2Q9 model described in Section 2.2.3, for each mesh point, we define its velocity vectors $\vec{e_i}$ as follows in the 3D world space:

$$\vec{e_i} = \begin{cases} 0 & , \quad i = 0 \\ cos(\theta_i)\ \vec{s} + sin(\theta_i)\ \vec{t} & , \quad i = 1,..,4 \\ \sqrt{2}\ cos(\theta_i)\ \vec{s} + \sqrt{2}\ sin(\theta_i)\ \vec{t} & , \quad i = 5,..,9, \end{cases} \tag{4.7}$$

where

$$\theta_i = \begin{cases} (i-1)\ \pi/2 & , \quad i = 1,..,4 \\ \pi/4 + (i-5)\ \pi/2 & , \quad i = 5,..,9. \end{cases} \tag{4.8}$$

4.2.2 Flatten the 1-Ring Neighborhoods

In flattening (see Figure 4.3), the positions of the ghost points G_k are computed as follows. First, we find such neighboring point P_1 that the angle between the edge $\overline{PP_1}$ and tangent plane Λ is the smallest. Second, we project edge $\overline{PP_1}$ onto Λ and scale the length of the projected edge to $|\overline{PP_1}|$. The resulting edge is $\overline{PG_1}$ and hence the position of G_1 is determined. Third, similar to [82], the positions of other ghost points are calculated in a way that all the edge lengthes are exactly preserved and the angles between two consecutive edges are preserved up to a common factor. We

denote the transformation matrix as M'_k which rotates $\overline{PP_k}$ to $\overline{PG_k}$ around point P with the vector $\overrightarrow{PP_k} \times \overrightarrow{PG_k}$ to be the rotation axis.

We also apply the transformation M'_k to the velocity vectors at each neighboring point P_k. After this transformation, the velocity vectors may not lie in P's tangent plane. Therefore, we apply an additional rotation M''_k to transform them into P's tangent plane. In the following description, $\vec{n_k}'$ denotes $M'_k\vec{n_k}$, where $\vec{n_k}$ is the normal vector at neighboring point P_k. M''_k rotates the vector $\vec{n_k}'$ into $\vec{n}$ around point G_k with the vector $\vec{n_k}' \times \vec{n}$ to be the rotation axis. Assuming the surface is smooth and the mesh resolution is high enough, the transformation $M''_k M'_k$ should cause no or only negligible distortions. The resulting vectors are defined as G_k's velocity vectors (see Figure 4.3).

4.2.3 Rotate and Align the Velocity Vectors

The local LBM computation can not yet be directly applied as on the 2D unstructured grid, because the velocity vectors of the ghost points have different orientations from those of point P's velocity vectors (see Figure 4.4).

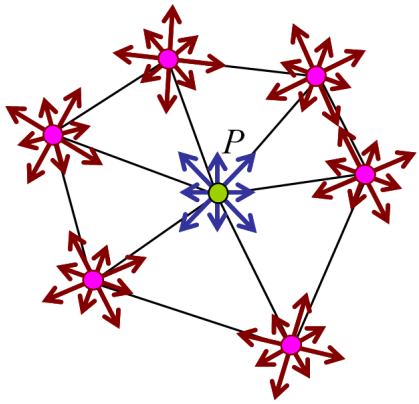


Figure 4.4: Vector alignment is needed, because the velocity vectors of the ghost points have different orientations from those of the point P's velocity vectors.

Our solution to this is to rotate the velocity vectors of the ghost points and align them with P's velocity vectors. Accordingly, the particle distribution functions need to be recomputed in order to preserve the flow properties, such as the fluid density ρ and the fluid velocity $\vec{\mathbf{u}}$ (see Figure 4.5). For a given ghost point G_k, we denote the

rotation angle as Θ, which is in the range of $[0, 2\pi]$. We denote the original velocity vectors and their associated particle distributions as e_i' and f_i' respectively. The target velocity vectors and the corresponding new particle distribution functions are denoted as e_i and f_i respectively.

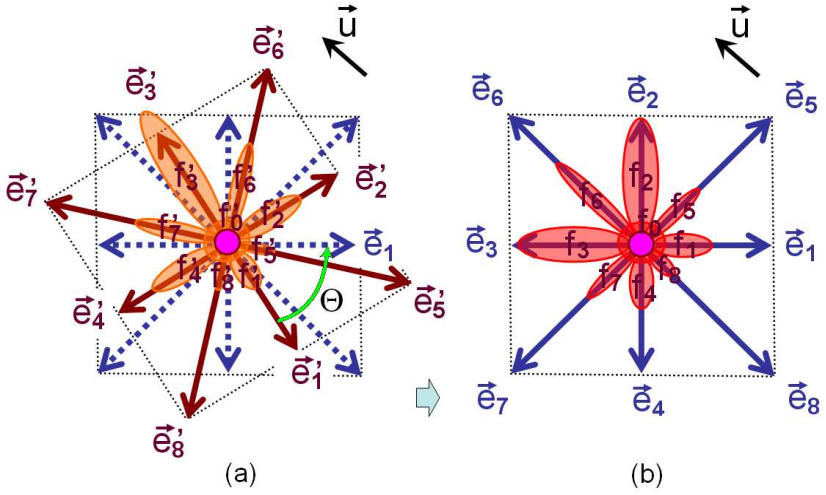


Figure 4.5: The illustration of vector alignment. (a) the velocity vectors e_i' of a ghost point are rotated and aligned with the velocity vectors e_i of point P. (Θ is the rotation angle. The particle distribution functions are shown as ellipses. The original particle distributions are denoted as f_i'.) (b) The new particle distributions f_i are recomputed for the rotated velocity vectors. They preserve the fluid density ρ and the fluid velocity $\vec{\mathbf{u}}$.

The following equations preserve the fluid density and the fluid velocity:

$$\sum_i f_i \;=\; \rho \;=\; \sum_i f_i', \tag{4.9}$$

$$\sum_i f_i \vec{e}_i \;=\; \vec{\mathbf{u}} \;=\; \sum_i f_i' \vec{e}_i', \tag{4.10}$$

They can be further supplemented with equations for preserving the energy, the stress tensor, and other flow properties. However, they are enough for our visual simulation and we have chosen the following way to satisfy them.

Without loss of generality, let's assume the rotation angle Θ is in the range of $[0, \pi/2]$, meaning that e_1 is between e_1' and e_2'. We let f_0 equal f_0'. The values f_1, .., f_4 are computed based on f_1', .., f_4'. In this computation, we first let

$$\bar{f} = (f_1' + f_2' + f_3' + f_4')/4.$$

Then we subtract $\bar{f}$ from each f_i' ($i = 1,..,4$) and get

$$\lambda'_i = f_i' - \bar{f}, \quad i = 1,..,4.$$

After that, we project the vectors $\lambda'_i e_i'$ and $\lambda'_{i+1} e_{i+1}'$ on the direction of e_i. The resulting vector length is

$$\lambda_i = \lambda'_i cos(\Theta) + \lambda'_{i+1} sin(\Theta), \quad i = 1,..,4.$$

Finally, we add $\bar{f}$ back and get

$$f_i = \bar{f} + \lambda_i, \quad i = 1,..,4.$$

It is easy to prove that $\sum_{i=1}^{4} f_i = \sum_{i=1}^{4} f_i' = 4\bar{f}$ and $\sum_{i=1}^{4} f_i e_i = \sum_{i=1}^{4} f_i' e_i'$. Similarly, we compute the values f_5, .., f_8 based on f_5', .., f_8'. Thus Equation 4.9 and 4.10 are satisfied.

4.3 Enhancements to Our Method

In this section, we introduce other elements that enhance our flow model over curved surfaces. They are boundary conditions, body forces, vorticity confinement on the unstructured grid, and multi-component fluids. Because these computations also only involve local operations and can be executed in flattened 1-ring neighborhoods, adapting them from previous 2D LBM models to our model is straightforward. From this part on, unless otherwise specially stated, all values and operations are presented in the tangent spaces $\langle \vec{s}, \vec{t} \rangle$ at mesh points.

4.3.1 Boundary Conditions

Ubertini et al. [143] have introduced three ways to handle static and moving boundary conditions in the 2D unstructured LBM. They are, listed in order of increasing implementation difficulty as well as physical accuracy: equilibrium method, mirror method, and covolume method. We have implemented the first method, which takes to set the particle distribution functions f_i as the equilibrium particle distribution functions f_i^{eq} for every boundary point. These f_i^{eq} are calculated using Equation 2.6, in which $\vec{\mathbf{u}}$ is set as the boundary velocity.

4.3.2 Body Forces

Body forces can be user applied force, gravity, vorticity confinement force, and etc. For each mesh point P, if the force vector is not in its tangent plane, we need to project it onto the tangent plane. We denote the resulting vector as $\vec{F}$. Then, this force affects the local particle distribution functions as follows, according to the previous work of LBM [10].

$$f_i \longleftarrow f_i + \frac{(2\tau - 1)}{2\tau} B\vec{F} \cdot e_i, \tag{4.11}$$

where B is the constant which has appeared in Equation 2.6.

4.3.3 Vorticity Confinement on Unstructured Grid

The vorticity confinement force adds small scale rolling features that are usually absent on coarse grid simulations. Fedkiw et al. [39] have firstly introduced to computer graphics the vorticity confinement method on the regular grid for the visual simulation of smoke.

Recently, a vorticity confinement formulation for the unstructured grid has been derived using dimensional analysis by L*ö*hner et al. [90]. We have incorporated this force into our model. The vorticity confinement force F_{vc} is expressed as a function of the local vorticity-based Reynolds-number $Re_{\omega,h}$, the local element size h, the vorticity ω, and the gradient of the absolute value of the voriticity.

$$F_{vc} = g(Re_{\omega,h}) c_\upsilon \rho h^2 \nabla|\omega| \times \omega, \tag{4.12}$$

$$g(Re_{\omega,h}) = max\left[0, min\left[1, \frac{Re_{\omega,h} - Re^0_{\omega,h}}{Re^1_{\omega,h} - Re^0_{\omega,h}}\right]\right], \tag{4.13}$$

$$Re_{\omega,h} = \frac{\rho|\omega|h^2}{\nu}, \tag{4.14}$$

$$\omega = \nabla \times \vec{\mathbf{u}}, \tag{4.15}$$

where c_υ is a constant regardless of the grid, and $Re^0_{w,h}$ and $Re^1_{w,h}$ are two parameters defining the effective range of $Re_{w,h}$. Note that calculations of Equation 4.12 and Equation 4.15 involve the finite difference operations ∇ and $\nabla\times$ respectively. The method to calculate these finite difference operations on the unstructured grid is introduced in the Appendix.

4.3.4 Multi-Component Fluids

The interaction of multiple-component fluids is a specially interesting and complex phenomenon, which has not been adequately addressed in computer graphics. In computational physics, there is a large literature for using the LBM to model this phenomenon, by taking the advantages of LBM in handling microscopic interactions. The fluids can be either immiscible or miscible. In our work, we focus on the immiscible two-component fluid, in which the interface between two fluids is always maintained. Adapting miscible fluids models into our model should be feasible as well.

Our method is based on the classic 2D LBM model for immiscible binary fluids [53]. Red and blue particle distribution functions f_i^r and f_i^b are introduced to represent two different components of the fluid. From them, the density and velocity of the two fluids can be computed by Equation 2.3. The total (or the color-blind) particle distribution function is defined as $f_i = f_i^r + f_i^b$. The collision is applied on f_i as usual. After this, a special two-step two-component collision rule maintains the interfaces that separate the different components.

The first step is to add a perturbation to the particle distribution near the interface which creates the correct surface-tension dynamics. The interface is located by computing the local color gradient $\vec{g}$, which is perpendicular to the interface. The perturbation to each color-blind particle distribution is given by:

$$\hat{f}_i = f_i + A|\vec{g}|cos2(\theta_i - \theta_g), \tag{4.16}$$

where θ_i is the angle of velocity vector e_i and θ_g is the angle of the local color gradient $\vec{g}$. This operation redistributes mass near the interface, depletes mass along lattice links parallel to the interface and adds mass to the links perpendicular to the interface, while the total mass and momentum are conserved.

The second step is to recolor the mass after perturbation in order to separate the two different components and maintain a clear interface. This is achieved by solving a maximization problem :

$$W(\hat{f}_i^r, \hat{f}_i^b) = max\left[(\sum_i (\hat{f}_i^r - \hat{f}_i^b)\vec{e}_i) \cdot \vec{g}\right]. \tag{4.17}$$

To conserve the total red mass and blue mass, $\sum_i \hat{f}_i^r$ must equal to the total amount

of red mass before collision. To conserve the mass in each lattice direction, $\hat{f}_i^r + \hat{f}_i^b = \hat{f}_i$ should be satisfied.

After the above collisions, the streaming is applied on $\hat{f}_i^r$ and $\hat{f}_i^b$, and then the color-blind particle distribution is recomputed for the next time step.

4.4 GPU Implementation

4.4.1 Preprocessing

In preprocessing, we first scale up/down the mesh size, making the average edge length to be one. The reason for doing this is to make uniform parameters regardless of the original mesh dimension. Then the following values at every mesh point are calculated: the local frame, the positions of ghost points and their rotation angles Θ (mentioned in Section 4.2.3), and the coefficients S_{ik} and C_{ik} used in Equation 4.2.

4.4.2 Algorithm Overview

The computational procedure of the flow simulation is listed as follows:

1: Initialize the values $\vec{\mathbf{u}}$, ρ, f_i^{eq}, and f_i for all mesh points
2: **while** the simulation is not terminated **do**
3: Update f_i for all ghost points (Section 4.2.3)
4: Compute $\vec{\mathbf{u}}$, ρ and f_i^{eq} for all ghost points
5: Apply streaming and collision (Equation 4.2) for all mesh points
6: Compute and apply body forces for all mesh points
7: Apply boundary condition for all boundary points
8: Compute $\vec{\mathbf{u}}$, ρ and f_i^{eq} for all mesh points
9: **end while**

This procedure is only for the single-component fluid. It can be just slightly modified as described in Section 4.3.4 to simulate the immiscible two-component fluid.

4.4.3 Data Packing

All steps in the algorithm (Section 4.4.2) are local and explicitly parallel. The major challenge, however, is that we need to pack the unstructured mesh in 2D textures. Because the connectivity of the unstructured mesh does not change over time, we use textures that store indices to express the connectivity of the unstructured mesh. Note that lines 3 and 4 in the algorithm are applied to all ghost points and lines 4, 6, and 8 are applied to all mesh points. Therefore, we store mesh points and ghost points in separate textures.

Figure 4.6 shows our data packing method. Group (a) stores the lattice data of all mesh points. Group (d) stores the lattice data of all ghost points. Note that the ghost points of each mesh point are stored contiguously in Group (d). For every mesh point, Group (b) stores the index of the first ghost point located in its flattened 1-ring neighborhood. For every ghost point, Group (c) store the index of the mesh point which the ghost point is on behalf of. The indices stored in group (b) allow the computation of every mesh point to access the data of the ghost points that are in its flattened 1-ring neighborhood. Likewise, the indices stored in group (c) allow the computation of every ghost point to access the data of the mesh point that the ghost point is on behalf of. The rest part of the implementation is similar to the GPU implementation of D2Q9.

Line 7 of the algorithm is a computation on all boundary points. The set of boundary points is a subset of the mesh points. We store the positions of all boundary points in a Vertexbuffer Object (VBO). In each simulation time step, we enable the boundary condition calculation fragment program and render the VBO as points into the textures of group (a). By doing so, the lattice data of the boundary points are updated accordingly.

4.5 Results

We list here the results of several examples simulated with our model. Figure 4.7(a) and Figure 4.7(b) show the flows over a dog surface and a two-hole torus surface respectively. We periodically deposit a material on the surfaces. Because the density of this material is larger than that of the fluid, gravity force drives the

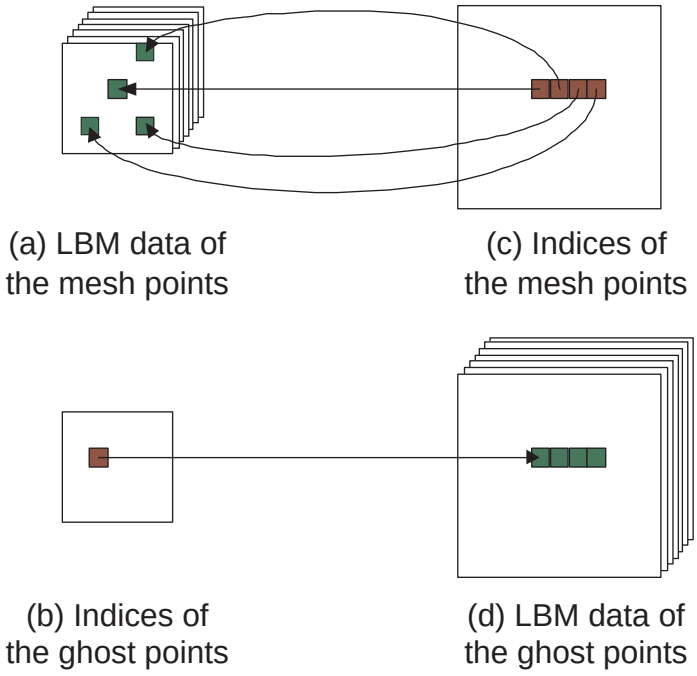


Figure 4.6: The unstructured LBM data are stored in four groups of textures.

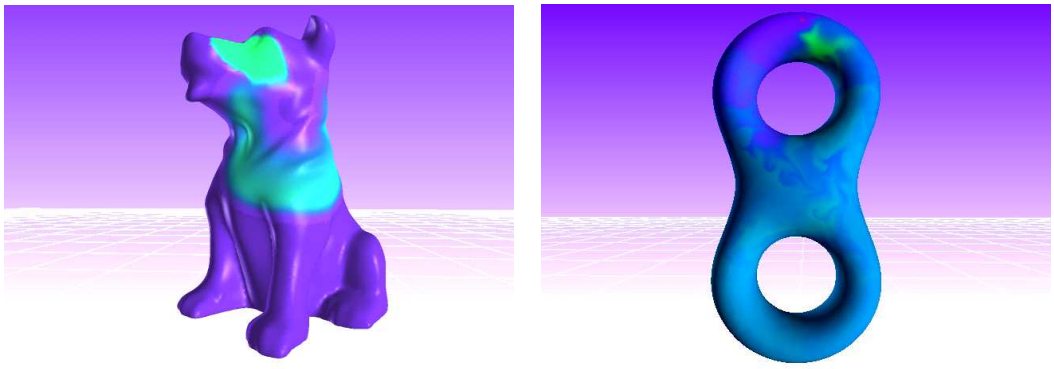


Figure 4.7: Flow motion due to gravity on (a) the dog model and (b) the two-hole torus model.

fluid to move, which causes the advection of the material. This advection is modeled by applying the semi-Lagrangian backtracing scheme with the flow velocity field in the flattened neighborhoods. Note that in some region, the fluid moves in the opposite direction of gravity due to incompressibility.

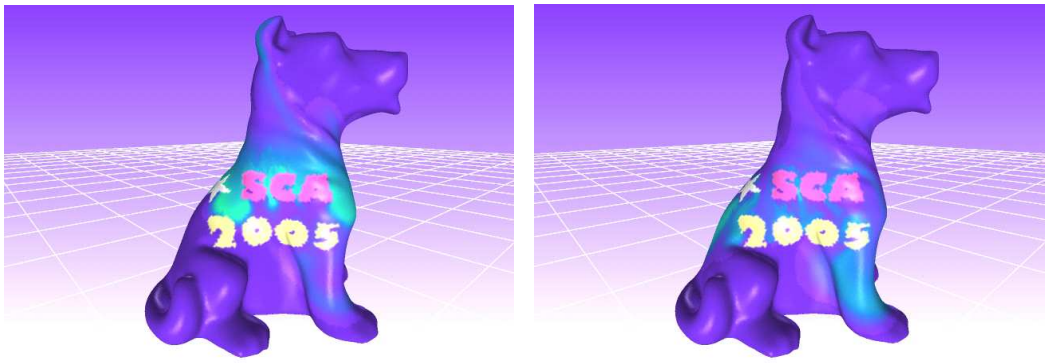


Figure 4.8: Flow motion due to gravity on the dog surface, with static boundaries.

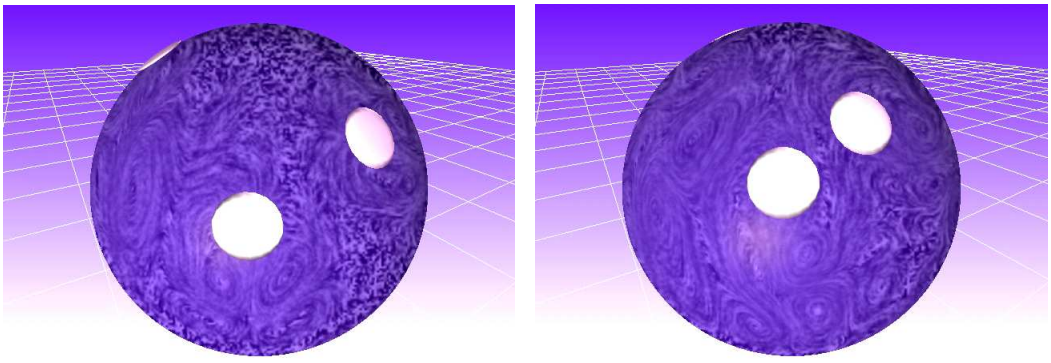
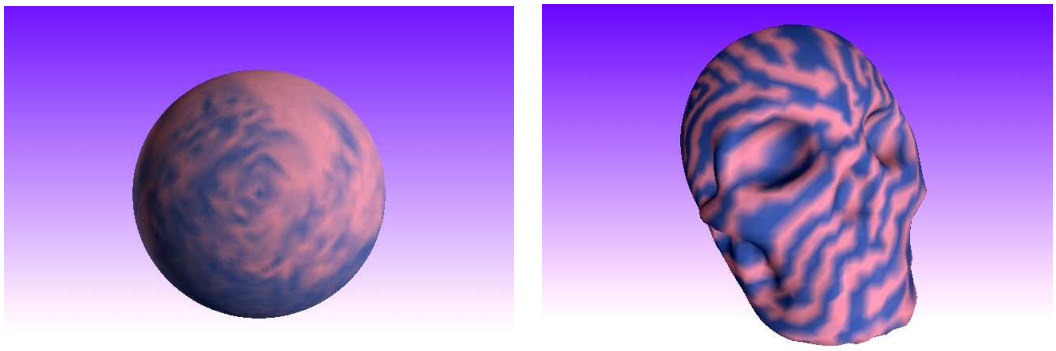


Figure 4.9: Flow motion caused by the animated boundary objects on the sphere.

Figure 4.8 and Figure 4.9 show interactions of the flow over surfaces with boundary objects. In Figure 4.8, static boundaries over the dog model are the text shape *SCA 2005* and the star shape. Figure 4.9 shows animated boundaries, the white objects moving inside and activating the flow on the sphere. We visualize the velocity field using the existing image-based flow visualization method for curved surfaces [147].

Figure 4.10(a) and Figure 4.10(b) shows the immiscible two-component fluids over surfaces. For the simulation of Figure 4.10(a), at the beginning, the left part of the sphere are full of blue fluid while the right part full of pink fluid. The blue fluid is denser than the pink one. Gravity causes a turbulent flow motion, resulting very complex interfaces between two parts. For the simulation of Figure 4.10(b), the blue and pink fluids have the same density. Two parts continuously inosculate



(a) (b)

Figure 4.10: Immiscible two-component fluids, colored in blue and pink: (a) a turbulent mixture of two components on the sphere, (b) a peaceful inosculation on the skull.

with each other, resulting in interesting dynamics.

Table 4.1 compares the performance of our GPU implementation and our software implementation. The performance has been measured on a PC with Intel Xeon 2.40GHz and NVIDIA Quadro FX 4500 GPU. The tested simulations are single-component fluid simulation. The GPU implementation has been about 6.5–10 times faster than the software implementation.

Table 4.1: Performance comparison of the adapted unstructured LBM on the CPU and GPU.

Surface	Vertices	Faces	FPS on CPU	FPS on GPU	Speedup
Sphere	4,098	8,192	15.0	97.0	6.5
Dog	37,502	75,000	1.7	16.3	9.6
Two-Hole Torus	49,998	100,000	1.3	12.6	9.7

Chapter 5

LBM on GPU Cluster

Because of the attractive FLOPS/dollar ratio and the rapid evolution of GPUs, we believe that a GPU cluster is promising for data-intensive scientific computing and can substantially outperform a CPU cluster at the equivalent cost. In this chapter, we present our GPU cluster built in 2004 and describe our LBM implementation on this cluster. We also discuss an application of airborne contaminants dispersion simulation in the Times Square area of New York City. Using 30 GPU nodes, our simulation can compute a 480x400x80 LBM in 0.31 second/step, a speed which is 4.6 times faster than that of our CPU cluster implementation. Although there have been some efforts to exploit the parallelism of a graphics PC cluster for interactive graphics tasks [50, 65, 66], to the best of our knowledge, we are the first to develop a scalable GPU cluster for high performance scientific computing and large-scale simulation.

5.1 The GPU Cluster

Figure 5.1 shows our cluster, called the Stony Brook Visual Computing Cluster. The first version was built in 2004 for two main purposes: as a GPU cluster for graphics and computation and as a visualization cluster for rendering large volume data sets. It has 32 computation nodes connected by a Gigabit Ethernet switch. Each node is an HP PC equipped with two Pentium Xeon 2.4GHz processors and 2.5GB memory. Each node has a GPU, the GeForce FX 5800 Ultra with 128MB

memory, used for GPU cluster computation.

Figure 5.1: The Stony Brook Visual Computing Cluster.

Each node can boot under Windows XP or Linux. We use MPI for data transfer across the network during execution. Each port of the network switch has 1 Gigabit bandwidth. Besides network transfer, data transfer includes upstreaming data from GPU to PC memory and downstreaming data from PC memory to GPU for the next computation. This communication occurs over an AGP 8x bus, which has an asymmetric bandwidth (2.1GB/sec peak for downstream and 133MB/sec peak for upstream). We only use the fragment processing stage of the GeForce FX 5800 Ultra for computing, which features a theoretical peak of 16 Gflops, while the dual-processor Pentium Xeon 2.4GHz reaches approximately 10 Gflops. The theoretical peak performance of this cluster is $(16+10)\times 32 = 832$ Gflops.

5.2 The LBM Implementation

5.2.1 Domain Partitioning

To implement the LBM on the GPU cluster, we decompose the LBM lattice space into sub-domains, each of which is a 3D block processed by one GPU. The

computation hence is expressed in two-levels of parallelism. At the coarse level, multiple GPUs communicate and synchronize with each other. For each node, the velocity distributions at the border sites need to stream to adjacent nodes at every computation step. At the fine level, each GPU executes the LBM computation kernels in SIMD fashion on the lattice sites of its sub-domain. Our single GPU implementation (see Chapter 3) has been reused for the fine level parallelism. In Figure 5.2, the arrows show the communication among the GPUs. Black arrows indicate velocity distributions that stream axially to nearest neighbor nodes while blue arrows indicate velocity distributions that stream diagonally to second-nearest neighbor nodes.

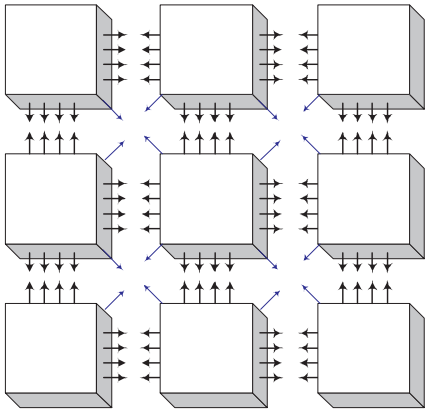


Figure 5.2: The LBM lattice is decomposed into sub-domains and each sub-domain is processed by one GPU. (The arrows show the communication among GPUs.)

In every simulation step, velocity distributions at border sites of the sub-domain may need to stream to adjacent nodes. This kind of streaming involves three steps: (1) Distributions are read out from the GPU; (2) They are transferred through the network to appropriate neighboring nodes; (3) They are then written to the GPU in the neighboring nodes. For ease of discussion, we divide these across-network streaming operations into two categories: streaming axially to nearest neighbors (represented by black arrows in Figure 5.2) and streaming diagonally to second-nearest neighbors (represented by blue arrows). Note that although Figure 5.2 only demonstrates 9 sub-domains arranged in 2 dimensions, our implementation is scalable and functions in a similar fashion for sub-domains arranged in 3 dimensions.

5.2.2 Optimization of Inter-GPU Communication

The primary challenge in implementing the LBM on the GPU cluster is to minimize the communication cost — the time taken for network communication and for transferring data between the GPU and the PC memory. Overlapping network communication time with the computation time is feasible, since the CPU and the network card are all standing idle while the GPU is computing. However, because each GPU can compute its sub-domain quickly, optimizing network performance to keep communication time from becoming the bottleneck is still necessary. Intuitively one might want to minimize the size of transferred data. One way to do this is to make the shape of each sub-domain as close as possible to a cube, since for block shapes the cube has the smallest ratio between boundary surface area and volume. Another idea that we have not yet studied is to employ lossless compression of transferred data by exploiting space coherence or data coherence between computation steps. We have found, however, that other issues actually dominate the communication performance.

The communication switching time has a significant impact on network performance. We performed experiments on the GPU cluster using MPI and replicated these experiments using communication code that we developed using TCP/IP sockets. The results were the same: (1) During the time when a node is sending data to another node, if a third node tries to send data to either of those nodes, the interruption will break the smooth data transfer and may dramatically reduce the performance; (2) Assuming the total communication data size is the same, a simulation in which each node transfers data to more neighbors has a considerably larger communication time than a simulation in which each node transfers to fewer neighbors.

To address these issues, we have designed communication schedules [138] that reduce the likelihood of interruptions. We have also further simplified the communication pattern of the parallel LBM simulation. In our design, the communication is scheduled in multiple steps and in each step certain pairs of nodes exchange data. This schedule and pattern are illustrated in Figure 5.3 for 16 nodes arranged in 2 dimensions. The same procedure works for configurations with more nodes and for 3D arrangement as well. The different colors represent the different steps. In the first step, all nodes in the $(2i)th$ columns exchange data with their neighbors to the

left. In the second step, these nodes exchange data with neighbors to the right. In the third and fourth steps, nodes in the $(2i)th$ rows exchange data with their neighbors above and below, respectively. Note that LBM computation requires that nodes need to exchange data with their second-nearest neighbors too. There are as many as 4 second-nearest neighbors in 2D arrangements and up to 12 in 3D D3Q19 arrangements. To keep the communication pattern from becoming too complicated, and to avoid additional overhead associated with more steps, we do not allow direct data exchange diagonally between second-nearest neighbors. Instead, we transfer those data indirectly in a two-step process. For example, as shown in Figure 5.3, data that node B wants to send to node E will first be sent to node A in step 1, then be sent by node A to node E in step 3. If the sub-domain in a GPU node is a lattice of size N^3, the size of the data that it sends to a nearest neighbor is $5N^2$, while the data it sends to a second-nearest neighbor has size of only N. Using the indirect pattern increases the packet size between nearest neighbors only by $\frac{c}{5N}$ (c is 1 or 2 for 2D arrangement and 1-4 for 3D arrangement). Since the communication pattern is also greatly simplified, particularly for 3D node arrangements, the network performance is greatly improved.

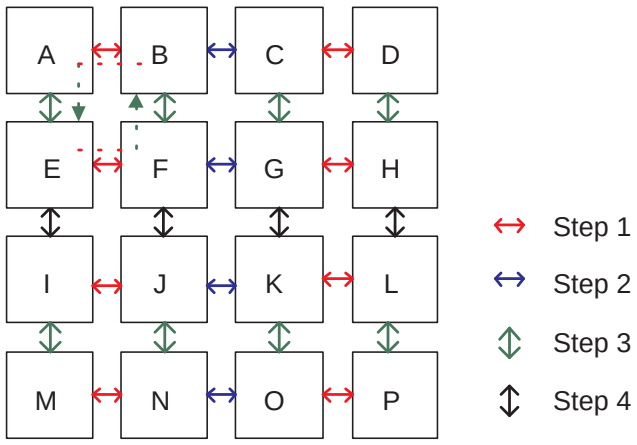


Figure 5.3: The optimized communication schedule and pattern of the parallel LBM simulation. (Different colors indicate the different steps in the schedule.)

We also found that for simulations with a small number of nodes (less than 16), synchronizing the nodes by calling MPI_barrier() at each scheduled step improves the network performance. However, if more than 16 nodes are used, the overhead

of the synchronization overwhelms the performance gained from the synchronized schedule.

The data transfer speed from GPU to CPU represents another bandwidth limitation. The velocity distributions that stream out of the sub-domain are stored in different texels and different channels in multiple textures. We have designed fragment programs which run in every time step to gather together into a texture all these data. Then they are read from the GPU in a single read operation (e.g., OpenGL function glGetTexImage()). In so doing, we minimize the overhead of initializing the read operations.

5.2.3 Performance of LBM on the GPU Cluster

In addition to the GPU cluster implementation, we have implemented the parallel LBM on the same cluster using the CPUs. The time and work taken to develop and optimize these two implementations were similar (about 3 man-months each). Note that although each node has two CPUs, for the purpose of a fair comparison, we used only one thread (hence one CPU) per node for computation.

In Table 5.1, we report the simulation execution time per step (averaged over 500 steps) in milliseconds on both the CPU cluster and the GPU cluster with 1, 2, 4, 8, 16, 20, 24, 28, 30 and 32 nodes. Each node evaluates an 80^3 sub-domain and the sub-domains are arranged in 2 dimensions. The timing for the CPU cluster simulation (shown in column 2 of table 5.1) includes only computation time because the network communication time was overlapped with the computation by using a second thread for network communication. The timing for the GPU cluster simulation (shown in column 6) includes: computation time, GPU and CPU communication time, and non-overlapping (non-O) network communication time. Network communication time (plotted as a function of the number of nodes in Figure 5.4) was partially overlapped with the computation because we let each GPU compute collision operation on inner cells of its sub-domain (which takes roughly 120 ms) simultaneously with network communication. If the network communication time exceeds 120 ms, the remainder will be non-overlapping and affect the simulation time. In column 5 we show this remainder cost along with a total network communication time in parenthesis.

Table 5.1: Per step execution time (in ms) for CPU and GPU clusters and the GPU cluster / CPU cluster speedup factor. (Each node computes an 80^3 sub-domain of the lattice.)

Nodes	CPU cluster	GPU cluster				Speedup
		Computation	AGP	Network: non-O (Total)	Total	
1	1420	214	-	-	214	6.64
2	1424	216	13	0 (38)	229	6.22
4	1430	224	42	0 (47)	266	5.38
8	1429	222	50	0 (68)	272	5.25
12	1431	230	50	0 (80)	280	5.11
16	1433	235	50	0 (85)	285	5.03
20	1436	237	50	0 (87)	287	5.00
24	1437	238	50	0 (90)	288	4.99
28	1439	237	50	11 (131)	298	4.83
30	1440	237	50	25 (145)	312	4.62
32	1440	237	49	31 (151)	317	4.54

The GPU cluster / CPU cluster speedup factor is plotted as a function of the number of nodes in Figure 5.5. When only a single node is used, the speedup factor is 6.64. This value projects the theoretical maximum GPU cluster / CPU cluster speedup factor which could be reached if all communication bottlenecks were eliminated by better optimized network and larger GPU/CPU bandwidth. When the number of nodes is below 28, the network communication will be totally overlapped with the computation. Accordingly the growth of the number of nodes only marginally increases the execution time due to the GPU/CPU communication and the curve flattens approximately at 5. When the number of nodes increase to 28 or above, the network can't be totally overlapped, resulting in a drop in the curve.

To quantify the scalability of the GPU cluster, Table 5.2 shows the computed efficiency of the GPU cluster as a function of the number of nodes. The efficiency values are also plotted in Figure 5.6.

Our simulation computes $640 \times 320 \times 80 = 15.6$M LBM cells in 0.317 second/step using 32 GPU nodes, resulting in 49.2M cells/second. This performance is comparable with supercomputers [99–101]. In the work of Martys et al. [99],

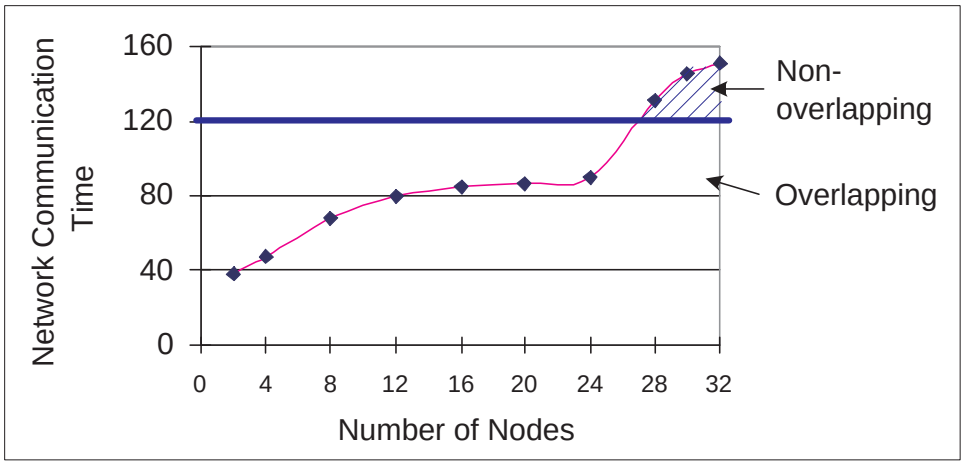


Figure 5.4: The network communication time measured in ms. (The area under the blue line represents the part of network communication time that was overlapped with computation. The shadow area represents the remainder.)

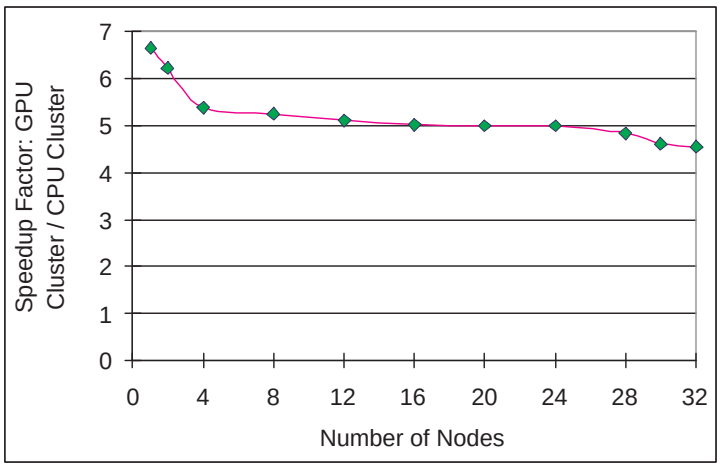


Figure 5.5: Speedup factor of the GPU cluster compared with the CPU cluster.

$128 \times 128 \times 256 = 4$M LBM cells were computed in about 5 seconds/step on IBM SP2 using 16 processors, which corresponds to 0.8M cells/second. In 2002, Massaioli and Amati [100] reported the optimized D3Q19 BGK LBM running on 16 IBM SP Nodes (16-way Nighthawk II nodes, Power3@375MHz) with 16GB shared memory using OpenMP. They computed $128 \times 128 \times 256 = 4$M LBM cells in 0.26

Table 5.2: The GPU cluster performance and the efficiency with respect to the number of nodes.

Number of Nodes	Number of cells computed per second	Speedup	Efficiency
1	2.3M	–	–
2	4.3M	1.87	93.5%
4	7.3M	3.17	79.3%
8	14.4M	6.26	78.3%
12	20.9M	9.09	75.8%
16	27.4M	11.91	74.4%
20	34.0M	14.78	73.9%
24	40.7M	17.70	73.8%
28	45.9M	19.96	71.3%
30	47.0M	20.43	68.1%
32	49.2M	21.39	66.8%

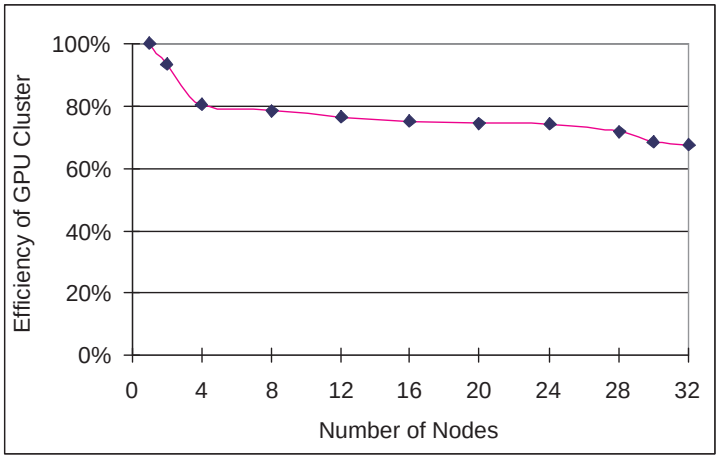


Figure 5.6: Efficiency of the GPU cluster with respect to the number of nodes.

second/step, which is 15.4M cells/second. They were able to further increase this performance to 20.0M cells/second using more sophisticated optimization techniques, such as (1) “fuse” the streaming and collision steps to reduce the memory accesses; (2) keep distributions “at rest” in memory and implement the streaming by

the indexes translation; (3) bundle the distributions in a way that relieves the Segment Lookaside Buffer (SLB) and Translation Lookaside Buffer (TLB) activities during address translation. In 2004, by using the above sophisticated optimization techniques and further taking advantage of vector codes, they achieved the performance of 108.1M cells/second on 32 processors with Power4 IBM [101]. Still, the GPU cluster is competitive with supercomputers at a substantially lower price.

In the above discussion, we have chosen to fix the size of each sub-domain as to maximize the performance of each GPU node. This means, using more nodes we can obtain more cycles to compute larger lattices within a similar time frame. However, another performance criterion for a cluster is to keep the problem size fixed, but increase the number of nodes to achieve a faster speed. However, we have found that in doing so, the sub-domains become smaller, resulting in a low computation/communication ratio. As a consequence, the network performance becomes the bottleneck. We thus may need a faster network in order to better exploit the computational power of the GPUs. We have tested this performance criterion with a $160 \times 160 \times 80$ lattice and started with 4 nodes. When the number of nodes increases from 4 to 16, the GPU cluster / CPU cluster speedup factor drops from 5.3 to 2.4. When more nodes are used, the GPU cluster and the CPU cluster gradually converge to achieve comparable performance.

5.3 Application: Dispersion Simulation in New York City

The LBM can easily accommodate complex-shaped boundaries of the urban environments characterized by sky-scrapers and deep urban canyons. As an example application, we have simulated airborne contaminant dispersion in the Times Square area of New York City. As shown in Figure 5.7, the simulation area extends North from 38th Street to 59th Street, and East from the 8th Avenue to Park Avenue. It covers an area of about 1.66 km $\times$ 1.13 km, consisting of 91 blocks and roughly 850 buildings. For large scale simulations of this kind, the combined computational speed of the GPU cluster and the linear nature of the LBM model create a powerful tool that can meet the requirements of both speed and accuracy. Beyond

enhancing our understanding of the fluid dynamics processes governing dispersion, the simulation will support the prediction of airborne contaminant propagation so that emergency responders can more effectively engage their resources in response to urban accidents or attacks.

Figure 5.7: The simulation area (enclosed by the blue contour) on the Manhattan map.

The geometric model of the Times Square area that we use is a 3D polygonal mesh that has considerable details and accuracy (see Figure 5.8). It covers an area of about 1.66 km $\times$1.13 km, consisting of 91 blocks and roughly 850 buildings. We model the flow using the D3Q19 BGK LBM with a $480 \times 400 \times 80$ lattice. This simulation is executed on 30 nodes of the GPU cluster (each node computes an 80^3 sub-domain). The urban model is rotated to align it with the LBM domain axes. It occupies a lattice area of 440×300 on the ground. As a result, the simulation resolution is about 3.8 meters / lattice spacing. We simulate a northeasterly wind with a velocity boundary condition on the right side of the LBM domain. The LBM flow model runs at 0.31 second/step on the GPU cluster. After 1000 steps of LBM computation, the pollution tracer particles begin to propagate along the LBM lattice links according to transition probabilities obtained from the LBM velocity distributions [94].

Figure 5.8 shows the velocity field visualized with streamlines at time step 1000. Red points indicate streamline origins. The blue color streamlines indicates that the direction of velocity is approximately horizontal, while the white color indicates a vertical component in the velocity as the flow passes over the buildings. Figure 5.9 shows snapshots of a smoke dispersion simulation in the Times Square Area of New York City.

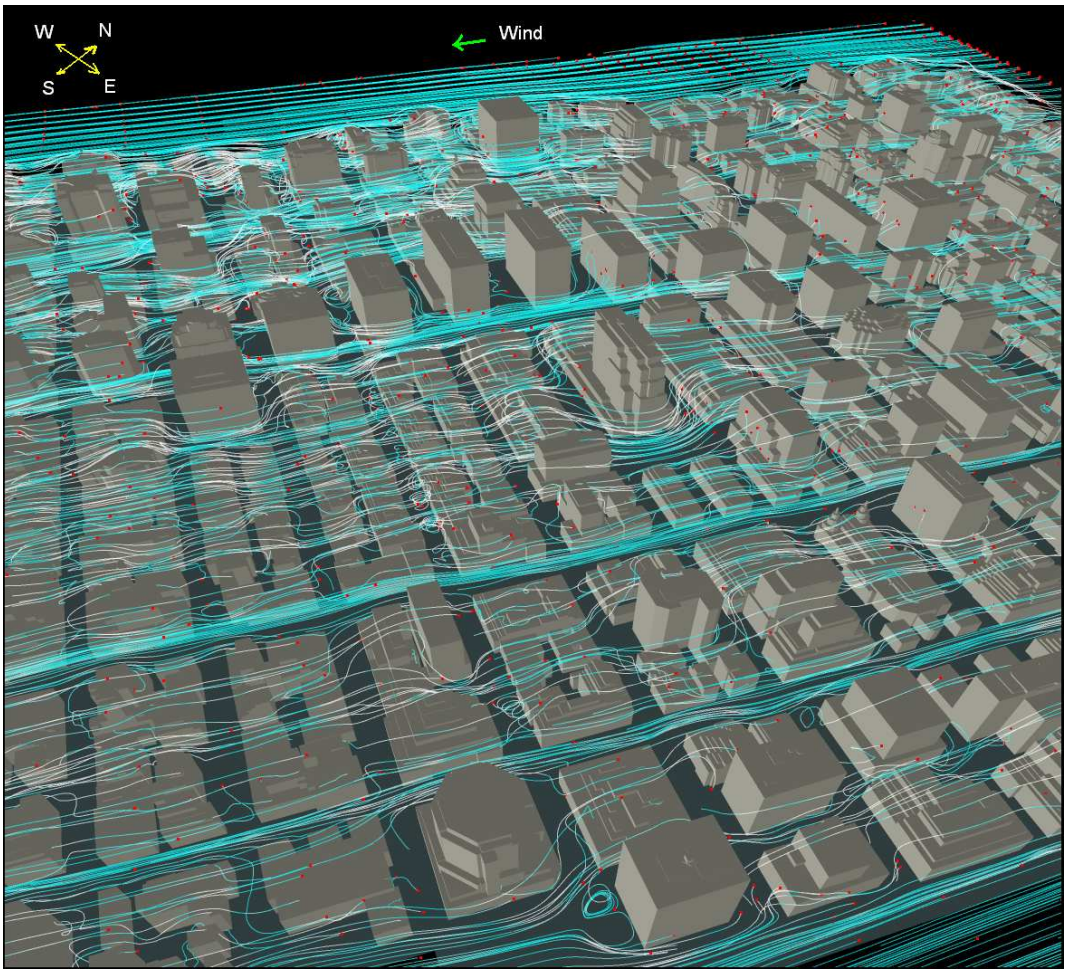
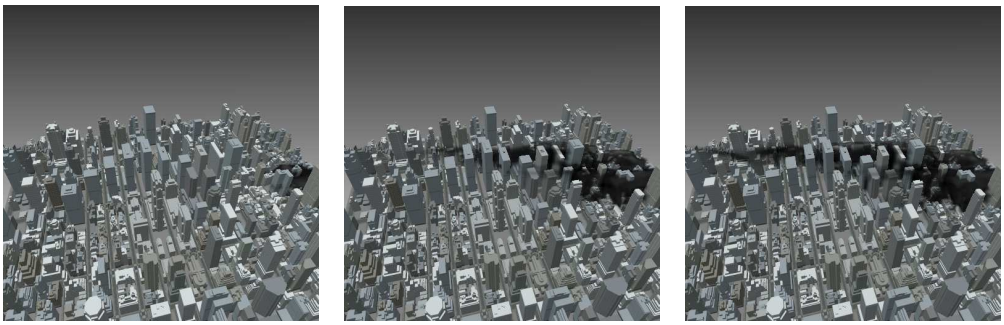


Figure 5.8: A snapshot of the simulation of air flow in the Times Square area of New York City at time step 1000, visualized by streamlines. (Simulation lattice size is $480 \times 400 \times 80$. Only a portion of the simulation volume is shown in this image.)

(a) (b) (c)



(d) (e) (f)

Figure 5.9: Smoke dispersion simulated in the Times Square Area of New York City. (a)-(c) are snapshots during navigation at different time steps. (d)-(f) are bird-eye views (in which the wind is blowing from right to left).

Chapter 6

LBM of Irregular-Shaped Simulation Domain on GPU Cluster

In this chapter, we present a simulation and visualization system for a critical application—analysis of the thermal fluid dynamics inside a pressurized water reactor (PWR) of a nuclear power plant when cold water is injected into the reactor vessel. We have worked closely with the PWR scientific engineers and have developed this visual simulation system. We employ a hybrid thermal lattice Boltzmann method (HTLBM) [81] for modeling the thermal fluid dynamics. The simulation demonstrates the formation of cold-water plumes in the reactor vessel. A set of interactive visualization tools, such as side-view slices, 3D volume rendering, thermal layers rendering, and panorama rendering, are provided to collectively visualize the structure and dynamics of the temperature field in the vessel.

In Chapter 5, we have presented an LBM implementation with a rectangular simulation domain on a GPU cluster. The simulation presented in this chapter is new and challenging in that we simulate the thermal fluid dynamics in the special geometry of an irregular-shaped PWR. If we straightforwardly use a rectangular simulation domain which contains the PWR geometry, only 5.8% of the lattice cells are useful. Hence, for efficient computation and memory consumption, we propose an LBM implementation of irregular-shaped simulation domain on the GPU cluster which packs only the nonempty LBM cells in GPU texture memories. To our knowledge, this is the first system that combines 3D simulation and visualization for analyzing pressurized thermal shock (PTS) risk in a pressurized water reactor.

This is also the first system that uses the LBM in PTS analysis with fast GPU cluster computation.

6.1 Background

The assessment of PTS risk in reactor vessels has been of great interest to the designers and constructors of pressurized water reactors in nuclear power plants. The purpose of the PTS investigations [25] is to assess whether the brittle fracture of the reactor vessel is credible during various off-normal or transient events. One of the concerns recently raised is the possibility of reactor vessel failure due to the formation of cold water plumes when cooling the reactor system at an excessive rate during these events.

In previous work, the U.S. Nuclear Regulatory Commission (NRC) has undertaken studies to investigate the performance of the reactor vessel during off-normal or transient events. Li and Modarres [85] have discussed in details the history of PTS studies. However, most of the studies are based on experiments rather than numerical simulations. The PTS investigation required the analysis of a large number of transient events to determine the thermal hydraulic conditions in the reactor vessel over a period of time. This thermal-hydraulic analysis has been performed in 2D with the RELAP5 computer code. Various reviewers of the PTS analysis have raised the issue of the possibility of cold water plume formation in the reactor vessel downcomer and the potential impact on vessel performance. Because of this possibility, it is necessary to further use 3D simulation to study whether the fluid circulation and mixing is sufficient to dissipate cold water plumes that may be formed. Martin and Bellet [98] have analyzed the qualification of finite element based and finite difference based CFD software for 3D simulation of the physical phenomena during PTS. In these PTS studies, the simulation results are visualized with the temperature history plots for several key positions and the surface color maps. With these elementary visualization tools, it is difficult for the designers and analysts to infer the structure and dynamics of the temperature field.

Unlike the previous work, our simulation uses the HTLBM which models the fluid at a mesoscopic level and hence have the following advantages. The boundary condition for the complex PWR vessel geometry can be easily handled. The

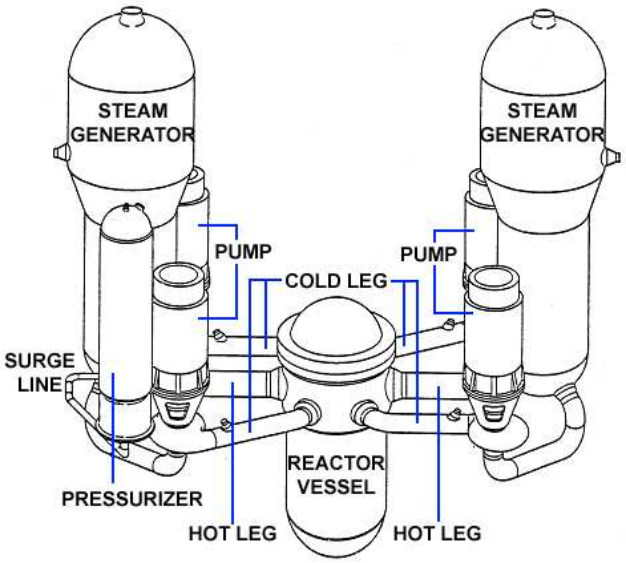


Figure 6.1: Typical layout of a combustion engineering PWR [91]

computation operations are local and linear, hence can be fully parallelized on the GPU and the GPU cluster. Moreover, our system provides a set of interactive 3D visualization tools to help understanding the thermal fluid dynamics and analyzing the structure and dynamics of the temperature field.

Figure 6.1 presents the general arrangement of a PWR with a combustion engineering plant design. The normal flow path is from the reactor vessel through the hot legs, the primary side of the steam generators, and back to the vessel through the reactor coolant pumps and cold legs. Plant designs from other PWR vendors differ in the number of steam generator loops and the steam generator design among other details. Figure 6.2 shows the internal structure of a typical reactor vessel. The downcomer is the annular narrow region between the core support barrel and the reactor vessel wall. The nozzle labeled “30 inch ID Inlet Nozzle” is the cold leg inlet and the nozzle labeled “42 inch ID Outlet Nozzle” is the hot leg connection. The region of interest for this analysis is the cold leg piping and reactor vessel downcomer.

One class of transients that is of interest to PTS risk analysis is the small-break loss of coolant accidents (SBLOCA). The LBM analysis focuses on a 5.08 cm (2.0 in) diameter pipe break in the surge line that connects to the hot leg (see

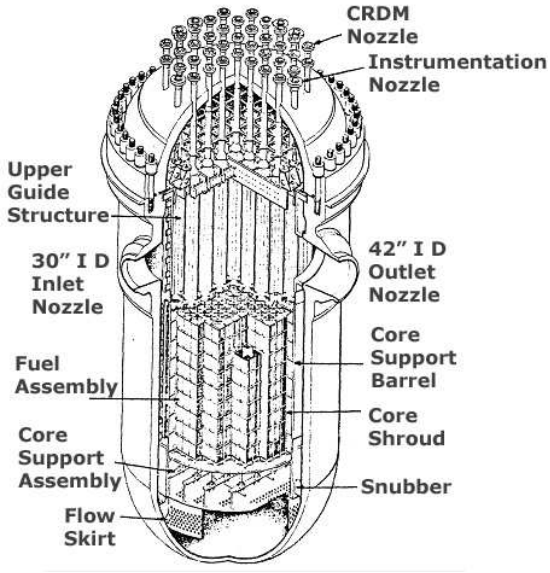


Figure 6.2: General arrangement of a typical combustion engineering PWR reactor vessel and internals [91]

Figure 6.1). When the break occurs, the reactor coolant system (RCS) pressure decreases rapidly from the normal operating value of 15.5 MPa (2250 psia) at first, and then more slowly as water-to-steam flashing occurs within the RCS.

The RCS depressurization leads to initiation of flow from the injection systems typically at a temperature of 305 K (90°F), which are designed to maintain water inventory in the reactor system during a LOCA. The decreasing RCS pressure ultimately results in the operator turning off the reactor coolant pump in each cold leg to prevent pump failure. With no pumped flow in the system, a transition from forced to natural circulation behavior occurs for a short time. This natural circulation flow keeps the primary system fluid well mixed (i.e., no cold plumes form in the downcomer). However, continued loss of RCS fluid inventory stops coolant loop natural circulation flow through both loops. The loss of natural circulation flow is referred to as "loop flow stagnation" and is a significant point of interest in the PTS analysis because afterward the influence of the cold injection water on the downcomer is the greatest. During the loop flow stagnation period, the fluid flow velocity around the loop, and particularly in the cold leg, becomes practically zero. The only mass flow through the cold leg during loop flow stagnation is that from

the high pressure injection (HPI) system and this flow is from the injection nozzle downstream of the pumps (see Figure 6.1) towards the downcomer. The temperature of the fluid in the reactor system at this time is 464 K (375°F) at a pressure of 4.34 MPa (630 psia).

The potential for plumes exists because cold water (305 K (90°F)) injected into the cold leg due to HPI operation will flow down the cold leg piping into the hot downcomer towards the lower plenum. Depending on the amount of mixing that occurs in the cold leg and downcomer, thermal stratification may occur in the cold leg providing a potential for the formation of thermal plumes in the downcomer. If thermal plumes exist in the downcomer, a larger thermal gradient through the vessel wall (relative to the average downcomer fluid temperature) may result affecting vessel performance. The LBM analysis focuses on determining whether cold water plumes form in the reactor vessel when the loop flow is stagnant.

6.2 Modeling of Thermal Fluid Dynamics

6.2.1 Multi-Relaxation-Time LBM

In Section 2.2.3, we have presented the basic LBM model. This LBM is called called single-relaxation-time LBM (SRTLBM). In Equation 2.4, only one parameter, τ, is used to control the collision of particle distribution functions. The SRTLBM is prone to unstable numerical computation when used for highly turbulent fluids or incorporated with temperatures or body forces. Therefore, we use a new version of the LBM, multiple-relaxation-time LBM (MRTLBM) [20]. It differs from the SRTLBM in that a new collision operator replaces the single-relaxation-time collision in Equation 2.4. The new collision operates in the space of hydrodynamic moments representing density, momentum, energy, etc. These moments, denoted as m_i, are mapped from the particle distributions as

$$|m\rangle = M|f\rangle, \tag{6.1}$$

where $|f\rangle = (f_0, f_1, ..., f_n)^T$, $|m\rangle = (m_0, m_1, ..., m_n)^T$ and n is the number of lattice links of a node to its neighbors. For the D3Q13 lattice, each of the 13 moments has a physical meaning: m_0 is the mass density; $m_{1,2,3}$ are the components of the

momentum vector; m_4 is the energy; and the other higher order moments are components of the stress tensor and other high order tensors. Although the values of the distributions and the moments vary over the lattice sites, M is simply a constant matrix for a given lattice structure. Mathematically, the MRTLBM collision is described as:

$$|f(\vec{r},t^+)\rangle = |f(\vec{r},t)\rangle - M^{-1}S[|m(\vec{r},t)\rangle - |m^{eq}(\vec{r},t)\rangle], \tag{6.2}$$

where S is a diagonal matrix. Its diagonal elements $\{s_i|i = 0,1,...,n\}$ are the relaxation rates [20]. It has been proven that the MRTLBM increases the numerical stability [81]. Body forces and heat effects can be easily added to the moments m_i, because the moments have explicit physical meanings, such as momentum and energy.

6.2.2 Hybrid Thermal Lattice Boltzmann Method

Based on the MRTLBM, Lallemand and Luo [81] have developed the HTLBM for coupling thermal effects to the LBM. The temperature evolution is governed by a diffusion-advection equation:

$$\partial_t T + \vec{u}\cdot\nabla T = \kappa\Delta T, \tag{6.3}$$

where κ is the thermal diffusivity of the material and u is the velocity. This equation is solved by finite-difference operators, where the x, y, and z components of the gradient ∇T and the Laplacian operator $\triangle T$ are computed by the finite difference operators [20].

The temperature is coupled to the LBM in order to model the interaction between the heat and the fluid dynamics. The moment m_4 represents the energy, therefore, the temperature can be added to the LBM when computing the equilibrium m_4^{eq}:

$$m_4^{eq} = n_1\rho + n_2\rho^2(\vec{u}\cdot\vec{u}) + n_3 T, \tag{6.4}$$

where the parameters, n_1 to n_3, are constants and physically defined by the linear analysis. After coupling T to m_4^{eq}, the method could model the thermal fluid dynamics.

Because the LBM lattice not only discretizes the simulation domain space but also subdivides the angular space, the complex surface of the irregular-shaped vessel can be effectively captured by the intersection points of the lattice links with the surface. Boundary conditions are the simple local rules applied after the streaming step. Such boundary conditions are treated with a bounce back rule. In this case, the outgoing particle distribution re-enters the grid at the same node, but associated with the opposite velocity. The rule is to copy the mirror image of the packet distributions of the nodes that are located on the fluid side of the boundary to those on the solid side. In the PTS analysis, the plume behavior is affected by the temperature of the vessel wall. We include in the simulation the heat transfer from the wall to the fluid. Initially, the vessel wall and the fluid are at the same temperature. As cold water is injected, heat exchange occurs between the wall and the fluid.

Typical LBM computation achieves second-order accuracy for solving Navier-Stokes equations [136]. The thermal LBM method we used has been applied and validated to simulate turbulent convective flows in 2D and 3D cases [145]. Validation studies [145] have been presented and compared to benchmark data for laminar and turbulent natural convection in a cavity. Results obtained for both the 2D and 3D case agree very well with existing benchmark data. The deviation from benchmark data is within 0.8%. They also show that the scheme yields quadratic convergence rate in space.

6.3 Simulation

6.3.1 Configuration

Figure 6.3 shows a 3D polygonal model created according to the specification of the vessel downcomer. The model is created with Maya Complete 6.0. It includes the downcomer, cold legs, and HPI injection pipes. We have carefully constructed this model based on the layout of the reactor vessel (Figure 6.1). The cold water is injected from the HPI injection pipes and flows through the cold legs into the lower plenum.

Using the LBM as a fluid solver, one has to determine the LBM configuration

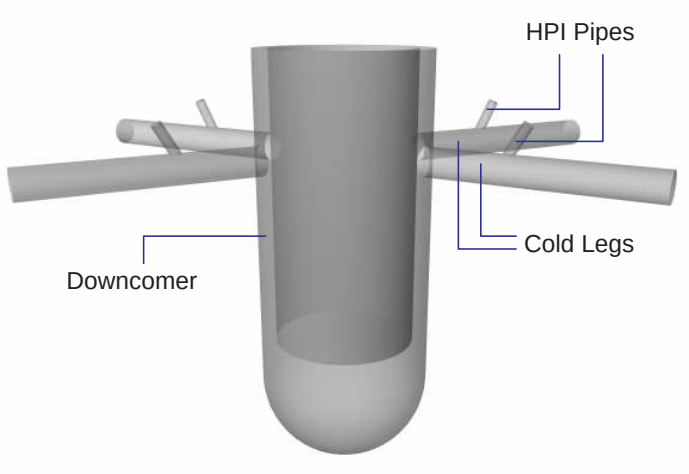


Figure 6.3: The geometric model of the vessel created for the simulation.

of grid size, time step, characteristic speed and size, etc., observing the configuration rules [150]. The following configuration is used in our simulation. We set the grid spacing (spacing between two neighbor lattice sites) $\delta x \doteq 0.02m$, and each simulation step $\delta t \doteq 0.01s$. The spacing and time step size determine the global lattice dimensions of the entire simulation domain at $800 \times 600 \times 480$.

The flow speed of the HPI injection is computed by the given flow rate, which is $35.0lb/s = 15.876kg/s$. The radius R of the HPI injection pipe is $R = 0.27m$ and the density of the water is $\rho \doteq 1 \times 10^3 kg/m^3$. The flow rate of the water can be computed as $\rho\pi R^2 U$; therefore, the flow speed of the HPI injection is computed as $U = 0.28m/s$. In the LBM simulation, the flow speed of the HPI injection is $U = 0.14$ in LBM units, which is computed from δx and δt. The viscosity is set as $\nu = 4 \times 10^{-3}$ in LBM units, therefore, the Reynolds number (which is the ratio of inertial forces to viscous forces) in the cold leg is approximately $Re \doteq 1400$.

For the thermal effects, the thermal diffusivity is computed as $\kappa = 4.05 \times 10^{-3}$ in LBM units from the value of viscosity ν because the Prandtl number, a dimensionless number approximating the ratio of momentum viscosity and thermal diffusivity, $Pr \doteq 1.0$. The initial fluid domain temperature is 190.56 °C and the cold water temperature is 31 °C. We use 0.1 in LBM units to represent initial fluid temperature, and the cold water temperature is set as 1.63×10^{-2} in the simulation,

which keeps the simulation stable and the effective Rayleigh number (a dimensionless number associated with the heat transfer within the fluid) in the cold leg is approximately $Ra \doteq 4.8 \times 10^4$.

6.3.2 Cell Classification and Packing

Figure 6.4 illustrates the inner geometry of the PWR, where the fluid flows inside the irregular-shaped vessel. Obviously, the empty cells waste much memory space and computation time in a rectangular simulation domain, since there is no discrepancy in treating the empty cells and the interior cells in the computation. Based on this observation, we propose a new LBM implementation with cell classification and packing. In this application, only 5.8% of cells are are stored and computed. Hence, it significantly improves the simulation performance in terms of storage and computation time.

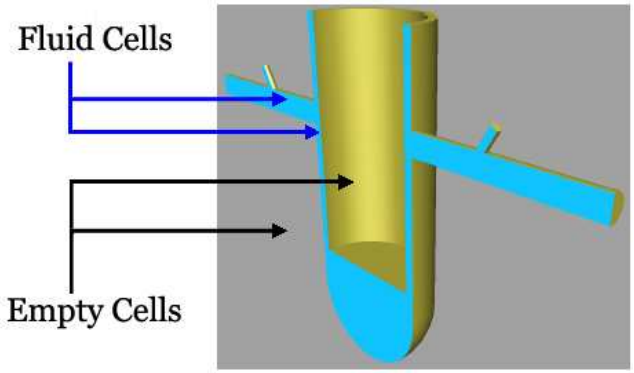


Figure 6.4: Because of the irregular shape of the reactor vessel, only 5.8% of cells are useful if we straightforwardly use a rectangular region as the simulation domain.

We classify the fluid and empty cells in a preprocessing stage. First, we mark the cells close to the vessel mesh as the border cells by an intersection test of the mesh and simulation cells. Second, starting from a given interior cell (the seed), a region-growing algorithm is implemented by a depth-first search to identify all the interior cells. The interior and border cells are then compactly stored in the memory for simulation. An indexing table is constructed to translate between the cell index in the memory and its position in the simulation domain.

6.3.3 GPU Cluster Implementation

Even with the cell classification, the computational resource consumption is still too extensive for a single CPU or GPU. We further accelerate our simulation by performing the computation on a GPU cluster. This cluster has 32 nodes connected by a Gigabit Ethernet. Each node has an NVIDIA Quadro FX 4500 GPU, dual Intel Xeon 3.6GHz CPUs, and a PCI-Express bus.

The simulation domain is divided into sub-domains and distributed to multiple nodes. Figure 6.5 shows the decomposition of the model. The LBM simulation on each sub-domain is computed simultaneously with proper synchronization. The communication between them is implemented on their interfaces. Therefore, we have also marked those cells on the boundary of each sub-domain as interface cells during the cell classification.

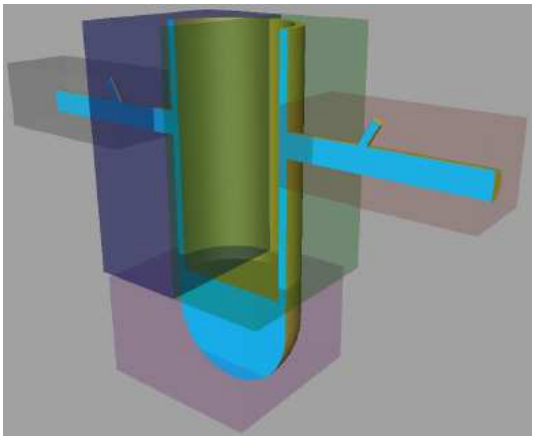


Figure 6.5: Decomposition of a simulation space into cuboidal sub-domains.

GPU data structures are generated in preprocessing. The non-empty cells of each sub-domain is packed into the 2D textures on a GPU. Some textures are used to store the flow properties, such as density, velocity, particle density functions, and temperature. An example of such a texture is shown in Figure 6.6(a). The region of the texture is divided into sub-regions that store interior cells, border cells, and ghost cells, respectively. An interior cell is inside the sub-domain assigned to the GPU and all of its neighboring cells are also inside this sub-domain. A border cell is inside the sub-domain while some of its neighboring cells belong to another

GPU. When a cell is located in another GPU but some of its neighboring cells belong to the sub-domain of the GPU, we use a ghost cell to represent it. During the simulation, the ghost cells are used to store data received from other GPUs, so that the computation only needs to access the local GPU data.

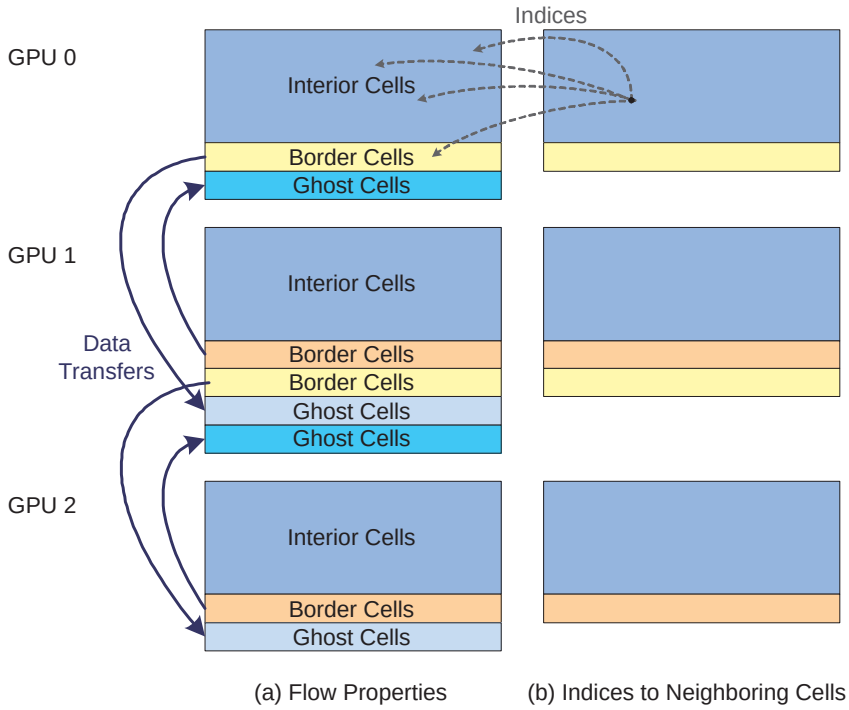


Figure 6.6: The data structures stored on the GPUs: (a) textures packing the flow properties, and (b) textures packing the indices of neighboring cells.

Because of the irregular shape of the simulation domain, the connection information among cells need to be stored in index textures (as shown in Figure 6.6(b)). Each texel stores the indices of neighboring cells for a given cell. However, storing this information for ghost cells is unnecessary because their flow properties do not have to be locally computed.

To initiate the simulation, multiple processes are launched in the cluster, each running on a cluster node and controlling a local GPU. In each process, every simulation step includes the following sub-steps: collision operation, communication with other processes, streaming operation, boundary conditions, and computation of the temperature field. The communication sub-step brings the data from remote

GPUs to the local GPU and the other sub-steps compute on the data locally.

During the communication sub-step, each process reads out from the GPU the flow properties of border cells and sends the data to the neighboring processes (see the data transfer in Figure 6.6). After receiving the data, each process writes the data to the GPU in order to update the flow properties of the ghost cells. In the communication process, the MPI functions MPI_Send and MPI_Recv are used to send or receive data on the network. To read the data out from the GPU, the corresponding texture is bound to an OpenGL FrameBuffer Object (FBO) and the function glReadPixels is used. Similarly, glWritePixels is used to write the data to the GPU.

The computation sub-steps are implemented with the GPU fragment programs. In operations, the textures of the source data are bound to texture units, while the destination texture to store results becomes renderable by being bound to an FBO. The algorithms specified in the fragment programs are executed on the flow properties in SIMD fashion. In some computation sub-steps, such as the streaming operation, the fragment program needs to access the flow properties of neighboring cells. The address is calculated based on the indices of the neighboring cells stored in the index textures.

Table 6.1 reports the simulation performance on our cluster with and without GPU acceleration. On 12 CPU nodes, the simulation of resolution $800 \times 600 \times 480$ takes 8.8 seconds in each simulation step. The GPU cluster implementation using the same number of nodes runs at 2.8 seconds per simulation step. About a three-time speedup has been achieved.

Table 6.1: Average computation time (in ms) of a single LBM step tested on 12 cluster nodes with and without GPU acceleration.

Resolution	CPU Cluster	GPU Cluster	Speedup
$\mathbf{800 \times 600 \times 480}$	8759	2810	3.1
$\mathbf{600 \times 450 \times 360}$	3927	1297	3.0
$\mathbf{400 \times 300 \times 240}$	1150	405	2.8

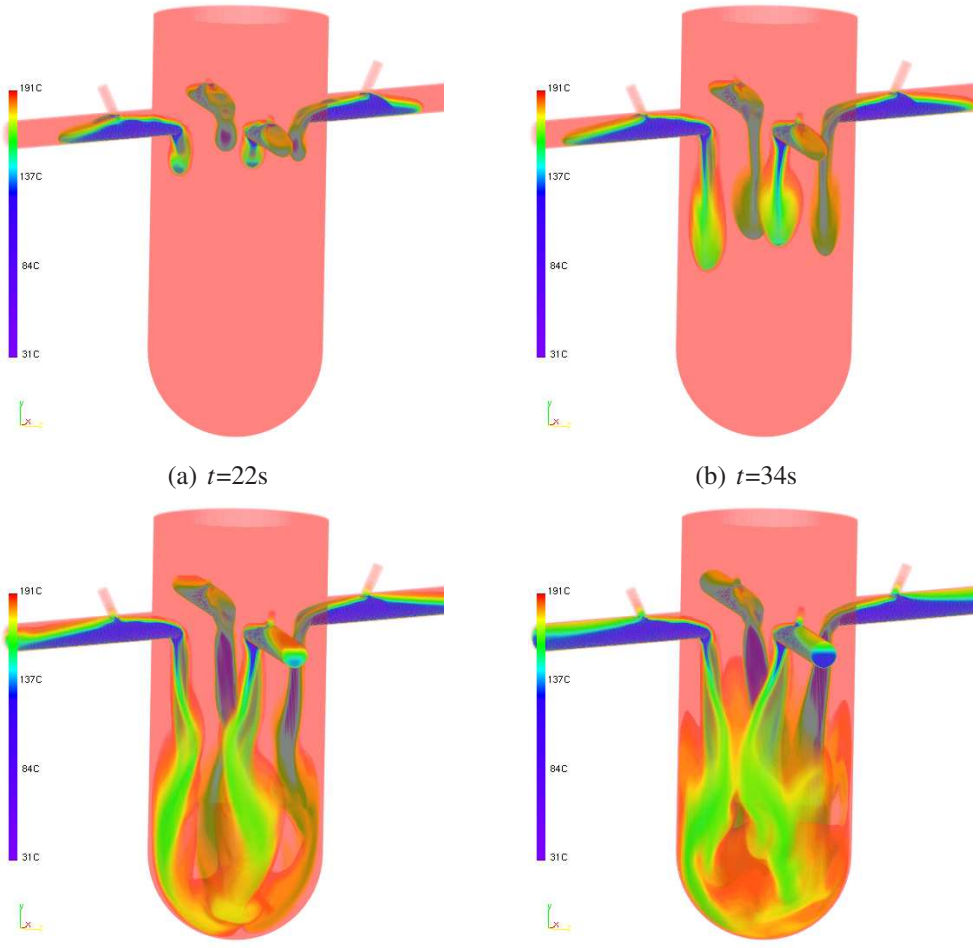
6.4 Visualization

6.4.1 3D Volume Rendering

The basic tool of our visualization system is a GPU-based 3D volume rendering program, which allows the user to interact with the volumetric temperature field in real-time. We employ a high quality rendering algorithm, the pre-integrated volume rendering [23]. The transfer function used in the volume rendering is based on the HSV color model. We store the transfer function in a 1D texture for translating temperature to color and opacity. Based on the user intuition, the lowest temperature is assigned a blue hue ($\frac{4}{3}\pi$) and the highest temperature is assigned a red hue (0) in default. Because the user is more interested in the distribution of low temperature water, the opacity is inversely proportional to the temperature. The user can change the transfer function by assigning the colors to temperatures freely in the HSV color space. Currently, the HSV color model is used because the scientists in nuclear reactor safety analysis are familiar with it. It is straightforward to change from HSV to a CIE *Lab* color model which is more intuitive to human perception.

We store the temperature field at each simulation time step to disks. After the simulation, these data are combined together from multiple cluster nodes, and a single GPU is used to visualize the data. For efficient storage, the floating point temperature is converted to unsigned byte data through scaling and biasing. To efficiently visualize the data in the irregular-shaped simulation domain, we use multiple 3D textures to store multiple blocks of data. The sizes and positions of the blocks have been shown in Figure 6.5. Each block represents the data copied from one cluster node. As described in Section 6.3.3, the simulation results are organized in an unstructured way using indirect addressing. Thus, in preprocessing of the visualization, we reorganize the simulation results and fill the data into the 3D blocks. During the visualization, the GPU renders the multiple blocks one by one and composites the rendering results into the final image.

Figure 6.7 shows a sequence of snapshots of the cold water plume evolution. The cold water is injected from the HPIs into the four cold legs. The cold water then flows down to the downcomer of the vessel. Due to the interaction between the cold water and the existing hot water, the four plumes appear and affect the temperature of the vessel wall. Finally, the four cold water plumes flow to the bottom plenum of



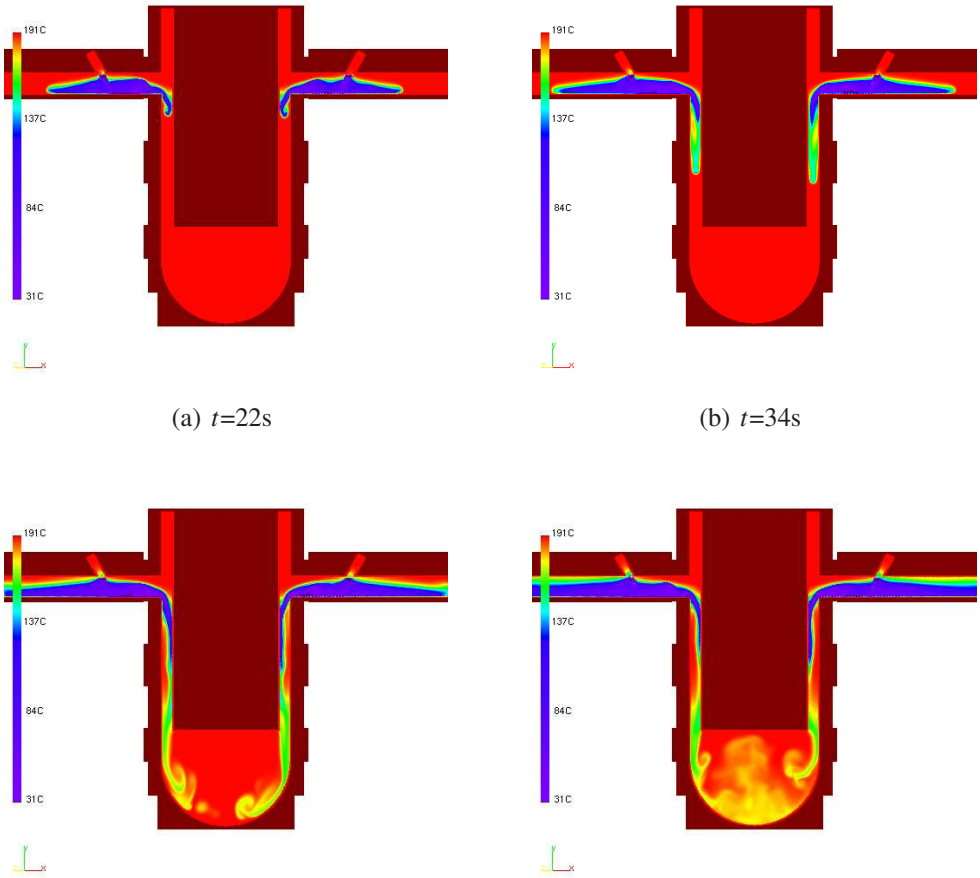
(a) t=22s (b) t=34s

(c) t=58s (d) t=82s

Figure 6.7: The simulation of cold water injected into the reactor vessel rendered using 3D volume rendering. (Time t in seconds is the simulation time.)

the vessel. They start to merge and induce more turbulent flows inside the bottom plenum.

The analysts often want to examine the temperature distribution at some specific positions for precise analysis. Our program also provides a slicing option that can show the temperature distribution on any arbitrary plane with user defined position and orientation in the whole modeling space. Figure 6.8 shows a sequence of snapshots of a particular vertical 2D slice cutting in the middle of the vessel.



(a) t=22s (b) t=34s

(c) t=58s (d) t=82s

Figure 6.8: The simulation of cold water injected into the reactor vessel on a vertical 2D slice in the middle of the vessel.

6.4.2 Thermal Layers Rendering

Volume rendering gives a good overview of the entire temperature field and shows the interesting turbulent flow. However, due to occlusion and composition, it does not clearly depict the internal structure and dynamics of the temperature field, especially the shape and dynamics of the cold water plumes. Scientists usually have in mind an intuitive understanding of the major structure of the temperature field. We have found that the structure is similar to the concept of iso-surfaces in the visualization community. Hence we use iso-surface volume rendering to visualize the structure of the temperature field, called the thermal layers. Our goal is to render multiple translucent iso-surfaces in real-time so that the user can interact with the temperature field and arbitrarily add, remove, and change the temperature iso-values.

One method is to explicitly extract the iso-surfaces and then use a view-independent translucent surface rendering algorithm, such as depth peeling [26], to display the iso-surfaces. However, the performance of both iso-surface extraction and translucent surface rendering greatly depend on the number and complexity of the iso-surfaces. For example, currently, the fastest Marching Cubes iso-surface extraction implementation on the GPU [21] can extract a single iso-surface from a volume of 16M voxels at a speed of 4 FPS to 16 FPS (depending on the percentage of cells intersected by the iso-surface). Our simulation domain contains 13M nonempty voxels. Supposing we want to show ten thermal layers simultaneously, the iso-surface extraction will be performance at approximately 0.5 FPS to 2 FPS, which does not meet our requirement of real-time speed. Note that the rendering of such complex multiple translucent surfaces is also time consuming and will decrease the frame rate.

We instead employ the iso-surface volume rendering algorithm based on the pre-integrated rendering algorithm [23]. Our implementation of pre-integrated volume rendering is reused here and only the transfer function is redesigned based on the selected iso-values. For each iso-value, a "$\bigcap$" shape is defined in the transfer function. The user can easily add, remove, and change the iso-values. The transfer function and the pre-integration table are updated accordingly. The advantage of this method is that the rendering performance is totally independent of the number and complexity of the thermal layers. As shown in the third row of Table 6.2, our

rendering method achieves 17 FPS speed on the GPU. Figure 6.9(a) shows our rendering results of five thermal layers. Compared with the 3D volume rendering in Figure 6.7, it shows less turbulence but more clearly depicts the main structure of the temperature field and the cold water plumes. Figure 6.9(b) and Figure 6.9(c) show two close views of the key locations, where the temperature field matches the prediction of scientists [98]. Figure 6.9(b) shows the development of thermal layers in the cold leg, where hot layer is on the top and cold layer is at the bottom. Figure 6.9(c) shows that when the water flows into the downcomer, the cold water layer penetrates the warm and hot water layers and touch towards the inner wall of the downcomer (core barrel) as opposed to the reactor vessel wall.

6.4.3 Panorama Rendering

Although the 3D views present the entire temperature field to the user, some of the information is lost due to occlusion and compositing. In this particular application, an important part to be analyzed is the small toroidal region between the inner and outer boundary of the downcomer, which is only a small part of the entire simulation domain. A possible solution is to employ an empty space skipping technique, but it is a complex algorithm and does not solve the occlusion problem. Instead, we propose a GPU-based panorama ray-casting algorithm which parameterizes the toroidal downcomer and maps it to a rectangle space. The algorithm efficiently displays the temperature distribution in the downcomer in a single image, so that the scientists can view the global distribution without occlusion and can easily find key positions for further investigation. Figure 6.10(a) demonstrates a panorama volume rendering result image. Figure 6.10(b) shows the temperature distribution on the surface of the vessel downcomer without compositing. The panoramic view also results in an intuitive user interface: it allows the scientists to drag the mouse over the view and see in a separate view the digital values of the temperature at interesting positions (see Figure 6.10(b)).

Figure 6.11 illustrates the algorithm. The rays are cast from the centerline of the vessel and they are perpendicular to the centerline. We construct a cylindrical coordinate system (r, θ, y) where θ is the angle between the ray and the x axis. The y axis is the centerline of the vessel. Given a point (u, v) in parameter space, its

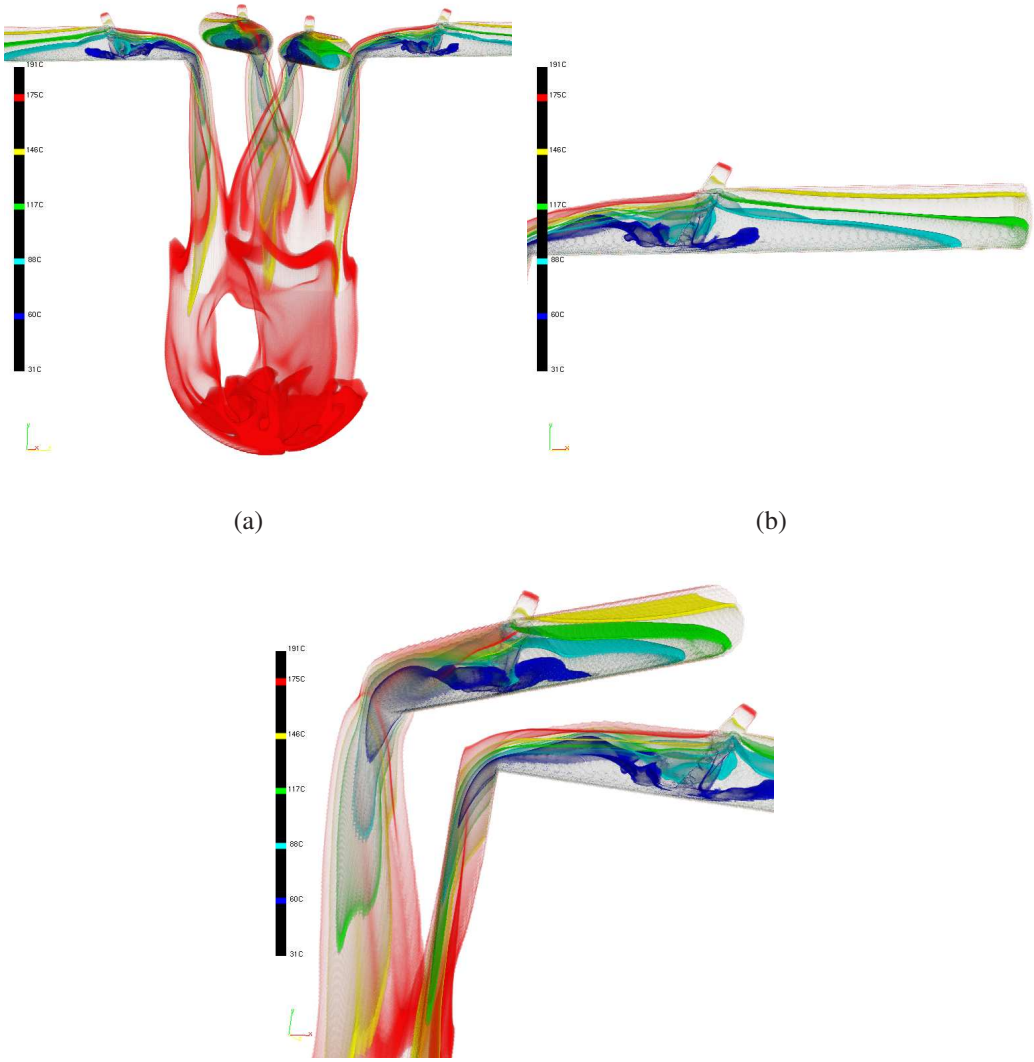
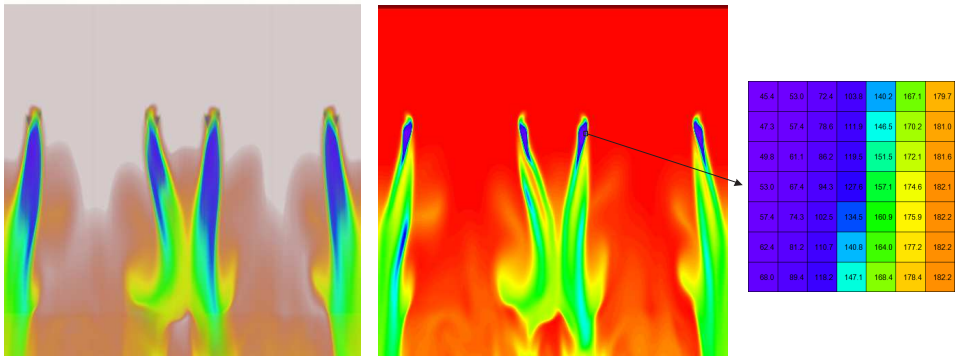


Figure 6.9: The rendering of five thermal layers: (a) an overview, (b) a close view showing the thermal layers developed in the cold leg, and (c) a close view showing the cold water layer penetrates the warm and hot water layers.



(a) (b)

Figure 6.10: Panorama rendering (a) with composing and (b) without compositing. (The user can drag the mouse over the panorama view and see the temperature values of interesting position in a separate view.)

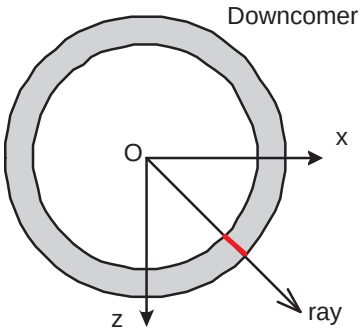


Figure 6.11: Illustration of panorama ray-casting. (This figure shows a horizontal cutting plane that intersects with the downcomer.)

corresponding ray segment starts from $(r_0, 2\pi u, v)$ and ends at $(r_1, 2\pi u, v)$, where r_0 and r_1 are the radii of inner and outer cylinder, respectively. In practice, a rectangle is drawn to the screen with proper texture coordinates ($[0..1] \times [0..1]$). The fragment program first calculates the ray direction and the origin for each fragment. Then, a single-pass ray casting method is used to calculate the accumulated color and transparency on each ray. The radii r_0 and r_1 are used for empty space skipping and ray termination. Because only a small portion of voxels are processed, the panorama rendering achieves a speed as high as 129 FPS on a GeForce 8800 GTX (shown in Table 6.2).

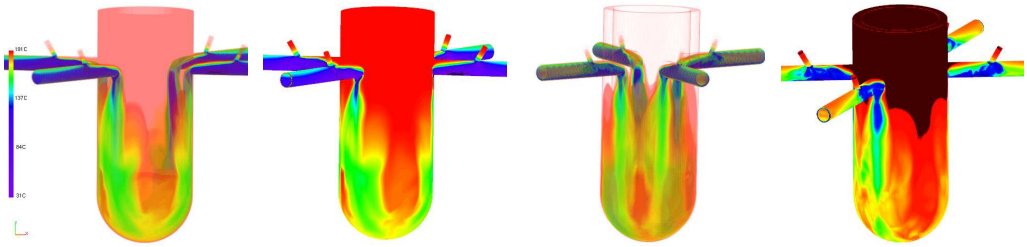
Table 6.2: Performance of our visualization methods, tested on a GeForce 8800 GTX. (The image size is 512^2. The step size of ray-casting is one grid unit.)

	Frames per second
3D volume rendering	17
Slice view rendering	880
Thermal layers rendering	17
Panorama rendering	129

6.4.4 Statistical Analysis

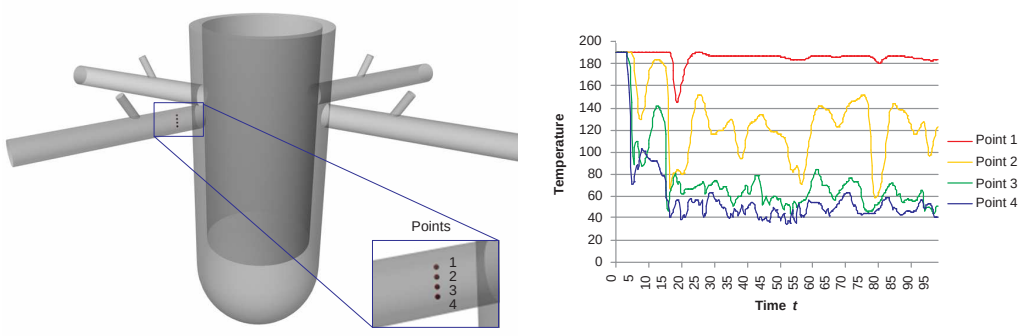
For a complete PWR safety analysis, our thermal fluid dynamics simulation results will be converted to inputs of a structural simulation for further examination of the vessel performance. In our current system, we provide statistical analysis tools to help the analysts identify the key regions before the structural simulation. Intuitively, the regions of the lowest temperature and/or highest temperature gradient are of most interest. Hence, we record the minimum temperature and maximum gradient magnitude for each lattice cell during the simulation. The volumes of minimum temperature and maximum temperature gradient magnitude are then downloaded to a GPU for visualization. The GPU renders the PWR geometry with the color map based on the data sampled from the 3D textures that store the minimum temperature and maximum temperature gradient. Figure 6.12(a) and Figure 6.12(b) show the volume rendering and polygonal rendering images of the minimum temperature over the simulation time. Figure 6.12(c) and Figure 6.12(d) show the maximum temperature gradient magnitude.

For positions of special interest to the scientists, the system can output accurate temperature readings for further analysis. Figure 6.13(a) shows four points specified by the user inside the cold leg. For these points, the temperature history is plotted in Figure 6.13(b). The figure shows that in the cold leg the upper points have higher temperature than the lower points. This phenomenon agrees with our thermal layer rendering result shown in Figure 6.9(b).



(a) (b) (c) (d)

Figure 6.12: In (a) and (b), the colors represent the minimum temperature over the simulation time, with volume rendering and surface rendering, respectively. In (c) and (d), the colors represent the maximum temperature gradient over the simulation time.



(a) (b)

Figure 6.13: (a) Four points (shown in the blue rectangle) in the cold leg defined by the user. (b) The plots of the temperature history at these points.

Chapter 7

Zippy: A General Framework for GPU Clusters

Recently, major GPU vendors have started to target the HPC market. For example, NVIDIA has announced the Tesla S1070 [2], a 1U rack-mount server with 4 GPUs dedicated to computation. Both NVIDIA Tesla S1070 and AMD FireStream 9170 [1] support double-precision floating point computation. At the same time, as researchers have accelerated a wide range of general-purpose applications on the GPU, the need arises for using the GPU cluster to achieve further acceleration and to support an increase in the problem size. From the point of view of hardware and application, using GPU clusters for general-purpose computation and visualization is becoming more and more attractive.

From the software point of view, however, programming GPU clusters is difficult, low-level, and error-prone. The complexity of programming also makes performance optimization difficult. In this chapter, we introduce the Zippy framework to simplify the GPU cluster programming. Zippy abstracts the GPU cluster with a two-level parallelism hierarchy and a non-uniform memory access (NUMA) model. Zippy preserves the advantages of both message passing and shared-memory models. It employs global arrays (GA) to simplify the communication, synchronization, and collaboration among multiple GPUs. Moreover, it exposes data locality to the programmer for optimal performance and scalability. We present three example applications developed with Zippy: sort-last volume rendering, Marching Cubes

isosurface extraction and rendering, and the LBM flow simulation with online visualization. They demonstrate that Zippy can ease the development and integration of parallel visualization, graphics, and computation modules on a GPU cluster.

7.1 Background

Researchers have explored GPU clusters for several specific applications. Besides our LBM flow simulation [32], GPU clusters have been used for volume compression and rendering [135], occlusion culling [50, 156], biological sequence search [62], surface flow visualization [6], N-body simulation, finite element computation [45], and data clustering [137]. In these implementations, the GPUs are programmed using the stream computing model [9, 102] and the communication among the cluster nodes are programmed using the Message Passing Interface (MPI). With MPI, the programmer can explicitly control the data locality and communication, hence application performance can be understood and optimized. However, the programming is tedious. MPI, sometimes referred to as "the assembly language of parallel computing," is low-level and error prone. For example, for every data transfer, the programmer needs to explicitly specify which processor sends and which receives the data. Some algorithms are complex to express in this way. Moreover, because MPI only deals with system memories of cluster nodes rather than GPU memories, integrating MPI and GPU computation is very inconvenient for the programmer.

Moerschell and Owens [105] have implemented a distributed shared-memory (DSM) abstraction for multi-GPU environments. This method virtualizes the distributed texture memories as a shared-memory and allows computation kernels to directly read and write remote data, hence simplifies the programming. However, currently it has only been implemented on a dual-GPU PC and the dual-GPU program has not outperformed the single-GPU program. An important contribution of their work is identifying the bottlenecks of the DSM and discussing possible solutions. One major bottleneck is the high latency overheads associated with maintaining the data consistency. Due to current GPU hardware and driver limitations, each program using the DSM must be separated into multiple passes that read data from or write data to GPUs. A page-based method is used to maintain the data

consistency; when a computation accesses a part of a remote page, the whole page will be transferred. Choosing a proper page size is important to reduce the performance loss caused by unneeded data transfers. Also, the DSM hides data locality and communication. A side effect is that it is unclear to the programmer how the application performance is affected by the data access patterns or how to improve the patterns.

Zippy combines the best features of the MPI+stream computing method and the DSM method. It abstracts a GPU cluster with two characteristics critical to high performance, the non-uniform memory accessing (NUMA) and the two-level parallelism hierarchy, and hides other details from the programmer. A GPU cluster is a NUMA environment: the bandwidth of remote GPU memory access is several orders of magnitude lower than local GPU memory access while the latency is several orders of magnitude higher. Therefore, Zippy allows the programmer to manage data locality and communication. Convenient coarse-grained communication operations are used to bring remote data to local GPU memory and kernels only operate on the local data. In other words, Zippy provides the programmer a clear view of two levels of parallelism, coarse-grained parallelism among all GPUs and fine-grained parallelism within each GPU, instead of mixing them.

To present the NUMA and the two-level parallelism hierarchy to the programmer, we adapt the Global Arrays (GA) programming model [113] to the GPU cluster and combine it with the stream computing model. Compared with the MPI-based method, our method has two advantages: (1) its high level data structures and API encapsulate the low-level communication details using global spaces and simplify the programming; (2) as the n-dimensional array is the common data structure for both GA and stream computing, they are seamlessly integrated into one framework. Compared with the DSM, our method also provides a virtual shared memory space but is simpler to implement. Moreover, it exposes data locality and communication to the programmer. The programmer has precise control of when and which data region is transferred, which is unavailable in the DSM. Therefore, better performance and scalability can be obtained.

Previous software systems [7, 66] for graphics clusters are specialized for parallel rendering and mainly handle the transfer of image data and graphics commands. With the programmability of modern GPUs, Zippy provides a general programming interface that blurs the boundaries between graphics, visualization, and computation, and to expressly support the graphics and visualization community with its tasks on GPU clusters. We demonstrate three example applications developed using Zippy: (1) sort-last volume rendering, a traditional parallel graphics application; (2) Marching Cubes isosurface extraction and rendering, a more complex graphics algorithm that benefits from our general programming interface; and (3) lattice Boltzmann model flow simulation and online visualization, an integration of general-purpose computation and visualization.

There are other related research directions. Fatahalian et al. [37] have proposed the Sequoia programming language and implemented it on a Cell workstation and a PC cluster. Unlike Sequoia that focuses on memory hierarchies and abstracts programs as trees of memory modules to address vertical communication, Zippy focuses on the two-level parallelism hierarchy and uses a regular GA data structure to simplify horizontal communication. However, both explicitly expose data locality and communication and encourage block data movement. Yamagiwa and Sousa [160] have proposed the Caravela system for programming GPU-based GRID environments, where the resources are loosely coupled and the primary goal is to efficiently manage a large amount of heterogeneous resources. In a GPU cluster, however, the resources are tightly coupled and the primary goal is to achieve the best possible application performance. Manzke et al. [96] are developing a hardware interconnection in a GPU cluster.

7.2 Zippy Overview

General-purpose computation on a GPU cluster often involves a two-layer communication among GPUs. If a GPU needs data from the memory of another GPU, the data on the source node is uploaded from the GPU memory to the system memory, transferred through the network, then downloaded to the GPU memory on the destination node. The two-layer communication would aggravate the difficulty of programming with MPI. However, to provide a virtually shared space to ease

the communication among multiple GPUs, we believe it is important to make the NUMA and the two-level parallelism hierarchy explicit to programmers.

Table 7.1 lists the theoretical bandwidth and latency of the GPU memory, PCI-Express bus, and the network. The actual data transfer performances are considerably lower. The last two rows show the theoretical bandwidth and latency for a GPU to access data in the memory of a remote GPU. Compared with the GPU memory, the bandwidth is 2-3 orders of magnitude lower and the latency is 2-4 orders of magnitude higher. In fact, due to hardware limitations, a GPU cannot transfer data through the PCI-Express bus when it is computing, which leads to much higher latency. Because of this NUMA nature, Zippy differentiates between the shared space and local space and exposes data locality to the programmer for optimal performance. To alleviate the effects of the high remote GPU memory access latency, the shared-space should be accessed only with coarse-grained operations such as block copy. Also, the stream computing kernels should only operate on the local space, so that the computation does not wait long for an expensive communication.

Table 7.1: Theoretical bandwidth and latency in a GPU cluster, which demonstrates a NUMA characteristic.

	Bandwidth	**Latency**
GPU memory	100 GB/sec	0.05 μsec
PCI-Express	4 GB/sec	0.5 μsec
Infiniband	2.5 GB/sec	1 μsec
GigaE	0.1 GB/sec	50 μsec
Remote GPU memory on Infiniband cluster	2.5 GB/sec	2 μsec
Remote GPU memory on GigaE cluster	0.1 GB/sec	51 μsec

Because of the large performance difference between transferring data among GPUs and within a GPU, Zippy abstracts the GPU cluster with the two-level parallelism hierarchy. Unlike the MPI+stream computing method, in Zippy the two levels are tightly coupled and seamlessly integrated. Unlike the DSM, in which inter-GPU communication can be implicitly triggered by GPU computation, in Zippy the boundary between two levels are clear to the programmer and the communication operations are explicitly initiated.

At the coarse level, multiple GPUs collaborate with each other. We employ the GA model [113] for programming this coarse-grained parallelism. GA has been developed in the parallel computing community and has been employed in applications, such as matrix multiplication, computational chemistry and physics, and electronic structure. It employs a global array data structure and a set of shared-memory style functions for convenient data transfers in the global array space. In addition, GA acknowledges the NUMA characteristics of distributed-memory architectures. It exposes data locality and the management of communication to the programmer. The programmer can make sure that only the needed data are transferred. The original GA toolkit was proposed for PC clusters and supercomputers. To adapt the GA model to a GPU cluster for programming the two-level parallelism hierarchy, we have implemented a new object-oriented library that encapsulates the details of the GPU computation and the two-layer communication.

At the fine level, existing GPGPU toolkits based on stream computing, such as Cg [97], GLSL [123], HLSL [131], Brook [9], Sh [102], NVIDIA CUDA [114], and RapidMind [3], have provided efficient ways to program computation kernels executed on a single GPU. In order to use them to program general-purpose computation on multi-GPU platforms, the programmer needs to manually create threads, handle each GPUs, and explicitly take care of inter-GPU data transfer. (Especially, in the GPU cluster, the programmer needs to take care of the network transfer using MPI.) Zippy presents a method of extending these stream computing toolkits to multi-GPU environments. Zippy programming model seamlessly integrates the fine-grained parallelism with coarse-grained parallelism because arrays are the fundamental data structures in GA and arrays are also natural for expressing stream computing.

7.3 Zippy Framework

The Zippy programmer writes the C++ program using the library classes and functions of Zippy. Multiple processes that execute the program are launched in the cluster. Each process with a unique ID runs on a cluster node. It controls a local GPU and collaborates with other processes. There is no centralized resource in the cluster.

```
// 3D global array, size 8 x 6 x 6
ZDimension oDimGA( 3, 8, 6, 6 );
// Each element is a 32-bit float
ZGlobalArrayType oType( oDimGA, 1, FLOAT32 );
// Divide axis X into three parts, size 3, 2, and 3.
int anSizes[3] = { 3, 2, 3 };
oType.SetSplit( 0, 3, anSizes );
// Divide both axes Y and Z evenly into two parts.
oType.SetSplit( 1, 2, NULL );
oType.SetSplit( 2, 2, NULL );

// Use global-array type to create global arrays
ZGlobalArray* GA0 = zCreateGlobalArray( ''GA0'', oType );
ZGlobalArray* GA1 = zCreateGlobalArray( ''GA1'', oType );
```

Listing 7.1: An example of creating global arrays.

7.3.1 Data Structures

The basic data structures are *n*-dimensional arrays, including local arrays and global arrays. The data of a local array reside in the texture memory of the local GPU. The data of a global array are distributed to the texture memories of multiple GPUs in the cluster. A global array is partitioned into rectangular regions, called chunks, each owned by one GPU. The global array provides a global space for easy data movement among GPUs. In addition, the local chunk of a global array implicitly defines a local array. The programmer may use a Zippy library function to obtain the pointer of this local array, so that the local chunk can be accessed in the same way as other explicitly created local arrays.

Listing 7.1 shows an example of specifying a partition pattern and creating global arrays. The function of creating a global array (zCreateGlobalArray) is a collective operation, which means it is called by all processes. Each process allocates its local chunk in its GPU memory and stores a copy of the partition pattern information in its system memory. Zippy provides functions for the program to query the data locality information at run time, such as which GPUs hold a particular region of a global array, which region a GPU owns, and the translation of a region between global space and local space.

7.3.2 Coarse Level Parallelism

Zippy provides a set of shared-memory style functions to easily move data. The data can be copied within a global array, from a global array to another, from a global array to a local array, and vice versa. The programmer only needs to specify the source and destination regions in the array space and does not need to be concerned about which GPUs send or receive the data. The low-level details are encapsulated in the Zippy implementation. Moreover, because the data locality is exposed to the programmer, he/she can assess the cost of communication and use it to optimize the performance.

Data movement can be carried out by collective blocking functions, collective non-blocking functions, and one-sided functions. The collective blocking/non-blocking functions should be called by all processes in order to get correct results. A collective blocking function call will not return until the process completes its task locally. For any uninvolved process the function call returns immediately. A collective non-blocking function only reads out appropriate data from the GPU if needed, starts the network transfers, and then returns. A wait function must be called later to wait until the network transfers are finished and to write the received data to the GPU. The programmer can insert local computation before the wait function call, so that the network transfers can be overlapped with the computation. The one-sided functions only need to be called by one process. In other processes, the responses are automatically provided by Zippy, and the implementation is transparent to the programmer.

Four collective functions are provided: copy, composite, ghost cell update, and chunk re-arrangement. The copy function moves a region of data from one global array to another. The source and destination arrays can be the same or different, and the regions can be disjoint or overlapping. Listing 7.2 shows an example and the procedure is illustrated in Figure 7.1. The composite function is similar to copy, except that it has an extra input parameter which is a pointer of a predefined or user-written computation kernel specifying how the source data are combined with the destination data.

The ghost cell support is for computations on regular grids that sample the neighboring grid points, such as physical simulations, and volume and image processing. When defining a global array type, the thickness of the ghost cell layer in

```
// 3D Region: e.g., [0, 4) x [0, 2) x [0, 3)
 ZRegion oRegion0( oDimGA, 0, 4,   0, 2,   0, 3 );
 ZRegion oRegion1(oDimGA, 4, 8,   2, 4,   0, 3 );
 ZRegion oRegion2( oDimGA, 2,6, 4, 6,   2, 5 );
 zCopy( GA0, oRegion0, GA1, oRegion1 );
 zCopy( GA0, oRegion0, GA0, oRegion2 );
```

Listing 7.2: An example of copy operations.

each dimension and either direction can be specified. With a layer of ghost cells, each stored chunk is slightly larger than the actual portion of data assigned to the process. In the ghost cell update function, each process reads out its boundary data from the GPU, sends the data to the neighbors, receives data from the neighbors, and updates the ghost cells with the received data. The chunk re-arrangement function actually involves no data movement. Instead, the program simply modifies the mapping between process IDs and chunk IDs, so that virtually all the chunks managed by different GPUs are re-arranged in the global array space.

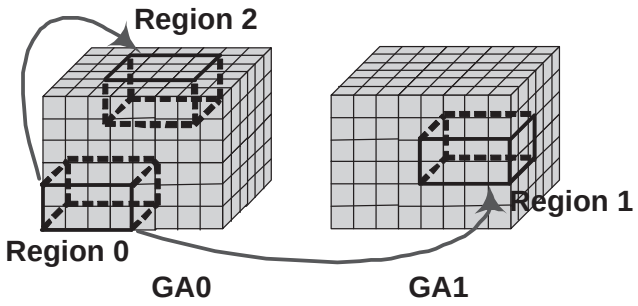


Figure 7.1: Data movement of the copy operations.

Two one-sided functions are provided. The get function moves a region of data from a global array to a local array and the put function moves data reversely. One-sided functions are called by one process. Compared with the collective functions, this kind of remote memory access is more convenient when the communication pattern is determined at run time. Zippy synchronization primitives include barrier and fence functions. Barrier synchronizes all the processes. Fence is used to execute and complete one-sided operations.

The data consistency model of Zippy is easy to understand. If a program only

```
ZKernel* pKernel = zFindOrLoadCgFP( ''Add.cg'' );
pKernel->Enable();
pKernel->SetParameter1f( ''AdditionalNum'', 0.5 );
zSetSourceArray( 0, pArray1 ); // the first source array
zSetSourceArray( 1, pArray2); // the second source array
zSetSourceRegion( 0, pRegion1 ); // the first source region
zComputeTo( pRegion0, pArray0 ); // Compute and write results to destination
pKernel->Disable();
```

Listing 7.3: An example of executing a kernel on local arrays.

uses local computation and collective data movement functions, the program behavior is deterministic and no additional synchronization is needed. To maintain data consistency, one just needs to understand what is locally completed by each function call. The collective blocking function completes the whole task locally. The collective non-blocking function completes the read of data from the GPU. The wait function completes the remaining tasks. All these operations are ordered in the program. Because Zippy uses a collective fence function call to execute and complete the queued one-sided operations, the order between any one-sided operation and other operations is also deterministic. The only indeterminable order is among multiple one-sided operations issued by different processes in the same fence. They can be executed in arbitrary order. If they conflict with each other, for example, put to overlapping regions, the programmer should use multiple fence function calls to separate them.

7.3.3 Fine Level Parallelism

On the GPU, the kernels operate on local arrays in SIMD fashion. The programmer specifies the computation kernel, source arrays, source regions, destination arrays, and a destination region. Listing 7.3 shows an example of adding the data elements in *Region1* of *Array1* and *Region1* of *Array2* plus a value of 0.5 and writing the results to *Region0* of *Array0*.

The programmer writes kernels with existing toolkits (e.g., Cg, GLSL). In Listing 7.3, because the mapping between the source regions and the destination region is simple, Zippy computes the texture coordinates that can be directly used

in the kernel to access the source data. In more complicated cases, the kernel may access arbitrary source data elements with the n-dimensional coordinates in the local array. Zippy provides address translation functions. The programmer needs to pass the packing information of the source local arrays to the kernel, and in the kernel, call the address translation to compute the texture coordinates.

7.3.4 Debugging Tool

Debugging has been difficult in GPU cluster programming. Zippy provides a debugging tool that manages all arrays and allows the programmer to select any array to view the data in a region. The debugging tool dynamically updates the data display at run time. It also allows to read the n-dimensional coordinates as well as the digital value of the mouse-picked data element. To use this tool, a debugging barrier operation needs to be added into the program. In the program, one process ID needs to be set as the control process. When the program is launched with debugging enabled, other processes are executed in the usual way. The control process, however, creates a debugging window (Figure 7.2), allowing to interactively forward the steps of the module, select an array, and view the data. The window contains 4 panels. In the top left panel, array data are displayed as a series of slices. When the programmer moves mouse cursor here, the corresponding coordinates and value are displayed in the bottom left panel. The programmer can click on the top right panel and use keyboard to forward the computation steps. The bottom right panel lists all arrays, from which the programmer can select one to view. For a local array, the data located in the GPU can be directly displayed. For a global array, data in the region of interest are brought to local with a get function called by the debugging barrier operation and then displayed. The example in Figure 7.2 shows the debugging of Marching Cubes isosurface extraction of a chair data.

7.4 Implementation

Zippy is implemented as a C++ library based on OpenGL, Cg and GLSL, and the Message Passing Interface (MPI). To run the program, the programmer needs to copy his/her executable program and related data files to all cluster nodes. A

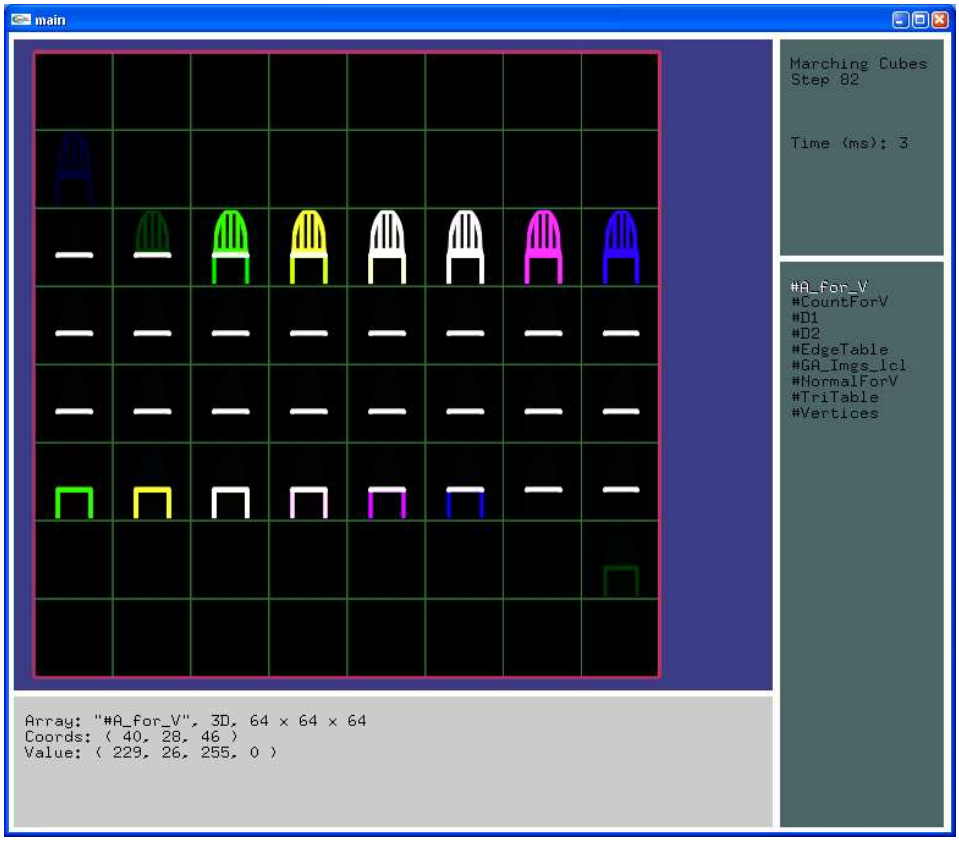


Figure 7.2: The debugging window of Zippy.

simple script file can fulfill this task. Then multiple processes that execute the same program are launched in the cluster through an MPI launcher command. The process ID is simply set to be the MPI rank.

7.4.1 Data Storage

Currently, Zippy supports 1D-7D arrays. Data of a local array are usually stored in a 2D texture, unless the programmer indicates to use a 3D texture. We apply a method similar to existing GPGPU toolkits [9, 83] to map an n-dimensional local array to a 2D texture. For any $n > 2$, assuming that we know how to map the $(n-1)$-dimensional array to a 2D patch, the n-dimensional array can be viewed as m arrays and mapped as $m1 \times m2$ patches, where m is the resolution in the n-th dimension, $m1 = ceil(sqrt(m))$ and $m2 = ceil(m/m1)$. By default, the i-th chunk

of a global array is stored in the texture memory of the GPU controlled by the i-th process. The local chunk defines a local array, hence is stored in the same way as an explicitly created local array. Every process stores in its system memory a copy of the partition pattern information of the global array, and this information is managed by Zippy.

7.4.2 Data Movement

Because every process has the data locality information of the global arrays, when calling a collective blocking function, every process has a global view of how data will be transferred in the cluster. Each process executes the following tasks. If either the source or destination region overlaps the region of the local chunk, the source and destination regions are decomposed into sub-regions based on data locality. Each pair of source and destination sub-regions represents a point-to-point data transfer in the cluster. If both sub-regions in a pair reside in the same GPU, the involved process copies the source data to a temporary Pixel Buffer Object managed by Zippy, so that the data can be later copied to the destination without going off the GPU.

The other pairs represent network data transfers. For each of these pairs, the source process binds the texture to a Framebuffer Object (FBO) and uses *glReadPixels* to read out the data from the GPU. All these pairs are sorted as follows. A graph $G = (V, E)$ is defined where V is the set of process IDs and $(i, j) \in E$ if and only if there is a data transfer between processes i and j. We find a maximum matching [44] M in G. All edges in M are removed from G and their corresponding pairs are appended to a list L. This procedure is repeated until E is empty. Each process selects from L only the pairs involving itself and executes the network transfers. The senders call the *MPI_Send* function and the receivers call the *MPI_Recv* function. Because all processes sort the pairs in L in the same way, deadlock will not happen. The senders and receivers reply on *MPI_Send* and *MPI_Recv* to be pairwise synchronized and there is no need for global synchronization. The above pair sorting also tends to allow maximal number of concurrent point-to-point data transfers. After receiving the data from the network, each receiver binds the appropriate

texture to an FBO and uses *glDrawPixels* to write the data. For the composite function, however, the destination and source data are copied to temporary local arrays and combined into the destination.

The implementation of a collective non-blocking operation is similar. The function call reads out data from the GPU if necessary, starts network transfers with the non-blocking *MPI_Isend* and *MPI_Irecv*, and returns. The wait function uses *MPI_Wait* to wait for the network transfers to finish, and writes the received data to the GPU, if necessary.

For a one-sided function, only the process requesting the data access needs to explicitly call the function. In other processes, the service provided by Zippy has to take control somewhere. For simplicity, we implemented a method similar to Thakur et al.'s [140] fence implementation for the MPICH2. When a get/put function is called, the program does nothing but locally queuing up the command of this operation. The queued operations are executed when a collective fence function is called. In the fence function, all processes send queued requests to others and execute services for others. With the optimization proposed by Thakur et al., barrier synchronization can be avoided.

7.4.3 Local Computation

The local computation interface has two layers: the upper provides the high-level abstraction while the lower encapsulates the GPU programming. Although in the latter only Cg and GLSL are supported now, incorporating other toolkits is feasible. In a computation, the shader program of the kernel is bound to fragment processors and textures of source arrays are bound to texture units. The textures of the destination arrays are attached to an FBO. The n-dimensional destination region is decomposed into a series of 2D slices, each representing a rectangle region to be rendered.

As mentioned in Section 7.3.3, two methods are available for a kernel to access the source data elements. In the first method, Zippy automatically translates the local array address to the texture coordinates. For each slice in the destination region, the positions of the corresponding slices of the source regions are computed on the CPU. With hardware interpolation, the texture coordinates of the source data

elements are computed and can be directly used in the kernel. This is convenient for computation with fixed data access patterns. In the second method, a set of optimized Cg and GLSL functions are provided for translating arbitrary n-dimensional coordinates to texture coordinates in the kernel. The packing information of the source arrays is a parameter of these functions. The programmer needs to call a Zippy function to pass this parameter to the kernel. This method is desired when arbitrary data access is needed, as in raycasting.

7.5 Example Applications

Three example applications have been implemented with Zippy and tested on a Gigabit Ethernet cluster. Each cluster node has an NVIDIA Quadro 4500 GPU, dual Intel Xeon 3.6GHz CPUs, and a PCI-Express bus.

7.5.1 Sort-Last Volume Rendering

For the sort-last volume rendering, two global arrays are defined. One is a 3D global array, representing the volume data. The other is a 2D global array that treats all partial images as a whole large image. The rendering of a local volume chunk into a partial image is a local computation of 3D texture based volume rendering. Then, the partial images are composited for the final image with the binary-swap algorithm [95]. At each step, partial images are transferred between pairs of GPUs. Using Zippy, the image compositing algorithm can be easily specified in the global space. First, based on the order of the distances between subvolumes and the view point, all partial images are virtually re-arranged using the chunk re-arrangement function. Then, the binary-swap is implemented with the composite and copy functions. The programmer only specifies the source and destination regions without concerning about how the GPU, system memory, and network are involved.

Figure 7.3 shows snapshots of the rendering of a $1875 \times 512 \times 512$ visible human CT data. With 8 GPUs, this data was rendered at 11 FPS. The user can interactively change the transfer function and explore the volume.

In our scalability experiment, each GPU renders a 512^3 subvolume, the image size is 1024^2, and the sampling step size is 1. The performance is reported in

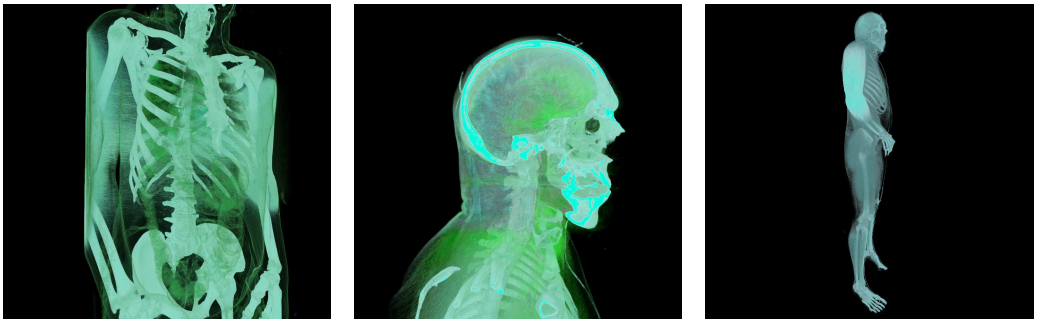


Figure 7.3: Snapshots from the sort-last volume rendering of the $1875 \times 512 \times 512$ visible human CT data.

Figure 7.4. With 16 GPUs, a 2GB volume can be rendered at 8 FPS, which gives a 16 GVox/sec overall performance. In Figure 7.5, the performance is plotted as a function of the number of GPUs . In another experiment, we disabled the local rendering and only tested the performance of the parallel image compositing, which is mainly determined by the communication cost. The image compositing achieves 16 FPS. Our performance is comparable with the fastest previous implementations. Houston [64] has implemented a system using Chromium. His implementation also exploited the GPUs for image compositing. On a 16-node Gigabit Ethernet GPU cluster, the overall performance was 8 GVox/sec and the image compositing was executed at 17 FPS.

7.5.2 Marching Cubes

Marching Cubes (MC) used to be difficult to implement on GPUs because each cube can generate 0 to 5 triangles and previous GPU shading programs did not support variable size outputs. Researchers [72, 73] have used the alternative method, Marching Tetrahedra, for GPU-based isosurface extraction. Now, the new DirectX 10 compatible GPUs support variable size outputs in the geometry shader. With this capability, Crassin [19] has implemented Marching Cubes on an NVIDIA GeForce 8800 GTS.

We have used another method to implement MC, which does not require the DirectX 10 compatible GPUs. The idea is to simulate variable size outputs using the GPU-based prefix-sum algorithms [61, 128]. Horn [61] has implemented the first

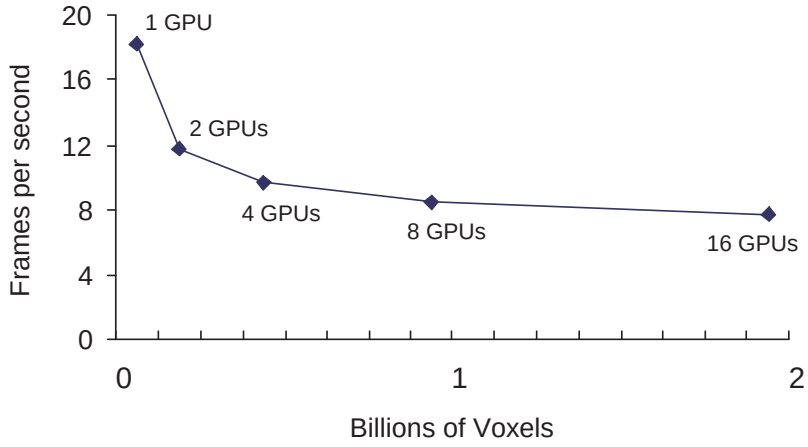


Figure 7.4: The performance of the sort-last volume rendering on our GPU cluster. (Each GPU renders a 512^3 subvolume. Note that the problem size scales up when more GPUs are used.)

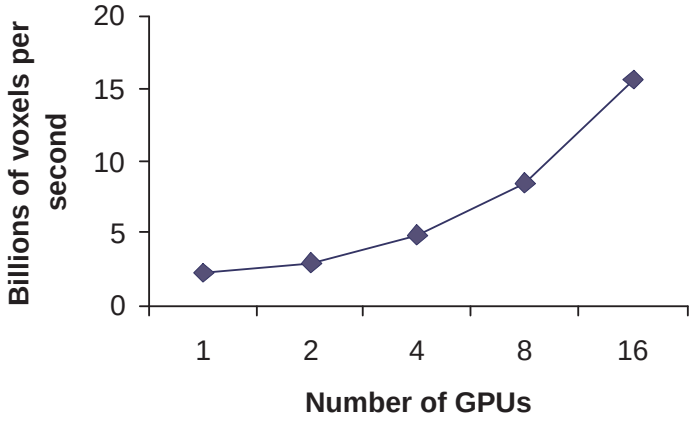


Figure 7.5: The performance (measured in billions of voxels rendered per second) is plotted as a function of the number of GPUs.

prefix-sum algorithm on the GPU. Based on it, he has efficiently simulated stream compaction, in which each input generates 0 or 1 output. Sengupta et al. [127, 128] have proposed a faster work-efficient GPU implementation of prefix-sum. Adopting Sengupta et al.'s prefix-sum method and modifying Horn's simulation of variable size outputs, we efficiently simulate stream amplification for MC, in which each

input generates a variable number of outputs.

Our idea is to directly generate the outputs in a densely packed array. The procedure is illustrated in Figure 7.6. Given an input array $\{I_i\}$ of size n, we compute the numbers of outputs and store them in array $\{M_i\}$. Then, the prefix-sum of $\{M_i\}$ is computed to array $\{P_i\}$. By the definition of prefix-sum, $P_i = \sum_{0 \le l < i} M_l$. The array $\{P_i\}$ gives two pieces of information: (1) the total number of outputs, denoted as N, equals $(P_{n-1} + M_{n-1})$; and (2) the outputs of I_i will start at position P_i and end at position $(P_i + M_i - 1)$ in the densely packed output array. By enabling a fragment program and drawing lines, array $\{(i,j)_k\}$ is computed. Each element of this array indicates that the corresponding output is the $(j+1)th$ output of I_i. With this information, the final outputs can be computed. The last two arrays in Figure 7.6 are virtually of size N. Their physical size, L, is no less than N. L is initially estimated and dynamically updated as needed. The update of L causes the reallocation of storage, but simple rules can make this reallocation happen infrequently. For example, if $N > L$, $L = 2 \times N$; if $N < L/10$, $L = max(N, 4096)$.

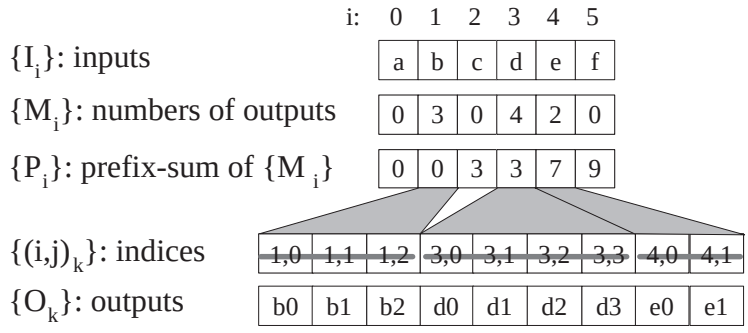
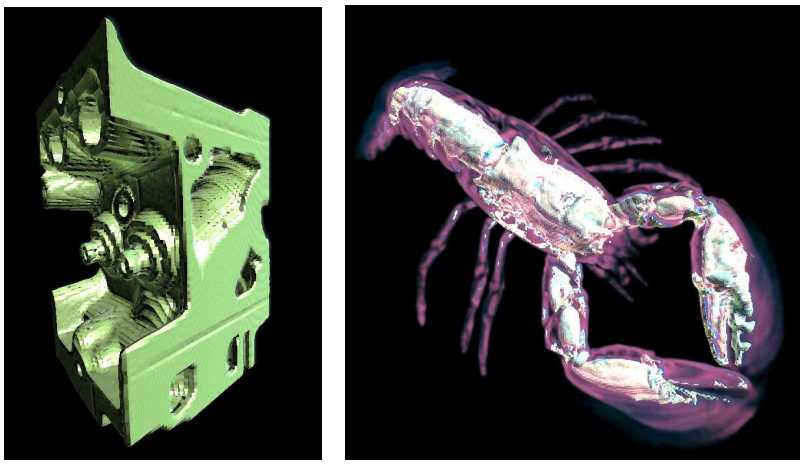


Figure 7.6: Directly generating outputs in a densely packed form for stream amplification.

With Zippy, the MC algorithm is easily translated to the kernels that operate on the local arrays. We have implemented a stream amplification function that encapsulates the above computation of array $\{(i,j)_k\}$ from array $\{M_i\}$. The local MC computation hence is fulfilled in three steps: (1) compute $\{M_i\}$, the number of triangles generated in each voxel; (2) call the stream amplification function to obtain array $\{(i,j)_k\}$; and (3) compute the output array of triangles based on array $\{(i,j)_k\}$. The output array is then copied to a Vertex Buffer Object and directly rendered with OpenGL. The volume is also rendered together with the isosurface.

We have tested our program on a separate machine that has an NVIDIA GeForce 8800 GTS and have compared the performance with Crassin's. For a 64^3 volume, Crassin's optimized implementation achieved 44 to 77 FPS and ours achieved 92 to 240 FPS for various isovalues.

To parallelize the MC is simple. A global array is defined for the whole volume. Each process independently computes the local isosurface in local arrays and renders the local data. The parallel image compositing module in Section 7.5.1 is reused here to get the final image. The system allows the user to change the isovalue and view the result in real-time. Figure 7.7 shows snapshots of a 128^3 engine CT data and a $256 \times 254 \times 57$ lobster CT data. The original image size is 800^2. The performance on our GPU cluster is reported in Figure 7.8. For the pure MC computation, there is no communication cost. However, the frame rate is not proportional to the number of GPUs because subproblems become smaller as more GPUs are used and our MC implementation is less efficient for smaller volumes.



(a) (b)

Figure 7.7: The Marching Cubes isosurfaces of (a) the engine dataset, and (b) the lobster dataset.

7.5.3 LBM Flow Simulation and Visualization

Using Zippy, we have implemented a new LBM program. The entire simulation is defined in the global space. Each GPU operates on its local chunk.

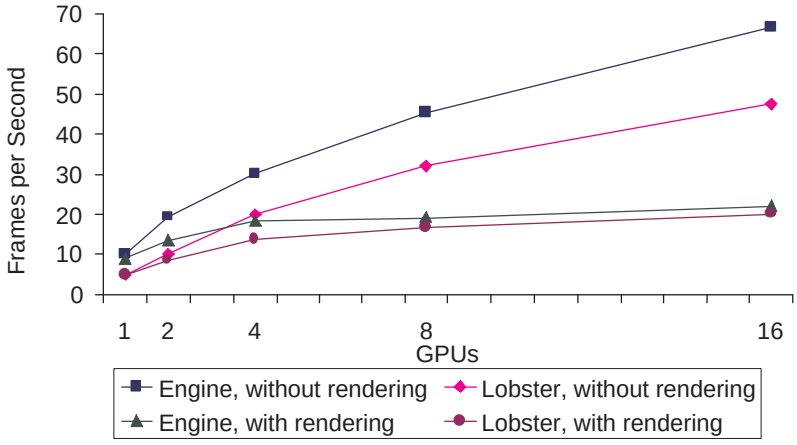


Figure 7.8: The performance of the Marching Cubes isosurface extraction on our GPU cluster for the 128^3 engine data and the $256 \times 254 \times 57$ lobster data. (The image size is 800^2. Isovalue 106 was used in all tests.)

With Zippy ghost cell support, a layer of ghost cells are defined around each local chunk. The communication is simply accomplished by non-blocking ghost cell update function calls. Based on a single-GPU version, which contains 750 lines of C++ code and 550 lines of Cg code, only 100 lines of C++ code are added to define the global arrays and update the ghost cells for the GPU cluster implementation. The dynamic simulation results on the GPUs can be directly visualized without going off the GPUs. Since the simulation and visualization modules are both developed using Zippy, the simulation module allows the rendering module to directly access its data. Figure 7.9 shows a snapshot of a simulation of grid size $400 \times 200 \times 200$. For an image size of 800^2, the overall speed is about 4.5 FPS on 16 GPUs. Compared with our previous MPI-based implementation on the same GPU cluster, the new implementation achieves better performance as shown in Figure 7.10.

In Figure 7.11, we divide the execution time of each example application into three parts: overhead time spent on barriers and Zippy logic, time waiting for data transfers through PCI-Express bus and the network, and time spent on local GPU computation. In the sort-last volume rendering and the LBM computation, the percentage of time spent on overhead operations is small. Both the percentages of time spent on overhead operations and time waiting for remote data increase slowly as

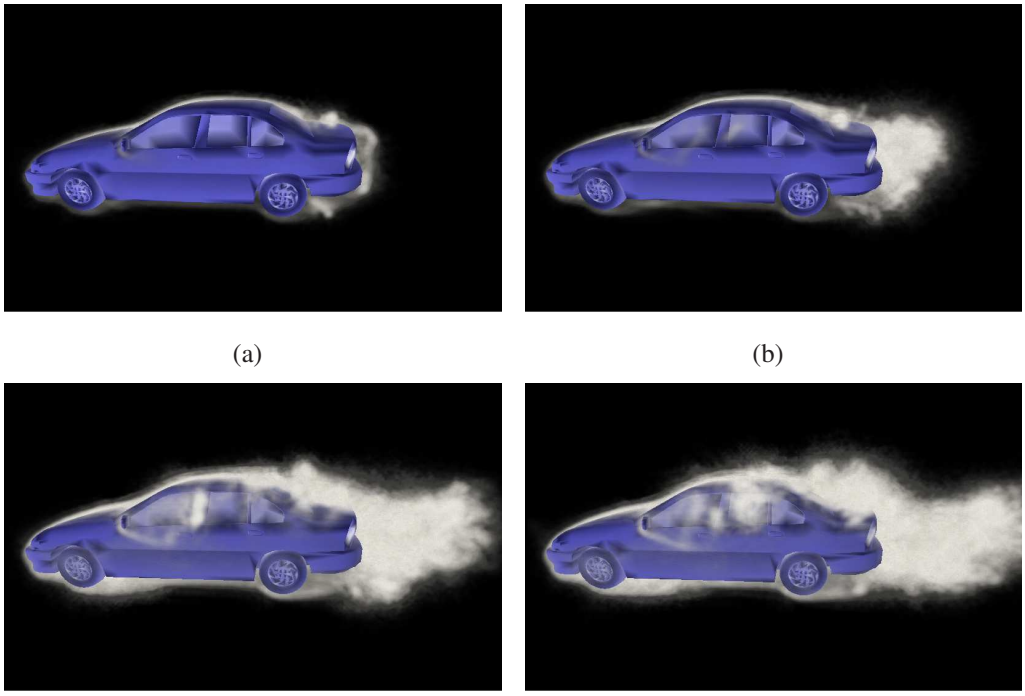


Figure 7.9: A sequence of snapshots of the LBM flow simulation. The vorticity magnitude is volume rendered to show the turbulence.

the number of GPUs increases. This shows good scalability as the computational power of GPUs can still be efficiently used when the number of GPUs increases. In both applications, each GPU solves a relatively large subproblem and the total problem size is proportional to the number of GPUs. In the MC isosurface extraction and rendering (benchmarked with lobster data), however, we have fixed the total problem size. Accordingly, the percentages of time spent on overhead operations and time waiting for remote data increase faster. In the GPU cluster, the time waiting for remote data transfers is the main bottleneck to the performance. Using a higher bandwidth network, such as Infiniband, will improve the performance. Our best implementation is the LBM computation. It shows the lowest percentage of waiting time, because we have used the non-blocking ghost cell update function to partially overlap the network communication with the computation. For other applications, this overlap is feasible for future implementation.

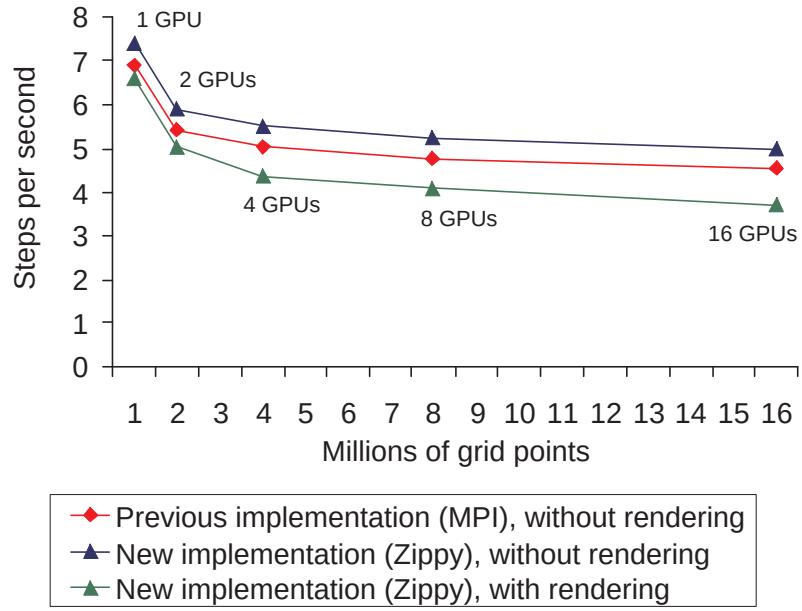


Figure 7.10: Performance comparison between our previous and new implementations on the same GPU cluster. (Each GPU manages an 100^3 sub-grid and the problem size scales up when the number of GPUs increases.)

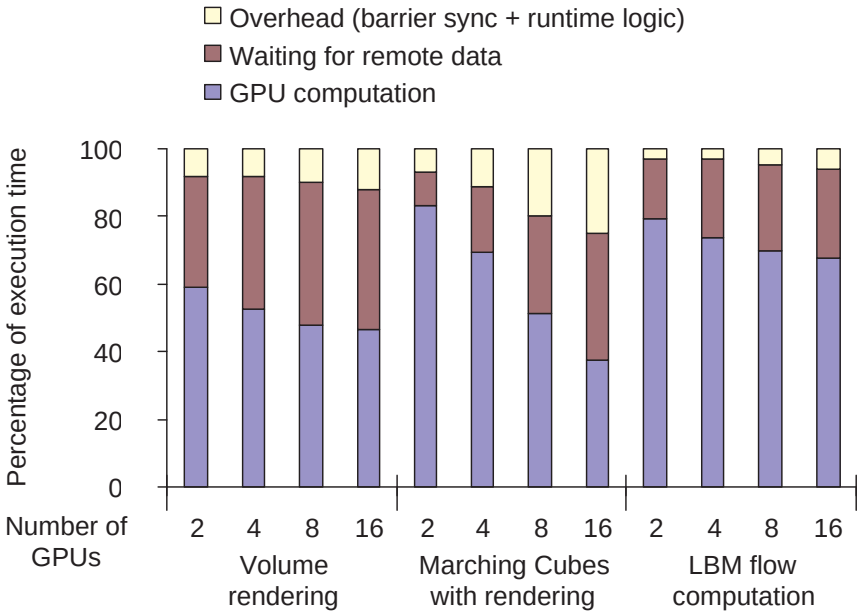


Figure 7.11: Percentages of time spent on different tasks.

Chapter 8

Conclusions

8.1 Summary

This dissertation has presented efficient ways to use GPUs and GPU clusters for the LBM flow simulation and visualization. The LBM is a discrete simulation method based on mescoscopic kinetic dynamics of particle distribution functions. A set of operations are applied to the lattice sites. These operations are local and linear, which makes the LBM friendly to the GPU and GPU cluster computation.

We have presented an optimized LBM implementation with complex boundary conditions on a single GPU and its applications in real-time amorphous phenomena modeling (Chapter 3). We have introduced a novel adapted unstructured LBM algorithm on the GPU that can effectively model fluid dynamics on 3D curved surfaces of arbitrary topology (Chapter 4). We have further described a method of implementing the LBM simulation on a GPU cluster (Chapter 5). This work has been further extended to an irregular-shaped simulation domain, which represents the reactor vessel of a nuclear power plant (Chapter 6). Finally, we have introduced Zippy, a general framework for programming parallel computation, visualization, and graphics modules on GPU clusters (Chapter 7).

Our methods have been applied to a wide range of interactive or real-time visual simulations, such as fire, smoke, wind, heat shimmering, airborne dispersion in a complex urban environment, thermal fluid dynamics in a pressurized water reactor of a nuclear power plant, and imaginary flows on 3D surfaces. The combination

of the high computational power of the GPU and GPU cluster and the linear nature of the LBM has created a powerful tool for visual simulation applications.

GPU clusters have several limitations. Because GPUs are less flexible than CPUs, not all applications can be implemented on or accelerated with GPU clusters. Also, most current GPUs do not support double precision floating point. Some scientific applications, however, require this high precision. For example, some CFD applications that use adaptive mesh refinement (AMR) to resolve the boundary layer usually need double precision to handle the fine mesh spacing. Fortunately, the LBM accurately captures the complex boundaries with the lattice links, hence the double precision is not crucial to the LBM. Density and heat generation are the other issues of GPUs, but we have seen improvements in recent years. A good example is the NVIDIA Tesla S870 1U rack-mount server. Nevertheless, because of the high performance/cost ratio and the fast performance growth of the GPU, we may see more and more general-purpose computation applications to be implemented on GPU clusters and to drive improvements of the GPU cluster hardware and software.

8.2 Future Work

8.2.1 Short-Term Future Work

We would like to explore three directions in the short-term future. The first direction is to increase the scalability of our GPU cluster implementations. A GPU cluster is a bandwidth-starve architecture. Our experiments have shown that network communication is the major bottleneck. We plan to use higher bandwidth and lower latency network, such as Infiniband, in our GPU cluster. With NVIDIA Tesla technology, multiple GPUs can be put in each node. The communication costs within nodes are lower than network communication costs. Therefore, designing and programming the GPU cluster as a three-level parallelism architecture will further increase the scalability in terms of the number of GPUs. Furthermore, we plan to implement GPU-based compression/decompression algorithms to reduce the data transfer. As computation is inexpensive on the GPUs, we can trade computation for communication performance by using compression/decompression

algorithms.

We also want to address a limitation of the LBM: it is unstable when modeling highly turbulent flows. A possible solution is to employ an implicit scheme LBM [12]. The implicit scheme LBM may be unconditionally stable and allow us to use a large time step. This method will increase the simulation speed and benefit the real-time applications. Another solution is to use the very large eddy simulation (VLES) method in the LBM. Exa Cooperation has used this method for its aerodynamics simulation [159]. Compared with the implicit scheme LBM, the VLES method may provide better accuracy.

We plan to extend Zippy with global irregular data structures so that more applications will be supported. For example, each GPU owns a portion of a global irregular data structure and uses well-defined coarse-grained communication functions to bring data from the global data structure to the local GPU memory. Also, as NVIDIA CUDA has become a popular programming toolkit on the GPU, we plan to build the local GPU interface of Zippy upon CUDA. As described in Chapter 7, our design of the local GPU interface is flexible; thus building Zippy upon CUDA is feasible. By doing so, Zippy will allow local computation to access a linear memory space that is much larger and much more flexible than the OpenGL textures. The programmer will write kernels more easily in the C language than in shading languages. The new GPU features, such as scatter and Parallel Data Cache, will be available to the programmer for performance optimization. Moreover, global irregular data structures would be much simpler to implement.

8.2.2 Long-Term Future Work

We plan to study other applications on the GPU cluster. Two interesting applications are the linear algebra computation and the real-time ray-tracing. The former is essential to scientific computation and the latter is one of the most important problems in computer graphics. Both are challenging because their data access patterns are more irregular than the LBM.

Hybrid clusters will be a trend for future high performance computing. For example, each node will have multiple multi-core CPUs and multiple GPUs. The GPUs and CPUs could work together, each executing the part of computation task

that it does best. Currently, our GPU cluster implementations use the CPUs only for communication and GPU management. The CPU computational power has not been fully exploited. A possible solution is to combine the vertical communication of Sequoia [37] and the horizontal communication of Zippy, which would help the programmer to exploit multiple memory levels on the cluster, including GPU memories, system memories, and disks. Specifically, the global arrays can be defined at any level so that the computational power of both CPUs and GPUs can be exploited.

Bibliography

[1] AMD FireStream 9170.
http://ati.amd.com/technology/streamcomputing/product_firestream_9170.html.

[2] NVIDIA Tesla S1070 computing system.
http://www.nvidia.com/object/tesla_s1070.html.

[3] Rapidmind multi-core development platform.
http://www.rapidmind.net/.

[4] Website of general purpose computation on GPUs.
http://www.gpgpu.org.

[5] B. Adams, M. Pauly, R. Keiser, and L. J. Guibas. Adaptively sampled particle fluids. *ACM Transactions on Graphics*, 26(3):48, 2007.

[6] S. Bachthaler, M. Strengert, D. Weiskopf, and T. Ertl. Parallel texture-based vector field visualization on curved surfaces using GPU cluster computers. *Eurographics Symposium on Parallel Graphics and Visualization*, pages 75–82, 2006.

[7] P. Bhaniramka, P. Robert, and S. Eilemann. OpenGL multipipe SDK: A toolkit for scalable parallel rendering. *IEEE Visualization*, pages 119–126, 2005.

[8] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22(3):917–924, 2003.

[9] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transcations on Graphics*, 23(3):777–786, 2004.

[10] J. M. Buick and C. A. Greated. Gravity in a lattice Boltzmann model. *Physical Review E*, 61(5):5307–5320, 2000.

[11] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *Symposium on Volume Visualization*, pages 91–98, 1994.

[12] N. Cao, S. Chen, S. Jin, and D. Martínez. Physical symmetry and lattice symmetry in the lattice Boltzmann method. *Physical Review E*, 55(1):R21–R24, 1997.

[13] M. Carlson, P. J. Mucha, and G. Turk. Rigid fluid: Animating the interplay between rigid bodies and fluid. *ACM Transactions on Graphics*, 23(3):377–384, 2004.

[14] N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. *ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 37–46, 2002.

[15] N. A. Carr, J. Hoberock, K. Crane, and J. C. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. *Graphics Interface*, pages 203–209, 2006.

[16] N. S.-H. Chu and C. Tai. Moxi: Real-time ink dispersion in absorbent paper. *ACM Transactions on Graphics*, 24(3):504–511, 2005.

[17] P. W. Cleary, S. H. Pyo, M. Prakash, and B. K. Koo. Bubbling and frothing liquids. *ACM Transactions on Graphics*, 26(3):97, 2007.

[18] K. Crane, I. Llamas, and S. Tariq. Real-time simulation and rendering of 3D fluids. In H. Nguyen, editor, *GPU Gems 3*, pages 633–676. Addison-Wesley, 2007.

[19] C. Crassin. OpenGL geometry shader marching cubes. *http://www.icare3d.org/content/view/50/9*, 2007.

[20] D. D'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.-S. Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Royal Society of London Philosophical Transactions Series A*, 360(1792):437–451, 2002.

[21] C. Dyken, G. Ziegler, C. Theobalt, and H.-P. Seidel. Histopyramids in isosurface extraction. Technical report, Max Planck Inst. für Infor., 2007.

[22] K. Engel, M. Hadwiger, J. M. Kniss, C. Rezk-salama, and D. Weiskopf. *Real-time Volume Graphics*. A K Peters, 2006.

[23] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 9–16, 2001.

[24] D. Enright, S. Marschner, and R. Fedkiw. Animation and rendering of complex water surfaces. *ACM Transactions on Graphics*, 21(3):736–744, 2002.

[25] M. EricksonKirk, M. Junge, W. Arcieri, B. Bass, R. Beaton, D. Bessette, T. Chang, T. Dickson, C. Fletcher, A. Kolaczkowski, S. Malik, T. Mintz, C. Pugh, F. Simonen, N. Siu, D. Whitehead, P. Williams, R. Woods, and S. Yin. Technical basis for revision of the pressurized thermal shock (PTS) screening limit in the PTS rules (10 CFR 50.61). *NUREG-1806*, 1, 2006.

[26] C. Everitt. Interactive order-independent transparency. Technical Report, NVIDIA Corporation, 2001.

[27] Z. Fan, Y. Kuo, Y. Zhao, F. Qiu, A. Kaufman, and B. Arcieri. Visual simulation of thermal fluid dynamics in a pressurized water reactor. Submitted, 2008.

[28] Z. Fan, W. Li, X. Wei, and A. Kaufman. GPU-based voxelization and its application in flow modeling. *ACM Workshop on General-Purpose Computing on Graphics Processors*, pages C–7, 2004.

[29] Z. Fan, C. Ma, and M. Oliveira. A sketch-based collaborative design system. *Brazilian Symposium on Computer Graphics and Image Processing*, pages 125–131, 2003.

[30] Z. Fan, M. Oliveira, C. Ma, and A. Kaufman. A sketch-based interface for collaborative design. *Eurographics Workshop on Sketch-Based Interfaces and Modelling*, pages 143–150, 2004.

[31] Z. Fan, F. Qiu, and A. Kaufman. Zippy: A framework for computation and visualization on a GPU cluster. *Computer Graphics Forum*, 27(2):341–350, 2008.

[32] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. *ACM/IEEE Supercomputing Conference*, page 47, 2004.

[33] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for scientific computing and large-scale simulation. *ACM Workshop on General-Purpose Computing on Graphics Processors*, pages C–32, 2004.

[34] Z. Fan, C. Vetter, C. Guetter, D. Yu, R. Westermann, A. Kaufman, and C. Xu. Optimized GPU implementation of learning-based non-rigid multi-modal registration. *SPIE Medical Imaging*, number 6914-107, 2008.

[35] Z. Fan, Y. Zhao, A. Kaufman, and Y. He. Adapted unstructured LBM for flow simulation on curved surfaces. *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 245–254, 2005.

[36] S. Fang and H. Chen. Hardware accelerated voxelization. *Computers & Graphics*, 24(3):433–442, 2000.

[37] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. *ACM/IEEE Supercomputing Conference*, page 4, 2006.

[38] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. *ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 133–137, 2004.

[39] R. Fedkiw, J. Stam, and H. Jensen. Visual simulation of smoke. *ACM SIGGRAPH*, pages 15–22, 2001.

[40] B. E. Feldman, J. F. O'Brien, and B. M. Klingner. Animating gases with hybrid meshes. *ACM Transactions on Graphics*, 24(3):904–909, 2005.

[41] N. Foster and R. Fedkiw. Practical animation of liquids. *ACM SIGGRAPH*, pages 23–30, 2001.

[42] N. Foster and D. Metaxas. Realistic animation of liquids. *Graphical Models and Image Processing*, 58(5):471–483, 1996.

[43] N. Foster and D. Metaxas. Modeling the motion of hot, turbulent gas. *ACM SIGGRAPH*, pages 181–188, 1997.

[44] Z. Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Computing Surveys*, 18(1):23–38, 1986.

[45] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. Buijssen, M. Grajewski, and S. Turek. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. *Parallel Computing*, 33(10–11):685–699, 2007.

[46] D. Göddeke, R. Strzodka, and S. Turek. Accelerating double precision FEM simulations with GPUs. *Symposium of Simulation Technique*, Frontiers in Simulation:139–144, 2005.

[47] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 102–111, 2003.

[48] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. *ACM International Conference on Management of Data*, pages 215–226, 2004.

[49] N. K. Govindaraju, S. Redon, M. C. Lin, and D. Manocha. Cullide: Interactive collision detection between complex models in large environments using graphics hardware. *ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 25–32, 2003.

[50] N. K. Govindaraju, A. Sud, S.-E. Yoon, and D. Manocha. Interactive visibility culling in complex environments using occlusion-switches. *ACM Symposium on Interactive 3D Graphics*, pages 103–112, 2003.

[51] X. Gu, S. J. Gortler, and H. Hoppe. Geometry images. *ACM Transactions on Graphics*, 21(3):355–361, 2002.

[52] E. Guendelman, A. Selle, F. Losasso, and R. Fedkiw. Coupling water and smoke to thin deformable and rigid shells. *ACM Transactions on Graphics*, 24(3):973–981, 2005.

[53] A. Gunstensen, D. Rothman, S. Zaleski, and G. Zanetti. Lattice Boltzmann model of immiscible fluids. *Physical Review A*, 43(8):4320–4327, 1991.

[54] T. Harada, S. Koshizuka, and Y. Kawaguchi. Smoothed particle hydrodynamics on GPUs. *Computer Graphics International*, pages 63–70, 2007.

[55] M. Harris, W. V. Baxter, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 92–101, 2003.

[56] M. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 109–118, 2002.

[57] W. Heidrich, R. Westermann, H. P. Seidel, and T. Ertl. Application of pixel textures in visualization and realistic image synthesis. *ACM Symposium on Interactive 3D Graphics*, pages 127–134, 1999.

[58] K. E. Hoff, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized voronoi diagrams using graphics hardware. *ACM SIGGRAPH*, pages 277–286, 1999.

[59] J. Hong and C. Kim. Discontinuous fluids. *ACM Transactions on Graphics*, 24(3):915–920, 2005.

[60] W. Hong, F. Qiu, S. Lakare, and A. Kaufman. Hybrid volumetric ray-casting. *ACM Workshop on General-Purpose Computing on Graphics Processors*, pages C–45, 2004.

[61] D. Horn. Stream reduction operations for GPGPU applications. In M. Pharr, editor, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 573–589. Addison Wesley, 2005.

[62] D. Horn, M. Houston, and P. Hanrahan. ClawHMMER: A streaming HMMer-search implementation. *ACM/IEEE Supercomputing Conference*, page 11, 2005.

[63] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-D tree GPU raytracing. *ACM Symposium on Interactive 3D Graphics and Games*, pages 167–174, 2007.

[64] M. Houston. Designing graphics clusters. Talk in Parallel Visualization Workshop of IEEE Visualization, 2004.

[65] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A scalable graphics system for clusters. *ACM SIGGRAPH*, pages 129–140, 2001.

[66] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics*, 21(3):693–702, 2002.

[67] G. Irving, E. Guendelman, F. Losasso, and R. Fedkiw. Efficient simulation of large bodies of water by coupling two and three dimensional techniques. *ACM Transactions on Graphics*, 25(3):805–811, 2006.

[68] C. Jiang and M. Snir. Automatic tuning matrix multiplication performance on graphics hardware. *International Conference on Parallel Architectures and Compilation Techniques*, pages 185–196, 2005.

[69] B. Jobard, G. Erlebacher, and M. Hussaini. Hardware-accelerated texture advection for unsteady flow visualization. *IEEE Visualization*, pages 155–162, 2000.

[70] B. Kim, Y. Liu, I. Llamas, X. Jiao, and J. Rossignac. Simulation of bubbles in foam with the volume control method. *ACM Transactions on Graphics*, 26(3):98, 2007.

[71] B. Kim, Y. Liu, I. Llamas, and J. Rossignac. FlowFixer: Using BFECC for fluid simulation. *Eurographics Workshop on Natural Phenomena*, pages 51–56, 2005.

[72] P. Kipfer and R. Westermann. GPU construction and transparent rendering of iso-surfaces. *Vision, Modeling, and Visualization*, pages 241–248, 2005.

[73] T. Klein, S. Stegmaier, and T. Ertl. Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. *Pacific Graphics*, pages 186–195, 2004.

[74] B. M. Klingner, B. E. Feldman, N. Chentanez, and J. F. O'Brien. Fluid animation with dynamic meshes. *ACM Transactions on Graphics*, 25(3):820–825, 2006.

[75] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. *ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 123–131, 2004.

[76] J. Krüger. *A GPU framework for interactive simulation and rendering of fluid effects*. PhD thesis, Technische Unversität München, 2006.

[77] J. Krüger, P. Kipfer, P. Kondratieva, and R. Westermann. A particle system for interactive visualization of 3D flows. *IEEE Transactions on Visualization and Computer Graphics*, 11(6):744–756, 2005.

[78] J. Krüger and R. Westermann. Acceleration techniques for GPU-based volume rendering. *IEEE Visualization*, pages 287–292, 2003.

[79] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, 2003.

[80] J. Krüger and R. Westermann. GPU simulation and rendering of volumetric effects for computer games and virtual environments. *Computer Graphics Forum*, 24(3), 2005.

[81] P. Lallemand and L.-S. Luo. Theory of the lattice Boltzmann method: Acoustic and thermal properties in two and three dimensions. *Physical Review E*, 68(3):036706, 2003.

[82] H. Lee, Y. Tong, and M. Desbrun. Geodesics-based one-to-one parameterization of 3D triangle meshes. *IEEE Multimedia*, 12(1):27–33, 2005.

[83] A. Lefohn, J. M. Kniss, R. Strzodka, S. Sengupta, and J. D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25(1):60–99, 2006.

[84] A. E. Lefohn, J. M. Kniss, C. D. Hansen, and R. T. Whitaker. A streaming narrow-band algorithm: Interactive deformation and visualization of level sets. *IEEE Transactions on Visualization and Computer Graphics*, 10(40):422–433, 2004.

[85] F. Li and M. Modarres. Probabilistic modeling for fracture mechanic studies of reactor vessels with characterization of uncertainties. *Nuclear Engineering and Design*, 235(1):1–19, 2005.

[86] S. Li, Z. Fan, X. Yin, K. Mueller, A. E. Kaufman, and X. Gu. Real-time reflection using ray tracing with geometry field. *Eurographics*, pages 29–32, 2006.

[87] W. Li, Z. Fan, X. Wei, and A. Kaufman. Flow simulation with complex boundaries. In M. Pharr, editor, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 747–764. Addison-Wesley, 2005.

[88] W. Li, K. Mueller, and A. E. Kaufman. Empty space skipping and occlusion clipping for texture-based volume rendering. *IEEE Visualization*, pages 317–324, 2003.

[89] W. Li, X. Wei, and A. Kaufman. Implementing lattice Boltzmann computation on graphics hardware. *The Visual Computer*, 19(7-8):444–456, 2003.

[90] R. Löhner, C. Yang, and R. Roger. Tracking vortices over large distances using vorticity confinement. *Symposium on Naval Hydrodynamics*, pages 950–962, 2003.

[91] P. Lobner, C. Donahoe, and C. Vavallin. Overview and comparison of U.S. commercial nuclear power plants. Technical Report NUREG/CR-5640, 1990.

[92] F. Losasso, F. Gibou, and R. Fedkiw. Simulating water and smoke with an octree data structure. *ACM Transactions on Graphics*, 23(3):457–462, 2004.

[93] F. Losasso, T. Shinar, A. Selle, and R. Fedkiw. Multiple interacting liquids. *ACM Transactions on Graphics*, 25(3):812–819, 2006.

[94] C. P. Lowe and S. Succi. Go-with-the-flow lattice Boltzmann methods for tracer dynamics. In *Lecture Notes in Physics*, chapter 9, pages 267–288. Springer-Verlag, 2002.

[95] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap image composition. *IEEE Computer Graphics and Applications*, pages 59–68, 1994.

[96] M. Manzke, R. Brennan, K. O'Conor, J. Dingliana, and C. O'Sullivan. A scalable and reconfigurable shared-memory graphics architecture. *ACM SIGGRAPH Sketches*, page 182, 2006.

[97] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22(3):896–907, 2003.

[98] A. Martin and S. Bellet. CFD-tools qualification for thermal-hydraulics pressurized thermal shock analysis. *Journal of Pressure Vessel Technology*, 125(4):418–424, 2003.

[99] N. Martys, J. Hagedorn, D. Goujon, and J. Devaney. Large scale simulations of single and multi-component flow in porous media. *International Symposium on Optical Science, Engineering, and Instrumentation*, pages 205–213, 1999.

[100] F. Massaioli and G. Amati. Optimization and scaling of an OpenMP LBM code on IBM SP nodes. Scicomp Talk, 2002.

[101] F. Massaioli and G. Amati. Performance portability of a lattice Boltzmann code. Scicomp Talk, 2004.

[102] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. *ACM Transactions on Graphics*, 23(3):787–795, 2004.

[103] R. Mei, L. S. Luo, and W. Shyy. An accurate curved boundary treatment in the Lattice Boltzmann method. *Journal of Computational Physics*, 155(2):307–330, 1999.

[104] R. Mei, W. Shyy, D. Yu, and L. S. Luo. Lattice Boltzmann method for 3-D flows with curved boundary. *Journal of Computational Physics*, 161(2):680–699, 2000.

[105] A. Moerschell and J. D. Owens. Distributed texture memory in a multi-GPU environment. *ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware*, pages 31–38, 2006.

[106] J. Monaghan. An introduction to SPH. *Computer Physics Communications*, 48:88–96, 1988.

[107] K. Mueller and R. Yagel. On the use of graphics hardware to accelerate algebraic reconstruction methods. *SPIE Medical Imaging*, number 3659-62, 1999.

[108] M. Mueller, D. Charypar, and M. Gross. Particle-based fluid simulation for interactive applications. *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 154–159, 2003.

[109] M. Müller, B. Solenthaler, R. Keiser, and M. Gross. Particle-based fluid-fluid interaction. *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 237–244, 2005.

[110] N. Neophytou, K. Mueller, K. T. McDonnell, W. Hong, X. Guan, H. Qin, and A. E. Kaufman. GPU-accelerated volume splatting with elliptical RBFs. *Eurographics/IEEE TCVG Symposium on Visualization*, pages 13–20, 2006.

[111] D. Nguyen, R. Fedkiw, and H. Jensen. Physically based modeling and animation of fire. *ACM Transactions on Graphics*, 21(3):721–728, 2002.

[112] H. Nguyen. *GPU Gems 3*. Addison Wesley, 2007.

[113] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. Advances, applications and performance of the Global Arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, 2006.

[114] NVIDIA. *Compute Unified Device Architecture Programming Guide*, 2007. Version 1.1, http://www.nvidia.com/object/cuda_develop.html.

[115] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[116] S. Park and M. J. Kim. Vortex fluid for gaseous phenomena. *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 261–270, 2005.

[117] G. Peng, H. Xi, G. Duncan, and S. H. Chou. Lattice Boltzmann method on irregular meshes. *Physical Review E*, 58(4):4124–4127, 1998.

[118] M. Pharr and R. Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[119] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002.

[120] F. Qiu, F. Xu, Z. Fan, N. Neophytos, A. Kaufman, and K. Mueller. Lattice-based volumetric global illumination. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1576–1583, 2007.

[121] F. Qiu, Y. Zhao, Z. Fan, X. Wei, H. Lorenz, J. Wang, S. Yoakum-Stover, A. Kaufman, and K. Mueller. Dispersion simulation and visualization for urban security. *IEEE Visualization*, pages 553–560, 2004.

[122] W. T. Reeves. Particle systems—A technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91–108, 1983.

[123] R. J. Rost. *OpenGL Shading Language (2nd Edition)*. Addison-Wesley Professional, 2006.

[124] M. Rumpf and R. Strzodka. Nonlinear diffusion in graphics hardware. *Eurographics/IEEE TCVG Symposium on Visualization*, pages 75–84, 2001.

[125] A. Selle, R. Fedkiw, B. Kim, Y. Liu, and J. Rossignac. An unconditionally stable MacCormack method. *Journal of Scientific Computing (in press)*.

[126] A. Selle, N. Rasmussen, and R. Fedkiw. A vortex particle method for smoke, water and explosions. *ACM Transactions on Graphics*, 24(3):910–914, 2005.

[127] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. *ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware*, pages 97–106, 2007.

[128] S. Sengupta, A. E. Lefohn, and J. D. Owens. A work-efficient step-efficient prefix sum algorithm. *Workshop on Edge Computing Using New Commodity Architectures*, pages D–26–27, 2006.

[129] J. Shade, S. Gortler, L. He, and R. Szeliski. Layered depth images. *ACM SIGGRAPH*, pages 231–242, 1998.

[130] L. Shi and Y. Yu. Inviscid and incompressible fluid simulation on triangle meshes. *Journal of Computer Animation and Virtual Worlds*, 15(3-4):173–181, 2004.

[131] S. St-Laurent. *The Complete Effect and HLSL Guide*. Paradoxal Press, 2005.

[132] J. Stam. Stable fluids. *ACM SIGGRAPH*, pages 121–128, 1999.

[133] J. Stam. Flows on surfaces of arbitrary topology. *ACM Transactions On Graphics*, 22(3):724–731, 2003.

[134] D. Stora, P. O. Agliati, M. P. Cani, F. Neyret, and J. D. Gascuel. Animating lava flows. *Graphics Interface*, pages 203–210, 1999.

[135] M. Strengert, M. Magallon, D. Weiskopf, S. Guthe, and T. Ertl. Hierarchical visualization and compression of large volume datasets using GPU clusters. *Eurographics Symposium on Parallel Graphics and Visualization*, pages 41–48, 2004.

[136] S. Succi. *The lattice Boltzmann equation for fluid dynamics and beyond*. Numerical Mathematics and Scientific Computation. Oxford University Press, 2001.

[137] H. Takizawa and H. Kobayashi. Hierarchical parallel processing of large scale data clustering on a PC cluster with GPU co-processing. *Journal of Supercomputing*, 36(3):219–234, 2006.

[138] A. T. C. Tam and C.-L. Wang. Contention-aware communication schedule for high-speed communication. *Cluster Computing*, 6(4), 2003.

[139] J. Teran, S. Blemker, V. N. T. Hing, and R. P. Fedkiw. Finite volume methods for the simulation of skeletal muscle. *Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, pages 68–74, 2003.

[140] R. Thakur, W. Gropp, and B. Toonen. Optimizing the synchronization operations in message passing interface one-sided communication. *International Journal of High Performance Computing Applications*, 19(2):119–128, 2005.

[141] N. Thürey and U. Ruede. Free surface lattice-Boltzmann fluid simulations with and without level sets. *Vision, Modeling, and Visualization*, pages 199–207, 2004.

[142] C. Trendall and A. J. Stewart. General calculations using graphics hardware with applications to interactive caustics. *Eurographics Workshop on Rendering Techniques*, pages 287–298, 2000.

[143] S. Ubertini, G. Bella, and S. Succi. Lattice Boltzmann method on unstructured grids: Further developments. *Physical Review E*, 68(1):016701, 2003.

[144] S. Ubertini and S. Succi. Recent advances of lattice Boltzmann techniques on unstructured grids. *Progress in Computational Fluid Dynamics*, 5(1/2):85–96, 2005.

[145] C. van Treeck, E. Rank, M. Krafczyk, J. Tölke, and B. Nachtwey. Extension of a hybrid thermal LBE scheme for large-eddy simulations of turbulent convective flows. *Computers & Fluids*, 35(8-9):863–871, 2006.

[146] J. J. van Wijk. Image based flow visualization. *ACM Transactions on Graphics*, 21(3):745–754, 2002.

[147] J. J. van Wijk. Image based flow visualization for curved surfaces. *IEEE Visualization*, pages 123–130, 2003.

[148] X. Wei, W. Li, K. Mueller, and A. Kaufman. Simulating fire with texture splats. *IEEE Visualization*, pages 227–234, 2002.

[149] X. Wei, W. Li, K. Mueller, and A. E. Kaufman. The lattice-Boltzmann method for simulating gaseous phenomena. *IEEE Transactions on Visualization and Computer Graphics*, 10(3):164–176, 2004.

[150] X. Wei, Y. Zhao, Z. Fan, W. Li, F. Qiu, S. Yoakum-Stover, and A. Kaufman. Lattice-based flow field modeling. *IEEE Transactions on Visualization and Computer Graphics*, 10(6):719–729, 2004.

[151] X. Wei, Y. Zhao, Z. Fan, W. Li, S. Yoakum-Stover, and A. Kaufman. Blowing in the wind. *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 75–85, 2003.

[152] D. Weiskopf, M. Hope, and T. Ertl. Hardware-accelerated Lagrangian-Eulerian texture advection for 2D flow visualization. *Vision, Modeling, and Visualization*, pages 77–84, 2002.

[153] D. Weiskopf, M. Hopf, and T. Ertl. Hardware accelerated visualization of time-varying 2D and 3D vector fields by texture advection via programmable per-pixel operations. *Vision, Modeling, and Visualization*, pages 439–446, 2001.

[154] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. *ACM SIGGRAPH*, pages 169–177, 1998.

[155] D. A. Wolf-Gladrow. *Lattice Gas Cellular Automata and Lattice Boltzmann Models: An Introduction*. Springer-Verlag, 2000.

[156] H. Xiong, H. Peng, A. Qin, and J. Shi. Parallel occlusion culling on GPUs cluster. *ACM International Conference on Virtual Reality Continuum and its Applications*, pages 19–26, 2006.

[157] F. Xu and K. Mueller. Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware. *IEEE Transactions on Nuclear Science*, 52(3):654–663, 2005.

[158] F. Xu and K. Mueller. Real-time 3D computed tomography reconstruction using commodity graphics hardware. *Physics in Medicine and Biology*, 52:3405–3419, 2007.

[159] V. Yakhot, S. Orszag, S. Thangam, T. Gatski, and C. Speziale. PowerFLOW3.2: Theory and benchmark results. *Physics of Fluids A*, pages 1510–1520, 1992.

[160] S. Yamagiwa and L. Sousa. Caravela: A novel stream-based distributed computing environment. *IEEE Computer*, 40(5):70–77, 2007.

[161] C. Yuksel, D. H. House, and J. Keyser. Wave particles. *ACM Transactions on Graphics*, 26(3):148, 2007.

[162] Y. Zhao, Y. Han, Z. Fan, F. Qiu, Y. Kuo, A. E. Kaufman, and K. Mueller. Visual simulation of heat shimmering and mirage. *IEEE Transactions on Visualization and Computer Graphics*, 13(1):179–189, 2007.

[163] Y. Zhao, F. Qiu, Z. Fan, and A. Kaufman. Flow simulation with locally-refined LBM. *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 181–188, 2007.

[164] Y. Zhao, X. Wei, Z. Fan, A. Kaufman, and H. Qin. Voxels on fire. *IEEE Visualization*, pages 271–278, 2003.

[165] K. Zhou, Z. Ren, S. Lin, H. Bao, B. Guo, and H.-Y. Shum. Real-time smoke rendering using compensated ray marching. *ACM Transactions on Graphics*, 27(3), 2008.

[166] H. Zhu, X. Liu, Y. Liu, and E. Wu. Simulation of miscible binary mixtures based on lattice Boltzmann method. *Computer Animation and Virtual Worlds*, 17(3–4):403–410, 2006.