NTNU
Norwegian University of
Science and Technology

The Lattice Boltzmann Simulation on Multi-GPU Systems

Thor Kristian Valderhaug

Master of Science in Computer Science
Submission date: June 2011
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The Lattice Boltzmann Method (LBM) is a fluid simulation algorithm which is used to simulate different types of flow, such as water, oil and gas in porous reservoir rocks. Some of the biggest challenges of Lattice Boltzmann algorithms are resolution and memory requirements when simulating large, irregular geometries. Modern GPUs have a limited amount of memory, which severely restricts the domain size which can be simulated on a single GPU. Spreading the domain over multiple GPUs will thus open for larger more realistic domains to be simulated.

This project will investigate how to utilize multiple GPUs when doing LBM simulation on large datasets. The investigations will include comparing thread-only vs. threading and MPI implementations in order to evaluate suitability for LBM on a cluster with GPUs.

The following issues may be further investigated: domain decomposition, overlapping computation and communication, modeling of communication and computation, auto tuning and load balancing. The code is expected to be developed in OpenCL making the work applicable to both ATI- and NVIDIA-based GPUs systems.

Assignment given: 17. January 2011
Supervisor: Anne Cathrine Elster, IDI

Abstract

The Lattice Boltzmann Method (LBM) is widely used to simulate different types of flow, such as water, oil and gas in porous reservoirs. In the oil industry it is commonly used to estimate petrophysical properties of porous rocks, such as the permeability. To achieve the required accuracy it is necessary to use big simulation models requiring large amounts of memory. The method is highly data intensive making it suitable for offloading to the GPU. However, the limited amount of memory available on modern GPUs severely limits the size of the dataset possible to simulate.

In this thesis, we increase the size of the datasets possible to simulate using techniques to lower the memory requirement while retaining numerical precision. These techniques improve the size possible to simulate on a single GPU by about 20 times for datasets with 15% porosity. We then develop multi-GPU simulations for different hardware configurations using OpenCL and MPI to investigate how LBM scales when simulating large datasets. The performance of the implementations are measured using three porous rock datasets provided by Numerical Rocks AS. By connecting two Tesla S2070s to a single host we are able to achieve a speedup of 1.95, compared to using a single GPU. For large datasets we are able to completely hide the host to host communication in a cluster configuration, showing that LBM scales well and is suitable for simulation on a cluster with GPUs. The correctness of the implementations is confirmed against an analytically known flow, and three datasets with known permeability also provided by Numerical Rocks AS.

Acknowledgements

This report is the result of work done at the HPC laboratory at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway.

I would like to thank my supervisor Dr. Anne C. Elster for introducing me to high-performance computing and enabling me to realize this project. This project would not have been possible without the resources and equipment she have made available for the HPC laboratory. Some of the GPUs used in this work was donated to the laboratory by NVIDIA, I wish to thank NVIDIA for their suport of the laboratory.

I would also like to thank Atle Rudshaug at Numerical Rocks AS for providing me with the datasets used to evaluate the implementations. Erik Ola Aksnes for providing me with his source code and answering questions about details in his implementation. I would like to thank Ian Karlin for giving me feedback on the content and writing style of this report. I would also like to thank Jan Christian Meyer for guidance, technical support and many interesting and helpful discussions during this project.

My greatest gratitude goes to Jan Ottar Valderhaug Anne Haagensen and Eli Maritha Carlsen for having the patience to prof-read this report and correct my many spelling errors.

Finally, I would like to thank my fellow students at the HPC group for great ideas and support during those long hours at the lab.

Thor Kristian Valderhaug
Trondheim, Norway June 20, 2011

Contents

Abstract **iii**

Acknowledgements **v**

Table of Contents **ix**

List of Figures **xi**

List of Tables **xii**

List of Abbreviations **xiii**

1 Introduction **1**
1.1 Project goal 2
1.2 Outline 2

2 Parallel Computing and The Graphical Processing Unit **3**
2.1 Parallel computing 3
2.1.1 Forms of Parallelism 4
2.2 Hardware configuration 5
2.2.1 Shared Memory Multiprocessor 5
2.2.2 Cluster 7
2.3 GPGPU Programming 9
2.4 OpenCL 10
2.4.1 OpenCL Architecture 10
2.5 Programming Models in Parallel Computers 14
2.6 Modeling Execution of Parallel Programs 15
2.7 Tesla S1070 17

3 Computational Fluid Dynamics and Porous Rocks **19**
3.1 Computational Fluid Dynamics 19

3.2 Lattice Boltzmann Method 20
3.2.1 Fundamentals 21
3.3 Related Work 26
3.4 Porous Rocks 27

4 Implementation **29**
4.1 Issues With Naive LBM on GPU 29
4.2 Tuning LBM for Execution on GPU 30
4.3 General implementation 33
4.3.1 Initialisation phase 35
4.3.2 Collision phase 36
4.3.3 Streaming phase 39
4.3.4 Border exchange phase 41
4.3.5 Boundary Conditions 41
4.3.6 Calculating Permeability 42
4.3.7 Convergence Check 42
4.4 Thread Implementation 43
4.5 MPI Implementation 44
4.6 Hybrid Implementation 45

5 Results and Discussion **47**
5.1 Test Environments and Methodology 47
5.2 Validation 49
5.2.1 Validation Against Poiseuille Flow 49
5.2.2 Porous Rock Measurements 51
5.3 Performance Measurements 52
5.3.1 Single GPU 52
5.3.2 Thread Implementation 54
5.3.3 MPI Implementation 57
5.3.4 Hybrid Implementation 59
5.4 Discussion 60

6 Conclusions and Future Work **63**
6.1 Future Work 64

Bibliography **64**

A Hardware Specification **70**

B Detailed Timings **73**
B.1 Single GPU 73
B.2 Thread Implementation 75

B.3 MPI Implementation 82
B.4 Hybrid Implementation 86

C Selected Source Code **89**
C.1 Collision Phase 89
C.2 Streaming Phase 90
C.3 Border Exchange Phase 92

List of Figures

2.1 A comparison of SIMD and SPMD kernels 5
2.2 Conceptual drawing of a shared memory multiprocessor 6
2.3 The thread model of OpenMP 7
2.4 Conceptual drawing of a cluster 8
2.5 Conceptual drawing of a shared memory multiprocessor cluster 9
2.6 OpenCL platform model 11
2.7 OpenCL execution model 12
2.8 OpenCL context with command queues and devices 12
2.9 OpenCL memory model 13
2.10 NVIDIA Tesla S1070 architecture 18
2.11 NVIDIA Tesla S1070 connection configuration 18

3.1 The three phases applied in every time step of LBM. 21
3.2 The LBM collision phase 21
3.3 The LBM streaming phase 22
3.4 The LBM Boundary condition. 22
3.5 Basic algorithm of the LBM 25

4.1 Memory requirement for D3Q19 LBM model, using double precision and temporary values. 30
4.2 Comparing the memory requirement of our and a naive implementation of LBM with the D3Q19 model. 33
4.3 The main phases in our implementations 34
4.4 Domain decomposition 35
4.5 The interleaved sequential copy pattern used in the border exchange phase 41

5.1 Fluid flow between two parallel plates 50
5.2 Comparison of known and simulated velocity profile for the Poiseuille flow using the three implementations 51
5.3 Speedup of using two Tesla C2070, Tesla C1060 and T10 compared to one 57

5.4 Speedup of using four T10 compared to one and two 58
5.5 Single iteration time of the MPI and thread implementations using 2 GPUs 61

List of Tables

5.1 Specifications of the three GPU types used 48
5.2 Technical data about the datasets used 48
5.3 Parameter values used during the Poiseuille flow simulations . 50
5.4 Parameter values used during the porous rocks measurements 52
5.5 Known and calculated permeability for Symetrical Cube, Square Tube and Fontainbleau 52
5.6 Time consumption in ms for the different phases of simulation on a single NVIDIA Tesla C2070 53
5.7 Time consumption in ms for the different phases of simulation on a single NVIDIA Tesla C1060 53
5.8 Time consumption in ms for the different phases of simulation on a single NVIDIA Tesla T10 53
5.9 Time consumption in ms for the different phases of simulation on a shard memory system with 2 Tesla C2070 55
5.10 Time consumption in ms for the different phases of simulation on a shard memory system with 2 Tesla 1060 55
5.11 Time consumption in ms for the different phases of simulation using two of the GPUs in the NVIDIA S1070 55
5.12 Time consumption in ms for the different phases of simulation using four GPUs on the NVIDIA S1070 56
5.13 Timings in ms on the cluster using all 4 GPUs (2 x Tesla 2070 on one node and 2 x Tesla 1060 on one) 59
5.14 Timings on a GPU cluster using 2 GPUs 59
5.15 Timings of using the hybrid implementation with 2 nodes and 4 GPUs 60

A.1 Shared memory multiprocessor multi-GPU feature 70
A.2 Cluster file server 71
A.3 Cluster compute node 1 71
A.4 Cluster compute node 2 72

List of Abbreviations

CFD Computational Fluid Dynamics

LBM Lattice Boltzmann Method

LGCA Lattice Gas Cellular Automata

MLUPS Million Lattice Updates Per Second

MFLUPS Million Fluid Lattice Updates Per Second

SLR Sparse Lattice Representation

OpenCL Open Computing Language

PE Processing Element

CU Compute Units

API Application Programming Interface

CUDA Compute Unified Device Architecture

SM Streaming Multiprocessors

DSP Digital Signal Processor

GPU Graphics Processing Unit

CPU Central Processing Unit

HPC High-Performance Computing

RAM Random Access Memory

HIC Host Interface Card

MPI Message Passing Interface

ILP Instruction-level Parallelism

MIMD Multiple Instruction Multiple Data

SIMD Single Instruction Multiple Data

SISD Single Instruction Single Data stream

MISD Multiple Instruction Single Data stream

SPMD Single Program Multiple Data

Chapter 1

Introduction

In recent years, the Graphics Processing Unit (GPU) has received a great deal of attention from the High-Performance Computing (HPC) community. Modern GPUs typically have hundreds of compute elements capable of performing operations in parallel. The high level of parallelism together with a high memory bandwidth makes the GPU particularly suited for data parallel computation. The introduction of frameworks such as NVIDIA's Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) made it possible to control the GPUs through high level C like programming languages. This makes it easier to use the highly parallel GPU to solve many compute and data intensive problems faced in HPC.

The lattice Boltzmann method for fluid simulation is frequently used to simulate physical phenomenas, in a wide variety of industries. In the oil industry the method is used to estimate the petrophysical properties, such as the permeability in porous rocks, to get a better understanding of the conditions that affect the oil production. The lattice Boltzmann method is highly data parallel, thereby making it suitable for acceleration on the GPU.

It is of great importance to the oil industry that the results obtained with the simulation are accurate, making it necessary to use big models requiring a large amount of memory. The limited amount of memory available on modern GPUs severely restricts the domain size that can be simulated on a single GPU.

1.1 Project goal

The main goal of this thesis is to investigate how to utilize multiple graphical processing units when doing LBM simulations on large datasets of porous rocks. The implementation is done in OpenCL making the work applicable for both ATI- and NVIDIA-based systems. First, to maximize the size possible to simulate on a single GPU, techniques for minimizing the memory footprint are implemented. Then the scalability properties of connecting multiple GPUs to a single host is investigated. Finally, the scaling properties of a cluster with GPUs is investigated.

1.2 Outline

The rest of this report is structured as follows:

In Chapter 2 we present some of the relevant background material and concepts in the field of parallel computing and GPU programming.

In Chapter 3 we give a brief introduction to the lattice Boltzmann method, as well as related work performed with LBM on GPU and multi-GPU systems. Techniques for calculation of permeability of porous rocks are also presented.

In Chapter 4 we describe the different multi-GPU implementations of the lattice Boltzmann method used in this thesis.

In Chapter 5 we present and discuss the validation and performance measurements of the implementations presented in the previous chapter.

In Chapter 6 we conclude with a summary of the results obtained in this thesis, and suggest areas of future work.

In Appendix A the specification and configurations of the hardware used in the simulations are presented.

In Appendix B more detailed timings from the test done are presented.

In Appendix C we present source code of the main GPU kernels developed for this thesis.

Chapter 2

Parallel Computing and The Graphical Processing Unit

This chapter introduces some of the important concepts for this thesis. In section 2.1 basic concepts of parallel computing are presented. Section 2.2 presents how parallel computers can be configured. Section 2.3 gives a brief summary of how GPU programming has evolved over time. Section 2.4 introduces some of the abstractions in OpenCL. In section 2.5 the programming models used as an inspiration for the implementations done in this thesis are presented. Section 2.6 highlights some of the aspects connected to modeling execution time on parallel systems. The chapter ends with a section describing one of the GPU systems used in the testing.

2.1 Parallel computing

From the very beginning of computer programming the trend has been to use a serial model to design and implement programs. However, modern computer hardware is inherently parallel. Even simple modern computers typically have five to six forms of parallelism [28]. To increase performance, compiler and hardware designers have tried to exploit implicit Instruction-level Parallelism (ILP) in serial code. ILP enables the processor to execute multiple instruction in parallel, but still preserving the serial programming model.

Unfortunately, ILP has reached the point of diminishing returns [4], and modern machines add more and more cores to increase compute capabilities.

Therefore, programmers need to explicitly express the parallelism withing the program, giving a renewed interest in ways for the programmer to control the parallelism within the program.

2.1.1 Forms of Parallelism

Parallel computing can be done in many different ways. The two most common classifications are task parallelism and data parallelism.

In task parallelism, the work is divided into independent tasks that can be performed in parallel by different processing elements. Separate instruction streams work on separate data streams in parallel. This parallelism form is usually called Multiple Instruction Multiple Data (MIMD). It is often hard to find a large number of separate tasks inside a program limiting the number of processing elements that can be used. Also these tasks are also usually not completely independent, making communication and synchronization between tasks necessary.

In data parallelism the data elements in a collection can be decomposed into chunks, where the same operation is performed on one or more elements from the different chunks in parallel. Data parallelism makes it possible to divide the work load among all processing elements available by decomposing the data collection into a appropriated number of chunks. The simplest form of data parallelism is called Single Instruction Multiple Data (SIMD) where the same instruction is preformed on multiple data elements in parallel. SIMD parallelism has many advantages both from the programming and the hardware point of view. For the programmer it is a natural extension to the serial programming model and from a hardware perspective it is easily implemented because only the data path has to be duplicated. As there is only one control flow, it is difficult to take advantage of special cases to avoid extra work when the amount of required computation differ between the data elements. A slightly more powerful model than SIMD is the Single Program Multiple Data (SPMD) model, a subcategory of the more general MIMD model. In SPMD the same program is applied to multiple data elements. As each program has its own control flow, it is easier to differentiate the amount of work done on each data element. Figure 2.1 shows how a SPMD kernel can take a different route trough the program based on the data, while a SIMD program has to apply the same instruction to every data element.

The SIMD and MIMD model together with the two other models, Single Instruction Single Data stream (SISD) and Multiple Instruction Single Data

stream (MISD) originate from Flynn's taxonomy [13]. SISD is the normal serial execution model, where a single instruction stream is applied to a single data element. In the fourth category MISD multiple instructions are applied to a single data element. This processing model is not widely used today [41], but as Flynn notes, it is found in ancient plug-board machines [13].

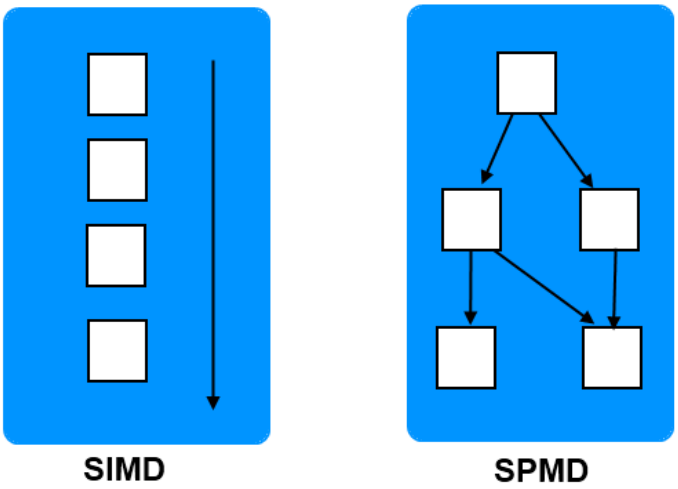


Figure 2.1: A comparison of SIMD and SPMD kernels. (Illustration based on [28])

2.2 Hardware configuration

There are multiple ways to build a parallel computing system. The two most common are shared memory multiprocessors and clusters. These two configurations can also be combined into clusters where each node is a shared memory multiprocessor.

2.2.1 Shared Memory Multiprocessor

A shared memory multiprocessor system, also called a shared memory system, is comprised of a set of processors and a set of memory modules that are connected through a interconnection network as shown in Figure 2.2. A shared memory multiprocessor system employs a single address space where

each location in main memory has a unique address used by every processor in the system to access that specific location. As every process in the system has access to the same memory, synchronization and communication between processes can be implemented simply by reading and writing to the same memory location. As the number of compute elements in the shared memory multiprocessor systems increases, it becomes harder to design certain components of the system, such as the interconnect between the processing and memory modules, making the shared memory multiprocessor model unsuitable for big systems [41].

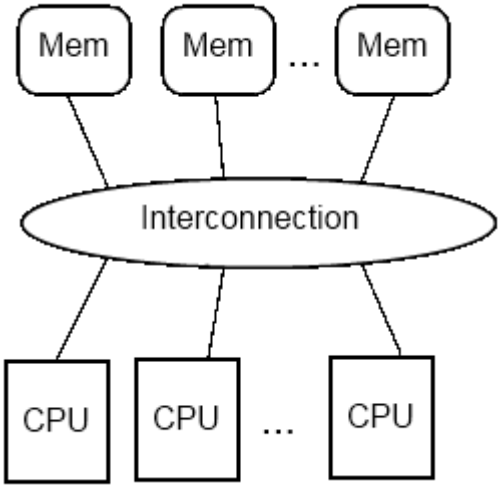


Figure 2.2: Conceptual drawing of a shared memory multiprocessor

Since each processor in a shared memory system has access to the same memory it is usual to use thread based programming on these systems. In thread based programming the same program is executed by all threads and resides in the shared memory. Each thread runs on its own core in the system and usually works on its own set of data.

Programming Shared Memory Systems

OpenMP is the de facto standard for easy parallel programming on shared memory systems [5]. It is based on a set of compiler directives, library routines, and environment variables giving the programmer the ability to parallelize a sequential code by injecting compiler directives in the areas that benefit from parallelization. The use of compiler directives makes it possible

to incrementally introduce parallelism into a sequential program with little implementation effort.

Figure 2.3 shows the fork-join threading model used by OpenMP. At the top we see the serial execution where each parallel task is preformed sequentially. The bottom part shows how the master OpenMP thread *forks* a set of worker threads when a parallel task is encountered. The parallel tasks are then divided among the working threads. When the tasks are completed, control is returned to the master thread.

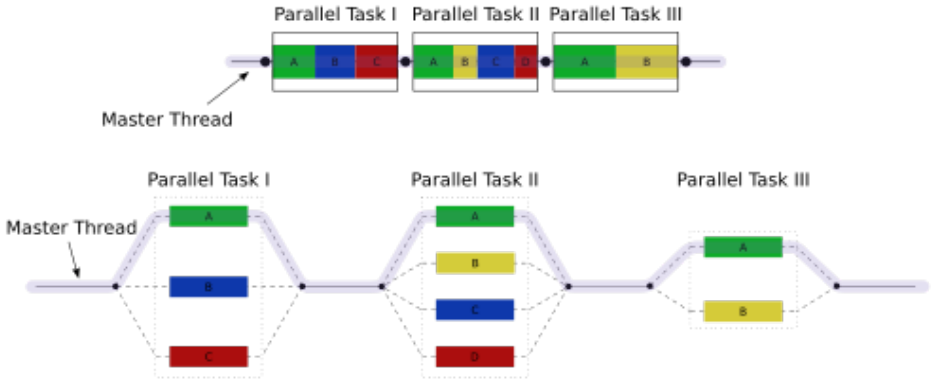


Figure 2.3: Thread model of openMP Figure from [40]

If the programmer needs more explicit control over the parallelization, one commonly used alternative is POSIX threads (Pthreads). In Pthreads the programmer has explicit control over creation, synchronization and termination of threads. The usage of Pthreads requires more effort in the development and debugging phase as the introduction of multiple threads can result in hard to find errors, such as race conditions.

2.2.2 Cluster

Another approach to building parallel systems is to connect several individual nodes each with its own computing element, memory module and memory address space, as shown in Figure 2.4. As each node in the cluster is a complete computer and has its own flow of control, it is common to use the Single Program Multiple Data (SPMD) processing model. Since each node of the system is completely independent from the other nodes, the cluster configuration is highly scalable and easy to construct and extend.

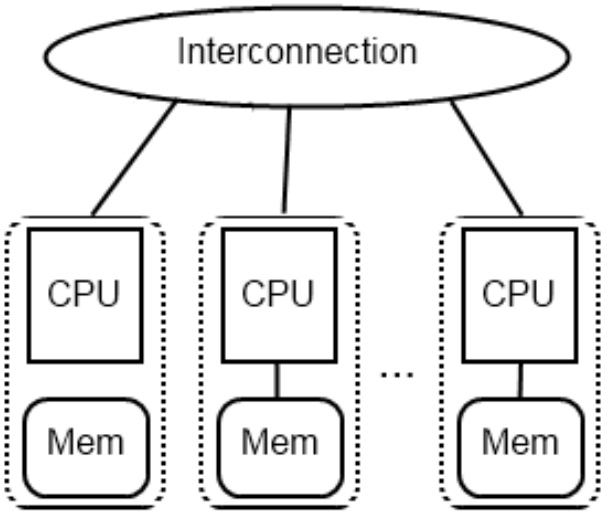


Figure 2.4: Conceptual drawing of a cluster

Clusters represent a large family of systems classified by how they are implemented. Beowulf clusters use low cost off-the-shelf components and connect the nodes with a low cost interconnect such as Ethernet, making it possible to design a powerful cluster at a relative low cost [41]. Another approach is a shared memory multiprocessor cluster, where each of the nodes in the cluster is a shared memory system, as shown in Figure 2.5. A shared memory multiprocessor cluster is usually constructed with a certain application in mind and usually use specially designed interconnects to lower the communication overhead.

Message Passing Interface

The de facto standard for programming communication and synchronization between nodes in a cluster is the Message Passing Interface (MPI). MPI is an Application Programming Interface (API) specification for how different invocations of the same program (processes) can communicate through sending and receiving of messages. Multiple implementations of the MPI specification exist, and vendors of high performance computing systems usually include an implementation of MPI specifically tuned to their hardware.

At the base of MPI is the concept of a communicator. A communicator is a group comprised of a subset, or all of the processes. Within the group each process is assigned a unique rank (number). Communication between

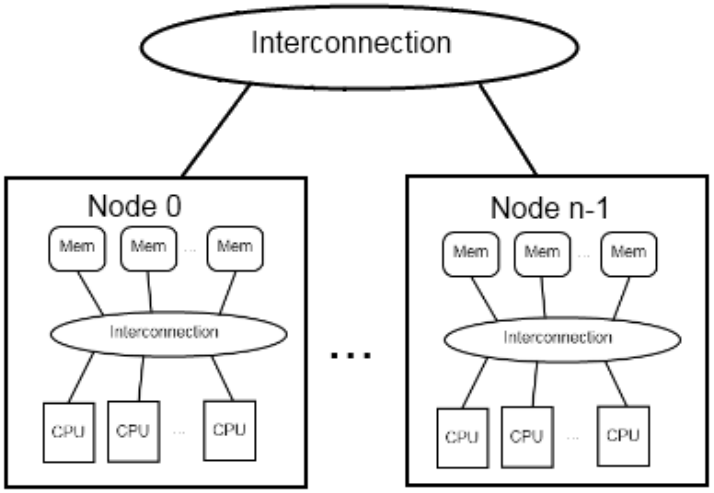


Figure 2.5: Conceptual drawing of a shared memory multiprocessor cluster

processes usually takes place within a single communicator and can either be point-to-point operations (e.g. send, receive) or collective operations (e.g. broadcast, reduce) [41].

The same communication patterns often occur multiple times within a program (e.g. the exchange of data between processes at the end of each iteration within a simulation). For such a situation MPI has functions for binding the argument list used in the communication to a persistent communicator request once, and then reusing this request. This construct allows for reduction of the overhead associated with the setup of communication between processes.

2.3 GPGPU Programming

This section is taken from the fall specialization project done by the author [39], and used with some changes.

The GPU was originally developed to offload the compute intensive work associated with 2D and 3D graphics from the Central Processing Unit (CPU). Graphical computation often involves performing the same operations on different elements of the input data. These operations are usually independent and can be performed in parallel.

The early GPU was a fixed-function pipeline processor, capable of performing common graphical operations with a high degree of hardware parallelism [33]. The stages of the pipeline later become programmable. The GPU was still designed for graphical processing. But the introduction of programmable pipeline stages meant that the HPC community could express their problems as sets of graphical operations to take advantage of the highly parallel GPU. Frameworks such as NVIDIA's CUDA [30] and the ATI Stream technology [2] enable the programmer to control the GPU through C like general purpose programming languages. This makes GPU programming more accessible to the HPC community. To ease the development for heterogeneous systems, frameworks such as OpenCL have been introduced making development independent of the underlying hardware.

2.4 OpenCL

This section is taken from the fall specialization project done by the author [39], and used with some changes.

Open Computing Language (OpenCL) is an open royalty-free standard for general purpose computing on heterogeneous platforms. OpenCL consists of an architecture, a programming language and an API. Implementations of OpenCL are available on NVIDIA devices supporting the CUDA architecture [31] and ATI devices with the ATI Stream technology SDK [1].

2.4.1 OpenCL Architecture

The OpenCL architecture defines three models depicting: the computation environment containing the host and the compute devices (platform model), the execution model and the memory hierarchy.

Platform Model

The OpenCL platform model is depicted in Figure 2.6. The *host* system is connected to one or more *devices*. Each device consists of one or more Compute Units (CU) each having a number of Processing Element (PE). On the devices the actual computation is done by the PEs. Typical devices include GPUs, Digital Signal Processor (DSP), IBM and Sony's CELL BEs and multicore CPUs.

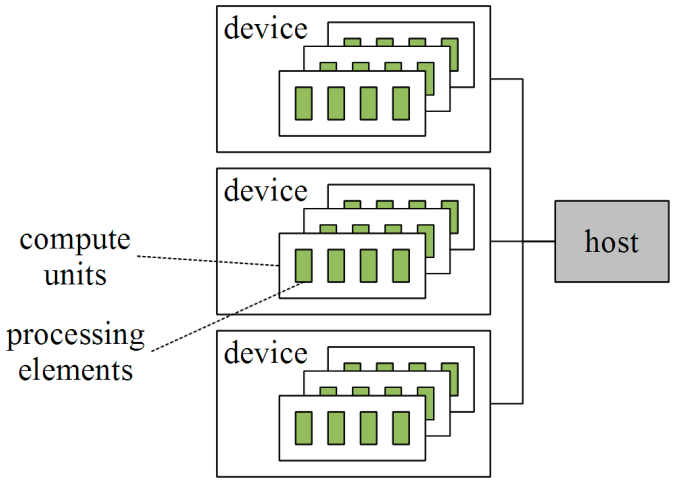


Figure 2.6: OpenCL platform model (from [27], used with permission)

Execution Model

A program written for OpenCL consists of two parts: a host program that executes on the host and a set of *kernels* that execute on one or more devices. The OpenCL execution model, as depicted in Figure 2.7, defines the execution of kernels on a device. The kernel is executed simultaneously by multiple threads, called a *work-item* in OpenCL. Work-items are organized into one, two or three dimensional groups, in OpenCl called a *work-group*. These work-groups are organized into a *NDRange* of the same dimension as the work groups. Each work-item is assigned a unique index within the work-group, local work-item ID, and a unique index within the NDRange, global work-item ID.

The host program creates and maintain a *context*, as shown in Figure 2.8, holding information about devices, command queues and memory, available to the host. To assign work to the device the host code enqueues *command* onto the *command queue*, the commands available in OpenCL are: Kernel execution, memory transfer, and synchronization. The execution of commands can be either in-order - commands are executed and finished in the order they appear in the queue, or out-of-order - the commands are started in the order they appear in the queue, but not guaranteed to complete in order. Transferring data between host and device can be made either non-blocking or blocking depending on whether the control should be returned to

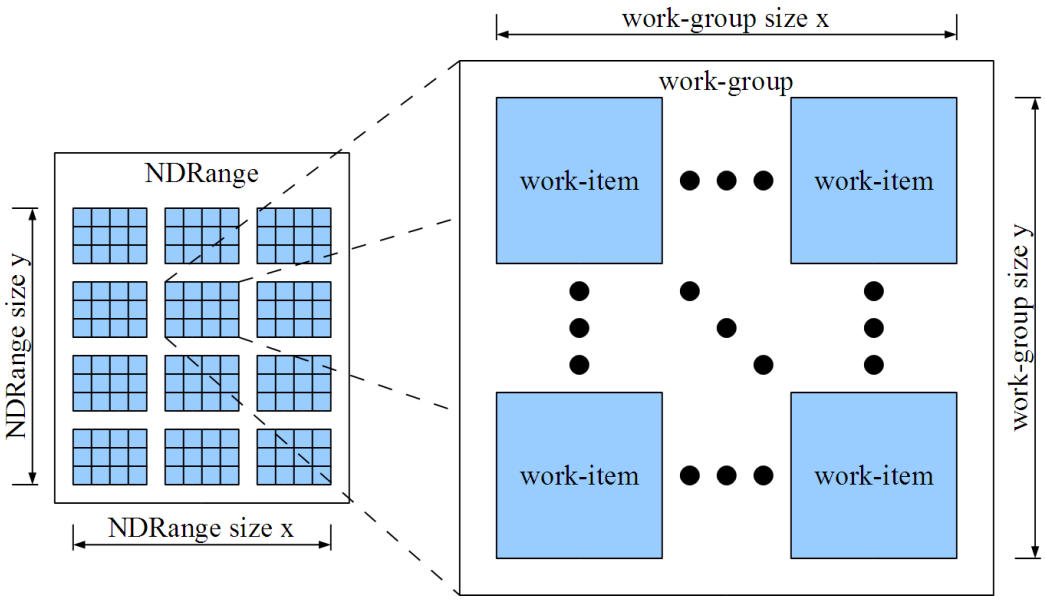


Figure 2.7: OpenCL execution model (from [27], used with permission)

the host code as soon as the command has been placed on the queue. Kernel execution is always non-blocking returning the control to the host code immediately after the command has be placed on the queue.

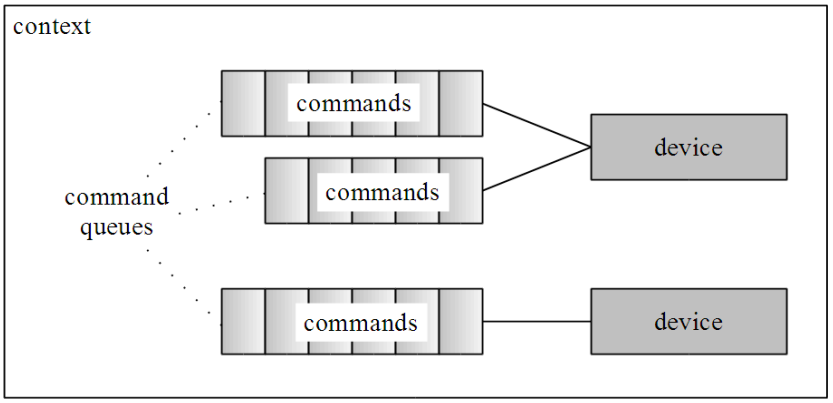


Figure 2.8: OpenCL context with command queues and devices (from [27], used with permission)

Memory Hierarchy Model

The memory hierarchy of OpenCL, shown in Figure 2.9 is divided into four distinct parts: Global, constant, local and private. Private and local memory are not directly accessible by the host. Global memory together with the read-only constant memory is shared among all the work-groups and is directly accessible by the host system. Private memory is individual to each work-item, while local memory is shared between the work-items within a work-group. Global memory is the main memory of the device and is large, but with a low bandwidth. Local and constant memory are often small amount of on-chip memory with a high bandwidth.

To control the device memory the host code uses *Memory objects*, which are parts of device memory together with its attributes. OpenCL defines two types of memory objects *Buffers* and *Images*. *Buffers* store data in a sequential array, and are accessed as a stream of bytes. *Images* are two or three dimensional arrays intended for storing images and textures, and are usually optimized for two and three dimensional access patterns.

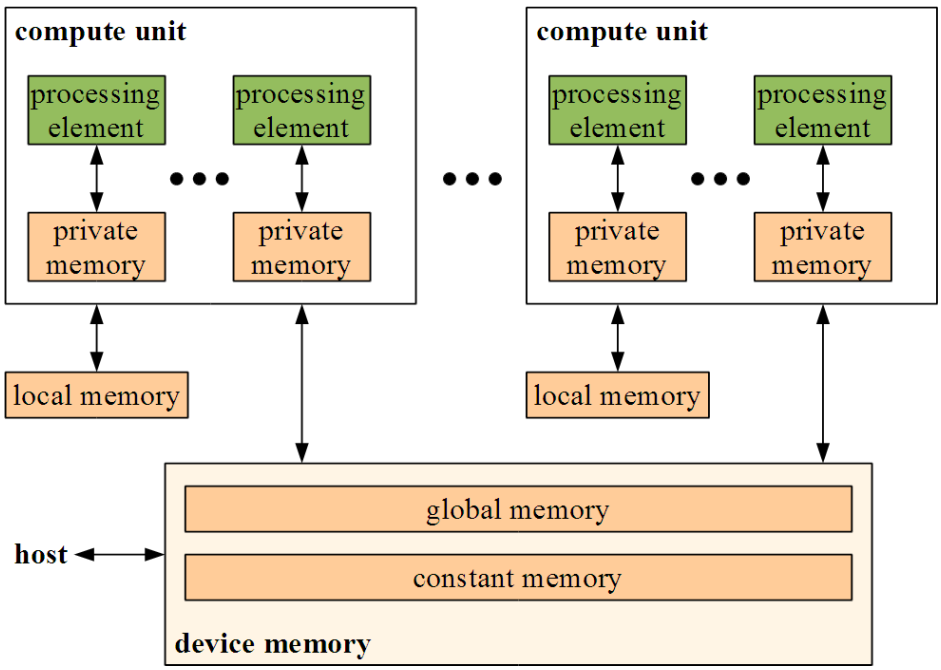


Figure 2.9: OpenCL memory model (from [27], used with permission)

2.5 Programming Models in Parallel Computers

When programming systems with multiple forms of parallelism, like the shared memory multiprocessor cluster, the programmer needs to combine the programming style for the different parts of the system. In the shared memory multiprocessor cluster case the problem is to combine the distributed memory programming used in inter-node communication, usually programmed with MPI, with the shared memory programming used inside the node, usually programmed with OpenMP. A classification on MPI and openMP based parallel programming schemes for hybrid system is given by Rabenseifner [35]. One can use pure MPI, pure OpenMP on top of a virtual distributed shared memory system or some combination of MPI and OpenMP.

Pure MPI: Each CPU in each node in the cluster runs its own MPI process. The MPI library can be optimized to use the shared memory between MPI processes within the same node, and the interconnect between MPI processes located on different nodes. The main drawback of a pure MPI approach is that it adds unnecessary complexity to the intra-node communication. This drawback becomes less off a performance issue when the intra-node communication is significantly faster than inter-node communication.
Pure OpenMP: Requires a single memory address space shared between every CPU in the system. The single address space has to be simulated by making what is called a virtual distributed memory system. The main drawback with a pure openMP approach is that the simulation of a single address space introduce extra overhead, and also makes the execution time dependent on where in memory the data is located.
Hybrid: One MPI process is executed on each node in the system. The communication between nodes is handled by the MPI process, while the work done on a single node is parallelized with OpenMP. This category can be subdivided into two. One where the OpenMP threads are sleeping while the communication is taking place, and one where calls to the MPI routines are overlapped with application code preformed by the OpenMP threads. The latter subcategory requires that the application code be divided into two parts: the code that is not dependent on the data exchange and therefore can be overlapped with the communication, and the code that depends on data exchange and has to be deferred until the data is received. The main drawback with both of the hybrid models is that as only one thread on each node is communicating they are not able to fully utilize the inter-node bandwidth [35]. The non-overlapping model also suffers from bad load balancing,

as most of the threads are idle for large amounts of the execution time.

A recent trend in HPC is to build clusters where most of the computational power comes from GPUs connected to some or all of the nodes. When using multiple GPUs in a computing system there are three principal components: host nodes, GPUs and interconnect. Since most of the calculation is supposed to be preformed by the GPUs the support infrastructure, (host systems and interconnection) has to be able to match the performance characteristics of the GPUs in order not to create bottlenecks. Kindratenko el al. [23] argue than in order to fully use the system it is important to have a one-to-one mapping between the CPU and GPU count in the system. Also that the total amount of host memory is at least as much as the the combined amount of memory on the GPUs.

Spampinato [36] argues that a multi-GPU system has many similarities to a cluster of shared memory nodes. It follows from his arguments that multi-GPU systems can be programmed using the same models as used for a cluster of shared memory multiprocessor nodes.

2.6 Modeling Execution of Parallel Programs

When the execution of a single program is distributed across multiple processing elements, the run time can be divided into two components: the time used for computation and the time used for communication. On a shared memory system the communication part consists mainly of the synchronization overhead, while on a cluster the communication part is dominated by the inter node message passing. The amount of time spent computing and communicating depends on the application, but is usually related to the size of the problem and the number of processes used. A simple model for the time usage for a parallel program is given in [41] as:

$$T_{par} = T_{comp}(n, p) + T_{comm}(n, p) \tag{2.1}$$

where n is the problem size parameter, p is the number of processes used, T_{comp} is the time used in computation and T_{comm} is the time spent communicating.

The communication of an application is usually composed of multiple communication flows between different nodes of the system. The total communication time is therefore calculated as the sum of the time spent in each

flow. The time used in a single communication flow can be modelled, by the Hockney model [18] as Equation 2.2:

$$T_{comm} = t_{startup} + w \times t_{word} \tag{2.2}$$

where $t_{startup}$ is the time used to send a message of zero bytes. This time includes the time used by both parties in packing and unpacking the message. This time is usually called the latency of the communication system, and is assumed to be constant. t_{word} is the time to send one data word over the communication channel. This value is usually given as the inverse bandwidth of the communication channel and usually assumed to be constant. w is the number of data words to be sent.

If the application code can be divided into, a data exchange dependent part and a non data exchange dependent part then it is possible to use the parts of the system not communicating to overlap some or all of the work independent of data exchange with the communication, thus decreasing the amount of time used.

A more detailed model taking the overlap between computation and communication into account, is presented by Barker *et al.* [7], and shown in Equation 2.3

$$T_{par} = T_{comp}(n, p) + T_{comm}(n, p) - T_{overlap} \tag{2.3}$$

where $T_{overlap}$ is the time when the system is both communicating and performing computation at the same time.

When analysing a system where most of the computational power comes from GPUs, it is necessary to extend the model for communication time to take account for the time spent transferring data between GPUs and hosts. As pointed out by Spampinato [36], the communication between two GPUs in a system can be estimated as Equation 2.4.

$$T_{comm} = 2 \times T_{GPU-host} + T_{host-host} \tag{2.4}$$

When transferring contiguous parts of memory between host and GPU Spampinato [36] finds that $T_{GPU-host}$ can be estimated as Equation 2.5

$$T_{GPU-host} = t_{startup} + w \times t_{word} \tag{2.5}$$

where $t_{startup}$, w and t_{word} are the startup and the inverse bandwidth for the interconnect between the host and GPU.

2.7 Tesla S1070

In this thesis, we use a Tesla S1070 computing system as a multi-GPU shared memory system. In this section, we describe the system because its structure has a direct influence the results obtained in our experiments.

The Tesla S1070 system is a 1U rack-mount system with four Tesla T10 GPUs. Each of the four GPUs contain 240 compute cores (in OpenCl termenology PEs) organized into 30 Streaming Multiprocessors (SM) (in OpenCl terminology CU).

Figure 2.10 shows a schematic view of the architecture of the NVIDIA Tesla S1070 computing system. As shown in the figure each GPU is connected to 4GB of Random Access Memory (RAM), giving the system as a total access to 16 GB RAM. Pairwise the GPUs are connected together in a NVIDIA switch that again is connected to the host via a Host Interface Card (HIC) connected to a PCI Express 1x or PCI Express 2x expansion slot on the host system. The HIC is capable of providing a transfer rate of up to 12.8 GB/s. To fully utilize the system from a single host both NVIDIA switches have to be connected to the same host as depicted in Figure 2.11 [32].

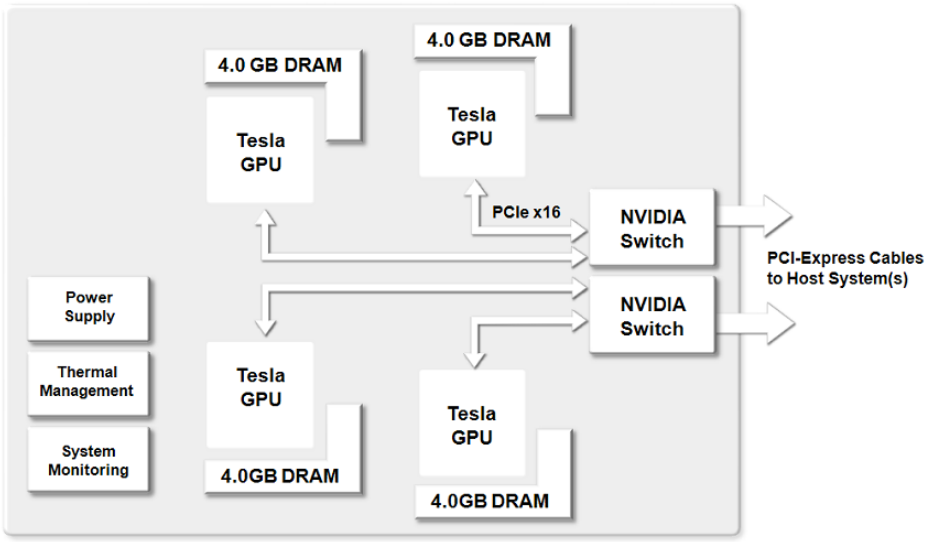


Figure 2.10: Schematic of NVIDIA Tesla S1070 architecture (from [32], used with permission)

Figure 2.11: NVIDIA Tesla S1070 connection configuration (from [32], used with permission)

Chapter 3

Computational Fluid Dynamics and Porous Rocks

In this chapter, we present an introduction of the theoretical background behind the lattice Boltzmann method used to simulate flow through porous rocks. In Section 3.1 we give a brief introduction to the field of computational fluid dynamics. Section 3.2 presents the theory and equations behind the lattice Boltzmann method. Section 3.3 presents some of the related work done using LBM simulation on GPUs. The chapter ends with a section describing the process of calculating the permeability of porous rocks.

3.1 Computational Fluid Dynamics

Fluid dynamics, the study of how fluid flows, is of great interest to both academics and industry. The large number of molecules, even in a small amount of fluid, makes it infeasible to track the movement of each individual molecule. Therefore, in continuum mechanics, the existence of molecules is ignored and the matter is treated as a continuous medium. Applying the laws of mass, momentum and energy conservation at sufficiently large time scales and distances eliminate the effect of the individual molecules, giving rise to a set of nonlinear partial differential equations known as the Navier-Stokes equations. These equations, also known as the conservation equations, describe the flow of fluids. Unfortunately, these equations can only be solved analytically if the geometry of the domain and the boundary conditions are not too complicated [26].

Computational Fluid Dynamics (CFD) is the use of computers to solve the Navier-Stokes equations numerically. One way these equations can be solved numerically is by using Lattice Gas Cellular Automata (LGCA) models. In LGCA the domain is divided into a lattice grid. Each lattice location, also known as cell, represents a location in space and is either a fluid or solid element. Within each cell the particles move along a set of distinct velocity vectors occupied by either zero or one particle. At each time step particles propagate to the neighboring lattice cell. When two or more particles collide at a lattice location particles, exchange momentum while conserving total mass and momentum [42].

One of the earliest attempts to numerically solve the Navier-Stokes equations based on LGCA was the HPP model by Hardy, Pomeau and de Pazzis [17]. This model used a square lattice grid with four velocity vectors at each cell connected to the closest neighbors, but not on the diagonal. The usage of square lattices and only four particles per location makes the model suffer from lack of rotational invariance [17]. To overcome the rotational invariance of the HPP model Frisch *et al.* [14] proposed the FHP model using a triangular lattice grid with six velocity vectors per cell. However, since some of the collisions in FHP have more than one outcome the model suffers from static noise.

3.2 Lattice Boltzmann Method

The Lattice Boltzmann Method (LBM) was introduced by McNamara and Zanetti [29] to remove statical noise from the LGCA methods. In LBM the boolean particle distributions (velocity vectors) used in LGCA is replaced with real number particle distribution functions. The flow of a fluid is then described as the interaction between these particle distribution functions in three separate phases, shown in Figure 3.1.

The collision step is designed to simulate the interaction between particles inside a lattice location. The existing and incoming particles collide and are distributed among the discrete velocities, see illustration 3.2. The collision is constructed to conserve mass, energy and momentum within the simulation. In the streaming step the particle distribution functions propagate to the neighbor lattice as shown in Figure 3.3.

Special care needs to be taken when handling lattice nodes either at the edge of the simulation domain or lattices adjacent to a solid lattice location.

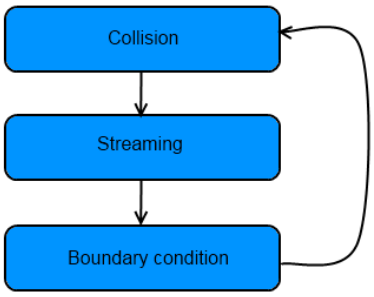


Figure 3.1: The three phases applied in every time step of LBM.

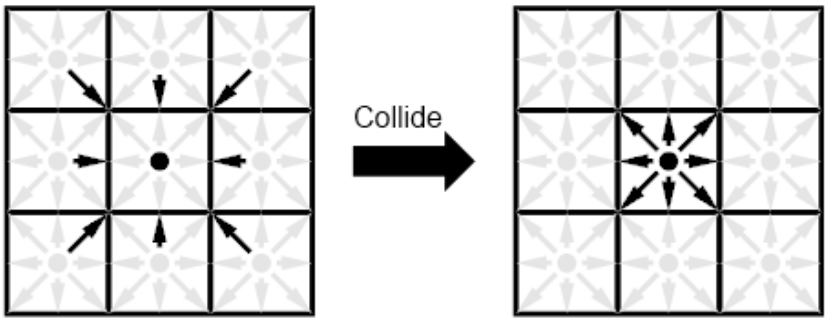


Figure 3.2: The LBM collision phase (based on [24])

The standard way of handling the interface between fluid and solid cells is a no-slip boundary condition (also called a bounce back boundary condition), shown in Figure 3.4. With this boundary condition particles moving into a solid cell are reflected back in the opposite direction. The result is zero velocity at the wall, located half-way between the last fluid cell and the first wall node, and ensures that there is no flux across the wall [24]. It is also common to use a periodic domain, with particles exiting through one side of the domain being reinserted at the opposite side.

3.2.1 Fundamentals

The LBM is not just an extension of the LGCA methods. It has been shown (e.g. Körner et *et al.* [24]) that LBM can be derived directly from the underlying physical model, the Boltzmann equation, and that the Navier Stokes flow can be recovered in the macroscopic limit [24].

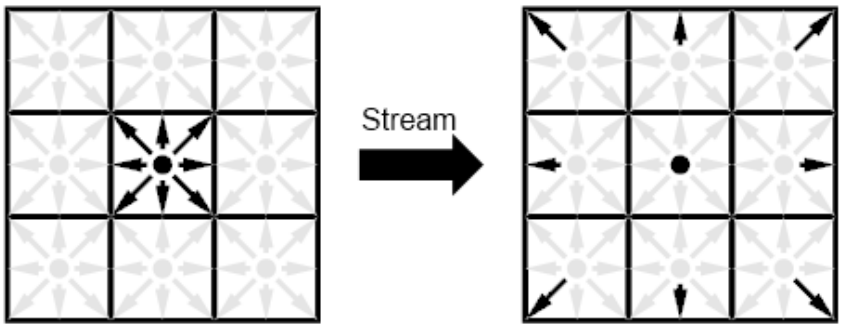


Figure 3.3: The LBM streaming phase (based on [24])

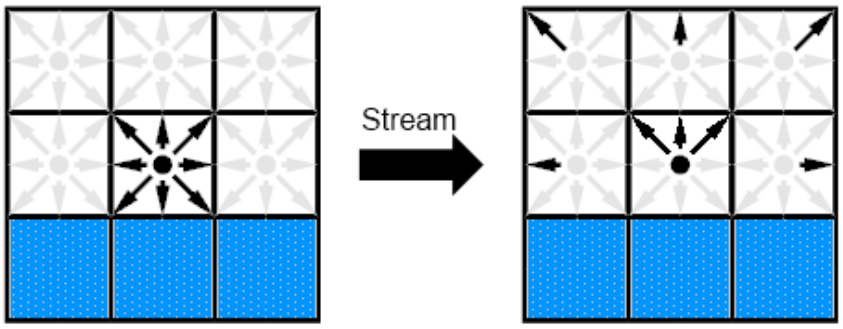


Figure 3.4: The LBM Boundary condition (based on [24])

The Boltzmann equations without external forces can be written as Equation 3.1 [24]:

$$\frac{\partial f}{\partial t} + e\frac{\partial f}{\partial x} = Q(f, f) \tag{3.1}$$

where f is the particle distribution function, e is the particle velocity and $Q(f, f)$ is the collision operator describing the interaction between colliding particles. The collision operator consists of a complex intergrodifferential expression, and is often simplified with the Bhatnagar-Gross-Krook (BGK) operator, Equation 3.2 [24]:

$$Q(f, f) = -\frac{1}{\lambda}(f - f^0) \tag{3.2}$$

where f^0 is the Maxwell-Boltzmann equilibrium distribution function, and λ is the relaxation time controlling the rate of approaching equilibrium. By applying the BGK approximation to Equation 3.1 we get Equation 3.3 [24]

$$\frac{\partial f}{\partial t} + e\frac{\partial f}{\partial x} = -\frac{1}{\lambda}(f - f^0) \tag{3.3}$$

To solve Equation 3.3 numerically we first discretize the velocity into a finite set of velocity vectors $e_i (i = 0, ..., N)$. A common classification of the discretization used is the DaQb model, where Da is the number of dimensions and Qb is the number of distinct velocities $\vec{e_i}$. Typical discretizations are D2Q9, D3Q15 and D3Q19. Applying a velocity discretization to Equation 3.3 give us the discrete Boltzmann equation Equation 3.4 [24]:

$$\frac{\partial f_i}{\partial t} + e_i\frac{\partial f_i}{\partial x} = -\frac{1}{\lambda}(f - f_i^{eq}) \tag{3.4}$$

where f_i^{eq} is the equilibrium particle function in the i direction. For all the DaQb models described here the equilibrium function in direction i becomes Equation 3.5:

$$f_i^{eq} = \rho w_i \left[1 + \frac{3}{c^2}\vec{e_i} \cdot \vec{u} + \frac{9}{2c^2}(\vec{ei} \cdot \vec{u})^2 - \frac{3}{2c^2}\vec{u} \cdot \vec{u}\right] \tag{3.5}$$

where $c = \frac{\Delta x}{\Delta t}$, and w_i is a weighting for each distribution function, dependent on the lattice model. c is normally normalized to 1.

The macroscopic values of the density ρ, momentum $\rho\vec{u}$, velocity $\vec{u}$ and kinematic viscosity ν can be calculated from the lattice locations using Equations 3.6, 3.7, 3.8 and 3.9.

$$\rho = \sum_{i=0}^{N} f_i \tag{3.6}$$

$$\rho\vec{u} = \sum_{i=0}^{N} \vec{e_i} f_i \tag{3.7}$$

$$\vec{u} = \frac{1}{\rho}\sum_{i=0}^{N} \vec{e_i} f_i \tag{3.8}$$

$$\nu = \frac{2\tau - 1}{6} \tag{3.9}$$

where N is the number of distinct vector velocities.

The next step in solving Equation 3.3 numerically is to discretize Equation 3.4 in time and space, giving rise to Equation 3.10

$$f_i(\vec{x} + \vec{e_i}, t + \Delta t) - f_i(\vec{x}, t) = -\frac{1}{\tau}\left[f_i(\vec{x}, t) - f_i^{eq}(\vec{x}, t)\right] \tag{3.10}$$

where $\tau = \frac{\lambda}{\Delta t}$ is the single relaxation parameter $\vec{x}$ is a point in the discretized space.

When implementing the LBM it is often advantageous or necessary to split Equation 3.10 into a collision and a streaming step. The splitting results in Equations 3.11 and 3.12

$$f_i^{out}(\vec{x}, t) = f_i^{in}(\vec{x}, t) - \frac{1}{\tau}\left[f_i^{in}(\vec{x}, t) - f_i^{eq}(\vec{x}, t)\right] \tag{3.11}$$

$$f_i^{in}(\vec{x} + \vec{e_i}, t + \Delta t) = f_i^{out}(\vec{x}, t) \tag{3.12}$$

where f_i^{out} is the distribution values after the collision step, but before the streaming step, and f_i^{in} is the distribution values after both the collide and streaming step are completed.

The bounce back boundary condition applied at the interface between fluid and solid cells can then be expressed by applying Equation 3.13 instead of Equation 3.11 at the solid cells [24].

$$f_i^{out}(\vec{x}, t) = f_{i_{oposit}}^{in}(\vec{x}, t) \tag{3.13}$$

Applying the bounce back boundary condition temporary stores the distribution function in the solid cell and returns the distribution back to the fluid cell with the opposite momentum in the next time step.

A basic implementation of the LBM consists of a series of nested loops, over the three dimensions, treating the collision and streaming phase separately. First a nested loop would apply the collision phase to each cell writing the result back to a temporary array (f^{out}). In a separate nested loop the values in the temporary array would be propagated to the neighboring cell in the original array (f^{in}). Figure 3.5 shows the operations involved in each phase.

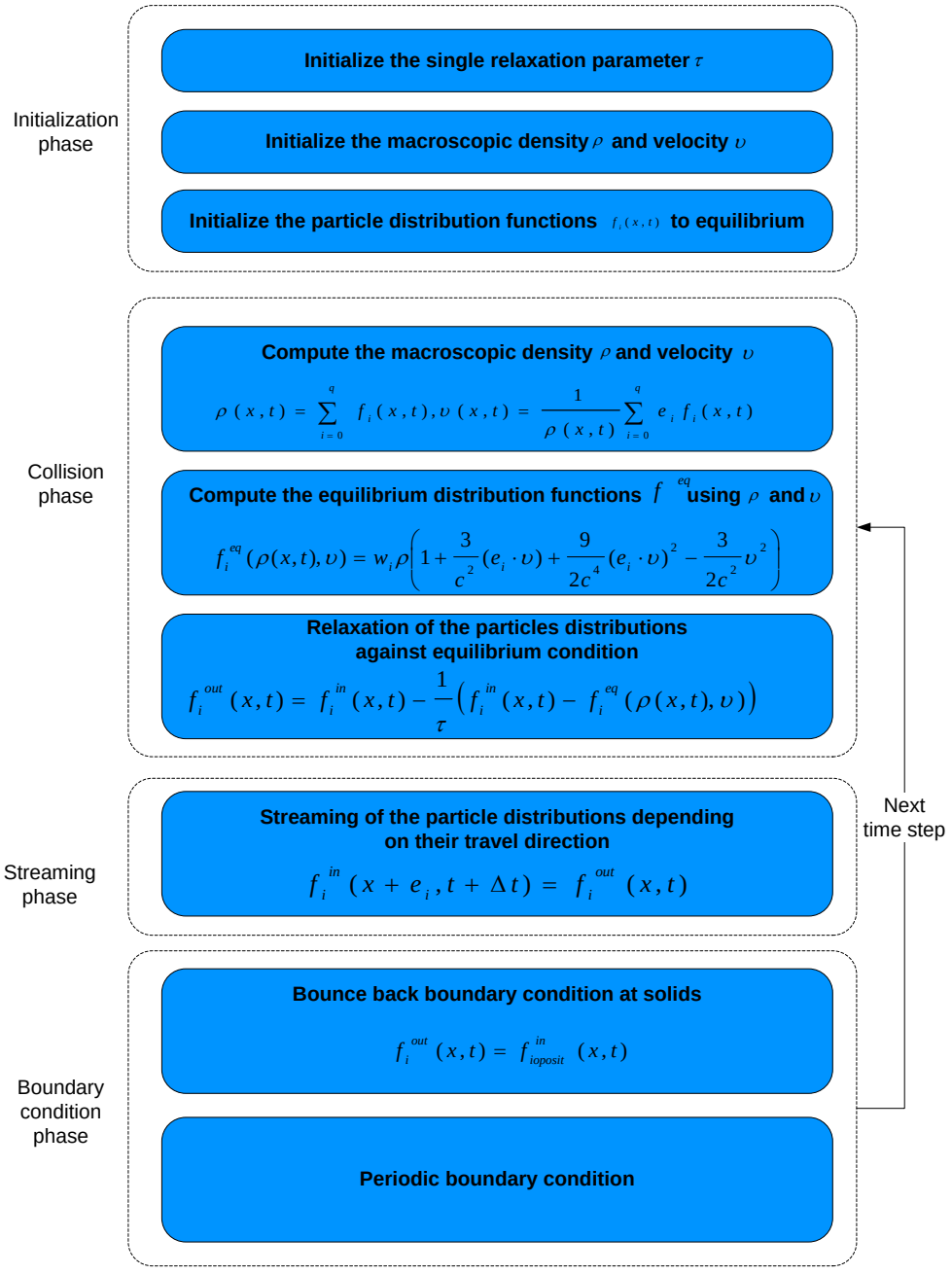


Figure 3.5: Basic algorithm of the LBM (based on [24])

3.3 Related Work

In this section, we present some of the related work performed in the field of LBM. Since this thesis investigate how to use multiple GPUs when simulating LBM, the focus is on LBM on GPU and multi-GPU systems. Two common metric used to evaluate and compare the performance of different LBM implementation is Million Lattice Updates Per Second (MLUPS) and Million Fluid Lattice Updates Per Second (MFLUPS). MLUPS indicate the number of lattice elements, including solids, that is updated in one second. While MFLUPS indicate the number of fluid elements updated in one second.

Tölke [38] implemented the Lattice Boltzmann method using the D2Q9 model and CUDA on a NVIDIA 8800 Ultra card, achieving a peak performance of 670 MLUPS. Tölke and Krafczyk [37] implemented the LBM method on a NVIDIA GeForce 8800 Ultra, using the D3Q13 model. By using single precision and paying special attention to the memory layout and the memory access pattern they where able to achieve a maximum performance of 592 MLUPS. Habich [16], implemented the D3Q19 model using CUDA on a GeForce8800 GTX achieving 250 MLUPS. Bailey el al. [6] improved the performance of the implementation used by Habich [16] to 300 MLUPS. Bailey *et al.* [6] also propose a space-efficient storage method reducing the GPU RAM requirement by 50%, doubling the size of domain possible to solve on the GPU. By using Bastien's technique for reducing the round off error when using single precision, Aksnes [3] was able to use single precision in the D3Q19 model to estimate the permeability of porous rocks, achieving a maximum of 184 MLUPS on a NVIDIA Quadro FX 5800 card.

The higher performance of the implementation by Tölke and Krafczyk and Tölke, is partially explained by the D2Q9 and D3Q13 models requiring less memory traffic than the D3Q19 model.

Fan *et al.* [11] used a 32 node cluster with each node containing a GeForce FX 5800 Ultra, to simulate the dispersion of airborne contamination in the Times Square area. The simulation was performed using the LBM and the D3Q19 model. Communication between nodes was handled by MPI resulting in a speedup of 4.6 compared to their CPU cluster implementation. Feichtinger *et al.* [12] implement a heterogeneous solver capable of running on both GPUs and CPUs. On a single Tesla C1060 they are able to achieve 300 MFLUPS. The implementation also achieves almost perfect weak scaling. Xian and Takayuki [43] report on the impact of different domain decomposition configurations when using a multi-GPU implementation of LBM using a D3Q19 model and MPI to communicate between nodes.

3.4 Porous Rocks

Permeability is a measure of how easily fluid flows through a porous rocks. In a rock with low permeability it is only possible to extract oil in close proximity to the well bore, while in a rock with high permeability oil can flow to the well bore from thousands of meters away [10]. By calculating the permeability and other petrophysical properties, the oil industry is able to better understand the factors that affect the production. It is therefore of great interest to be able to calculate the permeability of porous rocks from a simulation. The permeability of a rock can be calculated from the flow of fluid using Darcy's law expressed in Equation 3.14 [19]

$$q = -\frac{k}{\rho\nu}\frac{\Delta P}{L} \tag{3.14}$$

where k is the permeability of the porous medium, ρ is the fluid density, ν is the fluid kinematic viscosity, $\frac{\Delta P}{L}$ is the total pressure drop along the sample length L, and q is the volumetric fluid flux. In porous media q can be expressed as $q = \phi\vec{u}$ where ϕ is the sample porosity and $\vec{u}$ is the average velocity of the fluid. The porosity of a rock is defined as Equation 3.15.

$$\phi = \frac{V_p}{V_t} \tag{3.15}$$

where V_p is the volume of the pore space, and V_t is the total volume of the rock. The permeability of the rock can be found by Equation 3.16

$$k = \frac{\vec{u}\phi\rho\nu}{\frac{\Delta P}{L}} \tag{3.16}$$

Chapter 4

Implementation

This chapter presents our three LBM implementations. Section 4.1 describes some of the performance and space issues a naive LBM implementation has on a GPU. Section 4.2 presents the three techniques used to improve the performance of LBM on GPUs. Section 4.3 describes the parts common to all our implementations, before the chapter ends with three sections describing each implementation in more detail.

4.1 Issues With Naive LBM on GPU

Normally, in order to get the required accuracy, double precision is required in LBM simulations. There are mainly two problems with a naive implementation: First the usage of double precision decreases the performance of modern GPUs as they usually have significantly lower peak performance when using double precision as compared to single precision. Second the amount of memory used severely limits the size of the domain it is possible to simulate on a GPU.

Since real numbers are stored using a limited number of bits in a computer, approximations and rounding errors occur. These round off errors are greatest when the value of the numbers involved differ greatly, especially when using single precision. In the collision phase of LBM the equilibrium function is calculated from values with large differences making this phase particularly vulnerable for round off errors, and normally making it necessary to use double precision to store the particle distributions.

When discretizing the velocity space in a LBM simulation a smaller model

(e.g. D3Q15) means a smaller memory footprint, but less accuracy. To achieve the needed accuracy it is common to use the D3Q19 model. The usage of double precision and 19 particle distribution functions per lattice location consumes a large amount of memory. In a naive implementation it is also common to store a temporary value of each particle distribution, to avoid losing data in the streaming phase, further increasing the memory requirement. Figure 4.1 shows how the memory consumption increases with lattice size, when using the D3Q19 model with double precision and temporary values. In the figure we see that the largest simulation that fits on a single modern GPU contains approximately 256^3 lattice locations, as the largest total memory available in current GPUs are 6 GB.

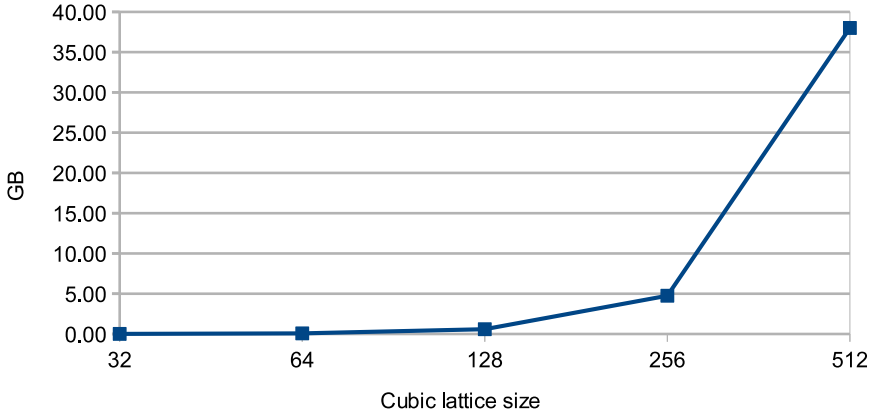


Figure 4.1: Memory requirement for D3Q19 LBM model, using double precision and temporary values.

4.2 Tuning LBM for Execution on GPU

To maximize the size of the dataset that can fit on a single GPU we used a series of performance tuning techniques to reduce the memory footprint without significant loss of accuracy.

By swapping data between the source and destination during the streaming phase [25], we were able to implement the streaming step without using temporary space, reducing the memory requirement by 50%.

To reduce rounding errors and make it possible to use single precision and still achieve high accuracy, we used the technique described by Bastien [8].

Bastien's approach makes the factors in the collision operator (Equation 3.11) closer to each other, thereby decreasing round off errors. Bastien achieves this by subtracting the constant $\rho_0 w_i$ from each term of the equation, where ρ_0 corresponds to the average density of the system, and w_i is the weighting of the particle distribution function. After variable substitution the equation for the collision operator becomes Equation 4.1

$$h_i^{out} = h_i^{in} - \frac{1}{\tau}\left[h_i^{in} - h_i^{eq}\right] \tag{4.1}$$

where $h_i^{out} = f_i^{out} - w_i\rho_0$, $h_i = f_i - w_i\rho_0$ and $h_i^{eq} = f_i^{eq} - w_i\rho_0$

The macroscopic values of the density ρ and velocity $\vec{u}$ can the be expressed in terms of h_i as Equations 4.2 and 4.3

$$\begin{aligned} \Delta\rho &= \sum_{i=0}^{N} h_i \\ \rho &= \Delta\rho + \rho_0 \end{aligned} \tag{4.2}$$

$$\vec{u} = \frac{\sum_{i=0}^{N} h_i \vec{e_i}}{\rho_0 + \sum_{i=0}^{N} h_i} \tag{4.3}$$

The original equilibrium function (Equation 3.5) can be rewritten as Equation 4.4

$$f_i^{eq} = \rho w_i \left[1 + F_i(u)\right] \tag{4.4}$$

with

$$F(u) = \frac{\vec{e_i} \cdot \vec{u}}{c_s^2} + \frac{1}{2c_s^2} Q_{i\alpha\beta u\alpha u\beta} \tag{4.5}$$

where c_s^2 is the speed of sound in the fluid medium, in our simulations given the value $\sqrt{(\frac{1}{3})}$ [24]. By substituting Equation 4.4 into the expression for h_{eq} the new equilibrium distribution function becomes Equation 4.6

$$h_i^{eq} = \Delta\rho\omega_i + w_i(\Delta\rho + \rho_0)F_i(u) \tag{4.6}$$

By setting $F(u)$ equal to the corresponding part of the original equilibrium function (in our D3Q19 model this corresponds to $\frac{1}{c_s^2}\vec{e_i}\cdot\vec{u} + \frac{1}{2c_s^4}(\vec{e_i}\cdot\vec{u})^2 - \frac{1}{2c_s^2}\vec{u}\cdot\vec{u}$)

and solving for $Q_{i\alpha\beta u\alpha u\beta}$ the final version of h_i^{eq}, used in our simulation, becomes Equation 4.7

$$h_i^{eq} = \Delta\rho\omega_i + \omega_i(\Delta\rho + \rho_0)(\frac{1}{c_s^2}\vec{e_i} \cdot \vec{u} + (\vec{e_i} \cdot \vec{u})^2 - c_s^2\vec{u} \cdot \vec{u}) \tag{4.7}$$

By using this technique we were able to use single precision while retaining numerical precision. Saving memory and increasing the performance of the implementation.

In a naive implementation it is common to always store 19 particle distributions for each lattice whether it is a solid or fluid cell. Storing all values greatly simplifies the implementation by simplifying the mapping between domain positions and memory location of the particle distribution functions. However, the distribution functions in a solid cell are always zero, thereby making it unnecessary to store solid cells. By only storing the fluid cells we lose the simple mapping between lattice position and memory position, making an indirect addressing scheme necessary. However, since the porosity of rocks interesting to the petroleum industry is relatively small, usually around 10% - 15%, the gains of only storing fluid cells overcomes the added memory requirement due to the needed addressing scheme. The addressing scheme used in this thesis is based on the Sparse Lattice Representation (SLR) implementation by Pan *et al.* [34] and is described in more detail in Section 4.3.1.

Figure 4.2 shows the memory requirements of our solution and a naive one, when simulating a dataset with a porosity of 15%. In the figure we see that we are able to reduce the memory requirement by about 20 times.

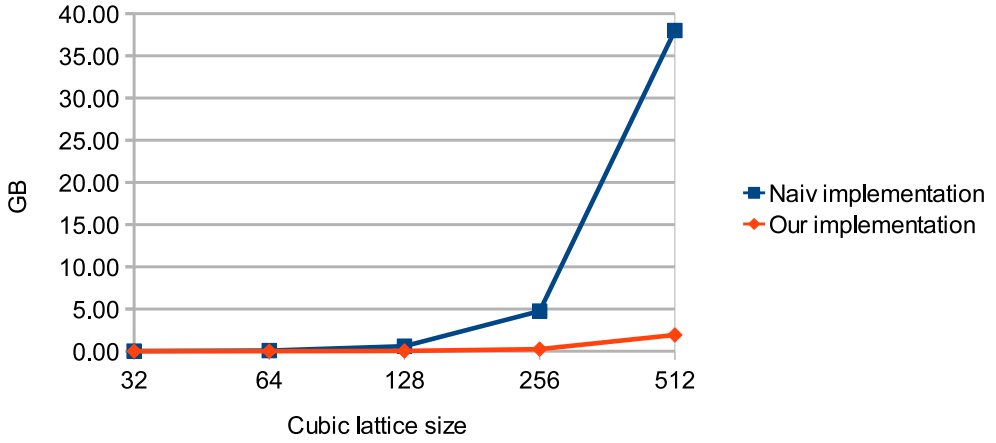


Figure 4.2: Comparing the memory requirement of our and a naive implementation of LBM with the D3Q19 model.

4.3 General implementation

In this section, we describe how we implemented the different phases of LBM. Figure 4.3 shows that we add an extra *Border exchange phase*, to the usual phases described in Section 3.2.1. All three implementations use the same OpenCL *Kernels*, which are described in this section. The main differences between the implementations is the way the GPUs is controlled and how the communication between the GPUs is handled, and this is described in the following sections. The code implemented for this thesis is based on the CUDA code created by Aksnes [3]. Therefore, some parts of the code have some similarities with the code used in his work.

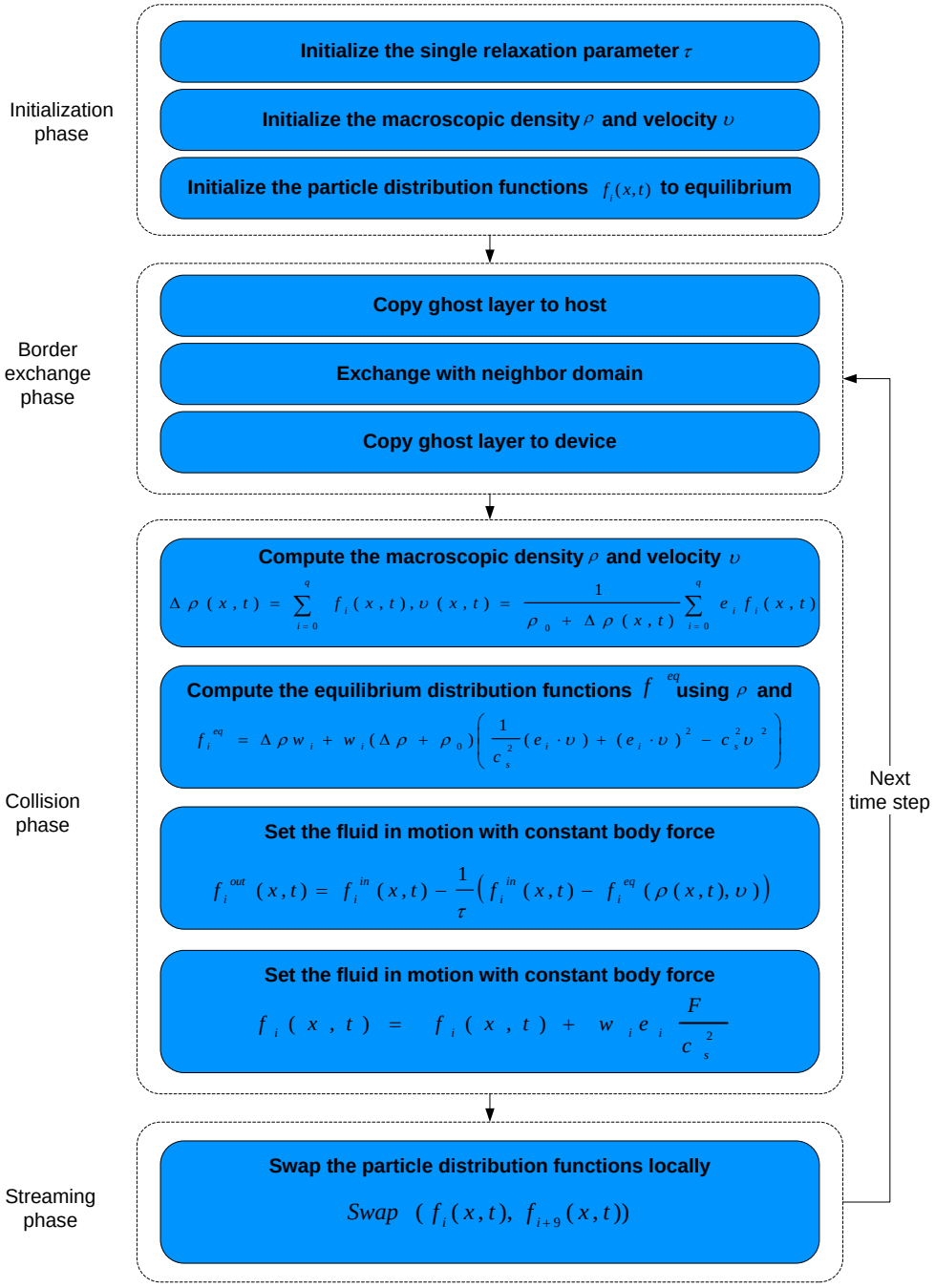


Figure 4.3: The main phases in our implementations

4.3.1 Initialisation phase

In the initialisation phase the domain (of lattice size $d_x \times d_y \times d_z$) is split into equal sub domains, each with $l_x \times l_y \times l_z$ lattice cells. As shown in Figure 4.4 we use a striped partitioning along the z axis, because this results in a simple decomposition and a sequential access pattern when transferring data to and from the device. All other possible decompositions, including a block partition, result in strided access patterns, increasing the time it takes to transfer data between the GPU and the host [36].

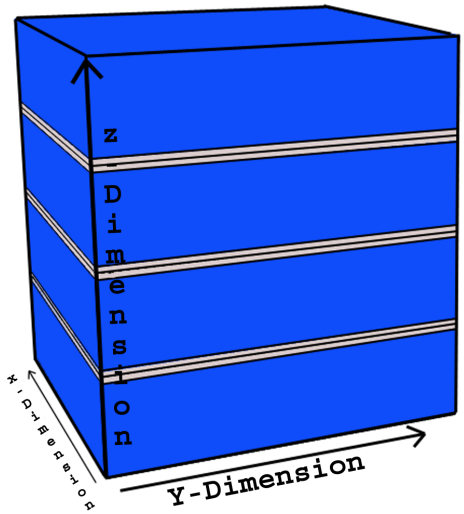


Figure 4.4: Domain decomposition used in our implementations. The gray layer in each sub domain is the ghost layer from the neighbors

As stated earlier, to save storage, only the fluid elements within the domain are stored on the GPU, making a mapping between lattice location and memory location necessary. This mapping is implemented as a three dimensional integer lookup table, (solid_d in the pseudo code for Algorithm 1).

The geometry information for each sub domain, including the ghost layers of the north and south neighbors, is used by the host to calculate the number of fluid elements and create the lookup table for each sub domain. As shown in Algorithm 1, the host iterates through the domain, including ghost layers, setting the lookup value of each cell. The lookup value of each solid cell is set to -1, while the value of a fluid cell is set to the next number in a sequence. For storing the particle distribution functions we use a structure-of-array layout with 19 *Buffers*, one for each particle distribution function. By saving

the lookup table as an *Image* we are able to use the caching capabilities usually implemented on reads and writes of *Images*.

Algorithm 1 Pseudo code of the SLR lookup table.

```
1:  fluidIndex ← 0
2:  antFluids ← 0
3:  index ← 0
4:  for z = 0 to l_z do
5:    for z = 0 to l_y + ghostLayersNorth + ghostLayersSouth do
6:      for x = 0 to l_x do
7:        if dataset[index] is fluid then
8:          solid_d[index] ← fluidIndex
9:          fluidIndex ← fluidIndex + 1
10:         antFluids ← antFluids + 1
11:       else
12:         solid_d[index] ← −1
13:       end if
14:       index + +
15:     end for
16:   end for
17: end for
```

4.3.2 Collision phase

Pseudo code for our implementation of the collision phase is shown in Algorithm 2. First the density ρ and the velocity $\vec{u}$ are calculated (Algorithm 2: Line 7-15). Then these two values are used to calculate the equilibrium function (Algorithm 2: Line 17), and the particle distribution function is relaxed against the equilibrium (Algorithm 2: Line 18). Finally, the particle distribution functions are locally swapped inside the cell (Algorithm 2: Line 22).

The flow in the simulation is created by applying an external force $\vec{F}$ to the fluid. The force is constructed to produce the same momentum as a true pressure gradient $\frac{\Delta P}{L}$, which is the total pressure drop along the sample length L. To achieve this a constant value is added to each particle distribution moving along the pressure gradient and subtracted from the particle distribution in the opposite direction.

There are several methods for incorporating the external force into the particle distribution. The most commonly used method is described by Guo *et*

al. [15] where the value added to the distribution functions (Algorithm 2: Line 19) is expressed as Equation 4.8.

$$F_i = w_i \vec{e_i} \cdot \frac{\vec{F}}{c_s^2} \tag{4.8}$$

where $\vec{F}$ is the external force, e_i is the direction of the i particle distribution and w_i is the weighting of the i particle distribution. The method described above is the one used in this thesis.

The rate of change towards the equilibrium state is controlled by the relaxation parameter τ. Due to stability requirements in the simulation, the relaxation parameter is chosen between $0.51 \leq \tau \leq 2.5$ [19].

From Algorithm 2 we see that the computation done on every lattice cell is independent and therefore can be done in parallel. The collision phase is implemented as a OpenCL *kernel*. The *kernel* is invoked in a 3D *NDRange* with the same dimension as the sub domain, including the ghost layers from the neighbors, each *work-item* performed the collision phase for a single cell.

On GPUs the global memory has a much higher latency and lower bandwidth than on-chip memory, so a typical programming pattern is to minimize the number of global memory accesses by loading data from global memory into the local memory [31]. In our implementations each *work-item* loads the 19 particle distribution functions into local memory before performing any calculations. To use the maximum amount of memory bandwidth it is important to get coalesced memory accesses, so the hardware is able to serve multiple work-items in a single memory transaction. Coalesced memory accesses are obtained differently on different hardware, but generally coalescing is obtained by letting successive work-items access data sequentially. Since we only store the fluid elements, the memory access pattern is directly dependent on the geometry of the dataset being simulated. Only storing fluid elements makes it difficult, if not impossible, to design a memory layout giving coalesced memory access in every situation. Non-coalesced memory access decreases the performance, but since the goal of this thesis is to simulate large datasets, the increased maximum size gained with only storing fluid elements is valued higher than the decrease in performance.

The large amount of local memory used per *work-item* severely limits the number of *work-items* that can be scheduled per compute unit, lowering the performance. Because the latency of local memory is orders of magnitude smaller than the latency of global memory, the reduction in global memory accesses makes this trade off profitable. By reversing the order in which the

Algorithm 2 Pseudo code of the collision phase.

1: **for** $z = 0$ to d_z **do**
2: **for** $z = 0$ to d_y **do**
3: **for** $x = 0$ to d_x **do**
4: $solidIndex \leftarrow x + y \times d_x + z \times d_x \times d_y$
5: $current \leftarrow solids[solidIndex]$
6: **if** $current \neq -1$ **then**
7: **for** $i = 0$ to 18 **do**
8: $\rho \leftarrow \rho + f_i[current]$
9: $u_x \leftarrow u_x + f_i[current] \times e_{ix}$
10: $u_y \leftarrow u_y + f_i[current] \times e_{iy}$
11: $u_z \leftarrow u_z + f_i[current] \times e_{iz}$
12: **end for**
13: $u_x \leftarrow \frac{u_x}{\rho}$
14: $u_y \leftarrow \frac{u_y}{\rho}$
15: $u_z \leftarrow \frac{u_z}{\rho}$
16: **for** $i = 0$ to 18 **do**
17: $f_i^{eq} \leftarrow w_i \times \rho \times (1.0 + 3.0 \times (e_{ix} \times u_x + e_{iy} \times u_y + e_{iz} \times u_z) + 4.5 \times (e_{ix} \times u_x + e_{iy} \times u_y + e_{iz} \times u_z)^2 - 1.5 \times (u_x \times u_x + u_y \times u_y + u_z \times u_z))$
18: $f_i[current] \leftarrow f_i[current] - \frac{1}{\tau} \times (f_i[current] - f_i^{eq})$
19: $f_i[current] \leftarrow f_i[current] + w_i \vec{e_{ix}} \times \frac{F_x}{c_s^2} + w_i \vec{e_{ix}} \times \frac{F_x}{c_s^2} + w_i \vec{e_{ix}} \times \frac{F_x}{c_s^2}$
20: **end for**
21: **for** $i = 1$ to 9 **do**
22: $swap(f_i[current], f_{i+9}[current])$
23: **end for**
24: **end if**
25: **end for**
26: **end for**
27: **end for**

particle distribution functions are copied back from local to global memory, we were able to implement the local swapping without extra memory traffic.

4.3.3 Streaming phase

In the streaming phase the movement of the fluid is created. The movement is achieved by letting each lattice cell interact with its neighbors swapping particle distribution functions (Algorithm 3: Line 9). As shown in Algorithm 4 the periodic boundary condition is implemented into the `getNeighbor` function by either allowing or disallowing interaction with the opposite side. The triple for loops in Algorithm 3 are completely independent of each other and can be done in any order. The streaming phase is implemented as a *kernel* on the GPU that is invoked in the same 3D *NDRange* as the collision phase.

Algorithm 3 Pseudo code of the streaming phase.

```
1: for z = 0 to d_z do
2:   for z = 0 to d_y do
3:     for x = 0 to d_x do
4:       thisIndex ← x + y × d_x + z × d_x × d_y
5:       if solids[thisIndex] ≠ −1 then
6:         for i = 1 to 9 do
7:           neighborIndex ← getNeighbor(x, y, z, i, d_x, d_y, d_z, p_x, p_y, p_z)
8:           if solids[neighborIndex] ≠ −1 then
9:             swap(f_{i+9}[thisIndex], f_i[neighborIndex])
10:          end if
11:        end for
12:      end if
13:    end for
14:  end for
15: end for
```

Algorithm 4 Pseudo code of the getNeighbor function.

1: $getNeighbor(x, y, z, i, d_x, d_y, d_z, p_x, p_y, p_z)$
2: $neghbour_x \leftarrow x + e_{ix}$
3: $neghbour_y \leftarrow y + e_{iy}$
4: $neghbour_z \leftarrow z + e_{iz}$
5: **if** p_x **then**
6: **if** $neghbour_x = -1$ **then**
7: $neghbour_x \leftarrow d_x - 1$
8: **end if**
9: **if** $neghbour_x = d_x$ **then**
10: $neghbour_x \leftarrow 0$
11: **end if**
12: **end if**
13: **if** p_y **then**
14: **if** $neghbour_y = -1$ **then**
15: $neghbour_y \leftarrow d_y - 1$
16: **end if**
17: **if** $neghbour_y = d_y$ **then**
18: $neghbour_y \leftarrow 0$
19: **end if**
20: **end if**
21: **if** p_z **then**
22: **if** $neghbour_z = -1$ **then**
23: $neghbour_z \leftarrow d_z - 1$
24: **end if**
25: **if** $neghbour_z = d_z$ **then**
26: $neghbour_z \leftarrow 0$
27: **end if**
28: **end if**
29: **return** $neghbour_z \times d_y \times d_x + neghbour_y \times d_x + neghbour_x$

4.3.4 Border exchange phase

Since the domain is split across multiple GPUs, possibly located on different physical hosts, border information needs to be exchanged between each iteration of the solver. Since GPUs can not directly communicate with each other, this exchange is handled by the hosts connected to the GPUs. Since the PCI bus has a relative high latency it is important to decrease the number of transfers. To decrease the number of transfers the exchange phase has been implemented in five steps: In the first step the 19 particle distribution functions for the elements designated to be sent to the north and south neighbor are copied, in a interleaved sequential way, into two temporary *buffers*, as shown in Figure 4.5. The step is implemented as a simple kernel launched twice. One invocation containing as many *work-items* as there are fluid elements that are transferred to the north neighbor, and the other invocation with as many *work-items* as the are elements to send to the south neighbor. Each *work-item* then loops through the 19 particle distributions and copies the values into the appropriate buffer. In the second step the two buffers are transferred to the host. In the third step the data are sent to the respective neighbors. When the transfer finishes the first two steps are preformed in the reverse order, constituting steps four and five of the exchange phase.

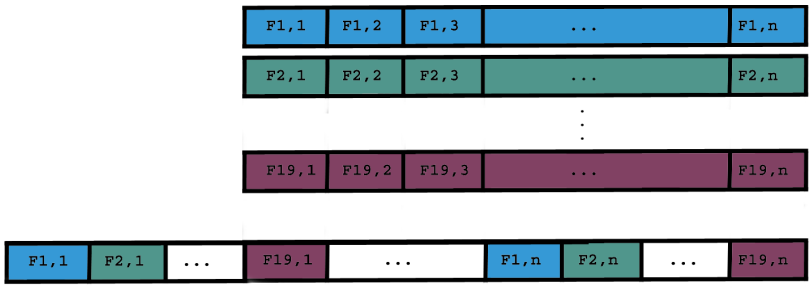


Figure 4.5: The interleaved sequential copy pattern used in the border exchange phase

4.3.5 Boundary Conditions

In addition to the periodic boundary condition implemented in the streaming phase we have implemented a bounce back boundary condition to handle interaction between fluid and solid cells. The bounce back boundary condition, as shown in Figure 3.4, is partly implemented in the collision phase and partly in the streaming phase. At the end of the collision step each particle

distribution, except the center one (particle distribution 0), is swapped with the particle distribution in the opposite direction. The streaming step only swaps particle distribution functions if both cells are fluid. A particle distribution going into a solid cell end up pointing in the opposite direction after the locale swapping at the end of the collision phase.

4.3.6 Calculating Permeability

The approach used to calculate the permeability of porous rocks in this thesis is taken from Guodong *et al.* [19]. The permeability of the porous rock is found by applying Darcy's law to the flow obtained from the lattice Boltzmann simulation. In our simulation the flow is created by an external force used to simulate a pressure drop $\frac{\Delta P}{L}$ along the sample length L. By adapting Equation 3.14 to take into account this force, the permeability of a porous rock is calculated using Equation 4.9

$$k = -a^2 \frac{\vec{u}\phi\rho\nu}{F} \tag{4.9}$$

where a is the cell spacing in the lattice. It is common to add extra layers containing only fluid cells at both sides in the direction of the flow, so that the periodic boundary can be applied. It is also usual to add an extra layer of solid elements on the other four boundaries [19]. In this thesis, we have used the definition of porosity given in Equation 3.15, taking into account both connected and disconnected (percolating) pore space. To get a better estimate of the permeability it is possible to use the Hoshen-Kopelman algorithm to identify the percolating pore space [19]. In this thesis permeability is only used to validate that the implementations are correct, thus the values could be further improved by using only the percolating pore space.

4.3.7 Convergence Check

Before applying Darcy's law to find the permeability of a rock the system has to reach a steady state. Several different approaches can be used to determine if the system has reached a steady state. A common method is to require that the change in average fluid velocity is smaller than a certain value [19, 3].

When using the method used by Aksnes [3] we observed that the computed average fluid velocity still fluctuates after the system has reached a steady

state. These fluctuations were temporary and dependent on the geometry of the dataset. The fluctuations are probably a consequence of the accumulative error associated with the global summation over all the fluid nodes in the system. By requiring that the convergence is stable for a period of time it is possible to somewhat eliminate the effect of random fluctuation. Checking that the system is stable for a period of time has the unwanted effect of altering the computation and communication ratio, since the convergence has to be checked at short intervals, and the computation of the fluid average velocity requires communication between all GPUs in a multi-GPU system.

In this thesis we use a different approached in the convergence check. We check the change in the maximum difference between two particle distribution functions in the streaming phase, using Equation 4.10

$$1 - \frac{|maxMovement(t_1)|}{|maxMovement(t_2)|} < \epsilon \tag{4.10}$$

where $maxMovement(t_1)$ and $maxMovement(t_2)$ represent the maximum difference between two particle distributions swapped in the streaming step in two iterations, and ϵ is the convergence criteria given as an input parameter to the simulation. By this method we eliminate the fluctuations and are able to run many iterations between each convergence check, thus saving time since the convergence check involves both calculation and global communication.

4.4 Thread Implementation

The original plan for the thread implementation was to use POSIX threads to create as many threads as there are GPUs in the system, with each thread controlling a single GPU. Unfortunately, this was not feasible because of the fact that in the current stable implementation of OpenCl from NVIDIA (version 1.0) not all of the API calls are thread safe [21]. Thread safe versions of all functions are required in the latest version of the OpenCL spec (version 1.1) [22]. NVIDIA's current implementation of OpenCl 1.1 still is in beta, and therefore using multiple threads is not investigated in this thesis.

Both functions for enqueueing kernel execution, and reading/writing from the memory of the GPU can be made non-blocking, returning the control to the host code as soon as the commands are successfully added to the queue. By creating a set of *command queues*, one per GPU, the host is able to control each GPU individually. By first iterating through all the GPUs,

enqueueing the appropriate amount of work, we are able to let all GPUs do work simultaneously. Then a new loop calls the `finish` function on each queue, causing the host to wait for all the GPUs to finish the enqueued work. By using the non-blocking capabilities of OpenCL we are able to let the GPUs work in parallel, but still have a simple means of synchronization.

By using the in-order execution mode of OpenCL the host code is able to enqueue both the collision and the streaming step on every GPU, and rely on OpenCL to enforce the internal dependency within the two phases, maximizing the amount of time the GPUs work simultaneously. Since most of the work is done by the GPUs, the single thread is able to coordinate all of the GPUs without being the bottleneck. Tests showing that this works well are performed with up to four GPUs, the largest number of GPUs connected to a single host available to us.

Since the GPUs are connected to the same host the exchange of ghost layers is handled by `memcpy` between the different buffers. Because the amount of data in each ghost area is relatively small, the host to host communication time becomes insignificant, leaving only the host to GPU communication as the overhead. Since this involves both the GPU and the host we are not able to hide this overhead. In the exchange phase we once again rely on the in-order execution and enqueue the gathering and scattering of data and the transfer from host to device before waiting for all GPUs to finish.

4.5 MPI Implementation

The MPI implementation is intended for use on a cluster where each node contains one or more GPUs. Each GPU in the system is controlled by its own MPI process, so communication and synchronization between GPUs is handled by the hosts through MPI. The MPI implementation take advantage of the in-order execution mode of OpenCL to transfer as much work as possible to each GPU before waiting for it to finish. Since multiple processes can be executed on a single node we use processor affinity to pin each process to a single processor core, thereby eliminating the possibility of processes destroying each others cached data.

During the initialization phase the process with rank 0 is responsible for reading in the rock geometry. The data for each sub domain along with information about the global size of the rock are then distributed to the other ranks through MPI. By using the parallel I/O feature of MPI version

2.0, and letting all ranks participate in reading the rock geometry, eliminating the need to distribute the sub domains from rank 0, it would be possible to lower this startup time. since the focus of this thesis is on the core simulation this has not been looked into.

The communication pattern used in the border exchange phase is the same for each iteration making it possible to use the persistent handler functionality of MPI. By setting up the necessary handlers in the initialization phase, via the `MPI_Send_init` and `MPI_Recv_init` functions, and then reusing this handle for every iteration, we are able to eliminate some of the overhead associated with communication between the process and the communication controller.

As explained in Section 2.6 the total execution time of an application on a multi-GPU system can be modeled using Equation 2.3. In a pure MPI configuration at least one pair of GPUs communicates over the inter-node interconnect, making host to host communication the dominating part of $T_{comm}(n, p)$ in Equation 2.3. Since only the host is involved in the exchange we are able to hide some of the communication cost by starting the collide phase on the inner lattice locations, making the $T_{overlap}$ equal to the minimum of the time used by the GPU to compute the collide phase of the inner lattice locations and the time to communicate the ghost layers between the hosts.

4.6 Hybrid Implementation

The Hybrid implementation is a combination of the thread and the MPI implementation. It is intended used on distributed memory systems where each node has multiple GPUs. The idea is to eliminate the overhead associated with using MPI to communicate internally on a single node. One MPI process is executed on each node handling the inter node communication, while internally on a node the control and communication is handled as in the thread implementation. This eliminates the MPI overhead between GPUs internally inside the node, but also increases the complexity of the implementation.

Chapter 5

Results and Discussion

In this chapter we present and discuss the validation and performance measurements done in this thesis. In section 5.1 we give a presentation of the test environments and test methodology used. Section 5.2 gives a description of how the results from the different implementations have been checked for correctness against the velocity profile of the known Poiseuille flow and datasets with known permeability. Section 5.2.2 presents and discusses the the performance measurements from different system configurations. The chapter ends with a section comparing the performance obtained with the three implementations.

5.1 Test Environments and Methodology

Table 5.1 gives the specifications of the three different types of GPU cards used to test the three implementations of LBM described in the previous chapter. During the testing we used two NVIDIA Tesla C2070 cards, two NVIDIA Tesla C1060 cards and four Tesla T10 cards. The four T10s were in form of a Tesla S1070 computing system connected to a single host. The two Tesla C2070 cards and the two Tesla C1060 cards were configured as a multi-GPU cluster with 2 nodes and 2 GPUs in each node. A more detailed description of the supporting hardware and connections can be found in Appendix A. The nodes in the cluster were connected to each other via a 1GB Ethernet switch. As can be seen from Table 5.1 we have only used NVIDIA based cards during the performance testing. The development has been done on a AMD ATI Radeon HD5870, but as we only had access to one ATI card this card has not been used during the performance testing.

Table 5.1: Specifications of the three GPU types used

GPU	**Tesla 2070**	**Tesla 1060**	**Tesla T10**
# Cores	448	240	240
Clock frequency	1.5 GHz	1.3 GHz	1.3 GHz
Memory	6 GB	4 GB	4 GB
Memory bandwidth	144 GB/s	102 GB/s	102 GB/s
Shared memory	48 KB	16 KB	16 KB

To validate our implementations and investigate how to utilize multiple GPUs in a LBM simulation five datasets provided by Numerical Rocks AS were used. The technical data for each of the datasets is presented in Table 5.2. The column *Memory usage* shows the amount of memory needed to store the datasets without ghost layers. From the table we see that all datasets, except Grid800 fit in the memory of a single GPU.

As explained in Section 4.3.2, the memory access pattern and performance depends on the geometry of the dataset. Therefore, to show how the implementations preform on realistic porous rocks, the three biggest datasets, Fontainebleau, Grid500 and Grid800 were used during the performance measurements. Fontainebleau, Symetrical Cube and Square Tube have known permeabilities and were used to validate the correctness of the implementations.

Table 5.2: Technical data about the datasets used

Name	**Lattice size**	**Porosity**	**Known permeability**	**Memory usage**
Symetrical Cube	$80 \times 80 \times 80$	16%	22 mD	0.01 GB
Square Tube	$200 \times 100 \times 100$	81%	216 D	0.13 GB
Fontaine-bleau	$300 \times 300 \times 300$	16%	1300 mD	0.42 GB
Grid500	$500 \times 500 \times 500$	15%	not known	1.86 GB
Grid800	$800 \times 600 \times 900$	15%	not known	6.44 GB

As described in Section 3.3, two commonly used metrics for measuring the performance of a LBM implementation are the Million Lattice Updates Per Second (MLUPS) and the Million Fluid Lattice Updates Per Second (MFLUPS). Since only fluid elements require computation in our implementations we report the MFLUPS, as this gives a better indication of the utilization of the

GPU. In this thesis MFLUPS is calculated using Equation 5.1

$$MFLUPS = \frac{n \times m}{t_{streaming} + t_{collision} + t_{border\ exchange}} \tag{5.1}$$

where n is the number of fluid nodes in millions, m is the number of iterations, and $t_{streaming}$, $t_{collision}$ and $t_{border\ exchange}$ is the time in seconds spent in the streaming, collision and border exchange phases.

To get a high resolution measurement of the time spent in each phase we used *cl_events* [21] to time the execution of the `collision` and `streaming` kernels. To time the border exchange phase we use a combination of *cl_events* and `gettimeofday`. *cl_events* was used to measure the execution time of the distribution/reduction kernels and the time to transfer data to and from GPU (see section 4.3.4) we used the `gettimeofday` function to measure the time spent sending data from host to host. The total simulation time is determined by the GPU finishing last, therefore all reported timings are taken form this GPU. The reported timings are the arithmetic average of 500 iterations.

5.2 Validation

The numerical correctness of our three implementations, thread, MPI, and hybrid were validated in two ways. First, the simulated velocity field from the Poiseuille flow was compared with the known analytically calculated velocity field. Second, the obtained permeability for each dataset was compared with the known permeability.

5.2.1 Validation Against Poiseuille Flow

One flow that it is possible to calculate numerically via the Navier-Stokes equations is the Poiseuille flow, which is the flow between two parallel plates with a distance of $2L$ between each other.

Figure 5.1 shows an example of a Poiseuille flow with a lattice dimension of 32^3.

The velocity profile of the Poiseuille flow can be calculated using Equation 5.2 [42, 9]

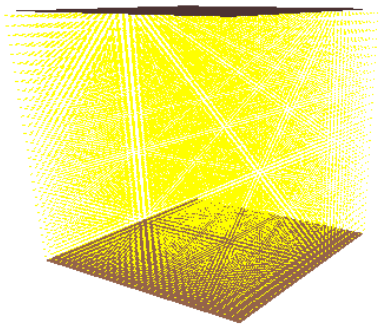


Figure 5.1: Fluid flow between two parallel plates

Table 5.3: Parameter values used during the Poiseuille flow simulations

Parameter	**Value**
τ	0.63
F_x	0.00001
F_y	0.0
F_z	0.0
Dimension	32^3
Convergence	$2.5e^{-05}$

$$u_x(y) = \frac{F}{2\nu}(L^2 - y^2) \tag{5.2}$$

where F is a constant external force working on the fluid, ν is the kinematic viscosity of the fluid and L is half the distance between the two plates.

During the simulation of the Poiseuille flow both types of boundary conditions were used: the solid elements in the plates (top an bottom) implemented the complete bounce back boundary condition, whereas the other dimensions (the sides) were periodic. The other parameters used during the simulation are given in Table 5.3.

To validate that the border exchange phase was correctly implemented in all our implementations, we ran each implementation using four GPUs. Figure 5.2 compares the simulated velocity profile with the known analytical solution. From the figure we see that all three implementations compute an identical velocity profile. The flow is simulated (squares in figures) with a deviation of $3.55e^{-05}$ compared to the known analytical solution (line in figures). As explained in Section 4.3 all three implementations use the same *Kernels*, and, therefore, produce the same numerical results during simula-

tion, if the exchange of data between GPUs is correctly implemented.

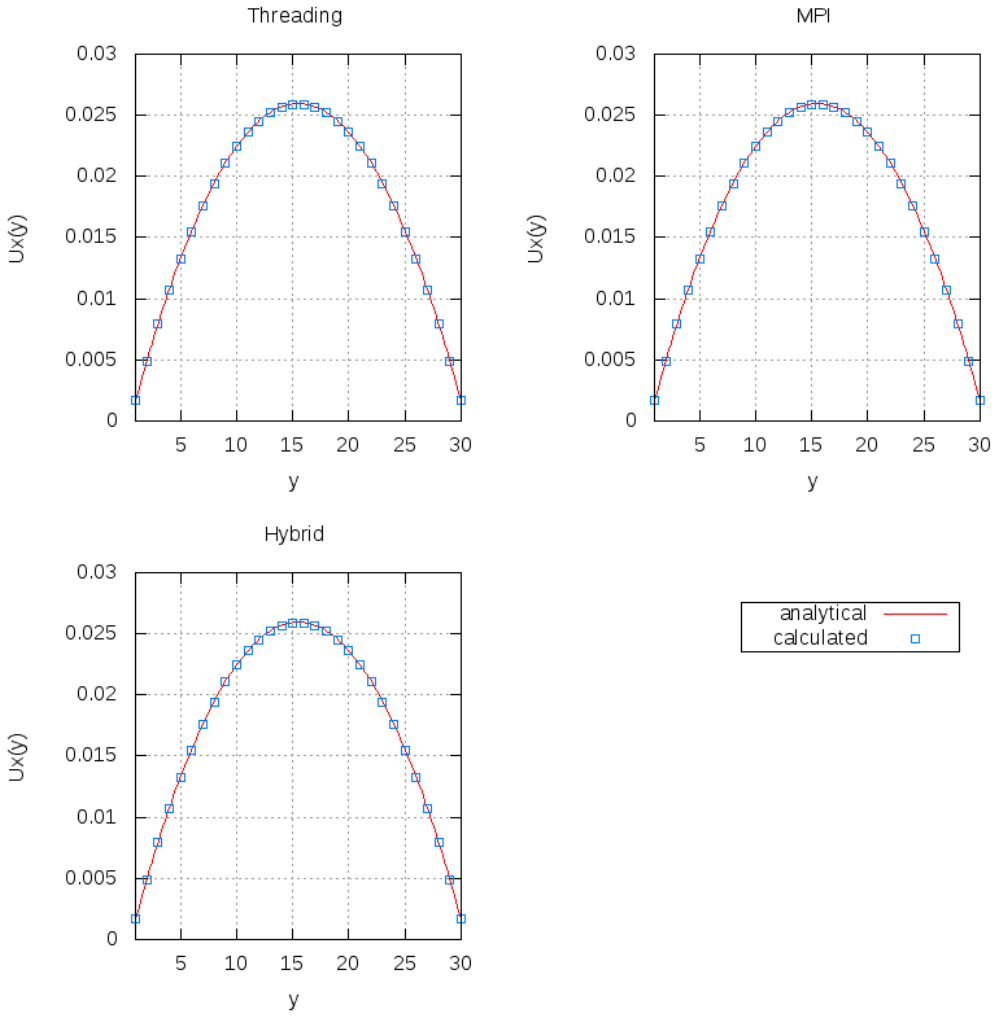


Figure 5.2: Comparison of known and simulated velocity profile for the Poiseuille flow using the three implementations

5.2.2 Porous Rock Measurements

In order to validate that our implementations calculate the permeability of porous rock correctly, the three datasets with known permeability were simulated. During the simulations five extra layers of fluid elements were added to the inlet and outlet, and one extra layer of solid elements was added to

the other four sides. Table 5.4 gives the value of the relaxation parameter, force and convergence criterion used during the simulations.

Table 5.4: Parameter values used during the porous rocks measurements

Parameter	**Value**
τ	$\frac{1}{1.5}$
F_x	0.00001
F_y	0.0
F_z	0.0
Convergence	$2.5e^{-3}$

Table 5.5 shows the known and simulated permeability values along with the deviation from the known permeability. From the table we see that the deviation is at most 7.8% between simulated and known permeability.

Table 5.5: Known and calculated permeability for Symetrical Cube, Square Tube and Fontainbleau

Name	**Known permeability**	**Obtained permeability**	**Deviation**
Symetrical Cube	22 mD	23.72 mD	7.8%
Square Tube	216 D	203.55 D	5.8%
Fontainebleau	1300 mD	1253.03 mD	3.6%

5.3 Performance Measurements

In this section we present the performance of simulating oil flow for the three largest datasets using the three implementations described in the previous chapter. The tests were run on the hardware described in Section 5.1.

5.3.1 Single GPU

To get a baseline to compare the performance of our three multi-GPU implementations against, we ran the kernels for the collision and streaming phase on each of the three card types presented in Table 5.1. Tables 5.6, 5.7 and 5.8 show the amount of time spent in each phase during a single iteration on each card. Since the Grid800 dataset requires 6.43 GB of memory, it was

too big to fit on a single GPU and we therefore do not have a single GPU baseline for it.

Table 5.6: Time consumption in ms for the different phases of simulation on a single NVIDIA Tesla C2070

	Fontainebleau	**Grid500**
Collide	46 (65.06%)	210 (63.81%)
Streaming	25 (34.94%)	119 (36.19%)
Total	**71 (100.00%)**	**330 (100.00%)**
MFLUPS	**62.94**	**58.22**

Table 5.7: Time consumption in ms for the different phases of simulation on a single NVIDIA Tesla C1060

	Fontainebleau	**Grid500**
Collide	98 (65.06%)	445 (67.26%)
Streaming	52 (34.94%)	217 (32.74%)
Total	**150 (100.00%)**	**662 (100.00%)**
MFLUPS	**29.95**	**28.99**

Table 5.8: Time consumption in ms for the different phases of simulation on a single NVIDIA Tesla T10

	Fontainebleau	**Grid500**
Collide	103 (66.79%)	465 (68.89%)
Streaming	51 (33.21%)	210 (31.11%)
Total	**154 (100.00%)**	**676 (100.00%)**
MFLUPS	**29.14**	**28.41**

Based on the cards' specifications we expected the Tesla C2070 should have the best performance and that the Tesla 1060 and Tesla T10 should performe similarly to each other. As shown in Tables 5.6, 5.7 and 5.8 this is the case, with the performance of the C2070 more than double of the two other cards. The tables also show that the T10 and C1060 have similar performance.

The specifications for the cards in Table 5.1 provides the explanation for this better performance. The Tesla C2070 has almost double the amount of cores

and three times the amount of shared memory compared to the other two. As explained in Section 4.3.2, the implementation of the collision phase uses a lot of local memory, limiting the number of work-groups per compute unit. The extra local memory available on the Tesla C2070 increased the number of work-groups per CU from 4 to 8, increasing occupancy from 25% to 33%, when simulating the Grid500 dataset. The calculation of work-groups per CU and occupancy were done using the NVIDIA Occupancy calculator. The differences in GPU performance creates load balancing issues when different GPUs are used together in the same system as shown in Section 5.3.3.

Table 5.7 shows that we are able to achieve a maximum of 29.85 MFLUPS when using a Tesla C1060, falling short of the 300 MFLUPS reported by Feichtinger *et al.* [12]. The difference in performance can be attributed to two factors: First, the performance of OpenCL kernel is less than the equivalent CUDA kernel when executed on NVIDIA cards [20]. Second, as described in section 4.3.2, we only store fluid elements making it impossible to achieve coalesced memory access which is a key factor in obtaining the high performance reported by Feichtinger *et al.* [12]. The main focus of this thesis is to investigate how to simulate large realistic domains. By only storing fluid elements we are able to lower the memory requirement, increasing the domain size possible to simulate on a single GPU.

5.3.2 Thread Implementation

As described in Section 4.4 the thread implementation uses one host thread to control all GPUs connected to a single host. Tables 5.9, 5.10, 5.11 and 5.12 show the timings for a single iteration of the three datasets using four configurations: two Tesla C2070s (compute node 2 in the cluster), two Tesla 1060s (compute node 1 in the cluster), two and four of the GPUs in the Tesla 1070 system. As described in Section 2.7 the GPUs in the Tesla 1070 pairwise share a PCI connection to the host. To ensure maximum performance we used GPUs from different connections when using only two GPUs.

Table 5.9: Time consumption in ms for the different phases of simulation on a shard memory system with 2 Tesla C2070

	Fontainebleau	**Grid500**	**Grid800**
Collide inner	23 (62.15%)	106 (62.76%)	314 (47.63%)
Collide border	0 (0.50%)	0 (0.27%)	1 (0.11%)
Streaming	13 (33.60%)	60 (35.48%)	340 (51.67%)
Communication	1 (3.75%)	2 (1.48%)	4 (0.59%)
Total	**38 (100.00%)**	**166 (100.00%)**	**659 (100.00%)**
MFLUPS	**119.43**	**113.84**	**99.35**

Table 5.10: Time consumption in ms for the different phases of simulation on a shard memory system with 2 Tesla 1060

	Fontainebleau	**Grid500**	**Grid800**
Collide inner	49 (62.51%)	230 (66.77%)	868 (69.93%)
Collide border	0 (0.44%)	1 (0.27%)	2 (0.15%)
Streaming	27 (34.73%)	110 (31.91%)	365 (29.42%)
Communication	2 (2.32%)	4 (1.04%)	6 (0.50%)
Total	**78 (100.00%)**	**345 (100.00%)**	**1241 (100.00%)**
MFLUPS	**57.51**	**56.65**	**52.75**

Table 5.11: Time consumption in ms for the different phases of simulation using two of the GPUs in the NVIDIA S1070

	Fontainebleau	**Grid500**	**Grid800**
Collide inner	51 (63.92%)	241 (67.95%)	891 (71.22%)
Collide border	0 (0.46%)	1 (0.28%)	2 (0.16%)
Streaming	27 (33.19%)	110 (30.95%)	353 (28.23%)
Communication	2 (2.42%)	3 (0.82%)	5 (0.39%)
Total	**80 (100.00%)**	**355 (100.00%)**	**1251 (100.00%)**
MFLUPS	**55.92**	**54.12**	**52.31**

Table 5.12: Time consumption in ms for the different phases of simulation using four GPUs on the NVIDIA S1070

	Fontainebleau	**Grid500**	**Grid800**
Collide inner	31 (60.09%)	118 (65.79%)	453 (66.41%)
Collide border	1 (1.69%)	2 (1.08%)	4 (0.59%)
Streaming	17 (32.55%)	53 (29.74%)	198 (29.07%)
Communication	3 (5.67%)	6 (3.39%)	27 (3.93%)
Total	**52 (100.00%)**	**180 (100.00%)**	**683 (100.00%)**
MFLUPS	**85.85**	**106.79**	**95.85**

From Tables 5.9 - 5.12 we see that GPU to GPU communication is less significant, consuming less than 4% (Table 5.9) of the total simulation time when using two GPUs and less than 6% (Table 5.12) when using four GPUs. We also see that the percentage of the time used in communication reduces as the dataset increases, becoming less than 1% when using two cards in the simulation of Grid800.

By comparing the time spent in communication when using two and four of the GPUs on the Tesla S1070 we see the effect of GPUs sharing connection to the host. When the load on the connection is small (Fontainebleau and Grid500) the communication time doubles, as expected, when we use four instead of two GPUs. When the load gets large (Grid800) the communication time increase by a factor of more than five. A doubling of the communication time is expected as two of the GPUs in a four GPU configuration would have two neighbors. The simulation has also been performed using two GPUs from the same connection, shown in Appendix B, giving execution times almost identical to the ones using two GPUs from different connection. It appears that as the load gets large the performance of the PCI express bus is strongly reduced. We can not see any obvious reason for this, but the same findings have been reported by Spampinato [36].

Figure 5.3 shows the speedup of using two GPUs as compared to one. From the figure we see that we achieve near linear speedup, with a maximum of 1.95 when using two Tesla 2070s in the simulation of Grid500. It cal also be noted that the speedup of using the Tesla C2070s is smaller than the two others for Fontainebleau. Even though the C2070 has higher performance than the two others, they all use the same bus between host and device, thus the percentage of time spent in communication for the Tesla C2070s is greater resulting in a smaller speedup.

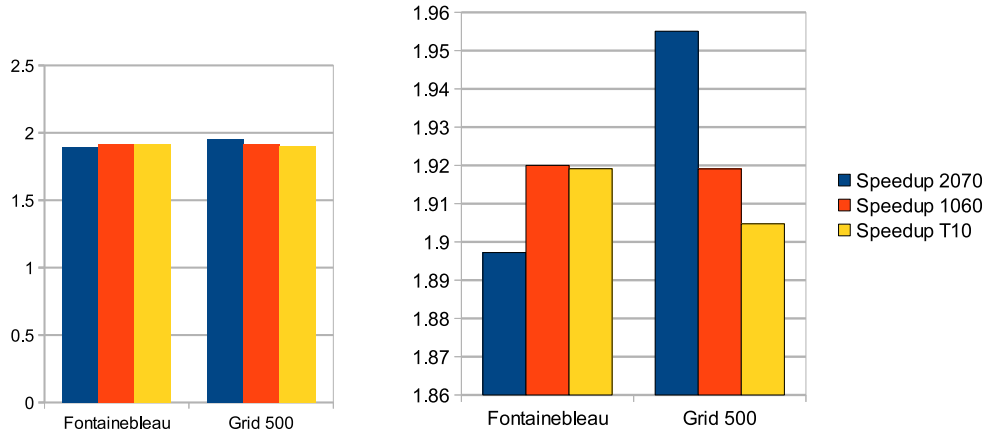


Figure 5.3: Speedup of using two Tesla C2070, Tesla C1060 and T10 compared to one

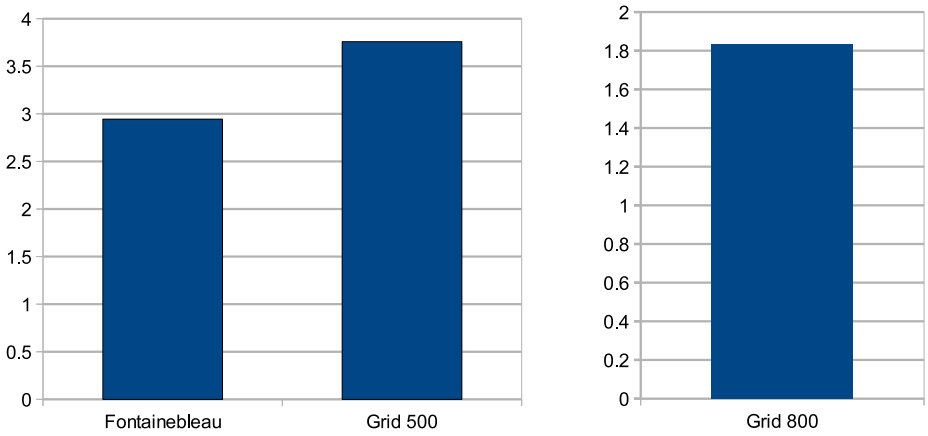
Figure 5.4 shows the speedup of using four GPUs in the simulations compared to one. Since the Grid800 dataset does not fit on a single GPU the speedup for this dataset is compared to using two GPUs. As shown in the figure we achieve a speedup of 2.94, 3.75 and 1.83 when simulating the Fontainebleau, Grid500 and Grid800 datasets respectively.

5.3.3 MPI Implementation

In the MPI implementation each GPU is controlled by its own MPI process. Tables 5.13 and 5.14 show the timings of a single iteration of the simulation for the three datasets using two different two node configurations: one using all four of the GPUs in the system and one using only one GPU on each node.

As explained in Section 4.5 we are able to overlap some of the host to host communication with computation by starting the next iteration while the communication is taking place. Both tables show that the total simulation time is dominated by the communication, using up to 95.96% (87.85% when excluding the overlap) of the run time for four GPUs to simulate the Fontainebleau dataset.

Another point to notice is that as the dataset increases, the percentage of time spent in communication decreases. As the dataset increases we are,



(a) Speedup of using four T10 compared to one

(b) Speedup of using four T10 compared to two

Figure 5.4: Speedup of using four T10 compared to one and two

therefore able to hide more of the communication with computation to the point where we are able to completely hide the communication when using two GPUs for the simulation of Grid800.

By looking at the detailed timings for the simulations (Appendix B), we observe two things. First, in the simulation using two GPUs we see that the Tesla C2070 always uses significantly less time to complete the iteration. As every GPU in the system has to complete each iteration before any GPU can continue the Tesla C2070 spends large amounts of time idle. Second, when using two GPUs on a single node we see that as one of the GPUs only communicate internally inside the node, this GPU uses less time on communication, and therefore, completes earlier resulting in idle time for this GPU. Information about the underlying hardware, obtainable through the OpenCL API, could be used to give the cards different amount of work resulting in a better load balancing. Because of time constraints this has not been looked into, and is left for further work.

Table 5.13: Timings in ms on the cluster using all 4 GPUs (2 x Tesla 2070 on one node and 2 x Tesla 1060 on one)

	Fontainebleau	**Grid500**	**Grid800**
Collide inner	15 (8.92%)	113 (27.27%)	440 (51.68%)
Collide border	0 (0.27%)	2 (0.44%)	4 (0.46%)
Streaming	7 (4.04%)	55 (13.23%)	199 (23.33%)
Communication	159 (95.96%)	368 (86.33%)	649 (76.21%)
Overlap	15 (−8.92%)	112 (−27.27%)	440 (−51.68%)
Total	**166 (100.00%)**	**413 (100.00%)**	**851 (100.00%)**
MFLUPS	**27.52**	**46.51**	**76.88**

Table 5.14: Timings on a GPU cluster using 2 GPUs (Tesla 2070 on one node and Tesla 1060 in one)

	Fontainebleau	**Grid500**	**Grid800**
Collide inner	49 (30.93%)	222 (48.47%)	855 (69.96%)
Collide border	0 (0.22%)	1 (0.20%)	2 (0.16%)
Streaming	27 (17.19%)	106 (23.20%)	365 (29.88%)
Communication	130 (82.59%)	351 (76.60%)	625 (51.17%)
Overlap	49 (−30.93%)	222 (−48.47%)	625 (−51.17%)
Total	**158 (100.00%)**	**458 (100.00%)**	**1222 (100.00%)**
MFLUPS	**28.46**	**41.89**	**53.57**

5.3.4 Hybrid Implementation

As described in Section 4.6 the hybrid implementation is intended to be executed on clusters with more than one GPU per node. Table 5.15 shows the timings from a single iteration of simulation for three datasets when using all four GPUs in the cluster.

By comparing the communication steps in Table 5.13 and Table 5.15 we see that the hybrid implementation eliminates the overhead introduced by using MPI to communicate between GPUs internally on one node. By using threaded GPU to GPU communication within nodes communication time is reduced from 649 ms to 622 ms (Grid800). The hybrid implementation is more complex than using only MPI, and hence one should consider carefully whether the improved performance is worth the extra programming effort.

Table 5.15: Timings of using the hybrid implementation with 2 nodes and 4 GPUs

	Fontainebleau	**Grid500**	**Grid800**
Collide inner	30 (20.03%)	112 (27.20%)	433 (52.86%)
Collide border	1 (0.56%)	2 (0.44%)	4 (0.47%)
Streaming	17 (11.26%)	55 (13.30%)	194 (23.72%)
Communication	133 (88.18%)	354 (86.26%)	622 (75.81%)
Overlap	30 (−20.03%)	112 (−27.20%)	433 (−52.86%)
Total	**151 (100.00%)**	**411 (100.00%)**	**820 (100.00%)**
MFLUPS	**29.72**	**46.75**	**79.82**

5.4 Discussion

In the previous section we presented the timings for the three different approaches to multi-GPU LBM simulation. In this section we discuss some of the advantages and disadvantages of the different approaches.

As shown in Section 5.3.4 the hybrid implementation have almost identical execution time as MPI. Hybrid only eliminates the overhead of using MPI between GPUs internally on one node, resulting in a small improvement of the communication time. The result of eliminating the overhead was smaller than first expected. As the hybrid implementation is more complex and gives only minor improvements over MPI, the hybrid will therefore be eliminated from further discussion.

The results show that the MPI and thread implementations have different performance characteristics, and therefore are suited for different scenarios. MPI is generally best suited for large datasets while thread implementation is best suited when the dataset fit the GPUs connected to one host.

Section 5.3.2 shows that the thread implementation achieves almost linear speedup up to four GPUs. The result of using four GPUs over two PCI connections show that it is important to have enough bandwidth between the host and GPUs, ideally a dedicated PCI slot to each GPU. Since the thread implementation depends on a single address space there is a physical limit on how many PCI express slots possible to connect to such systems. This limits the domain size possible to solve, but as long as the dataset fits in the memory of the GPUs possible to connect to a single host this is the preferred way.

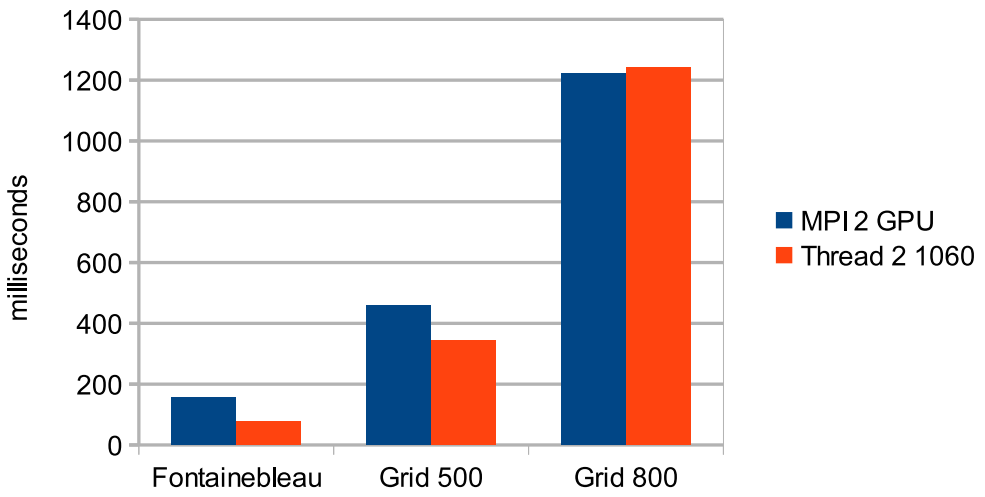


Figure 5.5: Single iteration time of the MPI and thread implementations using 2 GPUs

Figure 5.5 shows the total execution time of one iteration when using two GPUs in the MPI and thread implementation. From the figure we see that as the dataset increases the difference in execution time becomes smaller, resulting in the same execution time when simulating the Grid800 dataset. This indicates that as long as each GPU is given enough work the potential for overlapping can be exploited to completely hide the host to host communication resulting in perfect weak scaling. This conclusion is also reached by Feichtinger *el al.* [12] who achieve perfect parallel efficiency up to 60 GPUs on 30 nodes.

Chapter 6

Conclusions and Future Work

In this thesis we investigated how to utilize multiple GPUs in Lattice Boltzmann simulations of large datasets. To show how different hardware configurations impact the performance three different multi-GPU LBM simulations were implemented: one with communication handled via MPI on the host, one using the shared memory of the host, and a hybrid combining the two others.

Three techniques were applied to lower the memory requirement while retaining numerical precision. Bastien's approach to reduce the round off errors associated with the collision phase was applied [8]. We also eliminated the need for double buffering, using Latt's approach of swapping particle distribution functions in the streaming phase [25]. Combined with storing only fluid elements, this reduced the mermory requirement by about 20 times for rocks of 15% porosity, increasing the domain size possible to simulate on a single GPU.

By connecting multiple GPUs to a single host we were able to achieve near linear speedup for large datasets. Our tests show that a single host thread is sufficient to coordinate up to four GPUs without the host becoming the bottleneck.

Simulation on GPU-enhanced clusters showed that by providing enough work for each GPU, we are able to completely hide the cost of host to host communication. By exploiting the potential for overlapping communication with computation we were able to show that LBM has favorable weak scaling characteristics for clusters with GPUs.

Our tests showed that LBM is well suited for simulation of porous rocks

on clusters with GPUs, and that the preferred implementation in a cluster is MPI. A hybrid implementation performs slightly better, but is more complex.

6.1 Future Work

Our results present several interesting directions for future work. Current implementations only support a single layer of border communication. Varying the border thickness would alter the communication to computation ratio, which may reveal performance improvements. Domain decomposition is presently restricted to a striped partitioning. An interesting extension would be to investigate how alternative decompositions impact the balance of computation and communication. Timings of the MPI and hybrid implementations indicate that GPU idle time is significant on configurations where GPUs differ, or when subsets of GPUs share system buses. Incorporating a load balancing scheme in the domain decomposition could reduce this effect. There are also other areas that could benefit from auto tuning, e.g finding the domain decomposition resulting in the least amount of communication time. The available systems restricted our experiments to scaling only to systems of four GPUs. An attempt to verify our scalability results on larger systems would be informative. Moreover, although development was done on an AMD GPU, all performance testing was restricted to NVIDIA hardware. A performance comparison using AMD cards would be interesting.

Bibliography

[1] Advanced Micro Devices, Inc. *ATI Stream Computing OpenCL Programming Guide*, 2010. `http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf`, retrieved 2010-07-06.

[2] Advanced Micro Devices, Inc. *ATI Stream SDK Release Notes*, 2011. `http://developer.amd.com/gpu/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf`, retrieved 2011-05-14.

[3] Erik Ola Aksnes and Anne C. Elster. Simulation of fluid flow through porous rock on modern gpus. Master's thesis, Norwegian University of Science and Technology (NTNU), 2009.

[4] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[5] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Yuan Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404 –418, march 2009.

[6] P. Bailey, J. Myre, S.D.C. Walsh, D.J. Lilja, and M.O. Saar. Accelerating lattice boltzmann fluid flow simulations using graphics processors. In *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 550 –557, sept. 2009.

[7] K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, and J.C. Sancho. Using performance modeling to design large-scale systems. *Computer*, 42(11):42 –49, nov. 2009.

[8] Bastien Chopard. How to improve the accuracy of lattice boltzmann calculations. Technical report, LBMethod.org, 2008.

[9] Simon T. Engler. Benchmarking the 2D lattice boltzmann BGK model. *Complex Simulation Report*, 2008.

[10] Alphonsus Fagan. An introduction to the petroleumindustry. `http://www.nr.gov.nl.ca/nr/publications/energy/intro.pdf`, retrieved 2011-05-09, November 1991.

[11] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, Washington, DC, USA, 2004. IEEE Computer Society.

[12] Christian Feichtinger, Johannes Habich, Harald Köstler, Georg Hager, Ulrich Rüde, and Gerhard Wellein. A flexible patch-based lattice boltzmann parallelization approach for heterogeneous gpu-cpu clusters. *Parallel Computing*, In Press, Accepted Manuscript:–, 2011.

[13] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 – 960, sept. 1972.

[14] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the navier-stokes equation. *Phys. Rev. Lett.*, 56(14):1505–1508, Apr 1986.

[15] Zhaoli Guo, Chuguang Zheng, and Baochang Shi. Discrete lattice effects on the forcing term in the lattice boltzmann method. *Phys. Rev. E*, 65(4):046308, Apr 2002.

[16] Johannes Habich. Performance evaluation of numeric compute kernels on nvidia gpus. Master's thesis, Friedrich-Alexander-Universität, 2008.

[17] J. Hardy, Y. Pomeau, and O. de Pazzis. Time evolution of a two-dimensional model system. i. invariant states and time correlation functions. *J. Math. Phys.*, 14(12):1746–1759, 1973.

[18] Roger W. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Computing*, 20(3):389 – 398, 1994.

[19] Guodong Jin, Tad W. Patzek, and Dmitry B. Silin. Direct prediction of the absolute permeability of unconsolidated and consolidated reservoir rock. September 2004.

[20] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *CoRR*, abs/1005.2581, 2010.

[21] Khronos Group. *The OpenCL Specification 1.0*, 2010. `http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf`, retrieved 2011-05-02.

[22] Khronos Group. *The OpenCL Specification 1.1*, 2010. `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`, retrieved 2011-05-02.

[23] Volodymyr V. Kindratenko, Jeremy Enos, Guochun Shi, Michael T. Showerman, Galen W. Arnold, John E. Stone, James C. Phillips, and Wen mei W. Hwu. Gpu clusters for high-performance computing. In *CLUSTER*, pages 1–8, 2009.

[24] Carolin Körner, Thomas Pohl, Ulrich Rüde, Nils Thürey, and Thomas Zeiser. Parallel lattice boltzmann methods for CFD applications. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, Lecture Notes in Computational Science and Engineering, pages 439–466. Springer Berlin Heidelberg, 2006.

[25] Jonas Latt. Technical report: How to implement your DdQq dynamic with only q variables per node (insted of 2q). Technical report, Tufts University Medford, USA, 2007.

[26] Davidson David Lee. The role of computational fluid dynamics in process industries. *The Bridge*, 32(4), 2002.

[27] Holger Ludvigsen and Anne C. Elster. Real-time gpu-based 3d ultrasound reconstruction and visualization. Master's thesis, Norwegian University of Science and Technology (NTNU), 2010.

[28] M.D. McCool. Scalable programming models for massively multicore processors. *Proceedings of the IEEE*, 96(5):816 –831, May 2008.

[29] Guy R. McNamara and Gianluigi Zanetti. Use of the boltzmann equation to simulate lattice-gas automata. *Phys. Rev. Lett.*, 61(20):2332–2335, Nov 1988.

[30] Nvidia Corporation. *NVIDIA CUDA C Programming Guide*, 2010. `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf`, retrieved 2011-05-14.

[31] Nvidia Corporation. *OpenCL Programming Guide for the CUDA Architecture*, 2010. `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/OpenCL_Programming_Guide.pdf`, retrieved 2010-07-06.

[32] Nvidia Corporation. *Tesla S1070 GPU Computing System*, 2010. `http://www.nvidia.com/docs/IO/43395/SP-04154-001_v03.pdf`, retrieved 2011-03-12.

[33] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879 –899, may. 2008.

[34] Chongxun Pan, Jan F. Prins, and Cass T. Miller. A high-performance lattice boltzmann implementation to model flow in porous media. *Computer Physics Communications*, 158(2):89 – 105, 2004.

[35] Rolf Rabenseifner. Hybrid parallel programming: Performance problems and chances, 2003.

[36] Daniele Spampinato and Anne C. Elster. Modeling communication on multi-gpu systems. Master's thesis, Norwegian University of Science and Technology (NTNU), 2009.

[37] J. Tölke and M. Krafczyk. Teraflop computing on a desktop pc with gpus for 3d cfd. *Int. J. Comput. Fluid Dyn.*, 22:443–456, August 2008.

[38] Jonas Tölke. Implementation of a lattice boltzmann kernel using the compute unified device architecture developed by nvidia. *Comput. Vis. Sci.*, 2008.

[39] Thor Kristian Valderhaug. Real-time gpu-based freehand 3D ultrasound reconstruction, December 2010. Fall specialization project.

[40] Wikipedia. File:fork_join.svg. `http://en.wikipedia.org/w/index.php?title=File:Fork_join.svg&oldid=339669228`, retrieved 2011-06-13.

[41] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[42] Dieter A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models: An Introduction*. Springer, 2005.

[43] W. Xian and A. Takayuki. Multi-gpu performance of incompressible flow computation by lattice boltzmann method on gpu cluster. Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.007.

Appendix A

Hardware Specification

On the following pages are the specification and configuration of the different hardware used to simulate LBM on multiple GPUs.

Table A.1 gives the specification of the host and the biggest shared memory system comprised of one host and four GPUs in form of a Tesla S1070 Computing system

Host	
CPU	Intel Core i7 965
Clock speed	3.20 GHz
Cores	4, with two threads per core
Memory	12 GB DDR3
OS	Ubuntu 09.10, Linux kernel 2.6.31-22
Device (Tesla S1070)	
Number of GPUs	4, Tesla T10
Clock speed	1.30 GHz
Number of SPs	960 (240 SPs per GPU)
Device memory	16 GB (4 GB dedicated per GPU)
GPU-device memory bandwidth	up to 408 GB/s (102 GB/s per GPU)
GPU-host bandwidth	up to 12.8 GB/s
Connection to the host	Through 2 PCIe channels pairwise shared.

Table A.1: Shared memory multiprocessor multi-GPU feature

Tables A.2, A.3 and A.4, gives the configuration of the test cluster used in this thesis. Each node of the cluster was also used as a shared memory system with two GPUs.

Host	
CPU	Intel Core 2 Quad 9550
Clock speed	2.83 GHz
Cores	4
Memory	4 GB DDR3
OS	Ubuntu 10.04 , Linux kernel 2.6.32-24

Table A.2: Cluster file server

Host	
CPU	Intel Core i7 950
Clock speed	3.07 GHz
Cores	4, with two threads per core
Memory	6 GB DDR3
OS	Ubuntu 10.04 , Linux kernel 2.6.32-24
Device 1 (Tesla C1060)	
Clock speed	1.50 GHz
Number of SPs	448
Device memory	3 GB
GPU-device memory bandwidth	up to 144 GB/s
GPU-host bandwidth	up to 8 GB/s
Connection to the host	Through PCIe x16
Device 2 (Tesla C1060)	
Clock speed	1.3 GHz
Number of SPs	240
Device memory	4 GB
GPU-device memory bandwidth	up to 102 GB/s
GPU-host bandwidth	up to 8 GB/s
Connection to the host	Through PCIe x16

Table A.3: Cluster compute node 1

Host	
CPU	Intel Core i7 970
Clock speed	3.20 GHz
Cores	6, with two threads per core
Memory	24 GB DDR3
OS	Ubuntu 10.04 , Linux kernel 2.6.32-24
Device 1 (Tesla C2070)	
Clock speed	1.50 GHz
Number of SPs	448
Device memory	6 GB
GPU-device memory bandwidth	up to 144 GB/s
GPU-host bandwidth	up to 8 GB/s
Connection to the host	Through PCIe x16
Device 2 (Tesla C2070)	
Clock speed	1.50 GHz
Number of SPs	448
Device memory	6 GB
GPU-device memory bandwidth	up to 144 GB/s
GPU-host bandwidth	up to 8 GB/s
Connection to the host	Through PCIe x16

Table A.4: Cluster compute node 2

Appendix B

Detailed Timings

On the following pages are the timings form each GPU in the different tests performed in this thesis.

B.1 Single GPU

Single GPUs

Fontainebleau									
Ant iter	500								
Ant lattice	4485814								
	2070			1060			T10		
	total	Per iter		total	per iter		total	per iter	
Collide Inner	23180	46,360	65,06%	48890	97,780	65,29%	51411	102,821	66,79%
Collide border	0	0,000	0,00%	0	0,000	0,00%	0	0,000	0,00%
Streaming	12450	24,900	34,94%	25990	51,980	34,71%	25566	51,132	33,21%
Communication	0	0,000	0,00%	0	0,000	0,00%	0	0,000	0,00%
Total	**35630**	**71,260**	**100,00%**	**74880**	**149,760**	**100,00%**	**76976**	**153,953**	**100,00%**
MFLUPS	62,95			29,95			29,14		
Grid500									
Ant iter	500								
Ant lattice	19197136								
	2070			1060			T10		
	total	Per iter		total	per iter		total	per iter	
Collide Inner	105190	210,380	63,81%	222640	445,280	67,26%	232722	465,445	68,89%
Collide border	0	0,000	0,00%	0	0,000	0,00%	0	0,000	0,00%
Streaming	59660	119,320	36,19%	108350	216,700	32,74%	105098	210,197	31,11%
Communication	0	0,000	0,00%	0	0,000	0,00%	0	0,000	0,00%
Total	**164850**	**329,700**	**100,00%**	**330990**	**661,980**	**100,00%**	**337821**	**675,642**	**100,00%**
MFLUPS	58,23			29			28,41		

B.2 Thread Implementation

Thread Two Tesla C2070

Fontainebleau						
Ant iter	500					
Ant lattice	4485814					
	2070			2070		
	total	Per iter		total	Per iter	
Collide Inner	11671,05	23,342	62,15%	11537,27	23,075	62,18%
Collide border	94,45	0,189	0,50%	95,43	0,191	0,51%
Streaming	6310,94	12,622	33,60%	6234,04	12,468	33,60%
Communication	703,73	1,407	3,75%	686,97	1,374	3,70%
Overlap	0,000	0,000	0,00%	0,000	0,000	0,00%
Total	**18780,162**	**37,560**	**100,00%**	**18553,707**	**37,107**	**100,00%**
MFLUPS	119,43			120,89		
Grid500						
Ant iter	500					
Ant lattice	19197136					
	2070			2070		
	total	Per iter		total	Per iter	
Collide Inner	52922,68	105,845	62,76%	52822,87	105,646	62,74%
Collide border	229,53	0,459	0,27%	228,62	0,457	0,27%
Streaming	29920,18	59,840	35,48%	29942,62	59,885	35,56%
Communication	1246,96	2,494	1,48%	1201,98	2,404	1,43%
Overlap	0,000	0,000	0,00%	0,000	0,000	0,00%
Total	**84319,341**	**168,639**	**100,00%**	**84196,084**	**168,392**	**100,00%**
MFLUPS	113,84			114,00		
Grid800						
Ant iter	500					
Ant lattice	65450258					
	2070			2070		
	total	Per iter		total	Per iter	
Collide Inner	156713,3	313,427	47,61%	156889,47	313,779	47,63%
Collide border	359,68	0,719	0,11%	361,36	0,723	0,11%
Streaming	170111,69	340,223	51,68%	170179,96	340,360	51,67%
Communication	1975,33	3,951	0,60%	1957,57	3,915	0,59%
Overlap	0,000	0,000	0,00%	0,000	0,000	0,00%
Total	**329159,995**	**658,320**	**100,00%**	**329388,352**	**658,777**	**100,00%**
MFLUPS	99,42			99,35		

Thread Two Tesla C1060

Fontainebleau						
Ant iter	500					
Ant lattice	4485814					
	1060			1060		
	total	Per iter		total	Per iter	
Collide Inner	24461,33	48,923	63,61%	24377,89	48,756	62,51%
Collide border	170,27	0,341	0,44%	173,01	0,346	0,44%
Streaming	12931,1	25,862	33,62%	13545,69	27,091	34,73%
Communication	894,97	1,790	2,33%	903,32	1,807	2,32%
Overlap	0,000	0,000	0,00%	0,000	0,000	0,00%
Total	**38457,661**	**76,915**	**100,00%**	**38999,924**	**78,000**	**100,00%**
MFLUPS	58,32			57,51		
Grid500						
Ant iter	500					
Ant lattice	19197136					
	1060			1060		
	total	Per iter		total	Per iter	
Collide Inner	115168,05	230,336	66,77%	111061,89	222,124	66,71%
Collide border	474,24	0,948	0,27%	462,89	0,926	0,28%
Streaming	55032,72	110,065	31,91%	53158,95	106,318	31,93%
Communication	1798,4	3,597	1,04%	1804,29	3,609	1,08%
Overlap	0,000	0,000	0,00%	0,000	0,000	0,00%
Total	**172473,408**	**344,947**	**100,00%**	**166488,015**	**332,976**	**100,00%**
MFLUPS	55,65			57,65		
Grid800						
Ant iter	500					
Ant lattice	65450258					
	1060			1060		
	total	Per iter		total	Per iter	
Collide Inner	431130,27	862,261	70,52%	433839,2	868	69,93%
Collide border	960,16	1,920	0,16%	959,92	2	0,15%
Streaming	176218,56	352,437	28,82%	182553,81	365	29,42%
Communication	3033,82	6,068	0,50%	3074,14	6	0,50%
Overlap	0,000	0,000	0,00%	0,000	0,000	0,00%
Total	**611342,800**	**1222,686**	**100,00%**	**620427**	**1241**	**100,00%**
MFLUPS	53,53			52,75		

Thread Two Tesla T10 Different PCI

Fontainebleau						
Ant iter	500					
Ant lattice	4485814					
	T10			T10		
	total	Per iter		total	Per iter	
Collide Inner	25775,75	51,552	64,90%	25639,77	51,280	63,92%
Collide border	182,65	0,365	0,46%	184,47	0,369	0,46%
Streaming	12973,06	25,946	32,66%	13313,69	26,627	33,19%
Communication	787,52	1,575	1,98%	972,15	1,944	2,42%
Overlap	0,000	0,000	0,00%	0,000	0,000	0,00%
Total	**39718,98**	**79,438**	**100,00%**	**40110,07**	**80,220**	**100,00%**
MFLUPS	56,47			55,92		
Grid500						
Ant iter	500					
Ant lattice	19197136					
	T10			T10		
	total	Per iter		total	Per iter	
Collide Inner	120518,83	241,038	67,95%	116074,47	232,149	68,45%
Collide border	497,57	0,995	0,28%	485,46	0,971	0,29%
Streaming	54888,43	109,777	30,95%	51599,65	103,199	30,43%
Communication	1453,9	2,908	0,82%	1424,88	2,850	0,84%
Overlap	0,000	0,000	0,00%	0,000	0,000	0,00%
Total	**177358,725**	**354,717**	**100,00%**	**169584,456**	**339,169**	**100,00%**
MFLUPS	54,12			56,6		
Grid800						
Ant iter	500					
Ant lattice	65450258					
	T10			T10		
	total	Per iter		total	Per iter	
Collide Inner	443441,39	886,883	71,32%	445516,66	891,033	71,22%
Collide border	991,2	1,982	0,16%	986,97	1,974	0,16%
Streaming	174953,73	349,907	28,14%	176615,51	353,231	28,23%
Communication	2417,33	4,835	0,39%	2463,16	4,926	0,39%
Overlap	0,000	0,000	0,00%	0,000	0,000	0,00%
Total	**621803,640**	**1243,607**	**100,00%**	**625582,291**	**1251,165**	**100,00%**
MFLUPS	52,63			52,31		

Thread Two Tesla T10 Same PCI

Fontainebleau						
Ant iter	500					
Ant lattice	4485814					
	T10			T10		
	total	Per iter		total	Per iter	
Collide Inner	25775,68	51,551	64,75%	25637,94	51,276	63,92%
Collide border	182,57	0,365	0,46%	184,5	0,369	0,46%
Streaming	12972,98	25,946	32,59%	13313,66	26,627	33,20%
Communication	874,98	1,750	2,20%	971,28	1,943	2,42%
Overlapp	0,000	0,000	0,00%	0,000	0,000	0,00%
Total	**39806,21**	**79,612**	100,00%	**40107,38**	**80,215**	100,00%
MFLUPS	56,35			55,92		
Grid500						
Ant iter	500					
Ant lattice	19197136					
	T10			T10		
	total	Per iter		total	Per iter	
Collide Inner	120521,91	241,044	68,51%	116084,02	232,168	68,44%
Collide border	497,77	0,996	0,28%	486,43	0,973	0,29%
Streaming	54886,31	109,773	31,20%	51599,22	103,198	30,42%
Communication	1456,69	2,913	0,83%	1438,46	2,877	0,85%
Overlapp	0,000	2,913	-0,83%	0,000	0,000	0,00%
Total	**177362,675**	**351,812**	100,00%	**169608,133**	**339,216**	100,00%
MFLUPS	54,12			56,59		
Grid800						
Ant iter	500					
Ant lattice	65450258					
	T10			T10		
	total	Per iter		total	Per iter	
Collide Inner	443448,79	886,898	71,32%	445506,49	891,013	71,22%
Collide border	990,54	1,981	0,16%	987,31	1,975	0,16%
Streaming	174961,18	349,922	28,14%	176609,52	353,219	28,23%
Communication	2375,34	4,751	0,38%	2425,61	4,851	0,39%
Overlapp	0,000	0,000	0,00%	0,000	0,000	0,00%
Total	**621775,84**	**1243,552**	100,00%	**625528,93**	**1251,058**	100,00%
MFLUPS	52,63			52,32		

Thread Four Tesla T10

Fontainebleau													
Ant iter	500												
Ant lattice	4485814												
	T10			T10			T10			T10			
	total	Per iter		total	Per iter		total	Per iter		total	Per iter		
Collide Inner	12856,83	25,714	63,45%	13519,12	27,038	60,14%	15711,04	31,422	60,09%	12493,88	24,988	63,39%	
Collide border	183,65	0,367	0,91%	381,38	0,763	1,70%	441,89	0,884	1,69%	184,16	0,368	0,93%	
Streaming	6439,76	12,880	31,78%	6968,32	13,937	31,00%	8509,06	17,018	32,55%	6305,33	12,611	31,99%	
Communication	783,29	1,567	3,87%	1611,01	3,222	7,17%	1482,31	2,965	5,67%	725,43	1,451	3,68%	
Overlap	0,000	0,000	0,00%	0,000	0,000	0,00%	0,000	0,000	0,00%	0,000	0,000	0,00%	
Total	**20263,52**	**40,527**	**100,00%**	**22479,82**	**44,960**	**100,00%**	**26144,3**	**52,289**	**100,00%**	**19708,79**	**39,418**	**100,00%**	
MFLUPS	110,69			99,77			85,79			113,8			
Grid500													
Ant iter	500												
Ant lattice	19197136												
	T10			T10			T10			T10			
	total	Per iter		total	Per iter		total	Per iter		total	Per iter		
Collide Inner	58511,65	117,023	67,03%	59137,76	118,276	65,79%	58501,6	117,003	65,30%	57081,74	114,163	67,42%	
Collide border	487,07	0,974	0,56%	969,13	1,938	1,08%	959,66	1,919	1,07%	471,91	0,944	0,56%	
Streaming	26608,09	53,216	30,48%	26727,62	53,455	29,74%	27123,43	54,247	30,27%	25451,06	50,902	30,06%	
Communication	1679,16	3,358	1,92%	3048,06	6,096	3,39%	3010,68	6,021	3,36%	1657,77	3,316	1,96%	
Overlap	0,000	0,000	0,00%	0,000	0,000	0,00%	0,000	0,000	0,00%	0,000	0,000	0,00%	
Total	**87285,98**	**174,572**	**100,00%**	**89882,57**	**179,765**	**100,00%**	**89595,37**	**179,191**	**100,00%**	**84662,47**	**169,325**	**100,00%**	
MFLUPS	109,97			106,79			107,13			113,37			

Thread Four Tesla T10

Grid800													
Ant iter	500												
Ant lattice	65450258												
	T10				T10			T10			T10		
	total	Per iter			total	Per iter		total	Per iter		total	Per iter	
Collide Inner	221971,33	443,943	71,11%		222773,63	445,547	69,45%	226623,31	453,247	66,41%	224083,4	448,167	68,49%
Collide border	990,76	1,982	0,32%		1990,49	3,981	0,62%	2026,41	4,053	0,59%	1003,33	2,007	0,31%
Streaming	86780,72	173,561	27,80%		87652,11	175,304	27,33%	99196,93	198,394	29,07%	95077,1	190,154	29,06%
Communication	2414,9	4,830	0,77%		8355,72	16,711	2,60%	13405,71	26,811	3,93%	7025,26	14,051	2,15%
Overlap	0,000	0,000	0,00%		0,000	0,000	0,00%	0,000	0,000	0,00%	0,000	0,000	0,00%
Total	**312157,695**	**624,315**	**100,00%**		**320771,949**	**641,544**	**100,00%**	**341252,367**	**682,505**	**100,00%**	**327189,094**	**654,378**	**100,00%**
MFLUPS	104,84				102,02			95,9			100,02		

B.3 MPI Implementation

MPI Two GPUs

Fontainebleau						
Ant iter	500					
Ant lattice	4485814					
	2070			1060		
	total	Per iter		total	Per iter	
Collide Inner	11667,71	23,335	14,96%	24376,95	48,754	30,93%
Collide border	94,13	0,188	0,12%	172,94	0,346	0,22%
Streaming	6271,13	12,542	8,04%	13545,88	27,092	17,19%
Communication	71619,92	143,240	91,84%	65100,09	130,200	82,59%
Overlap	11667,712	23,335	-14,96%	24376,953	48,754	-30,93%
Total	**77985,18**	**155,970**	**100,00%**	**78818,9**	**157,638**	**100,00%**
MFLUPS	28,76			28,46		
Grid500						
Ant iter	500					
Ant lattice	19197136					
	2070			1060		
	total	Per iter		total	Per iter	
Collide Inner	52923,77	105,848	25,19%	111057,11	222,114	48,47%
Collide border	228,5	0,457	0,11%	463,16	0,926	0,20%
Streaming	29715,49	59,431	14,14%	53156,71	106,313	23,20%
Communication	180159,31	360,319	85,75%	175519,25	351,038	76,60%
Overlap	52923,770	105,848	-25,19%	111057,108	222,114	-48,47%
Total	**210103,296**	**420,207**	**100,00%**	**229139,113**	**458,278**	**100,00%**
MFLUPS	45,68			41,89		
Grid800						
Ant iter	500					
Ant lattice	65450258					
	2070			1060		
	total	Per iter		total	Per iter	
Collide Inner	156740,95	313,482	31,68%	427354,9	854,710	69,96%
Collide border	357,64	0,715	0,07%	946,93	1,894	0,16%
Streaming	170184,76	340,370	34,40%	182552,29	365,105	29,88%
Communication	324202,64	648,405	65,53%	312559,66	625,119	51,17%
Overlap	156740,953	313,482	-31,68%	312559,664	625,119	-51,17%
Total	**494745,037**	**989,490**	**100,00%**	**610854,119**	**1221,708**	**100,00%**
MFLUPS	66,15			53,57		

MPI Four GPUs

Fontainebleau														
Ant iter	500													
Ant lattice	4485814													
	2070						1060							
	total	Per iter					total	per iter						
Collide Inner	7236,72	14,473	68,44%	7413,66	14,827	8,92%	13299,42	26,599	16,32%	15502,82	31,006	65,17%		
Collide border	112,16	0,224	1,06%	226,86	0,454	0,27%	365,9	0,732	0,45%	222,75	0,446	0,94%		
Streaming	3224,89	6,450	30,50%	3357,25	6,715	4,04%	6693,45	13,387	8,21%	8061,56	16,123	33,89%		
Communication	882,41	1,765	8,35%	79544,93	159,090	95,69%	74432,68	148,865	91,34%	4156,58	8,313	17,47%		
Overlap	882,413	1,765	-8,35%	7413,661	14,827	-8,92%	13299,419	26,599	-16,32%	4156,583	8,313	-17,47%		
Total	**10574**	**21,148**	**100,00%**	**83129**	**166,258**	**100,00%**	**81492**	**162,984**	**100,00%**	**23787**	**47,574**	**100,00%**		
MFLUPS	212,12			26,98			27,52			94,29				
Grid500														
Ant iter	500													
Ant lattice	19197136													
	2070			2070			1060			1060				
	total	Per iter		total	Per iter		total	per iter		total	per iter			
Collide Inner	23071,69	46,143	60,15%	23122,37	46,245	11,51%	56267,86	112,536	27,27%	55084,58	110,169	67,38%		
Collide border	199,82	0,400	0,52%	403,81	0,808	0,20%	917,29	1,835	0,44%	455,06	0,910	0,56%		
Streaming	15084,28	30,169	39,33%	15241,8	30,484	7,59%	27293,54	54,587	13,23%	26215,07	52,430	32,07%		
Communication	2099,23	4,198	5,47%	185247,9	370,496	92,21%	178147,08	356,294	86,33%	3404,54	6,809	4,16%		
Overlap	2099,230	4,198	-5,47%	23122,366	46,245	-11,51%	56267,857	112,536	-27,27%	3404,54	6,809	-4,16%		
Total	**38356**	**76,712**	**100,00%**	**200893,508**	**401,787**	**100,00%**	**206358**	**412,716**	**100,00%**	**81755**	**163,509**	**100,00%**		
MFLUPS	250,25			47,78			46,51			117,41				

MPI Four GPUs

Grid800														
Ant iter	500													
Ant lattice	65450258													
	2070				2070			1060			1060			
	total	Per iter			total	Per iter		total	per iter		total	per iter		
Collide Inner	78171,01	156,342	47,76%		77886,45	155,773	18,84%	219995,91	439,992	51,68%	218190,3	436,381	68,78%	
Collide border	361,03	0,722	0,22%		725,08	1,450	0,18%	1962,68	3,925	0,46%	973,05	1,946	0,31%	
Streaming	85150,24	170,300	52,02%		85832,85	171,666	20,77%	99317,93	198,636	23,33%	98043,38	196,087	30,91%	
Communication	3314,32	6,629	2,02%		326790,29	653,581	79,06%	324402,29	648,805	76,21%	9627,10	19,254	3,03%	
Overlap	3314,318	6,629	-2,02%		77886,451	155,773	-18,84%	219995,912	439,992	-51,68%	9627,10	19,254	-3,03%	
Total	**163682,272**	**327,365**	**100,00%**		**413348,222**	**826,696**	**100,00%**	**425682,898**	**851,366**	**100,00%**	**317206,728**	**634,413**	**100,00%**	
MFLUPS	199,93				79,17			76,88			103,17			

B.4 Hybrid Implementation

Hybrid Four GPUs

Fontainebleau													
Ant iter	500												
Ant lattice	4485814												
	2070			2070				1060			1060		
	total	Per iter		total	Per iter			total	per iter		total	per iter	
Collide Inner	5802,97	11,606	64,16%	5889,66	11,779	7,73%	15111,47	30,223	20,03%	12003,6	24,007	64,57%	
Collide border	92,42	0,185	1,02%	188,91	0,378	0,25%	422,76	0,846	0,56%	175,96	0,352	0,95%	
Streaming	3148,49	6,297	34,81%	3280,45	6,561	4,31%	8494,05	16,988	11,26%	6409,33	12,819	34,48%	
Communication	711,82	1,424	7,87%	72718,21	145,436	95,45%	66538,62	133,077	88,18%	822,32	1,645	4,42%	
Overlap	711,815	1,424	-7,87%	5889,658	11,779	-7,73%	15111,473	30,223	-20,03%	822,318	1,645	-4,42%	
Total	**9043,879**	**18,088**	**100,00%**	**76187,564**	**152,375**	**100,00%**	**75455,432**	**150,911**	**100,00%**	**18588,884**	**37,178**	**100,00%**	
MFLUPS	248			29,44			29,72			120,66			
Grid500													
Ant iter	500												
Ant lattice	19197136												
	2070			2070				1060			1060		
	total	Per iter		total	Per iter			total	per iter		total	per iter	
Collide Inner	26390,1	52,780	63,58%	26429,8	52,860	13,26%	55845,82	111,692	27,20%	55382,1	110,764	67,06%	
Collide border	225,35	0,451	0,54%	458,81	0,918	0,23%	912,14	1,824	0,44%	453,64	0,907	0,55%	
Streaming	14891,54	29,783	35,88%	15201,79	30,404	7,62%	27300,16	54,600	13,30%	26755,5	53,511	32,40%	
Communication	1541,78	3,084	3,71%	183730,51	367,461	92,15%	177084,73	354,169	86,26%	1783,74	3,567	2,16%	
Overlap	1541,776	3,084	-3,71%	26429,802	52,860	-13,26%	55845,815	111,692	-27,20%	1783,735	3,567	-2,16%	
Total	**41506,990**	**83,014**	**100,00%**	**199391,111**	**398,782**	**100,00%**	**205297,030**	**410,594**	**100,00%**	**82591,244**	**165,182**	**100,00%**	
MFLUPS	231,25			48,14			46,75			116,22			

Hybrid Four GPUs

Grid800														
Ant iter	500													
Ant lattice	65450258													
	2070				2070			1060			1060			
	total	Per iter			total	Per iter		total	per iter		total	per iter		
Collide Inner	78359,18	156,718	47,81%		77714,32	155,429	19,31%	216721,05	433,442	52,86%	212700,49	425,401	70,06%	
Collide border	357,07	0,714	0,22%		724,48	1,449	0,18%	1923,12	3,846	0,47%	945,42	1,891	0,31%	
Streaming	85173,13	170,346	51,97%		85548,12	171,096	21,26%	97232,42	194,465	23,72%	89955,44	179,911	29,63%	
Communication	2058,09	4,116	1,26%		316160,78	632,322	78,56%	310807,84	621,616	75,81%	3544,16	7,088	1,17%	
Overlap	2058,094	4,116	-1,26%		77714,324	155,429	-19,31%	216721,045	433,442	-52,86%	3544,156	7,088	-1,17%	
Total	**163889,372**	**327,779**	**100,00%**		**402433,391**	**804,867**	**100,00%**	**409963,372**	**819,927**	**100,00%**	**303601,357**	**607,203**	**100,00%**	
MFLUPS	199,68				81,32			79,82			107,79			

Appendix C

Selected Source Code

In this appendix the main GPU kernels used in this thesis are listed. Complete source code is available upon request to thor.kristian@tksoftware.no

C.1 Collision Phase

The following code listing shows the kernel implementing the collision phase.

```
1  __kernel void collide_and_swap(__read_only image3d_t solids,
2                                 __global real_t *f0,
3                                 __global real_t *f1,
4                                 __global real_t *f2,
5                                 __global real_t *f3,
6                                 __global real_t *f4,
7                                 __global real_t *f5,
8                                 __global real_t *f6,
9                                 __global real_t *f7,
10                                __global real_t *f8,
11                                __global real_t *f9,
12                                __global real_t *f10,
13                                __global real_t *f11,
14                                __global real_t *f12,
15                                __global real_t *f13,
16                                __global real_t *f14,
17                                __global real_t *f15,
18                                __global real_t *f16,
19                                __global real_t *f17,
20                                __global real_t *f18,
21                                __local real_t *lattice,
22                                real_t omega,
23                                real_t force,
24                                real_t avrage_dencity,
25                                int ant_vec,
26                                int offset_x,
27                                int offset_y,
28                                int offset_z )
29 {
30
31     const int x = get_global_id(0) + offset_x;
32     const int y = get_global_id(1) + offset_y;
33     const int z = get_global_id(2) + offset_z;
34
35     const int nx = get_global_size(0);
36     const int ny = get_global_size(1);
37     const int nz = get_global_size(2);
38
```

```
39      const int lnx = get_local_size(0);
40      const int lny = get_local_size(1);
41      const int lnz = get_local_size(2);
42
43      const int lx = (get_local_id(2) * lny * lnx) + (get_local_id(1) * lnx) +
            get_local_id(0);
44
45      int4 this_coord = (int4) (x,y,z,0);
46      int this_index =  read_imagei(solids,sampler,this_coord).x;
47
48      const int local_index = lx * ant_vec;
49
50      //Make pointer to this threads starting pos in the shared lattice memory
51      __local real_t *local_lattice = &lattice[local_index];
52
53      if (this_index != -1) {
54              copy_to_shared(f0,f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,
55                      f11,f12,f13,f14,f15,f16,f17,f18,
56                      local_lattice,this_index);
57      }
58      barrier(CLK_LOCAL_MEM_FENCE);
59
60      if (this_index == -1) {
61              return;
62      }
63
64      // Compute density from the particle distribution functions
65      real_t delta_rho = get_rho_shared(local_lattice);
66      real_t rho = delta_rho + avrage_dencity;
67
68      //Compute velocity in x, y, and z directions from the particle distribution
            functions
69      real_t ux = get_ux_shared(local_lattice,rho);
70      real_t uy = get_uy_shared(local_lattice,rho);
71      real_t uz = get_uz_shared(local_lattice,rho);
72
73
74      //precalculeting some of the constants used later
75      real_t uSqr = ux*ux + uy*uy + uz*uz;
76      real_t u_sqr = (1.0f/3.0f) * uSqr;
77      real_t f_eq;
78      real_t cu;
79      real_t external_force = 0.0f;
80
81
82    #define C_3D(i,j) c_3d[3*(i)+(j)]
83    for(int i = 0; i < 19 ; i++){
84
85      //Compute local equalibrium
86      cu = C_3D(i,0)*ux + C_3D(i,1)*uy + C_3D(i,2)*uz;
87      f_eq = t_3d[i]*delta_rho + t_3d[i]*rho*((3.f*cu) + (cu*cu) - u_sqr);
88      external_force = C_3D(i,0) * 3.0f * force * t_3d[i] * avrage_dencity; //Flyttall
89
90      local_lattice[i] += omega * (f_eq - local_lattice[i]) + external_force;
91    }
92  #undef C_3D
93
94    barrier(CLK_LOCAL_MEM_FENCE);
95
96    //swap when coppying back to global memory
97    copy_to_global(f0,f10,f11,f12,f13,f14,f15,f16,f17,
98                   f18,f1,f2,f3,f4,f5,f6,f7,f8,f9,
99                   local_lattice,this_index);
100 }
```

C.2 Streaming Phase

The following code listing shows the kernel implementing the streaming phase.

```
1  __kernel void stream_and_swap(__read_only image3d_t solids,
2                                __global real_t *f0,
3                                __global real_t *f1,
4                                __global real_t *f2,
```

```
5                                   __global real_t *f3,
6                                   __global real_t *f4,
7                                   __global real_t *f5,
8                                   __global real_t *f6,
9                                   __global real_t *f7,
10                                  __global real_t *f8,
11                                  __global real_t *f9,
12                                  __global real_t *f10,
13                                  __global real_t *f11,
14                                  __global real_t *f12,
15                                  __global real_t *f13,
16                                  __global real_t *f14,
17                                  __global real_t *f15,
18                                  __global real_t *f16,
19                                  __global real_t *f17,
20                                  __global real_t *f18,
21                                  __global real_t *max_change)
22
23 {
24     const int x = get_global_id(0);
25     const int y = get_global_id(1);
26     const int z = get_global_id(2);
27
28     const int nx = get_global_size(0);
29     const int ny = get_global_size(1);
30     const int nz = get_global_size(2);
31
32     int4 this_coord = (int4) (x,y,z,0);
33     int4 neighbour_coord;
34     real_t _max = 0.0;
35
36     int this_index =  read_imagei(solids,sampler,this_coord).x;
37     int neighbour_index;
38
39     if (this_index == -1) {
40         return;
41     }
42
43
44 #define periodicX true
45 #define periodicY true
46 #define periodicZ false
47
48     // Swap 1 and 10
49     neighbour_coord = (int4)(get_next_x(x,nx,1,periodicX),
50                              get_next_y(y,ny,1,periodicY),
51                              get_next_z(z,nz,1,periodicZ),
52                              0);
53     if(neighbour_coord.z >= 0 && neighbour_coord.z < nz){
54         neighbour_index = read_imagei(solids,sampler,neighbour_coord).x;
55         if(neighbour_index >= 0){
56             swap_global(&f10[this_index],&f1[neighbour_index],&_max);
57         }
58     }
59     // Swap 2 and 11
60     neighbour_coord = (int4)(get_next_x(x,nx,2,periodicX),
61                              get_next_y(y,ny,2,periodicY),
62                              get_next_z(z,nz,2,periodicZ),
63                              0);
64     if(neighbour_coord.z >= 0 && neighbour_coord.z < nz){
65         neighbour_index = read_imagei(solids,sampler,neighbour_coord).x;
66         if(neighbour_index >= 0){
67             swap_global(&f11[this_index],&f2[neighbour_index],&_max);
68         }
69     }
70     // Swap 3 and 12
71     neighbour_coord = (int4)(get_next_x(x,nx,3,periodicX),
72                              get_next_y(y,ny,3,periodicY),
73                              get_next_z(z,nz,3,periodicZ),
74                              0);
75     if(neighbour_coord.z >= 0 && neighbour_coord.z < nz){
76         neighbour_index = read_imagei(solids,sampler,neighbour_coord).x;
77         if(neighbour_index >= 0){
78             swap_global(&f12[this_index],&f3[neighbour_index],&_max);
79         }
80     }
81     // Swap 4 and 13
82     neighbour_coord = (int4)(get_next_x(x,nx,4,periodicX),
83                              get_next_y(y,ny,4,periodicY),
84                              get_next_z(z,nz,4,periodicZ),
85                              0);
86
87     if(neighbour_coord.z >= 0 && neighbour_coord.z < nz){
```

```
88          neighbour_index = read_imagei(solids,sampler,neighbour_coord).x;
89          if(neighbour_index >= 0){
90              swap_global(&f13[this_index],&f4[neighbour_index],&_max);
91          }
92      }
93      // Swap 5 and 14
94      neighbour_coord = (int4)(get_next_x(x,nx,5,periodicX),
95                               get_next_y(y,ny,5,periodicY),
96                               get_next_z(z,nz,5,periodicZ),
97                               0);
98
99      if(neighbour_coord.z >= 0 && neighbour_coord.z < nz){
100         neighbour_index = read_imagei(solids,sampler,neighbour_coord).x;
101         if(neighbour_index >= 0){
102             swap_global(&f14[this_index],&f5[neighbour_index],&_max);
103         }
104     }
105     // Swap 6 and 15
106     neighbour_coord = (int4)(get_next_x(x,nx,6,periodicX),
107                              get_next_y(y,ny,6,periodicY),
108                              get_next_z(z,nz,6,periodicZ),
109                              0);
110
111     if(neighbour_coord.z >= 0 && neighbour_coord.z < nz){
112         neighbour_index = read_imagei(solids,sampler,neighbour_coord).x;
113         if(neighbour_index >= 0){
114             swap_global(&f15[this_index],&f6[neighbour_index],&_max);
115         }
116     }
117     // Swap 7 and 16
118     neighbour_coord = (int4)(get_next_x(x,nx,7,periodicX),
119                              get_next_y(y,ny,7,periodicY),
120                              get_next_z(z,nz,7,periodicZ),
121                              0);
122
123     if(neighbour_coord.z >= 0 && neighbour_coord.z < nz){
124         neighbour_index = read_imagei(solids,sampler,neighbour_coord).x;
125         if(neighbour_index >= 0){
126             swap_global(&f16[this_index],&f7[neighbour_index],&_max);
127         }
128     }
129
130     // Swap 8 and 17
131     neighbour_coord = (int4)(get_next_x(x,nx,8,periodicX),
132                              get_next_y(y,ny,8,periodicY),
133                              get_next_z(z,nz,8,periodicZ),
134                              0);
135
136     if(neighbour_coord.z >= 0 && neighbour_coord.z < nz){
137         neighbour_index = read_imagei(solids,sampler,neighbour_coord).x;
138         if(neighbour_index >= 0){
139             swap_global(&f17[this_index],&f8[neighbour_index],&_max);
140         }
141     }
142     // Swap 9 and 18
143     neighbour_coord = (int4)(get_next_x(x,nx,9,periodicX),
144                              get_next_y(y,ny,9,periodicY),
145                              get_next_z(z,nz,9,periodicZ),
146                              0);
147     if(neighbour_coord.z >= 0 && neighbour_coord.z < nz){
148         neighbour_index = read_imagei(solids,sampler,neighbour_coord).x;
149         if(neighbour_index >= 0){
150             swap_global(&f18[this_index],&f9[neighbour_index],&_max);
151         }
152     }
153   max_change[this_index] = _max;
154 }
```

C.3 Border Exchange Phase

The following code listing shows the kernel used to gather and scatter the particle distribution function into a single buffer for transferring to and from the host in the border exchange phase.

```
__kernel void copy_to_from_ghost_layer(__global real_t *f0,
                                       __global real_t *f1,
                                       __global real_t *f2,
                                       __global real_t *f3,
                                       __global real_t *f4,
                                       __global real_t *f5,
                                       __global real_t *f6,
                                       __global real_t *f7,
                                       __global real_t *f8,
                                       __global real_t *f9,
                                       __global real_t *f10,
                                       __global real_t *f11,
                                       __global real_t *f12,
                                       __global real_t *f13,
                                       __global real_t *f14,
                                       __global real_t *f15,
                                       __global real_t *f16,
                                       __global real_t *f17,
                                       __global real_t *f18,
                                       __global real_t *buffer,
                                       int ant_vec,
                                       int start_index,
                                       int to)
{

  const int offset = get_global_id(0);
  const int ant_element = get_global_size(0);
  const int index = start_index + offset;


  __global real_t *f[19];
  f[ 0] = f0;
  f[ 1] = f1;
  f[ 2] = f2;
  f[ 3] = f3;
  f[ 4] = f4;
  f[ 5] = f5;
  f[ 6] = f6;
  f[ 7] = f7;
  f[ 8] = f8;
  f[ 9] = f9;
  f[10] = f10;
  f[11] = f11;
  f[12] = f12;
  f[13] = f13;
  f[14] = f14;
  f[15] = f15;
  f[16] = f16;
  f[17] = f17;
  f[18] = f18;

  for (int i = 0 ; i <ant_vec; i++){
    if(to == 1){
      buffer[(i * ant_element) + offset] = f[i][index];
    }else{
      f[i][index] = buffer[(i * ant_element) + offset];
    }
  }
}
```