NTNU
Norwegian University of
Science and Technology

Simulation of Fluid Flow Through Porous Rocks on Modern GPUs

Eirik Ola Aksnes

Master of Science in Computer Science
Submission date: July 2009
Supervisor: Anne Cathrine Elster, IDI

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

Problem Description

This project evaluates how Graphical Processing Units (GPUs) may be utilized to offload computations i large petroleum engineering applications. In particular, the task is to develope and implement a parallelized version of the Lattice Boltzman Method (LBM) for simulation fluid flow through porous rocks for modern GPUs taking advantage of NVIDIA's CUDA environment. Testing will be done using datasets provided by the oil industry on HPC-LAB's new NVIDIA Quadro FX 5800 graphics card, the NVIDIA Tesla s1070 and/or other appropriate systems.

Assignment given: 26. January 2009
Supervisor: Anne Cathrine Elster, IDI

NTNU

Norwegian University of
Science and Technology

Master Thesis

Simulation of Fluid Flow Through Porous Rocks on Modern GPUs

Eirik Ola Aksnes
eirikola@stud.ntnu.no

Department of Computer and Information Science
Norwegian University of Science and Technology, Trondheim
(Norway)

July 2009

Supervisor
Dr. Anne C. Elster

Abstract

It is important for the petroleum industry to investigate how fluids flow inside the complicated geometries of porous rocks, in order to improve oil production. The lattice Boltzmann method can be used to calculate the porous rock's ability to transport fluids (permeability). However, this method is computationally intensive and hence begging for *High Performance Computing* (HPC). Modern GPUs are becoming interesting and important platforms for HPC. In this thesis, we show how to implement the lattice Boltzmann method on modern GPUs using the NVIDIA CUDA programming environment. Our work is done in collaborations with Numerical Rocks AS and the Department of Petroleum Engineering at the Norwegian University of Science and Technology.

To better evaluate our GPU implementation, a sequential CPU implementation is first prepared. We then develop our GPU implementation and test both implementation using three porous data sets with known permeabilities provided by Numerical Rocks AS. Our simulations of fluid flow get high performance on modern GPUs showing that it is possible to calculate the permeability of porous rocks of simulations sizes up to 368^3, which fit into the 4 GB memory of the NVIDIA Quadro FX 5800 card. The performances of the CPU and GPU implementations are measured in MLUPS (million lattice node updates per second). Both implementations achieve their highest performances using single floating-point precision, resulting in the maximum performance equal to 1.59 MLUPS and 184.30 MLUPS. Techniques for reducing round-off errors are also discussed and implemented.

Acknowledgements

This master thesis was written at the Department of Computer and Information Science at the Norwegian University of Science and Technology in collaboration with Numerical Rocks AS and the Department of Petroleum Engineering.

This thesis would not have been possible without the support of numerous people. I wish to thank my supervisor Dr. Anne C. Elster who was incredibly helpful and offered invaluable assistance and guidance with her extensive understanding of the field. She has been a source of great inspiration, with her always encouraging attitude and generosity through providing resources needed for this project from her own income. I wish to thank Ståle Fjeldstad, and Atle Rudshaug at Numerical Rocks AS for providing me with three porous data sets to evaluate my implementations, valuable assistance, and giving me the possibility to work on this subject, which has been an outstanding experience. I wish to thank Egil Tjåland at the Department of Petroleum Engineering and Applied Geophysics, and the Center for Integrated Operations in the Petroleum Industry for letting me use their microcomputed tomography device. I wish to thank Pablo M. Dupuy at the Department of Chemical Engineering at the Faculty of Natural Sciences and Technology, without his valuable advice-giving and guidance in computational fluid dynamics this thesis would not have been successful. I wish to thank Jan Christian Meyer for guidance, technical support, and many interesting discussions during this project. I wish to thank Thorvald Natvig for advice-giving and guidance throughout the process. I wish to thank fellow co-student Rune Erlend Jensen for many interesting discussions, and ideas to this thesis work. I also with to thank my fellow co-students: Henrik Hesland, Rune Johan Hovland, Åsmund Herikstad, Safurudin Mahic, Robin Eidissen, Daniele Giuseppe Spampinato, and many other peoples for many interest-

ing and entertaining discussions throughout the semester. Finally, I wish to thank NVIDIA for providing several GPUs to Dr. Anne C. Elster and her HPC-lab through her membership in the NVIDIA's professor partnership program.

Eirik Ola Aksnes
Trondheim, Norway, July 12, 2009

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

CFD	Computational Fluid Dynamics
HPC	High Performance Computing
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General-Purpose Computation On Graphics Processing Units
CUDA	Compute Unified Device Architecture
OpenCL	Open Computing Language
UMA	Uniform Memory Access
NUMA	Non Uniform Memory Access
MIMD	Multiple-Instruction Multiple-Data
SIMD	Single-Instruction Multiple-Data
SPMD	Single-Program Multiple-Data
ILP	Instruction Level Parallelism
SM	Streaming Multiprocessors
SP	Stream Processors
SFU	Special Function Units
LBM	Lattice Boltzmann Method
LGCA	Lattice Gas Cellular Automata
BGK	Bhatnagar-Gross-Krook
VBO	Vertex Buffer Object
GLEW	OpenGL Extension Wrangler Library
GLUT	The OpenGL Utility Toolkit

Nomenclature

τ	Single relaxation parameter
ρ	Mass density of a fluid particle
u_x	Velocity in the x direction of a fluid particle
u_y	Velocity in the y direction of a fluid particle
u_z	Velocity in the z direction of a fluid particle
f^{eq}	Equilibrium distribution function
f_i^{eq}	Discrete equilibrium distribution function in the i direction
$\bar{u}$	Average fluid velocity
d_x	Lattice dimension in the x direction
d_y	Lattice dimension in the y direction
d_z	Lattice dimension in the z direction
f	Particle distribution function
f_i	Discrete particle distribution function in the i direction
w_i	Discrete weight factor in the i direction
e	Particle velocity
e_i	Discrete particle velocity in the i direction
c_s	Speed of Sound
F_x	External force in the x direction
F_y	External force in the y direction
F_z	External force in the z direction
ν	Kinematic viscosity
Ω	Collision operator
R	Universal gas constant
D	Dimension of the space
T	Temperature
q	Volumetric fluid flux

k	Permeability
$\frac{\Delta P}{L}$	Pressure drop along the sample length L
ϕ	Porosity
V_p	Volume of pore space
V_t	Total volume of the porous media

Chapter 1

Introduction

A great many problems in science and engineering are too difficult to solve analytically in practice, but with today's powerful computers it is possible to analyze and solve those problems numerically. Computer simulations are important to the field of fluid dynamics, as their equations are often complicated referred to as *Computational Fluid Dynamics* (CFD). To solve the most complex problems in fluid dynamics, computer simulations performed by systems with huge performance capability are necessary. In the field of *High Performance Computing* (HPC), researchers are interested in maximizing the computing power available in systems with huge performance capability. Systems are typically made from clusters of workstations, large expensive supercomputers, or *Graphics Processing Units* (GPUs), to be able to solve complex problems in science and engineering.

With the recent introduction of NVIDIA's *Compute Unified Device Architecture* (CUDA) programming environment for the NVIDIA Tesla architecture, as well as heterogeneous programming standards such as *Open Computing Language* (OpenCL), GPUs are becoming interesting and important platforms for HPC. Modern GPUs have typically their own dedicated memory [1], and are optimized for performing floating-point operations in parallel, which are much used in games, multimedia, and scientific applications. Today, even commercial GPUs can have very high floating-point compute capacity at very low cost, available as off-the-shelf products. The state-of-the-art in GPUs can provide computing power equal to small supercomputers [2]. Compared with

[1] NVIDIA Quadro FX 5800 have total memory size of whole 4 GB.

[2] NVIDIA has recently released the NVIDIA Tesla s1070 Computing System, that's

an up-to-date *Central Processing Unit* (CPU), modern GPUs utilize a larger portion of their transistors for floating-point arithmetic, and has higher memory bandwidth. With the recent developments and improvements in GPU hardware and software, many difficulties are eliminated, allowing GPUs to be used as accelerators for a wide range of scientific applications. Today, much attention is focused on how to utilize the GPUs huge performance capability for more than just graphics rendering, to accelerate computationally intensive problems, referred to as *General-Purpose computation on Graphics Processing Units* (GPGPU).

1.1 Project Goals

For the petroleum industry it is important to quantify the petrophysical properties of porous rocks, such as the rock illustrated in Figure 1.1, to gain better understanding of conditions that affect oil production [5]. It would be of great value for the petroleum industry if the petrophysical properties of porous rocks, such as the porosity and permeability, could be obtained directly through computer simulations, capable of fast and accurate analysis.

The main objective in this thesis is to investigate the use of the graphics processing unit for simulations of fluid flows through the internal pore geometry of natural and computer generated rocks, in order to compute the rock's ability to transport fluids (permeability).

In this thesis, we have chosen to look at the lattice Boltzmann method for the simulation of fluid flow through porous rocks, offloaded to the GPU. The permeability of porous rocks is obtained directly from the generated velocity fields of the lattice Boltzmann method, together with using Darcy's law for the flow of fluids through porous media [24].

provide parallelism of total 960 streaming processor cores across 4 GPUs, with total of 4 teraflops compute capability [13].

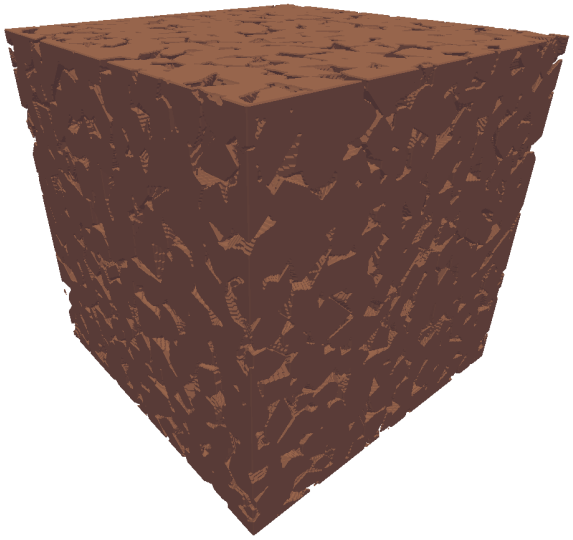


Figure 1.1: Porous rock.

1.2 Outline

This thesis is structured in the following manner:

In Chapter 2, **Parallel Computing and The Graphics Processing Unit**, we will highlight several advantages with parallel computing, and describe various forms of parallelism. We will describe modern GPUs, what tasks that suit GPUs, and explain some differences between GPUs and CPUs. We will also give a brief introduction to the NVIDIA CUDA programming model and the NVIDIA Tesla architecture.

In Chapter 3, **Computational Fluid Dynamics and Porous Rocks**, we will present and explain the necessary background theory for fluid flow through porous rocks using the lattice Boltzmann method. We will also give a brief description of porous rocks, and how to calculate their permeability.

In Chapter 4, **Implementations**, we will describe how the lattice Boltzmann method has been implemented. We will also give a brief description of

the Marching Cubes algorithm, used for visual analysis of how the fluid will flow through the internal pore geometry of porous rocks.

In Chapter 5, **Benchmarks**, we will present performance benchmarks of our implementation of the lattice Boltzmann method, and estimations of porous rock's ability to transmit fluids. We will compare differences between both performance and permeability estimations using NVIDIA GPUs and CPUs, with both single and double floating-point precision.

In Chapter 6, **Conclusions and Future Work**, we will summarize the results we achieved, and discuss future work to improve our implementations of the lattice Boltzmann method.

Chapter 2

Parallel Computing and The Graphics Processing Unit

The topics covered in this chapter were also covered by the author and Henrik Hesland in [3], the joint fall upon which this project was built. It describes some of the basic concept of parallel computing as well as the Graphics Processing Unit (GPU).

In particular, Section 2.1 gives a brief introduction to parallel computing, and explains different forms of parallelism. Section 2.2 explains modern GPUs and what tasks they are suited for, and motivates for why one should use GPUs instead of the CPUs for some compute intensive tasks. At the end of this chapter, the NVIDIA CUDA programming model and the NVIDIA Tesla architecture are presented.

2.1 Parallel Computing

Moore's law predicts that integration of transistors doubles in 18 months. Today, to double the number of transistors placed on integrated circuits in 18 months has become difficult. We have already hit the Power and Frequency Walls. Whatever the peak performance of today's processors, there will always be some problems that require or benefit from better processor speed. As explained in [6], there is a recent renaissance in parallel computing development. Due to the Power Wall, increasing clock frequency is no longer

the primary method of improving processor performance. Today, parallelism has become the standard way to increase overall performance for both the CPU and GPU. Both modern GPUs and CPUs are concerned with increasing power dissipation, and want to increase absolute performance, but also improve efficiency through architectural improvements by means of parallelism.

Parallel computing often permits a larger problem or a more precise solution of a problem to be found within a practical time. Parallel computing is the concept of breaking up a larger problem into smaller units of tasks that can be solved. However, problems often cannot be broken up perfectly into independent parts, so interactions are needed among the parts, both for data transfer and synchronization. The problem characteristicts affects how easy it is to parallelize. If possible, there would be no interaction between the separate processes, each process requiring different data and produce results from its input data without need for results from the other processes. However many problems are to be found in the middle, neither fully independent nor synchronized [54].

There are two basic types of parallel computers, when categorized by their memory architecture [54]:

- Shared memory systems that have a single address space, which means that all processing elements can access the same global memory. It can be very hard to implement the hardware to achieve *Uniform Memory Access* (UMA) by all the processors with a larger number of processors, and therefore many systems have *Non Uniform Memory Access* (NUMA).

- Distributed memory systems that are created by connecting computers together through an interconnection network, where each computer has its own local memory that cannot be accessed by the other processors. The access time to the local memory is faster than the access time to the non-local memory.

Distributed memory will physically scale more easily than shared memory, as its memory is scalable with the increased number of processors.

2.1.1 Forms of parallelism

Most of the information found in this section is based on ref by Mccool [37].

There are several ways to do parallel computing. Two frequently used methods are task parallelism and data parallelism. Task parallelisms, also called *Multiple-Instruction Multiple-Data* (MIMD), focus on distributing separate tasks across different parallel computing nodes that operate on separate data in parallel. It can typically be difficult to find independently tasks in a program and therefore task parallelism can have limited scalability ability. The interaction between different tasks occurs through either message passing or shared memory regions. Communication through shared memory regions poses the problem of maintaining memory cache coherency with increased number of cores, as most modern multi core CPUs use caches for memory latency hiding. Ordinary sequential execution of a single thread is deterministic, making it understandable. Task parallelism on the other hand is not even if the program is correct. Task parallelism is subject to errors such as race conditions and deadlocks, as correct synchronization is difficult. Such faults are difficult to identify, which can make development time overwhelming. Multicore CPUs are capable of running entirely independent threads of control, and are therefore great for task parallelism [37].

Data parallelism is a form of computation that implicitly has synchronization requirements. In a data parallel computation, the same operation is performed on different data elements concurrently. Data parallel programming is very convenient for two reasons. It can be easy to program and it can scale easily to large problems. The *Single-Instruction Multiple-Data* (SIMD) is the simplest type of data parallelism. It operates by having the same instruction execute in parallel on different data elements concurrently. It is convenient from a hardware standpoint. It gives an efficient hardware implementation, because it only needs to replicate the data path. However, it has difficulty of balance variable work load, since it does not support efficient control flow. The SIMD models have been generalized to the *Single-Program Multiple-Data* (SPMD), which include some control flow. With the SPMD it is possible to avoid and adjust the work load if there are variable amounts of computation in the different parts of a program [37], as illustrated in Figure 2.1. Data parallelism is essential for the modern GPU as a parallel processor, as it is optimized to carry out the same operations on a lot of data in parallel.

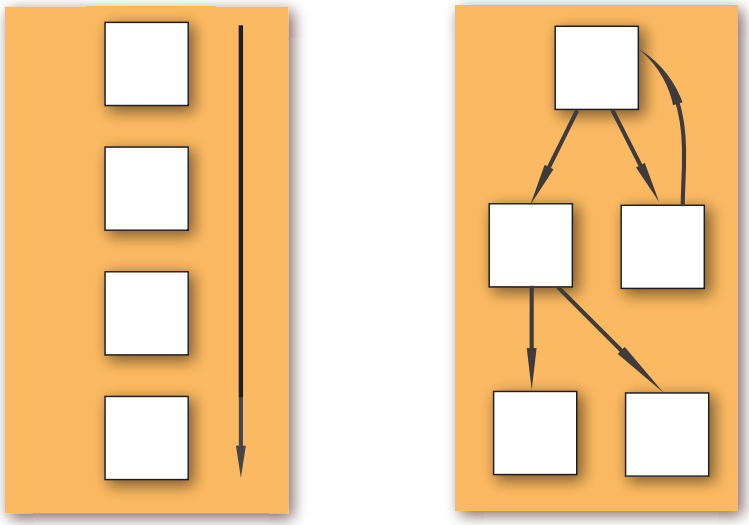


Figure 2.1: SPMD support control flow, and SIMD does not. Based on [37].

2.2 The Graphics Processing Unit

The *Graphics Processing Unit* (GPU) is a special-purpose processor dedicated to rendering computer graphics. With their potential for computing several hundred instructions simultaneously, these accelerators are becoming very interesting architectures also for *High Performance Computing* (HPC). As special-purpose accelerators, the GPUs are primarily designed to accelerate graphics rendering, and most development has been targeted towards improved graphics for game consoles, personal computers, and workstations. Over the last 40 years, the GPU has undergone significant changes in its functionality and capability, driven primarily by an ever increasing demand for more realistic graphics in computer games. The GPU has evolved from a fixed processor only capable of doing restricted graphics rendering, into a dedicated programmable processor with huge performance capability. Modern GPU's theoretical floating-point processing power and memory bandwidth, has exceeded the *Central Processing Units* (CPUs) [12].

Figure 2.2: Transistors dedicated for data processing: CPU vs. GPU. Figure is taken with permission from [12].

The CPU is designed to maximize the performance of a single thread of sequential instructions. It operates on different data types, performs random memory accesses, and branching. *Instruction Level Parallelism* (ILP) allows the CPU to execute several instructions at the same time or even alter the order in which the instructions are executed. To increase performance, the CPU uses many of its transistors to avoid memory latency with data caching, sophisticated flow control, and to extract as much ILP as possible. There is a limited amount of ILP that is possible to identify and take advantage of in a sequential stream of instructions, to keep the execution units active. This is also known as the ILP Wall, and ILP causes tremendous increase in hardware complexity and related power consumption, without linear speedup in application performance [36].

The GPU is dedicated to rendering computer graphics, and the primitives, pixel fragments and pixels can largely be processed independently in parallel (the fragment stage is typically the most computationally demanding stage [39]). The GPU differs from the CPU in the memory access pattern, as memory access in the GPU is very coherent. When a pixel is read or written, the neighboring pixel will be read or write a few cycles later. By organizing memory correctly and hide memory access latency by doing calculations instead, there is no need for big data caches. GPUs are designed such that the same instruction operates on collections of data, and therefore only need simple flow control. GPUs may therefore dedicate more of its transistors to data processing than the CPU, as illustrated in Figure 2.2.

The modern GPU is a mixture of programmable and fixed function units, allowing programmers to write vertex, fragment and geometry programs for sophisticated surface shading, and lighting effects. The instruction sets of the vertex and fragment programs have converged, now all programmable units in the graphics pipeline share a single programmable hardware unit. Into the unified shader architecture, where the programmable units share their time among vertex work, fragment work, and geometry work [39]. GPUs differentiate themselves from traditional CPU designs by prioritizing high-throughput processing of many parallel operations over the low-latency execution of a single thread. Quite often in scientific and multimedia applications there is a need to do the same operation on a lot of different data elements. GPUs support a massive number of threads, typically 61440 on a NVIDIA GeForce GTX 295, running concurrently and support the SPMD model to be able to suspend and use threads to hide the latency with uneven workloads in the programs. The combination of high performance, low-cost, and programmability has made the modern GPU attractive for applications traditionally executed by the CPU, for *General-Purpose Computation On GPUs* (GPGPU). With the unified shader architecture, the GPGPU programmers can target the programmable units directly, rather than split up task to different hardware units.

Since GPUs are first and foremost made for graphics rendering, it is natural that the first attempts of programming GPUs for non-graphics where through graphics APIs. This makes it tougher to use as a HPC platform since programmers have to master the graphics APIs and languages. To simplify this programming task and to hide the graphics APIs overhead, several programming models have recently been created. The latest release from the GPU supplier NVIDIA is NVIDIA's *Compute Unified Device Architecture* (CUDA), initially released in November 2006 [12].

2.2.1 NVIDIA CUDA Programming Model

Most of the information found in this section is based on NVIDIA's programming guide for NVIDIA CUDA [12].

There are a few difficulties with the traditional way of doing GPGPU, with the graphics API overhead that are making unnecessary high learning

curve and the difficult to debugging. In NVIDIA CUDA, programmers do not need to think in terms of the graphics APIs for developing applications to run on the GPU. It also reveals the hardware functions of the GPUs directly, giving the programmer better control. NVIDIA CUDA is a programming model that focuses on low learning curve for developing applications that are scalable with the increase number of processor cores. It is based on a small extension to the C programming language, making it easier to get started with for developers familiar with the C language. Since there are currently millions of PCs and workstations with NVIDIA CUDA enabled GPUs, developing techniques to harvest the GPUs power make it feasible to accelerate applications for a broad range of users. Parts of programs that have little parallelism execute on the CPU, while parts that have rich parallelism execute on the GPU. To a programmer, a system in the NVIDIA CUDA programming model consists of a host that is a traditional CPU, and one or more compute devices that are massively data-parallel coprocessors. Each device is equipped with a large number of arithmetic execution units, has its own DRAM, and runs many threads in parallel.

Kernels And Execution

In NVIDIA CUDA, programmers are allowed to define data-parallel functions, called kernels, that run in parallel on many threads [43]. These kernels are invoked from the host, and are defined using a special syntax, which indicates how they are executed on the GPU. To invoke a kernel, programmers need to specify the number of thread blocks, and threads within these thread blocks, between triple angle brackets, and to define the kernels using the *__global__* declaration sign. This is illustrated in Algorithm 1. In NVIDIA CUDA, there exist several qualifiers for functions:

- *__global__* defines functions that can only be called from the host, and that execute on the device.
- *__device__* defines functions that can only be called from the device, and that execute on the device.
- *__host__* defines functions that can only be called from the host, and that execute on the host.

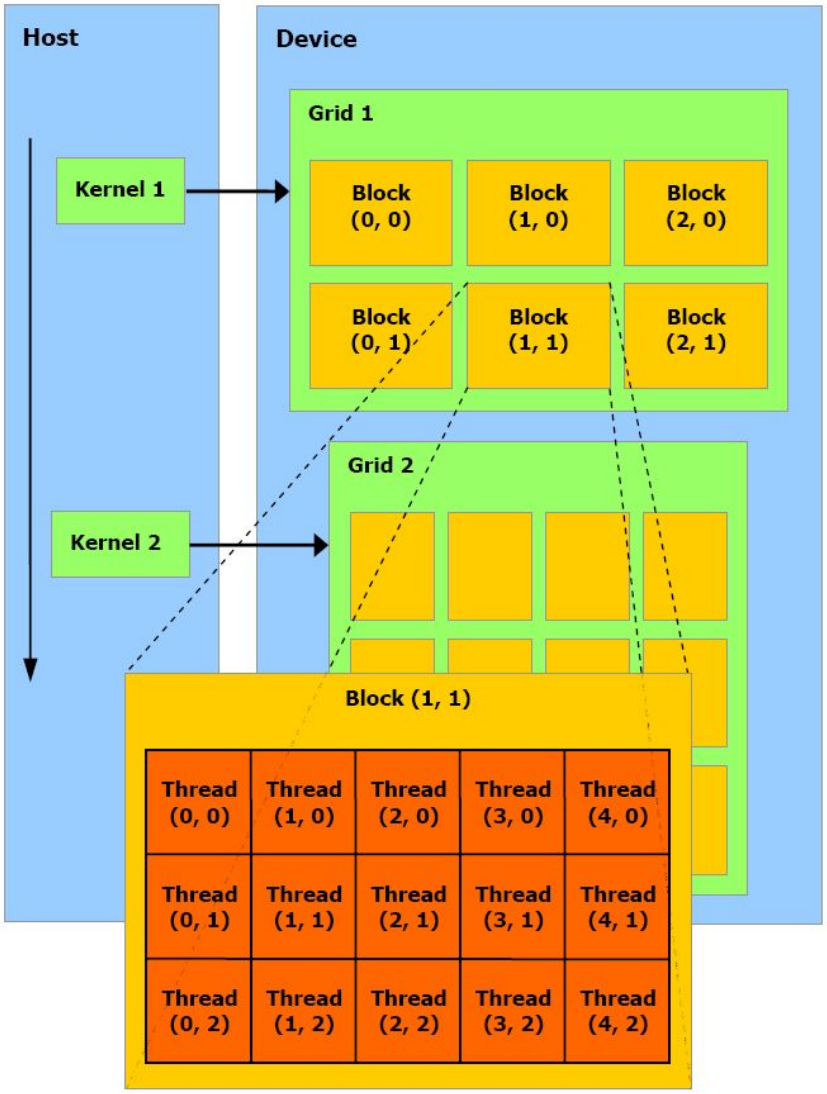


Figure 2.3: NVIDIA CUDA thread hierarchy. Figure is taken with permission from [12].

Algorithm 1 Definition and execution of NVIDIA CUDA kernels.

```
// Kernel definition
__global__ void kernelName()
{
  ...
}

void main()
{
  // Defines how the GPU kernel is executed
  kernelName<<<gridDim,blockDim>>>();
}
```

Threads are grouped into a three level hierarchy during execution, as illustrated in Figure 2.3. Every kernel executes as a grid of thread blocks in one or two dimensions, where each thread block has a unique identification index in the grid. Each thread block is an array of threads, in one, two or three dimensions, where each individual thread has a unique identification index in the thread block. Threads within the same thread block can synchronize, by calling *__syncthreads()*. To help programmers, several built-in variables are available:

- *gridDim*, which holds the dimension of the grid.
- *blockDim*, which holds the dimension of the thread block.
- *threadIdx*, which contains the index to the current thread within the current thread block.
- *blockIdx*, which contains the index to the current thread block within the grid.

How to use the build-in variables to calculate the index of a thread in a two-dimensional block is illustrated in Algorithm 2.

Algorithm 2 Build-in variables in NVIDIA CUDA kernels.

```
// Kernel definition
__global__ void kernelName()
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  int i = x + y * blockDim.x;

  ...
}
```

Memory

All threads in a GPU kernel can access data from diverse places during execution. Each thread has its private *local memory*, and the architecture allows effective sharing of data between threads within a thread block by using the low latency *shared memory*. There are also two additional read-only memories accessible by all threads, the *texture memory* and *constant memory*. The texture memory is optimized for various memory accesses patterns. Finally, all threads have access to the same large, high latency *global memory*, and it is possible to transfer data to and from the GPUs global memory and the hosts memory using different API calls.

Variables may also have different qualifiers, used to differentiate where in the memory the variables should be stored:

- *__device__* defines a variable to be stored in the global memory, available from all threads, and available to the host through API calls.
- *__constant__* defines a variable to be stored in the constant memory, available from all threads, and available to the host through API calls.
- *__shared__* defines a variables to be stored in the shared memory for a current thread block, available only to the threads within the same thread block.

In addition, with the *__device__* and *__constant__* declarations the variable has

the life time of the application, and with the *__shared__* declaration the variable has the life time of the current block.

2.2.2 NVIDIA Tesla Architecture

Most of the information found in this section is based on NVIDIA's programming guide for NVIDIA CUDA [12].

For high performance, knowledge of NVIDIA's GPUs hardware architecture is important. The latest generations of NVIDIA's GPUs are based on the NVIDIA Tesla architecture that supports the NVIDIA CUDA programming model. The NVIDIA Tesla architecture is built around a scalable array of *Streaming Multiprocessors* (SMs), and each SM consists of several *Stream Processors* (SPs), two *Special Function Units* (SFU) for complex calculations (sine, cosine and square root), a multithreaded instruction unit, on-chip shared memory, texture cache, some registers, and a constant cache [12], as illustrated in the Figure 2.4. GPUs from NVIDIA based on the NVIDIA Tesla architecture have the same architectural foundation, but support different degree of parallelism equal to the number of SM. Some of the latest generations of desktop GPUs from NVIDIA, based on the NVIDIA Tesla architecture are listed in Table 2.1.

Table 2.1: NVIDIA GPUs based on the NVIDIA Tesla architecture. Taken from [13] and [12].

GPU Model	8800 GT	9800 GX2	GTX 295
Number of GPUs	1	2	2
Streaming Multiprocessors	14	32	60
Stream Processors	112	256	480
Graphics Clock	600 MHz	600 MHz	576 MHz
Processor Clock	1500 MHz	1500 MHz	1242 MHz
Memory	512 MB	1 GB	1792 MB
Memory Bandwidth	57.6 GB/s	128 GB/s	223.8 GB/s
Compute Capability	1.1	1.1	1.3

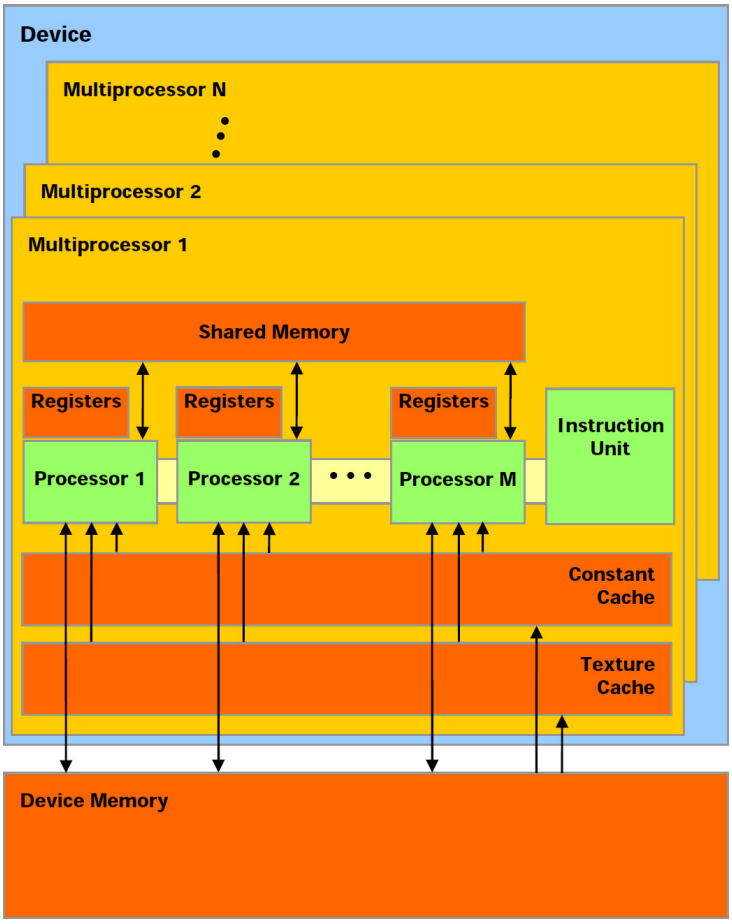


Figure 2.4: NVIDIA Tesla architecture. Figure is taken with permission from [12].

Compute Capability

The GPUs from NVIDIA based on the NVIDIA Tesla architecture have different compute capability. As of April 2009, indicated by a version number from 1.0 to 1.4, which describes the NVIDIA's GPUs technical specifications and features supported, for among other things it can indicate support for double precision floating-point numbers. Today, GPUs from NVIDIA have a number of general features and technical specifications listed in Table 2.2, but also some differences listed in Table 2.3.

Table 2.2: NVIDIA's GPUs general compute capability. Taken from [12].

Compute capability	1.1-1.4.
Maximum size of the x dimension of a thread block	512
Maximum size of the y dimension of a thread block	512
Maximum size of the z dimension of a thread block	64
Maximum number of threads per thread block	512
Maximum number of active thread blocks per SM	8
Amount of shared memory available per SM	16 KB
Total amount of constant memory	64 KB
Maximum number of thread blocks per grid	65535

Table 2.3: NVIDIA's GPUs points of distinction in compute capability. Taken from [12].

Compute capability	1.1	1.4
Double-precision float-point support	no	yes
Maximum number of active threads per SM	768	1024
Maximum number of active warps per SM	24	32
Maximum number of threads per thread block	512	1024
Number of registers per SM	8192	16384
Support for atomic functions	no	yes

Execution

For high performance, NVIDIA's GPUs exploit *massive multithreading*, a hardware technique which executes thousands of threads simultaneously, to utilize the large number of computational cores and overlap memory transactions with computation. This is possible because NVIDIA's GPUs threads have very little creation overhead, and it is possible to switch between threads that execute with near zero cost [48]. When a GPU kernel is invoked, thread blocks from the kernel grid are distributed to SMs with available execution capacity. As one thread block terminates, new thread blocks are lunched on the SM [12]. Therefore to keep the SM busy, one needs to have enough thread blocks in the grid, and threads in thread blocks. There is, however, a

limit on how many thread blocks a SM can process at once, one need to find the right balance between how many registers per thread, how much shared memory per thread block and what number of simultaneously active threads are required for a given kernel [48]. The programming guide for NVIDIA CUDA states that the minimum number of blocks should be at least twice the number of SMs in the device, preferably larger than 100, and that 64 is the minimum of threads within a block. During execution, threads within a thread block are grouped into warps, which are 32 threads from continuous sections of a thread block. Even though warps are not explicit declared, knowledge of them may improve performance. NVIDIA's GPUs support the SPMD model where all threads execute the same program although they don't need to follow the same path of execution. The SM executes the same instruction for every thread in a warp, so only threads that follow the same execution path can be executed in parallel. If none of the threads in a warp have the same execution path, all of them must be executed sequentially [48]. There are several memory types with different latency available through the NVIDIA Tesla architecture, such as the global memory and the shared memory. For high performance, it is essential to know some details of the different memory types.

Global Memory

For maximum global memory bandwidth, which can be very high, it is important to to access the global memory with the right access pattern. This is achieved through coalescing, and result in a single memory transaction for simultaneous accesses against the global memory by threads in a half-warp. To achieve coalescing, the programming guide for NVIDIA CUDA state some conditions that must be met. For NVIDIA GPUs with compute capability 1.0 and 1.1, there are four conditions that global memory access must meet to achieve coalescing, which are illustrated in Figure 2.5:

- Threads must access:

  4-byte words, resulting in one 64-byte memory transaction.

  8-byte words, resulting in one 128-byte memory transaction.

  16-byte words, resulting in two 128-byte memory transactions.

- Every 16 words stand in the same segment in global memory.

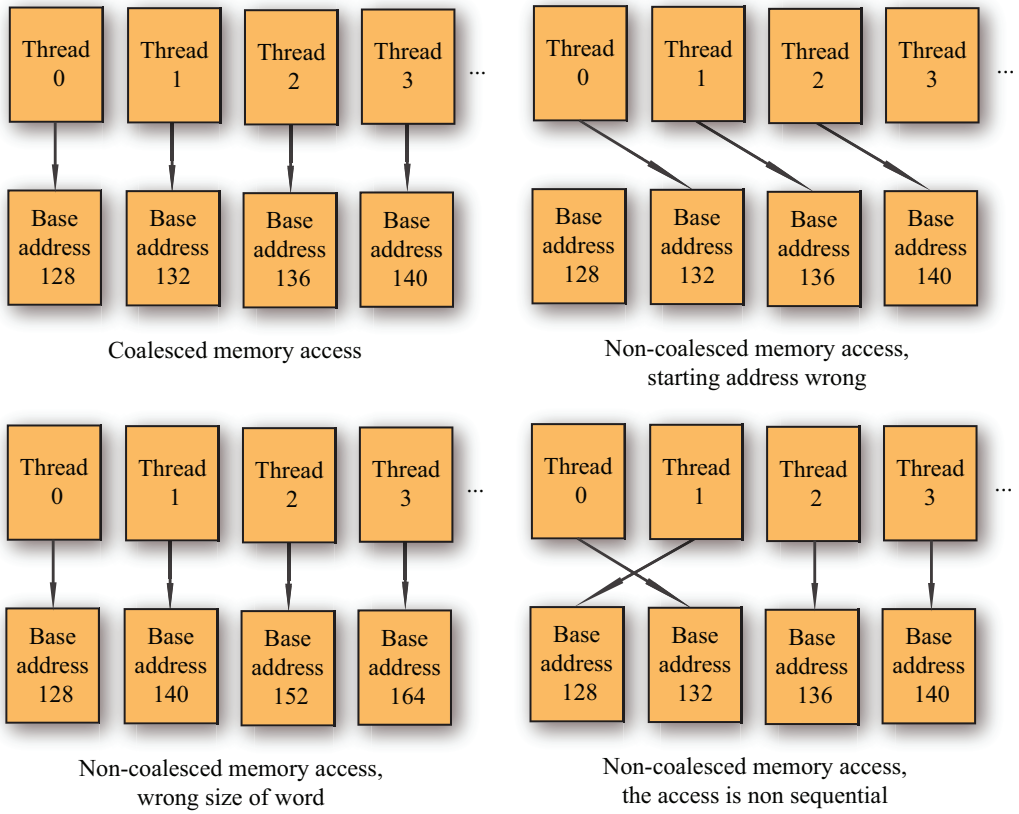


Figure 2.5: Examples of coalesced and non-coalesced global memory access patterns. Reproduced from [12].

- Threads must access the words in sequence, the k^{th} thread must access the k^{th} word.

With compute capability 1.2 and higher, threads do not need to access global memory in sequence to achieve coalescing, and access to the global memory over one segment will not directly result in 16 separate accesses. To achieve coalescing:

- Thread access the global memory with words stand in the same aligned segment of required size:

  32-bytes for threads access 2-byte words

  64-bytes for threads access 4-byte words

128-bytes for threads access 8-byte or 16-bytes words

Shared Memory

The 16 KB of on-chip shared memory is divided into 16 banks, where access to the same bank only can be done sequentially, but access to different banks can be done in parallel. To get the maximum performance, the addresses of memory requests must fall into separate banks, or bank conflicts will arise. The shared memory is organized so that a sequence of four bytes words is assigned to different banks, as illustrated in Figure 2.6. A common access pattern organized to avoid memory bank conflicts, are illustrated in Algorithm 3, where tid is the thread index and s some stride. To ensure that there will be no bank conflicts, the programming guide for NVIDIA CUDA state that s must be odd [12], or bank conflicts will aris as illustrated in Figure 2.7.

Algorithm 3 Memory access pattern without bank conflicts. Taken from [12].

```
__shared__ float shared[32];
float data = shared[BaseIndex + s * tid];
```

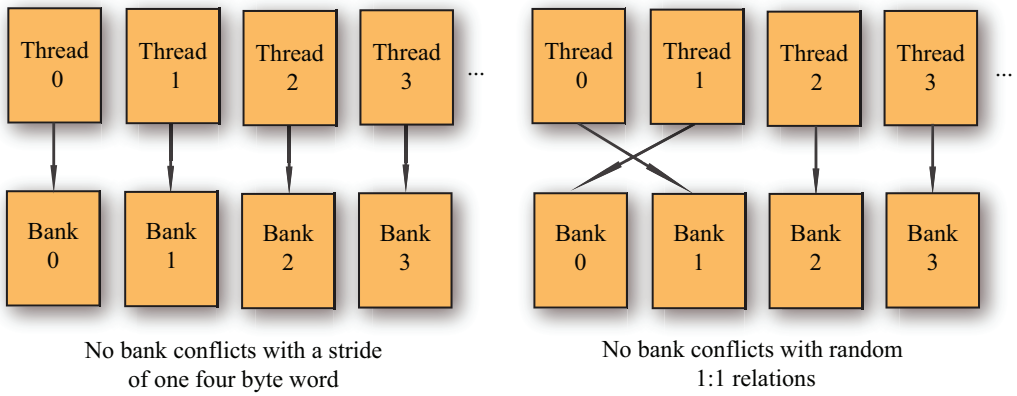
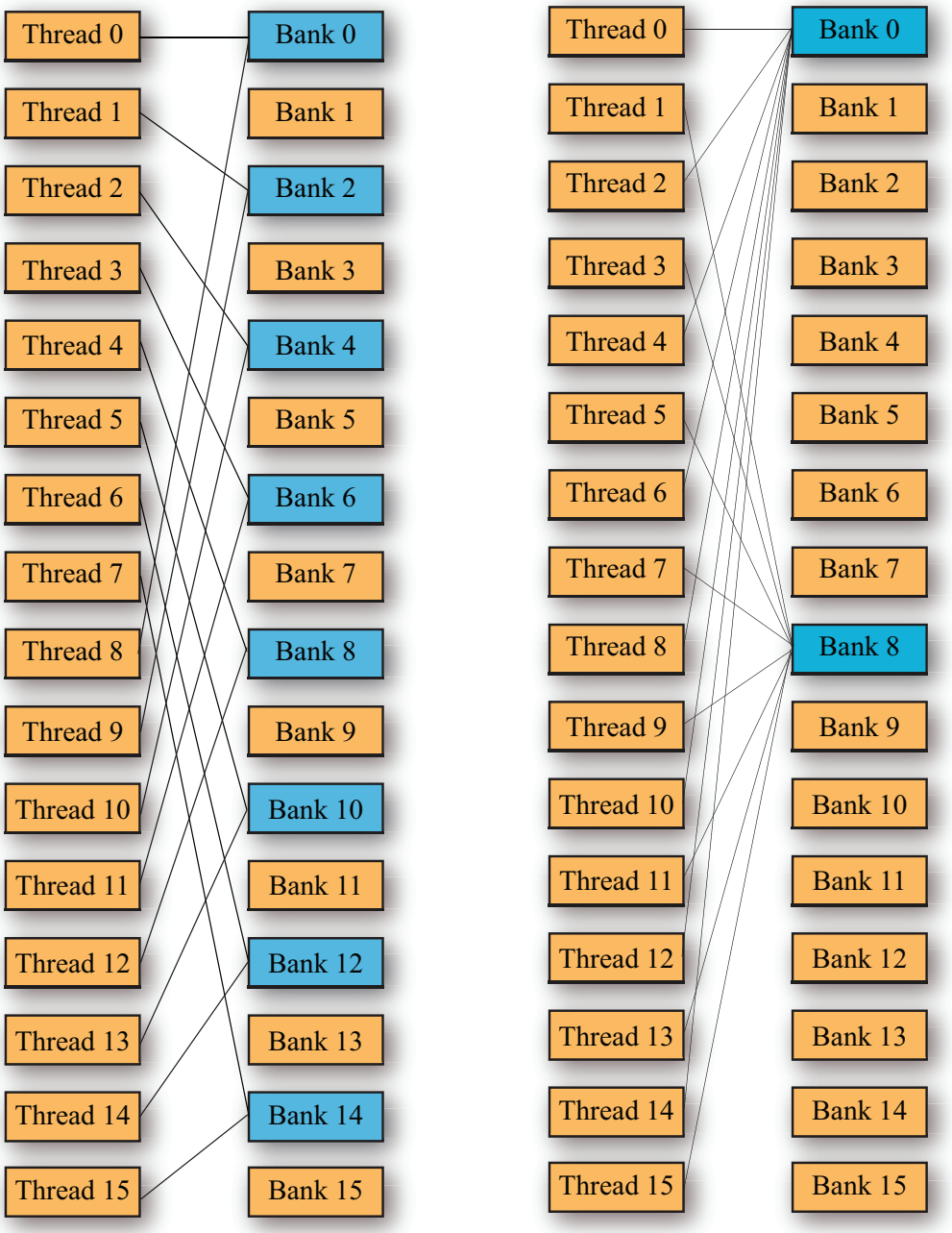


Figure 2.6: Examples of shared memory access patterns without bank conflicts. Reproduced from [12].



Bank conflicts with linear addressing with a stride of two four byte words

Bank conflicts with linear addressing with a stride of eight four byte words

Figure 2.7: Examples of shared memory access patterns with bank conflicts. Reproduced from [12].

Chapter 3

Computational Fluid Dynamics and Porous Rocks

This chapter highlights the background theory we used for simulating fluid flow through porous rocks, including the lattice Boltzmann method.

In particular, Section 3.1 gives a brief introduction to computational fluid dynamics. Section 3.2 presents the theory of the lattice Boltzmann method and explains the meaning of the most important equations of the method. Section 3.3 gives a brief description of porous rocks, and how to calculate the porosity and permeability in these.

3.1 Computational Fluid Dynamics

In the field of fluid dynamics, researchers study fluid flows. Similar to a lot of physical phenomena in classical mechanics, fluid flows must satisfy the principles of conservation of mass and momentum, together with energy, which are expressed in terms of mathematical equations in form of differential equations. These equations are known as the conservation equations. Analytical solutions to these equations are of interest, but given that these equations can be very complex to describe mathematically, and therefore can be difficult to solve analytically, it is useful to use computers to solve these equations numerically. *Computational Fluid Dynamics* (CFD) is the process of analyzing and solving different equations describing fluid flow, and other related phe-

nomena numerically on a computer [17]. Fluid flows through porous rocks are of importance to the petroleum industry, and with CFD analysis it is possible to investigate how the fluids behave inside the complicated geometries of porous rocks. This gives a better understanding of how to harvest the oil.

The fundamental equations for CFD are the Navier-Stokes equations. The Navier-Stokes equations are nonlinear partial differential equations that are too difficult to solve analytically in practice, but with today's powerful computers it is possible to analyze and solve approximations to these equations. In order to solve the Navier-Stokes equations numerically, the equations needs to be discretized using finite differences, finite elements, finite volumes or spectral methods [55].

In this thesis, the lattice Boltzmann method is used for simulating fluid flow through porous rocks, which is an alternative CFD approach to the Navier-Stokes equations, based on microscopic models and the Boltzmann equation [27]. The lattice Boltzmann method might be considered mesoscopic CFD approach, and has several desired properties for performing CFD simulations of fluid flows through porous media, particularly the ability to deal with complex geometries without significant penalty in speed and efficiency [45].

3.2 The Lattice Boltzmann Method

The intention of this section is to give a brief introduction to the theory of the lattice Boltzmann method applied in this thesis. For a more comprehensive overview of the method we refer to [47] and [55].

The Boltzmann equation, formulated by Ludwig Boltzmann, uses classical mechanics and statistical physics, to describe the evolution of a particle distribution function. The lattice Boltzmann method is a solver of the Boltzmann equation in a fixed lattice. The particle distribution function gives the probability of finding a fluid particle located at the location x, with velocity e, at time t [4]. The complex interactions of all these particles manifest as a fluid on a macroscopic scale. The Boltzmann equation without external

forces, can be written as Equation 3.1 [27].

$$\frac{\partial f}{\partial t} + e\nabla f = \Omega \tag{3.1}$$

where f is the particle distribution function, e is the particle velocity, and Ω is the collision operator that describes the interaction of collided particles.

The basic idea in the lattice Boltzmann method is that fluid flows can be simulated by interacting particles within a lattice in one, two or three dimensions. These particles perform successive streaming and collision over the lattice in discrete time steps. Instead of taking into consideration every individual particle's position and velocity as in microscopic models, fluid flows are described by tracking the evolution of the particle distribution functions [44, 41]. The statistical treatment in the lattice Boltzmann method is necessary because of the large number of particles interacting in a fluid [47]. It leads to substantial gain in computational efficiency compared to the microscopic models (molecular dynamics). The macroscopic density, pressure and velocity can be obtained from these particle distribution functions [18], and it has been shown [42], [55] that the macroscopic properties of the fluid obtained through the lattice Boltzmann method is equivalent to solving the Navier-Stokes equations [4].

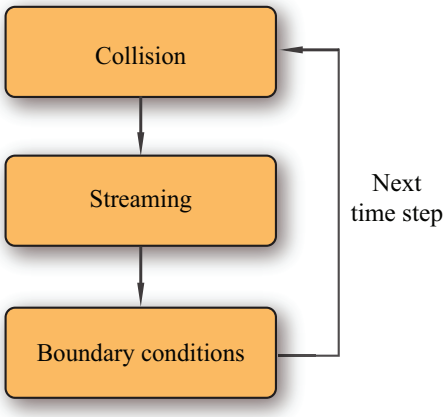


Figure 3.1: The three phases of the lattice Boltzmann method.

In the lattice Boltzmann method, fluids flows are simulated by the stream-

ing and collision of particles within the lattices, often together with some boundary conditions that must be fulfilled. As illustrated in Figure 3.1 these operations must be carried out for each discrete time step. The particles within the lattices can move within certain discrete velocities from one discrete lattice location to another. The discrete lattice locations correspond to volume elements that contain a collection of particles [18], and represents a position in space that holds either fluid or solid [35]. In the streaming phase, particles move to the nearest neighbor along their path of motion, as can be seen in Figure 3.2, where they collide with other arrived particles, as can be seen in Figure 3.3. The outcome of the collision is designed so that it is consistent with the conservation of mass, energy and momentum [45]. They are collision invariant. After each iteration, only the particle distribution changes, while the particle distribution function in the center of each lattice locations remains unchanged.

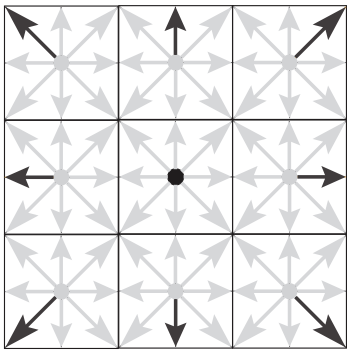


Figure 3.2: The streaming step: Particles spread out to their nearest nodes in the lattice [19].

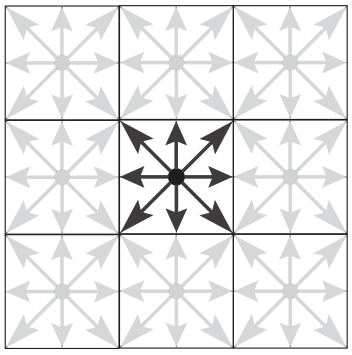


Figure 3.3: The collision step: Existing and entry particles are weighted [19].

3.2.1 Previous And Related Work

Historically, the lattice Boltzmann method is an outcome from the attempts to improve the Lattice Gas Cellular Automata (LGCA), even though the lattice Boltzmann method can be derived directly from the Boltzmann equation [27].

The first LGCA model named HPP was introduced in 1973 by Hardy, Pomeau and de Pazzis. In this model the lattice applied was square, and particles could not move diagonally. Because of this the model suffered from lack of rotational invariance [26]. In 1986, over 10 years later, the LGCA model named FHP was introduced by Frisch, Hasslacher, and Pomeua [23], who discovered the lattice symmetry. In this model the lattice applied was triangular and therefore did not suffer from lack of rotational invariance [55]. The main motivation for further development was to remove the static noise in the LGCA models, which makes computational precision difficult to achieve [8]. In 1988, two year later, McNamara and Zanetti introduced the lattice Boltzmann method [38], which completely removed the static noise found in the LGCA models, by replacing the Boolean representation of a particle by the particle distribution function. Further development and improvements where proposed by Chen[10] and Qian[42], with the use of the Bhatnagar-Gross-Krook (BGK) simplified collision operator.

Today, the lattice Boltzmann method has been applied on CPUs for fluid flows through porous media to determine the permeability of porous media [2, 24]. For a comprehensive overview of efficient implementations of the lattice Boltzmann method for CPUs we refer to [53], in view of the fact that the architecture of the GPU is quite different. The lattice Boltzmann method has been applied on GPUs using NVIDIA CUDA [25, 40, 50, 51]. Performance of lattice Boltzmann implementations is often measured in MLUPS (million lattice nodes updates per second). Tolke and Krafczyk [51], implemented the lattice Boltzmann method using the D3Q13 model on a NVIDIA GeForce 8800 Ultra card, which achieved the total of 592 MLUPS. Habich [25], implemented the lattice Boltzmann method using the D3Q19 model on a NVIDIA GeForce 8800 GTX card, which achieved the total of 250 MLUPS. Bailey, Myre, Walsh, Lilja, and Saar [40] implemented the lattice Boltzmann method using the D3Q19 model on a NVIDIA GeForce 8800 GTX card, which achieved the total of 300 MLUPS. The higher performance of Tolke and Krafczyk can to some extent be explained because the D3Q13 is a simpler lattice, compared to the D3Q19.

3.2.2 Fundamentals

In what follows, the starting point is the lattice Boltzmann equation with the Bhatnagar-Gross-Krook (BGK) simplified collision operator, that can be written as Equation 3.2.

$$\frac{\partial f}{\partial t} + e\nabla f = -\frac{1}{\tau}(f - f^{eq}) \tag{3.2}$$

where τ is the single relaxation time, and f^{eq} is the equilibrium distribution function, that can be written as Equation 3.3.

$$f^{eq} = \frac{\rho}{(2\pi RT)^{D/2}} exp\left[-\frac{(e-u)^2}{2RT}\right] \tag{3.3}$$

where R is the universal gas constant, D is the dimension of the space, ρ is the macroscopic density, u is the macroscopic velocity, and T is the macroscopic temperature [27].

In order to solve the Boltzmann equation numerically, the physical space is limited to a discrete lattice, only a discrete set of velocities are allowed, and the time is limited to discrete time steps. A widespread classification system used for the different lattices that exist is DaQb, where Da is the number of dimensions and Qb is the number of distinct discrete lattice velocities $\vec{e}_i$. In the lattice Boltzmann method, the underlying lattice must have enough symmetry to ensure isotropy, and typically lattices are D2Q9, D3Q13, D3Q15, and D3Q19.

In the lattice Boltzmann method, the time evolution of the particle distribution function is obtained, by solving the discrete Boltzmann equation, that can be written as Equation 3.4 [4].

$$f_i(\vec{x} + \vec{e}_i, t+1) - f_i(\vec{x}, t) = \Omega \tag{3.4}$$

where $\vec{e}_i$ are discrete lattice velocities, Ω is the collision operator, and $f_i(\vec{x}, t)$ is the discrete particle distribution function in the i direction. The simplified BGK collision operator is often used, that can be written as Equation 3.5 [42].

$$\Omega^{BGK} = -\frac{1}{\tau}\left(f_i(\vec{x}, t) - f_i^{eq}(\rho(\vec{x}, t), \vec{u})\right) \tag{3.5}$$

where τ is the single relaxation parameter, and $f_i^{eq}(\rho(\vec{x},t),\vec{u})$ is the equilibrium distribution functions in the i direction (also often called for the Maxwell-Boltzmann distribution function). The equilibrium distribution function can be written as Equation 3.6 [42].

$$f_i^{eq}(\rho(\vec{x},t),\vec{u}) = w_i\rho\left(1+\frac{3}{c^2}(\vec{e}_i\cdot\vec{u})+\frac{9}{2c^4}(\vec{e}_i\cdot\vec{u})^2-\frac{3}{2c^2}\vec{u}^2\right) \tag{3.6}$$

where c is equal to $\Delta x/\Delta t$, which are often normalized to 1, and w_i is weight factors that depends on the lattice model. The macroscopic kinematic viscosity ν of the fluids, can be found with Equation 3.7 [31].

$$\nu = \frac{2\tau-1}{6} \tag{3.7}$$

The macroscopic properties of the fluids can be computed from the particle distribution functions, such as the mass density $\rho(\vec{x},t)$, momentum $\rho(\vec{x},t)u(\vec{x},t)$ and velocity $\vec{u}(\vec{x},t)$ of a fluid particle as Equations 3.8, 3.9, and 3.10 [18].

$$\rho(\vec{x},t) = \sum_{i=0}^{q} f_i(\vec{x},t) \tag{3.8}$$

$$\rho(\vec{x},t)u(\vec{x},t) = \sum_{i=0}^{q} \vec{e}_i f_i(\vec{x},t) \tag{3.9}$$

$$\vec{u}(\vec{x},t) = \frac{1}{\rho(\vec{x},t)}\sum_{i=0}^{q} \vec{e}_i f_i(\vec{x},t) \tag{3.10}$$

where q is the number of distinct lattice velocities $\vec{e}_i$.

3.2.3 Boundary Conditions

The standard boundary condition applied at solid-fluid interfaces is the no-slip boundary condition (also called bounce back boundary condition), that can be written as Equation 3.11 [31].

$$f_i^{in}(\vec{x}+\vec{e}_i,t+1) = f_i^{out}(\vec{x},t) = f_i^{in}(\vec{x},t) \tag{3.11}$$

With this boundary condition applied, the particles close to solid boundaries do not move at all, resulting in zero velocity. The particles at the

solid-fluid interfaces are reflected [49], as illustrated in Figures 3.4 and 3.5. Periodic boundary condition is also common, and allows particles to be circulated within the fluid domain. With the periodic boundary conditions, outgoing particles at the exit boundaries will come back again into the fluid domain through the entry boundaries on the opposed side.

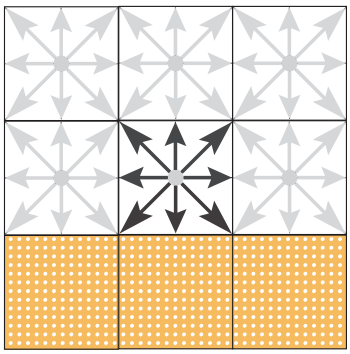


Figure 3.4: Bounce back boundary: Lattice node before streamin [49].

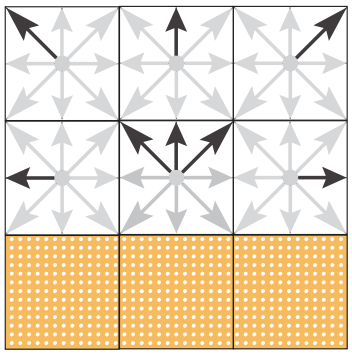


Figure 3.5: Bounce back boundary: Lattice node after streaming [49].

3.2.4 Basic Algorithm

Equations 3.4 and 3.5 can be split up into the two following Equations 3.12 and 3.13 [31].

$$f_i^{out}(\vec{x},t) = f_i^{in}(\vec{x},t) - \frac{1}{\tau}\left(f_i^{in}(\vec{x},t) - f_i^{eq}(\rho(\vec{x},t),\vec{u})\right) \tag{3.12}$$

$$f_i^{in}(\vec{x}+\vec{e}_i,t+1) = f_i^{out}(\vec{x},t) \tag{3.13}$$

where f_i^{out} represents the particle distribution value after collision, and f_i^{in} is the value after both the streaming and collision operations are finished. Equations 3.12 and 3.13 implement the collide-stream system, also known as the push method, described by [31]. The basic algorithm using the collide-stream system is illustrated in Figure 3.6.

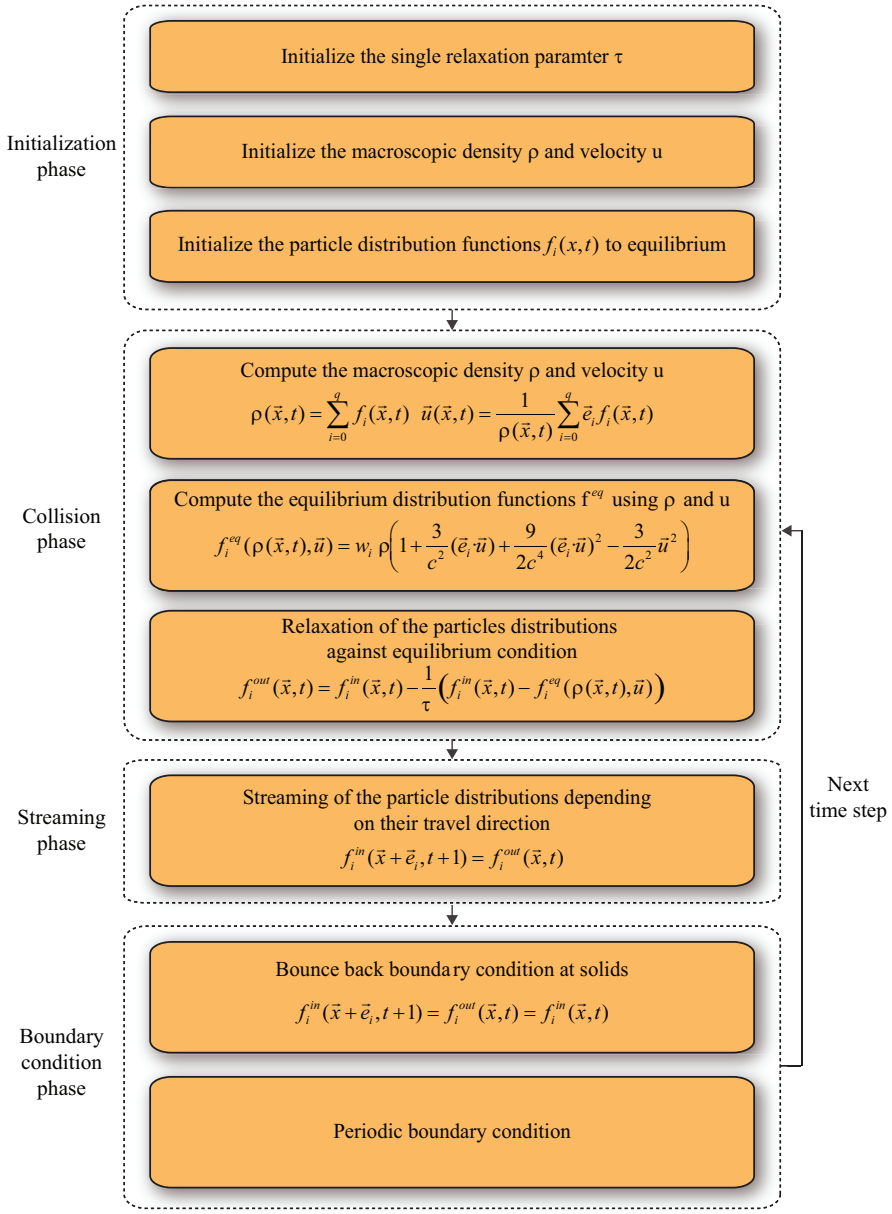


Figure 3.6: Basic algorithm of the lattice Boltzmann method. Based on [31].

3.3 Porous Rocks

This section about porous rocks is taken from earlier work made by Eirik Ola Aksnes and Henrik Hesland [3], and used with some changes.

The word petroleum, meaning rock oil, is important to humankind as it is the primary source of energy. Petroleum does not typically lie in huge pools or are found in underground rivers, but refers to the naturally occurring hydrocarbons that are found in porous rock formations beneath the surface of the earth. A petroleum reservoir or an oil and gas reservoir is an underground accumulation of oil and gas that is located within pore spaces of porous rocks. Not all rocks are capable of holding oil and gas. Reservoir rocks, which are capable of holding oil and gas, are characterized by having sufficient porosity and permeability, meaning that is has sufficient storage capacity for oil and gas, and has the ability to transmit fluids. The challenge is how to extract the oil and gas out from the porous rocks, and it is vital for the oil industry to analyze the petrophysical properties of reservoir rocks to gain improved understanding of oil production. Figure 3.7 illustrates the influence the pore geometry of porous rocks has on how the fluid will flow inside the porous rocks. In Figure 3.7 fluid will flow more slowly through the left rock, because of the lack of interconnected pore spaces within the rock, as the pore spaces are isolated from each other, and that the existing pore spaces are narrow [22].

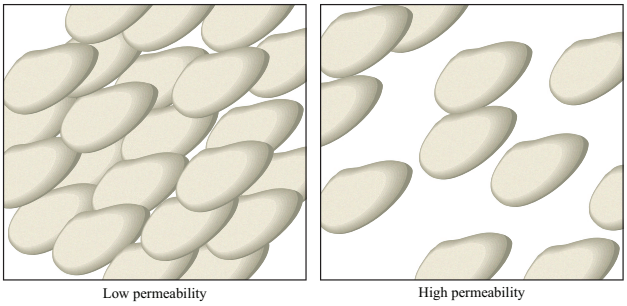


Figure 3.7: The left rock has low permeability, in contrast to the right rock, which allows fluid to flow easily.

3.3.1 Porosity

The porosity is defined as Equation 3.14 [8].

$$\phi = \frac{V_p}{V_t} \tag{3.14}$$

where V_p is the volume of pore space and V_t is total volume of the porous medium. Some actual material porosities are listed in Table 3.1.

Table 3.1: Typical porosity of some representative real materials. Taken from [8].

Material	Porosity
Sandstone	10-20 %
Clay	45-55 %
Gravel	30-40 %
Soils	50-60 %
Sand	30-40 %

3.3.2 Permeability

Darcy's law for the flow of fluids through porous media can be written as Equation 3.15 [24].

$$q = -\frac{k}{\rho\nu}\frac{\Delta P}{L} \tag{3.15}$$

where q is the volumetric fluid flux through the porous media, k is the permeability of the porous medium, $\frac{\Delta P}{L}$ is the total pressure drop along the sample length L, ρ is the fluid density, and ν is the fluid kinematic viscosity. In porous media, fluid will flow only through pores capable of transporting fluid, therefore the volumetric flux q is considered as Equation 3.16 [24].

$$q = \bar{u}\phi \tag{3.16}$$

where $\bar{u}$ is the average velocity of the fluid and ϕ is the porosity of the porous medium. The permeability can be found with Equation 3.17.

$$k = \frac{\bar{u}\phi\rho\nu}{\frac{\Delta P}{L}} \tag{3.17}$$

Chapter 4

Implementations

This chapter describes how we implemented the lattice Boltzmann method presented in the previous chapter on modern GPU hardware. Our simulation model is baded on [31]. On order to better analyze our results, both parallel CPU and GPU implementations of the lattice Boltzmann method were developed.

In particular, Section 4.1, describes the target platforms, libraries, and languages used for both our implementations. Section 4.2 describes support for porous rocks visualization, and of the Marching Cubes algorithm used to generate three dimensional models of porous rocks. Section 4.3 describes the simulation model used in both our implementations. Section 4.4 describes an approach used to reduce the rounding error in the simulation model used. Section 4.5 describes the data structure used to store the particle distribution functions for both our implementations. Sections 4.6 and 4.7 describe the CPU and GPU implementations of the lattice Boltzmann method, with optimization guidelines for high performance.

4.1 Platforms, Libraries, and Languages

Both the CPU and GPU implementation are programmed in C++. For the GPU implementation, NVIDIA GPUs are used to accelerate the lattice Boltzmann method, due to the well developed hardware and software support from NVIDIA GPUs for general purpose computation. With the latest generations of NVIDIA GPUs that supports the NVIDIA CUDA program-

ming model, several unnecessary difficulties with the traditional way of doing GPGPU are eliminated. The NVIDIA CUDA[1] are a natural choice, since it exposes the hardware functions of the NVIDIA GPUs, making it possible to target the programmable units directly for improved control. The OpenCL framework are also considered due to it's heterogeneous platform support, but NVIDIA CUDA are used, because OpenCL is still new. For graphics rendering, NVIDIA CUDA supports both the Direct3D and OpenGL graphics library. Since the Direct3D graphics library is only supported on the Microsoft Windows operating systems, the OpenGL[2] graphics library are used for platform independence. To access the expansions to the OpenGL graphics library for efficient graphics rendering, the *OpenGL Extension Wrangler Library*[3] (GLEW) are used. The *OpenGL Utility Toolkit* (GLUT) are used to handle mouse, keyboard, and window management. The OpenGL Shading Language (GLSL) are used to be able to use the programmable pipeline with vertex and fragment shaders. The CPU implementation also uses the OpenGL, GLEW, GLUT and GLSL library.

4.2 Visualization

Visualization of the pore geometry and how fluids flow inside the pore geometry of rocks are implemented for both our implementations, as illustrated in Figure 4.1. The yellow color indicates high velocity and the black color indicates low velocity. Visualization are implemented to ease debugging during development, and for visual analysis of the pore geometry of rocks. Since the pore geometry of rocks, how the pore spaces within the rocks are interconnected, and the size of the pore spaces have major influence on permeability [22].

4.2.1 Graphics Rendering

Vertex Buffer Object (VBO), which is an OpenGL extension provided through GLEW are used for graphics rendering. The traditional way of rendering geometric data is to transfer single data elements to the memory of the GPU. With VBO, it is possible to upload multiple data elements to the memory

[1] www.nvidia.com/cuda
[2] http://www.opengl.org/
[3] http://glew.sourceforge.net/

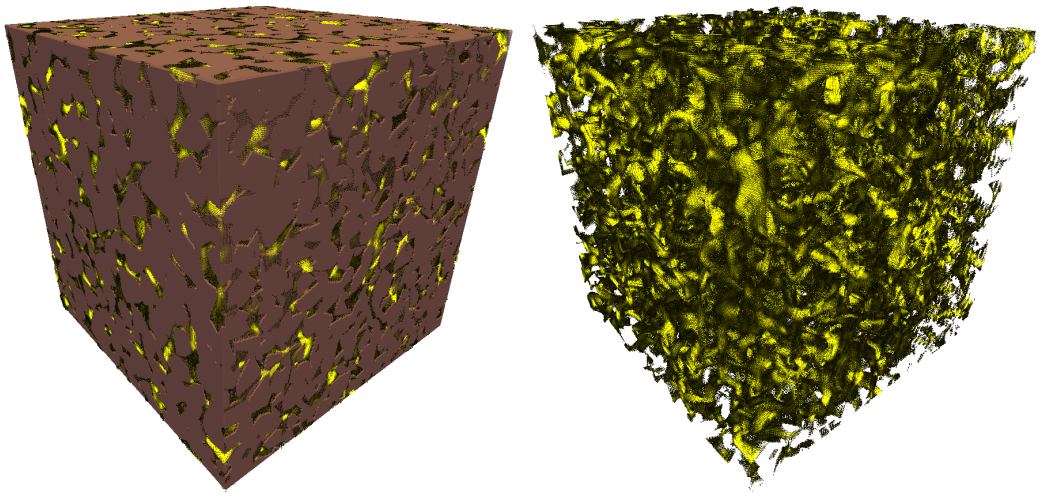


Figure 4.1: Visualization of porous rock left) the pore geometry right) how the fluids flow inside the pore geometry.

of the GPU simultaneously. It is also possible to render geometric data directly from the memory of the GPU, without transferring the data from the system memory to the memory of the GPU [15]. This can offer significant performance increase, particularly if the geometric data is static and does not need to be updated for every frame. The models of porous rocks used in this thesis only require to be created once at startup, and are therefore reused and directly rendered from the memory of the GPU. This increases the graphics rendering performance significantly.

4.2.2 Porous Rock Visualization

The Marching Cubes algorithm are used to generate the three dimensional models of the internal pore geometry of porous rocks, such as the rock illustrated in Figure 4.1, taken from earlier work made by Eirik Ola Aksnes and Henrik Hesland [3]. The Marching Cubes algorithm takes eight points at a time from a density field, forming cubes. Within these cubes, triangles are created. The density values of the eight points decide how the triangles will be generated. The points are one by one compared to a threshold value (also often called isovalue), where each points makes a bit value representing if it is inside or outside the volume. This creates an 8 bit lookup index. The Marching Cubes algorithm uses two lookup tables. The first is the edge

table that holds an overview (256 different combinations) over which of the 12 edges we have to interpolate. The second is the triangle table that determines the quantity (maximum five) and how the triangles inside each cube should be drawn for correct representation of the isosurface. The Marching Cubes algorithm repeats the treatment of cubes until the entire density field is treated. Figure 4.2 shows the main phases of the Marching Cubes algorithm used, with the value of the corner 3 above the selected isovalue.

4.2.3 Fluid Flow Visualization

In fluid simulations, velocity fields are represented as vector fields. Visualization of vector fields is used to describe the motion of fluids inside the pore geometry of porous rocks. To visualize vector fields, shader code and an NVIDIA CUDA kernel provided by Robin Eidissen were used, with some modifications [20]. For each lattice node, a single line segment is created, by representing the position and direction using two four-component vectors. The length of line segments represents flow velocity. The first three components of vectors are x, y, and z coordinates. The last component of vectors holds the flow velocity, which is used to determine the color of the line segment. A GLSL shader colors the line segments based on the flow velocity, with yellow color indicating high velocity and black color indicating low velocity.

4.3 Simulation Model

The lattice Boltzmann method with the simplified Bhatnagar-Gross-Krook collision operator is used as simulation model for the CPU and GPU implementations [42]. In this thesis, large portions of the simulations will be between solid-fluid interfaces, inside the complex geometries of porous rocks. One of the most cited benefits of the lattice Boltzmann method is that it can handle complex flow geometries without significant drop in computational speed and efficiency [46] and [45]. The method is therefore suitable for the simulations of fluid flows through the complex geometries of porous rocks (although simulating fluid flow through porous rocks is still very complex, including the flow of single-phase fluid).

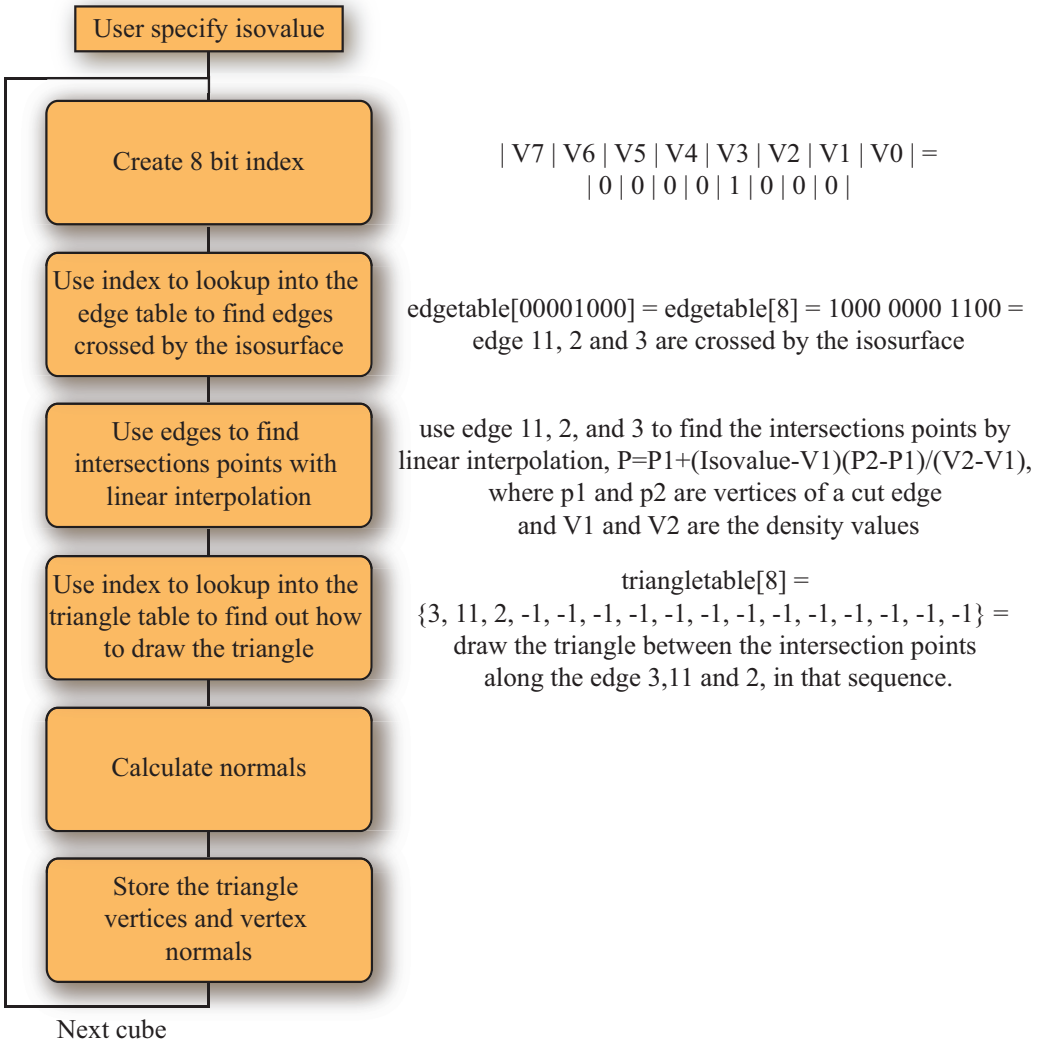


Figure 4.2: The Marching Cubes algorithm used. Based on [7] and [34].

4.3.1 Memory Usage

The simulation model makes use of the D3Q19 lattice, which is a three dimensional lattice with 19 discrete lattice velocities. Appendix A shows the configuration of the D3Q19 lattice used. Relatively large models of the internal pore geometry of porous rocks are often required to get accurate estimates

of their permeability. For every node in the lattice, implementations using the D3Q19 model often stores and uses 19 values for the particle distribution functions and 19 temporary values for the streaming phase, so that the particle distribution functions are not overwritten during the exchange phase between neighbor lattice nodes. Table 4.1 lists the memory consumption of the lattice Boltzmann method using temporary storage with varying lattice sizes, and with single and double floating-point precision. One can see how the memory consumption becomes gigabytes of memory with large lattice sizes, making the growth in memory requirements with lattice size visible.

Table 4.1: Memory usage D3Q19 model with temporary storage.

Lattice size	Single precision	Double precision
32^3	4 MB	10 MB
64^3	38 MB	76 MB
128^3	304 MB	608 MB
256^3	2.4 GB	4.8 GB
512^3	19.4 GB	38.8 GB

Since the lattice Boltzmann method uses a lot of memory resources, it is important to allocate and use the minimal amount of memory possible. This is particularly important for the GPU implementation, since memory capacity is limited and memory cannot simply be added. Therefore instead of duplicating the particle distribution functions to temporary storage in the streaming phase, another approach described by Latt [32] is used in both the implementations, where the source and destination particle distribution functions are instead swapped between neighbor lattice nodes. This approach reduces the memory requirements by 50 %, compared to using temporary storage.

4.3.2 Simulation Model Details

The details of how the lattice Boltzmann method is realized in our CPU and GPU implementation are discussed here. Figure 4.4 shows the main phases

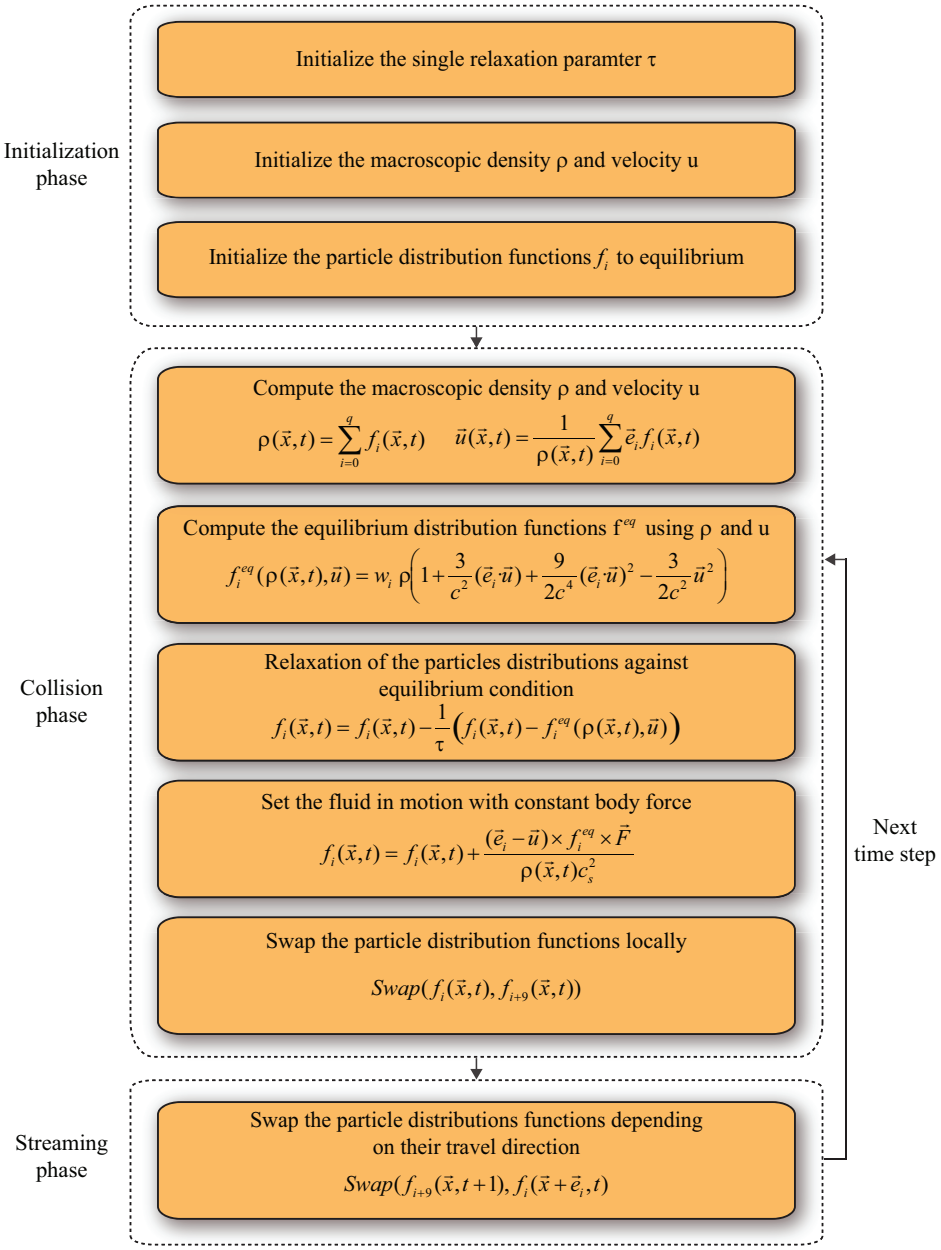


Figure 4.3: The main phases of the simulation model used. Based on [31].

of the simulation model. In the simulation model, the collisions of particles are evaluated first, and then particles streams to the lattice neighbors along the discrete lattice velocities.

Two types of boundary conditions are implemented: the standard bounce back boundary condition to handle solid-fluid interfaces, and periodic boundary condition to allow fluids to be circulated within the fluid domain. The periodic boundary condition is built into the streaming phase, and the bounce back boundary condition is built into the streaming and collision phase. The bounce back boundary condition emerges from swapping particle distribution functions between neighbors, because neighbors in the lattices only exchange particle distribution functions with other neighbors that are fluid elements. The different phases of the simulation model will now be discussed further, accompany by pseudo-code.

Initialization Phase

A common practice in the initialization phase is to initialize the lattice nodes using the equilibrium distribution function with the wanted initial macroscopic density ρ and macroscopic velocity $\vec{u}$. In this thesis, when estimating the permeability of porous rocks, the steady state of the velocity fields is important. Therefore is the lattice initialized with the density ρ equal to 1.0 and the velocity $\vec{u}$ equal to 0.0 for each node in the lattice that is fluid element (Algorithm 4: Line 7). The velocity $\vec{u}$ becomes 0.0 with each node in the lattice initialized with the density ρ equal to 1.0.

Collision Phase

The collision phase is, in contrast to the streaming phase, completely local, but it is also the most computational demanding phase. In the collision phase, the macroscopic density ρ (Algorithm 5: Line 7) and the macroscopic velocity $\vec{u}$ (Algorithm 5: Line 8-10) need to be computed, together with the equilibrium distribution functions that need to be solved (Algorithm 5: Line 13), and the relaxation of the particles distributions against equilibrium condition (Algorithm 5: Line14). In the collision phase, the fluids are set in motion, with some constant external force that is added to every fluid

Algorithm 4 Pseudo code of the initialization phase.

```
1: for z = 0 to d_z do
2:    for y = 0 to d_y do
3:       for x = 0 to d_x do
4:          current ← x + y × d_x + z × d_x × d_y
5:          if f_i[current] is fluid then
6:             for i = 0 to 18 do
7:                f_i[current] ← 1.0
8:             end for
9:          end if
10:      end for
11:   end for
12: end for
```

element in the lattice (Algorithm 5: Line 15), using equation 4.1.

$$f_i(\vec{x}, t) = f_i(\vec{x}, t) + \frac{(\vec{e}_i - \vec{u}) \times f_i^{eq} \times \vec{F}}{\rho(\vec{x}, t) c_s^2} \tag{4.1}$$

$\vec{F}$ is the external force, and c_s^2 is the speed of sound, given the value of $\sqrt{(1/3)}$ [28]. The external force will add some constant value to the particle distribution functions moving along the fluid flow direction, and subtracting a corresponding value from the particle distribution functions moving exactly the opposite direction [24]. At the end in the collision phase, the particle distribution functions are swapped locally (Algorithm 5: Line 17).

The collision phase is responsible for approximating the particle collisions that happen in real fluids. The rate of change toward equilibrium is controlled by the relaxation time τ, and determines the viscosity [4] of the fluids [45]. In the lattice Boltzmann method, the relaxation time τ is normally chosen between $0.51 \leq \tau \leq 2.5$ [49], due to the requirements of numerical stability for the simulations [24].

[4] Viscosity is the measure of a fluid's resistance to flow.

Algorithm 5 Pseudo code of the collision phase.

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

Streaming Phase

The streaming phase is responsible for the movements of particle distribution functions within the lattice. In the streaming phase, lattice nodes need

to interact with neighbors in the lattice to exchange particle distribution functions. In the simulation model, this is mostly swapping of particle distribution functions between memory locations (Algorithm 6: Line 9). Before the exchange of particle distribution functions, the lattice nodes need to find the neighbors in the lattice (Algorithm 6: Line 7).

Algorithm 6 Pseudo code of the streaming phase.

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

To find the coordinates to neigbors in the lattice, the coordinates of the current lattice nodes are used (Algorithm 7: Line 1-3).

$$\begin{aligned} neighbor_x &= current_x + e_{ix} \\ neighbor_y &= current_y + e_{iy} \\ neighbor_z &= current_z + e_{iz} \end{aligned} \tag{4.2}$$

In our two implementations, the periodic boundary condition is built into the finding of neighbors, by allowing or denying the lattice nodes to interact with the opposite side of the lattice when they come close to the borders (Algorithm 7: Line 5-27).

Algorithm 7 Pseudo code for the find neigbhbor algorithm.

getNeighbor(x,y,z,i,p_x,p_y,p_z)

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

4.3.3 Calculate Permeability

The approach used to calculate the permeability of porous rocks and most of the information found in this section is taken from [24].

The details of how the calculation of the permeability of porous rocks is

realized in our CPU and GPU implementation is discussed in this section. Figure 4.4 shows the expansion of the simulation model for the calculation of permeability of porous rocks.

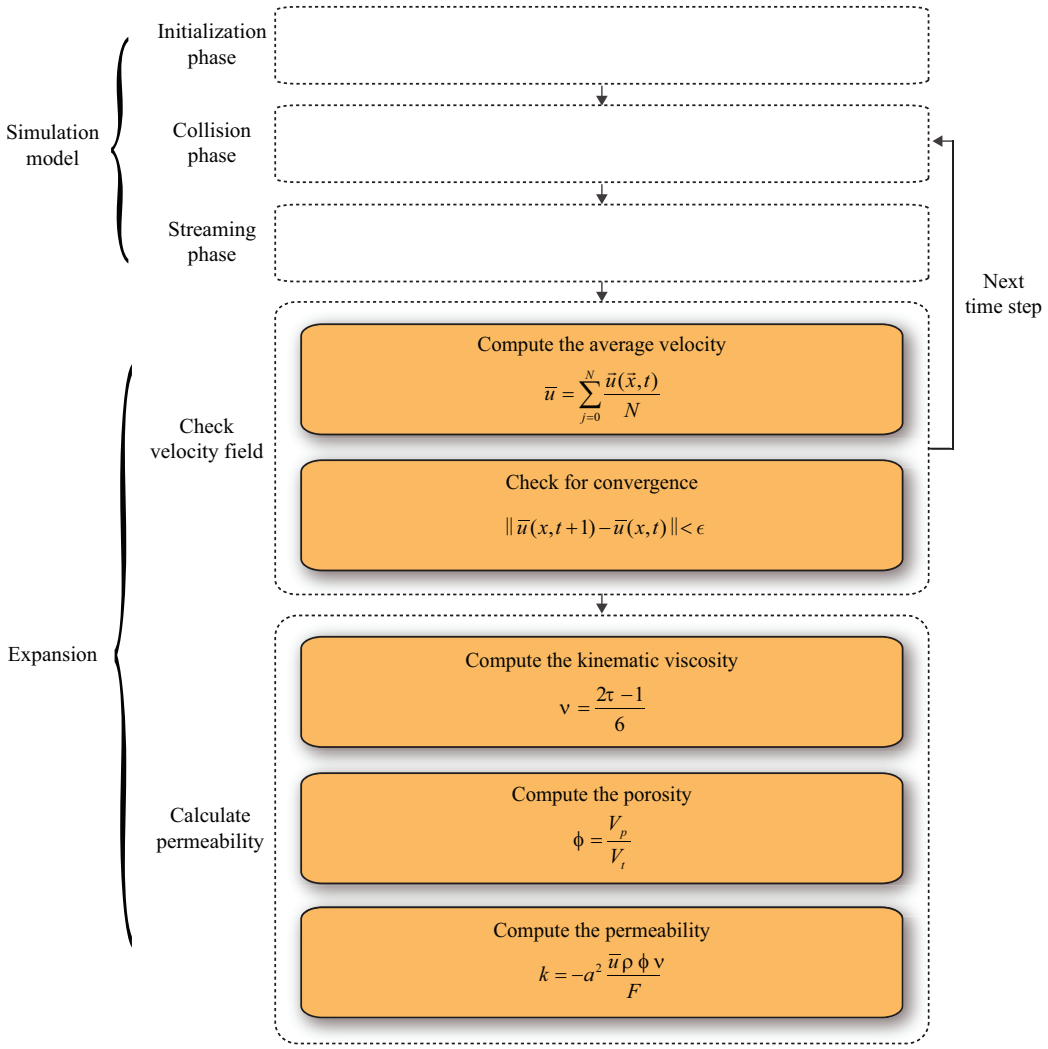


Figure 4.4: Expansion of the simulation model for permeability calculations.

The permeability of the porous rocks is obtained directly from the generated velocity fields of the lattice Boltzmann method, together with using

Darcy's law for the flow of fluids through porous media [24]. The fluid flow is driven by some external force in the simulation model, but it could also be driven by pressure on the boundaries. The external force is expected to give the same change in momentum as the true $\frac{\Delta P}{L}$, which is the total pressure drop along the sample length L. By adapting Equation 3.17 the permeability of porous rocks are directly obtained from the lattice Boltzmann simulations, using Equation 4.3 [24].

$$k = -a^2 \frac{\bar{u}\rho\phi\nu}{F} \tag{4.3}$$

where a is the node resolution equivalent to the lattice spacing. Driven by some external force, the permeability is always obtained when the velocity field is at steady state. The simulations is considered to have converged if the change of the average velocity meets the condition described in Equation 4.4.

$$|\bar{u}(x, t+1) - \bar{u}(x, t)| < \epsilon \tag{4.4}$$

where the convergence threshold ϵ is chosen to be 10^{-9}. The average velocity is computed using Equation 4.5.

$$\bar{u} = \sum_{j=0}^{N} \frac{\vec{u}(\vec{x}, t)}{N} = \sum_{j=0}^{N} \frac{\sum_{i=0}^{q} \vec{e}_i f_i(\vec{x}, t)/\rho(\vec{x}, t)}{N} \tag{4.5}$$

where N is the number of fluid elements within the lattice.

Figure 4.5 illustrates the two types of boundary conditions that are used in the calculation of the permeability of porous rocks. In the calculations of the permeability, the boundaries parallel to the flow direction are made solid, and with bounce back boundary conditions. The entry and exit boundaries are applied with periodic boundary conditions. In order for the periodic boundary condition to be correct in the simulations, it is common practice to add some extra empty layers at both the entry and exit boundaries [24].

Only the interconnected (percolated) pore spaces within porous rocks transport fluid. In the expansion of the simulation model for porosity calculations, both the disconnected and interconnected pore spaces are considered. There is possible to identify the percolating pore spaces within porous rocks with the Hoshen-Kopelman algorithm [24].

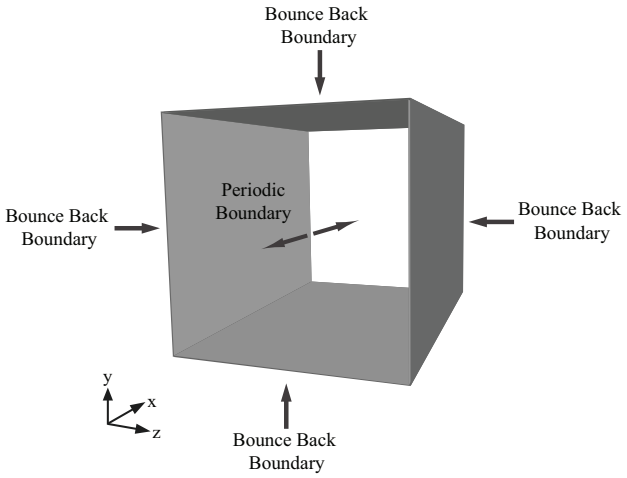


Figure 4.5: Configurations of the boundaries in permeability calculations.

4.4 Floating-point Precision

The Institute of Electrical and Electronics Engineers (IEEE) standard 754, defines a standard for floating-point arithmetic. The standard specifies the format of the numbers, the results of the basic floating-point operations and comparisons, and the rounding to nearest rule among many other things [29]. Since computers represent real numbers with a limited number of bits, approximations and rounding errors occur [52]. Large rounding errors occur between two very different numbers that are added or subtracted [29]. In the collision phase of the simulation model, the equilibrium distribution function needs to be computed with a mixture of large and small numbers that often leads to large amount of rounding error. It was necessary to reduce the rounding error in the simulation model when using single floating-point precision, to get precise permeability of porous rocks. The approach used to reduce the rounding errors used in the simulation model is taken from [11] [5]. It was left out from the previous descriptions and pseudo codes of the simulation model, due to readability. In this approach the simplified Bhatnagar-Gross-Krook collision operator becomes as Equation 4.6.

$$h_i^{out} = h_i + \omega(h_i^{eq} - h_i) \tag{4.6}$$

where $h_i^{out} = f_i^{out} - w_i p_0$, $h_i = f_i - w_i p_0$, and $h_i^{eq} - w_i p_0$. The equilibrium

[5] www.lbmethod.org

distribution function becomes as Equation 4.7.

$$h_i^{eq} = w_i \Delta p + w_i(p_o + \Delta p)\left(\frac{3}{c^2}(\vec{e}_i \cdot \vec{u}) + \frac{9}{2c^4}(\vec{e}_i \cdot \vec{u})^2 - \frac{3}{2c^2}\vec{u}^2\right) \tag{4.7}$$

The macroscopic density ρ and the macroscopic velocity $\vec{u}$ becomes as Equations 4.8 and 4.10.

$$\rho = p_0 + \Delta p \tag{4.8}$$

$$\Delta p = \sum_{i=0}^{q} h_i \tag{4.9}$$

$$\vec{u} = \frac{\sum_{i=0}^{q} h_i \vec{e}_i}{p_0 + \Delta p} \tag{4.10}$$

4.5 Data Structure

A structure-of-arrays are used to store the particle distribution functions for both our implementations. There are 19 arrays in the structure, since there are 19 discrete directions in the D3Q19 lattice, as illustrated in Figure 4.6. Each array stores one discrete direction of the particle distributions functions.

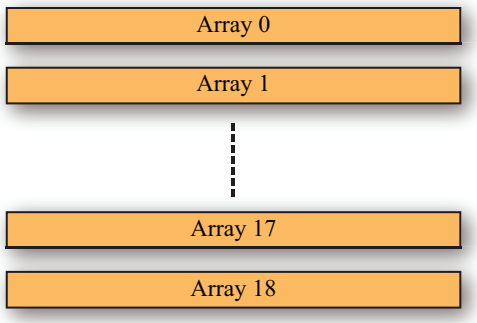


Figure 4.6: Structure-of-arrays.

4.6 CPU Implementation

Before describing the CPU implementation, we will present some optimization guidelines for high performance.

4.6.1 Optimizations Guidelines

This section is taken from earlier work made by Eirik Ola Aksnes and Henrik Hesland [3], and used with only some minor changes.

There are some general optimization techniques which programmers should be acquainted with for obtaning the highest possible performance. A very important technique for high performance is to avoid cache misses, since most modern CPUs uses fast cache memory to reduce average time to access main memory. The CPU will first check for the data needed in the cache memory, and if the data is not there, it must wait for the data to be fetched from the much slower main memory. Many programs are waiting for data to be fetched from main memory, and therefore waste much of their execution time. Caches rely on both spatial and temporal locality, items near a used item might be used soon, and recently used items will likely be used again in the near feature [30]. Data structures must therefore be set up so that the main memory is accessed in contiguous memory segments, and data elements need to be reused in a loop as frequently as possible [33]. Loop optimization techniques can have huge effect on cache performance, since most of the execution time in scientific applications is used on loops. Matrix multiplication is one example that can gain performance if the sequence of for loops is correctly arranged, by allowing an optimal data array access pattern. If the data needed is in continuous memory, the next elements needed are more likely to be in the cache. Cache locality can be improved by dividing the matrix into smaller sub matrixes, and chosing the optimal block size that will fit into cache. The size of the sub matrix will be system dependent, since the different systems have different cache size.

Another feature of today's processor architectures is the long execution pipeline, which offers significant execution speedups when kept full. By pipelining, several instructions can be overlapped in parallel. Branch predication is used to ensure that the pipeline is kept full, by guessing which way the branch is most likely to go. It has significant impact on performance, by

letting processors fetch and start instructions without waiting for the branch to be determined. A conditional branch is a branch instruction that can be either taken or not taken. If the processor makes the wrong guess in a conditional branch, the pipeline needs to be flushed, and all computation before the branch point becomes unnecessary. It can therefore be advantageous to eliminate such conditional branches in performance critical code. One technique that can be used to eliminate branches and data dependencies is to unroll loops [1]. One should also try to reduce the use of if-statements as much as possible inside the inner loops.

4.6.2 Details

The CPU implementation is similar to the pseudo code described in Section 4.7.3. It uses only one processor core, and are made to evaluate the computational exactness and performance of the GPU implementation. Algorithm 8 shows pseudo code of the CPU implementation. In the CPU implementation the lattice nodes are accessed sequential.

Algorithm 8 CPU implementation pseudo code.

```
1: Initialize lattice
2: Load porous dataset
3: Set boundaries on porous dataset
4: while new time step is needed do
5:    for all n nodes in lattice do
6:       n.collide()
7:       n.stream()
8:    end for
9:    Compute the average velocity
10:   Check for convergence
11: end while
12: Compute the kinematic viscosity
13: Compute the porosity
14: Compute the permeability
```

4.7 GPU Implementation

Before we describing the GPU implementation, we present some optimization guidelines for high performance.

4.7.1 Optimizations Guidelines

This section is taken from earlier work made by Eirik Ola Aksnes and Henrik Hesland [3], and used with only some minor changes.

For high performance of applications on NVIDIA GPUs using NVIDIA CUDA, the knowledge of the hardware architecture is important. The primary performance element of the GPU is the large number of computation cores. To exploit this, a massive multithreaded program should be implemented. The number of thread blocks used simultaneously on a *Streaming Multiprocessor* (SM) is restricted by the number of registers, shared memory, maximum number of active threads per SM, and number of thread blocks pr SM at a time should therefore be configured carefully in proportion to these. One of the problems with a GPU implementation is the communication latency. For computations, there should be as little communication as possible between the CPU and GPU, and often some time can be saved by doing the same computation several times, before loading the answers between the CPU and GPU. The massive parallelization of threads is also important for hiding latency. A modern GPU contains several types of memory, with differnt latiencies. To reduce bandwidth usage, it is recommended to use shared memory where it is possible. Shared memory is divided into banks, where access to the same bank only can be done sequentially, but access to different banks can be done in parallel. The threads should therefore be grouped to avoid this memory conflict. Another function that should be used as little as possible is synchronization. This can cause many threads to wait a long time for another thread to finish up, and can slow down the computation significantly.

4.7.2 Profiling

Most of the information found in this section is based on NVIDIA's programming guide for NVIDIA CUDA [12].

```
code {
  name =
  __globfunc__Z23streamAndSwapGPU_kernel14cudaPitchedPtr3PDFjiii
  lmem = 0
  smem = 124
  reg = 9
}
```

Figure 4.7: Section from Cubin file.

Several helpful profiling tools are available to gather information about the performance critical parts of the GPU kernels to achieve high performance.

NVIDIA CUDA Visual Profiler

With the NVIDIA CUDA Visual Profiler, it is possible to generate visual profiles of the kernel's resource usage during execution. The NVIDIA CUDA Visual Profiler is capable of giving precise statistics of several performance critical aspects. Some of the most useful statistics for high performance are [14]:

- Number of coalesced global memory loads and stores
- Number of non-coalesced global memory loads and stores
- Number of local memory loads and stores
- Kernel occupancy
- Number of divergent branches
- Number of warps serialized

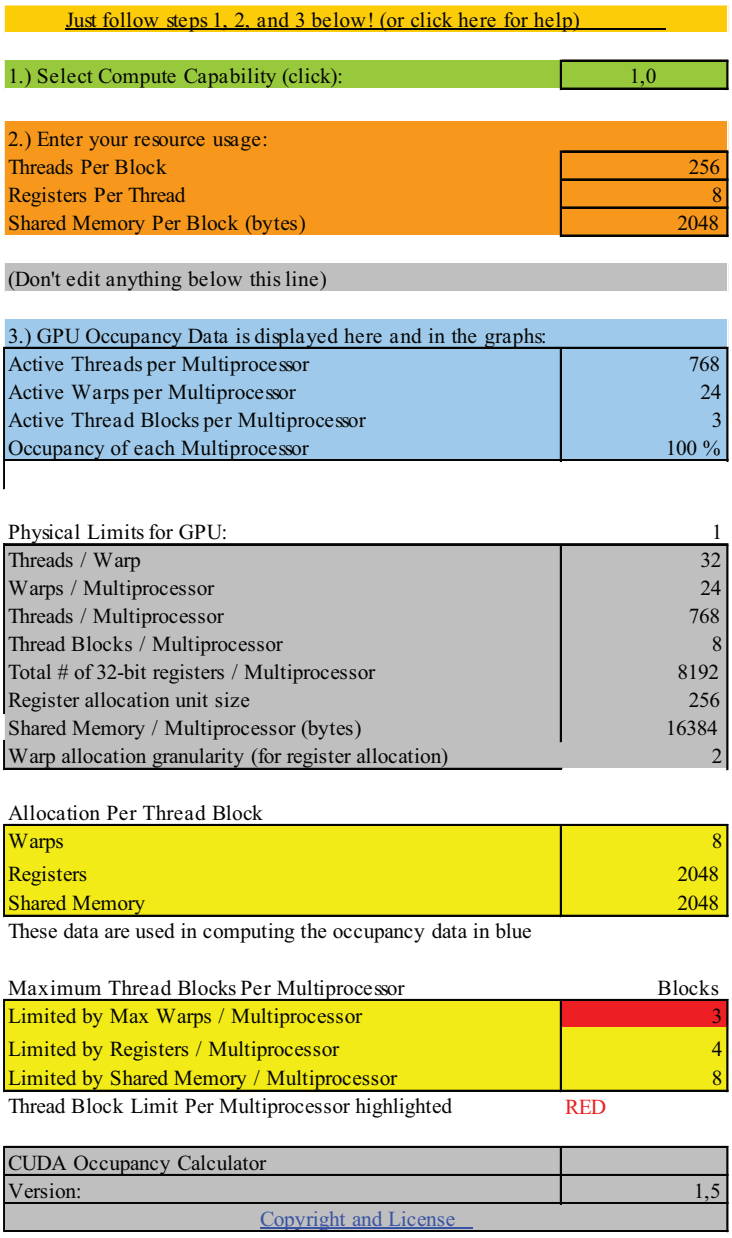
Cubin Files

By adding the extra parameter -cubin to the compiler, there is possible to generate so-called Cubin files from the NVIDIA CUDA source files. The generated files can be opened in regular text editors, and provide helpful information about kernel resource usage of particularly interest are the number of registers, shared memory and local memory used per thread [16, 12]. Figure 4.7 shows an example of section found in generated Cubin file.

In the Cubin file the lmen is the amount of local memory, smen is the amount of shared memory, and reg is the number of registers used per thread in the kernel streamAndSwapGPU_kernel.

NVIDIA CUDA Occupancy Calculator

The NVIDIA CUDA occupancy calculator is a spreadsheet provided by NVIDIA, which can be used to calculate the multiprocessor occupancy. The multiprocessor occupancy is the ratio of active warps per multiprocessor to the maximum number of active warps. The NVIDIA Quadro FX 5800 used in this thesis contains 16384 registers per streaming multiprocessor, and 16 KB of shared memory shared among all threads during execution. With the NVIDIA CUDA occupancy calculator, is possible to determine the maximum number of active threads given the limited number of registers and shared memory available. Higher multiprocessor occupancy results in more warps active, which can be used to hide the global memory latency [16, 12]. The NVIDIA CUDA occupancy calculator takes the number of registers, amount of shared memory and local memory used per thread in your kernel as input, which can be seen in the orange field of Figure 4.8. This information can be found within generated Cubin files. To achieve the highest possible multiprocessor occupancy, one must use the minimal amount of shared memory and registers. The NVIDIA CUDA occupancy calculator provides three graphs showing the changes in occupancy of kernels, due to changes in block size, register count, and shared memory usage, as illustrated in Figure 4.9.



Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	1,0

2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	8
Shared Memory Per Block (bytes)	2048

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	768
Active Warps per Multiprocessor	24
Active Thread Blocks per Multiprocessor	3
Occupancy of each Multiprocessor	100 %

Physical Limits for GPU:	1
Threads / Warp	32
Warps / Multiprocessor	24
Threads / Multiprocessor	768
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	8192
Register allocation unit size	256
Shared Memory / Multiprocessor (bytes)	16384
Warp allocation granularity (for register allocation)	2

Allocation Per Thread Block	
Warps	8
Registers	2048
Shared Memory	2048

These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	3
Limited by Registers / Multiprocessor	4
Limited by Shared Memory / Multiprocessor	8

Thread Block Limit Per Multiprocessor highlighted RED

CUDA Occupancy Calculator	
Version:	1,5
Copyright and License	

Figure 4.8: NVIDIA CUDA occupancy calculator.

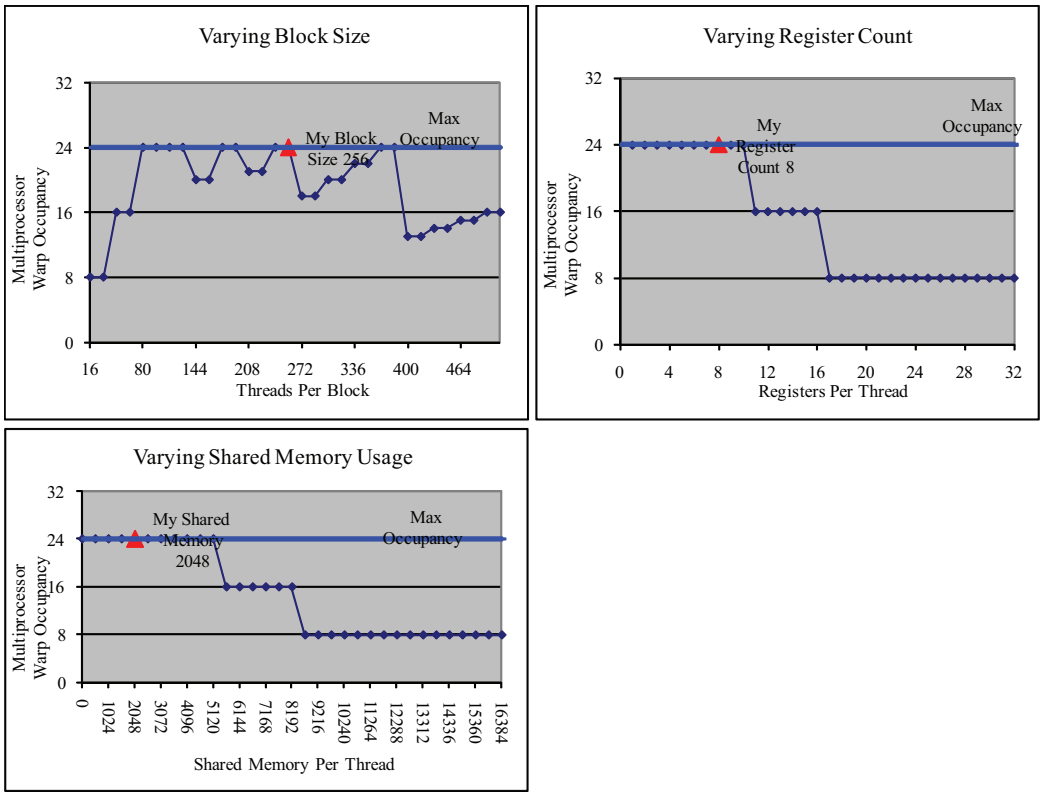


Figure 4.9: NVIDIA CUDA occupancy calculator.

4.7.3 Details

In the GPU implementation, every thread created during execution is responsible for performing the collision and streaming for a single lattice node. Figure 4.10 illustrates this one-to-one mapping between threads and lattice nodes.

To get high utilization of global memory bandwidth, the access pattern to the global memory must be correctly aligned to achieve coalescing. A structure-of-arrays has been proven by Habich [25] to be useful to achieve coalescing. Threads access the arrays in contiguous memory segments to obtain coalescing. This enables efficient reading and writing of particle distribution functions. Each array contains one discrete direction of the particle distributions functions. Arrays in the structure are three-dimensional, allocated as

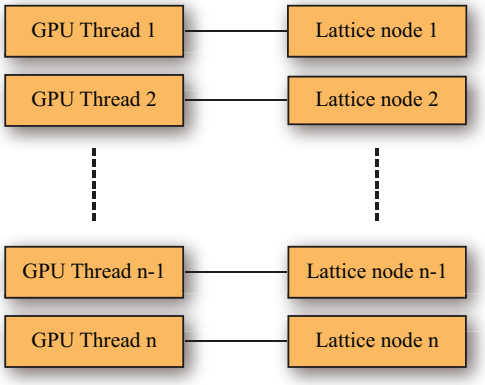


Figure 4.10: One-to-one mapping between threads and lattice nodes.

contiguous memory on the device using cudaMalloc3D. This function takes the width, height, and depth of simulations as input, and pads the allocation to meet the alignment requirements to achieve coalescing. The function returns a pitch (or stride), which is the width in bytes of the allocation. This pitch is used in the access of array elements [12].

The configuration of grids and thread blocks used to get coalesced global memory access, and to have simulations with large lattices are shown in the Algorithm 9 and Figure 4.12. Kernels execute as two dimensional grids of thread blocks, with the width and height of the grids equal to the simulation size in the y- and z- direction. Two-dimensional grids were necessary in order to simulate large lattices, due to NVIDIA's restriction of the maximum number of threads blocks to be 65535 in one direction of the grid. Figure 4.11 shows the number of thread blocks with varying cubic lattice sizes, and the largest lattice possible with one dimensional grid.

Thread blocks are one dimensional, with the number of threads equal to the simulation size in the x-direction. This configuration of thread blocks results in a maximum simulation size of 512 in the x-direction, due to NVIDIA's restriction of the maximum number of threads within thread blocks to be 512 in the x-direction. However, the memory usage of simulations will dominate the limitation of the number of threads per threads blocks in the x-direction. Simulation sizes of 512^3 without using temporary storage and with single floating-point precision, result in a memory consumption of 9.5 Gigabyte.

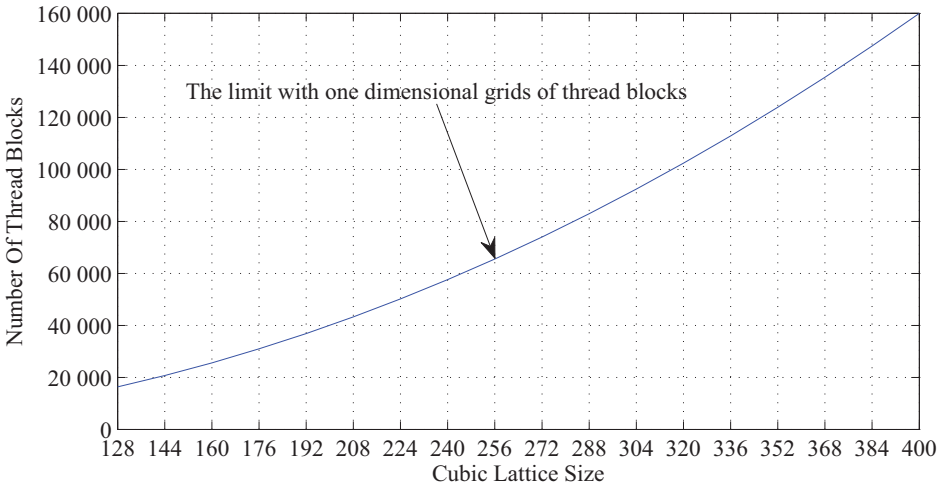


Figure 4.11: Number of thread blocks with varying cubic lattice size.

An additional GPU implementation using shared memory was made, where entire thread blocks load data into shared memory for re-use. The current maximum size of the shared memory available per multiprocessor is 16 KB, as explained in 2.2.2. This relation between the shared memory and the number of threads within thread blocks results in the maximum simulation size in the x-direction being no more than 210 lattice nodes for single floating-point precision and 105 lattice nodes for double floating-point precision. Because of this restriction, we decided not to continue the development and testing using shared memory.

Algorithm 9 The configurations of grids and thread blocks in kernels.

```
// Kernel definition
__global__ void kernelName()
{
  // Increase in the order x, y, and z to the global memory.
  int x = threadIdx.x;
  int grid_y = __mul24(blockIdx.y, gridDim.x);
  int z = (blockIdx.x + grid_y) / height;
  int y = (blockIdx.x + grid_y) - z * height;

  ...
}

void main()
{
  dim3 blockSize(dimX);
  dim3 nBlocks(dimY,dimZ);

  // Defines how the GPU kernel is executed
  kernelName<<<nBlocks,blockSize>>>();
}
```

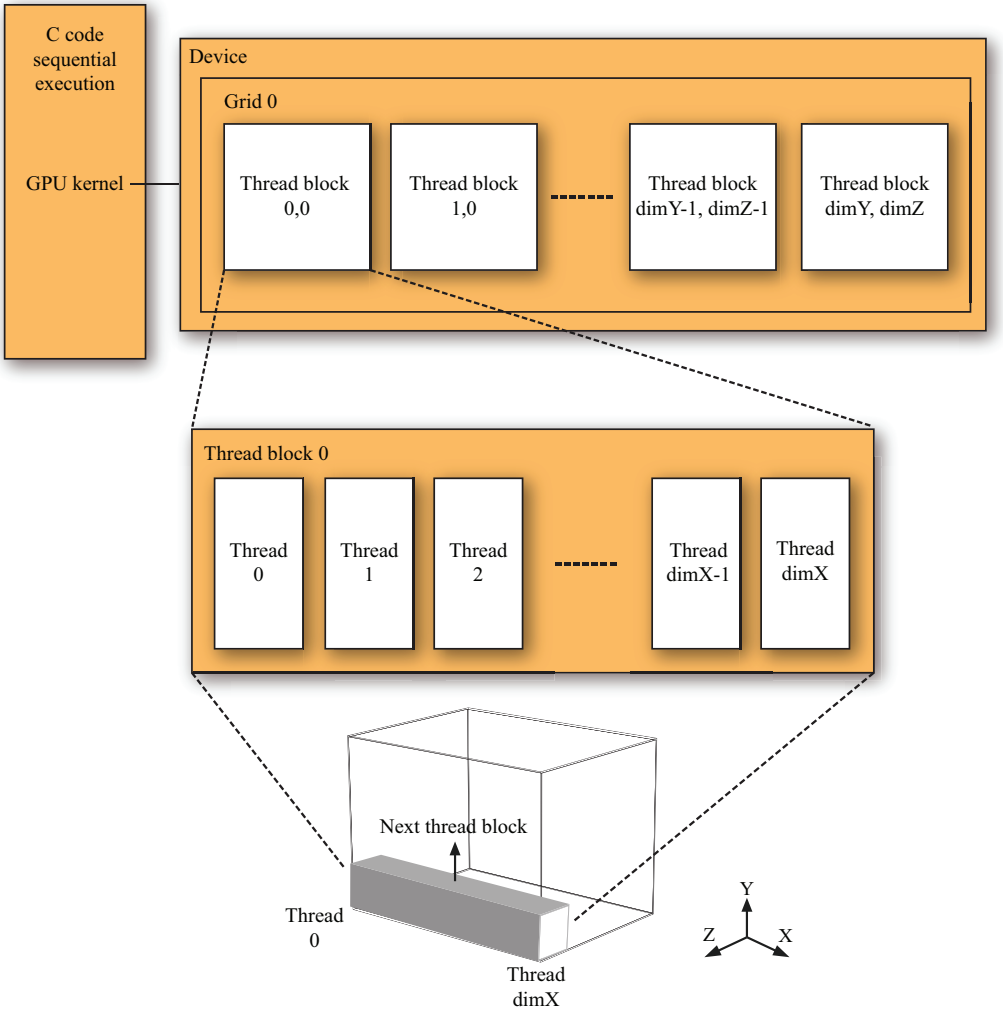


Figure 4.12: The configurations of grids and thread blocks in kernels. Adapted from [12].

Chapter 5

Benchmarks

This chapter presents and discusses several measurements of our implementations of the lattice Boltzmann method, described in the previous chapter.

In particular, Section 5.1 describes the software and hardware configuration of the machine that was used to obtain the measurements of our implementations, and the testing methodology. Section 5.2 validates the numerical exactness and correctness of our implementations, by comparing the numerical results of fluid flow between two parallel plates to known analytical solutions. Section 5.3 presents and discusses profiling results of the collision phase and streaming phase. Section 5.4 discusses several restrictions to the simulations sizes. Section 5.5 presents and discusses performance results of running our implementations with different cubic lattice sizes ranging from 8^3 up to 368^3. Section 5.6 discusses our implementations ability to calculate the permeability of porous rocks. Section 5.7 present visual results from our implementations of the lattice Boltzmann method.

5.1 Test Environment and Methodology

The machine that was used to obtain the measurements of our implementations is shown in Table 5.1. The NVIDIA Quadro FX 5800 GPU used is targeted towards professional graphics solutions, and provides high memory bandwidth, and large memory size. The NVIDIA Quadro FX 5800 is designed to handle the most demanding challenges for geophysicists, scientists, designers, engineers, and others that need ultra-high-end graphics solutions

[13].

Table 5.1: The test machine that was used to obtain the measurements.

CPU	Intel Core 2 Quad Processor Q9550
Processor Clock	2.83 GHz
Bus Speed	1333 MHz
L2 Cache Size	12 MB
L2 Cache Speed	2.83 GHz
GPU	NVIDIA Quadro FX 5800
Streaming Multiprocessors	30
Stream Processors	240
Processor Clock	1.3 GHz
Memory Size	4 GB
Memory Bandwidth	102 GB/s
Compute Capability	1.3
Memory	Corsair XMS3 DHX
Quantity	8 GB
Memory Type	DDR3 SDRAM
Memory Speed	1333 MHz
Memory Latency	9-9-9-24
Motherboard	ASUS P5E3 Premium

Measurements of our implementations were obtained using the Microsoft Windows XP 64-bit operating system. We had exclusive access to the system while collecting the measurements. Since graphics rendering is not vital to either performance measurements or the calculation of the permeability of porous rocks, it was disabled during the measurements. This avoided graphics rendering interference with the measurements results.

The performance of our implementations is measured in MLUPS (million lattice nodes updates per second), to an average of two decimal of precision. This metric indicates the number of lattice nodes that is updated in one second [40]:

$$MLUPS = \frac{n}{t_{streaming} + t_{collision}} \tag{5.1}$$

where n is the number of million lattice nodes, $t_{streaming}$ is the time used for the streaming phase and $t_{collision}$ is the time used for the collision phase of the lattice Boltzmann method. The QueryPerformanceCounter is used to measure in high-resolution the time used for the collision phase and the streaming phase of the lattice Boltzmann method. Presented performance measurements are based on the arithmetic average over 25 iterations.

To distinguish between the different implementations and the floating-point precision used in the measurements, abbreviations will be used henceforth, which are listed in Table 5.2.

Table 5.2: Measurements abbreviations.

Abbreviation	Description
CPU	Measurement of the the CPU implementation
GPU	Measurement of the the GPU implementation
32	32-bit floating-point precision used.
64	64-bit floating-point precision used.

5.2 Poiseuille Flow

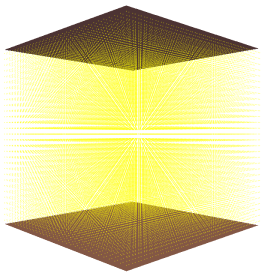


Figure 5.1: Fluid flow between two parallel plates.

In Poiseuille Flow the analytical solutions of the velocity profile are known. To validate the numerical correctness and exactness of our two implementations, the numerical velocity profile of fluid flow between two parallel plates

was compared to known analytical solutions. The fluid flow between two parallel plates that are shown in Figure 5.10, has the lattice dimension of 32^3. The analytical velocity profile of the fluid flow between two parallel plates, which is parabolic and with the flow highest in the middle, can be found with Equation 5.2 [55, 21].

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

where G is some constant external force that accelerate the fluid, μ is the dynamic viscosity of the fluid, and L is half the size of the lattice dimension in the y direction. In the Poiseuille Flow simulation, two types of boundary conditions were used: bounce back boundaries along the two parallel plates and periodic boundaries in the x, y, and z direction for conservation of fluid particles. The values of the parameters that were used in the Poiseuille Flow, for the single relaxation parameter and the external force are listed in Table 5.3.

Table 5.3: Parameter values used in the Poiseuille Flow.

Parameter	Value
τ	0.65
F_x	0.00001
F_y	0.0
F_z	0.0

The Poiseuille Flow simulations were performed with single and double precision. Figure 5.2 compares the velocity profile of the correct analytical solutions to the numerical solutions, showing that the fluid flows are recovered with great accuracy. The solid lines show the analytical solution, and the circles are the numerical results obtained from the Poiseuille Flow simulations. Table 5.4 shows the measured deviations between the numerical and analytical solutions.

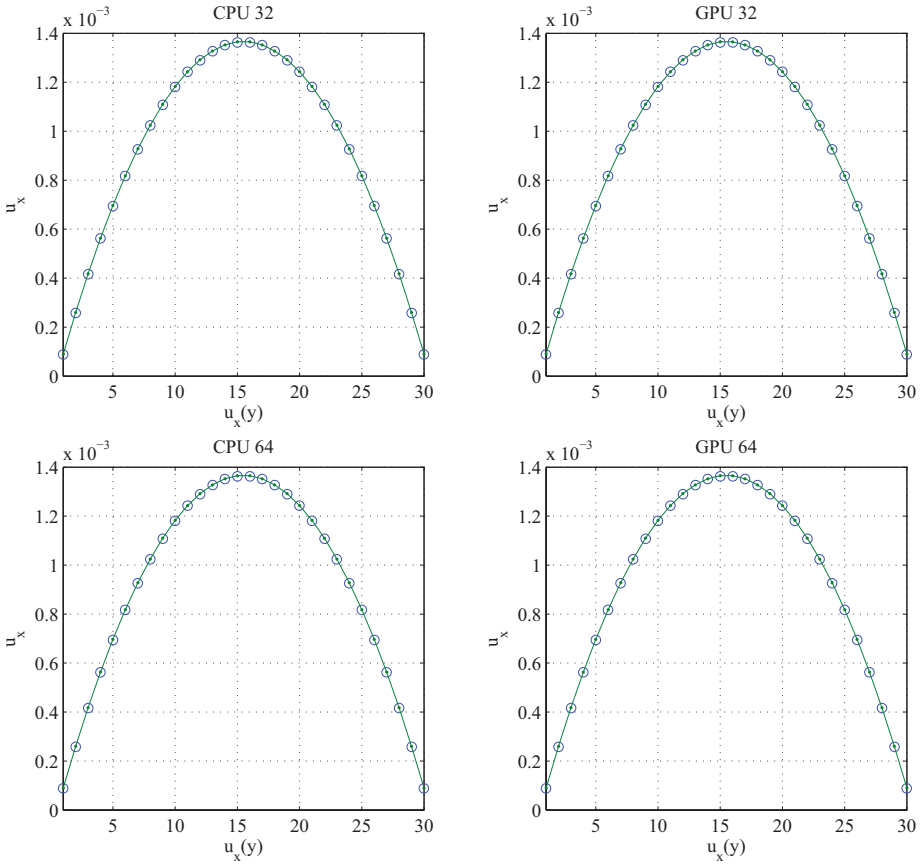


Figure 5.2: Comparison of numerical and analytical velocity profiles.

Table 5.4: Measured deviation in the Poiseuille Flow.

Implementation	Deviation
CPU 32	1.680030e-006
CPU 64	1.622238e-006
GPU 32	1.680030e-006
GPU 64	1.622254e-006

5.3 Kernel Profiling

The GPU implementation was profiled to find the processing time of the collision phase and streaming phase with single floating-point precision. We used a cubic lattice of size 80^3, filled only with fluids elements. Only the initial phase, collision phase, and the streaming phase were enabled during profiling. Figure 5.3 shows the results of the relative distribution of the processing time between the collision phase and the streaming phase. The streaming phase used almost 36% of the processing time, which is because the lattice nodes interact with neighbors in the lattice to exchange particle distribution functions. The collision phase used almost 63% of the processing time, due to the calculations that takes places with the relaxation of the particles distributions against equilibrium condition. The memory copying used almost 1% of the processing time, which is because of the transfer of the initial lattice configuration to GPU memory once at startup. This memory copying is unnecessary, and the lattice could have been initialized directly on the GPU.

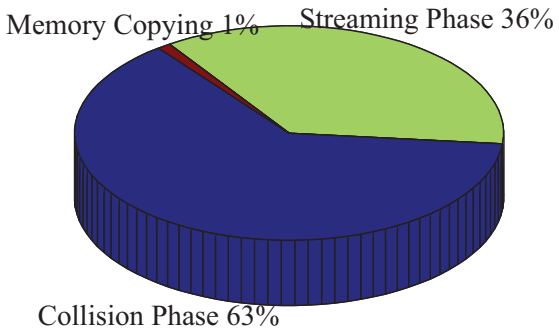


Figure 5.3: GPU implementation processing time of the collision phase and streaming phase.

5.4 Simulation Size Restrictions

The NVIDIA Quadro FX 5800 card used has 16384 registers and 16 KB shared memory available per multiprocessor. Threads of all thread blocks running on a multiprocessor must share these registers and shared memory during execution [25]. Kernels will fail to lunch if threads uses more registers or shared memory than available per multiprocessor [12]. Table 5.5 shows the number of available registers per thread with varying block size.

Table 5.5: Registers available with varying block size.

Block size	256	320	384	448	512	576	640	704	768
Registers available	64	51	42	36	32	28	25	23	21

In the stream and collide kernels, thread blocks are one dimensional with the number of threads equal to simulation size in the x-direction. To find the amount of local memory, shared memory and number of registers used per thread in the stream and collide kernels, we generated Cubin files from the NVIDIA CUDA source files. The registers count and shared memory usage of the stream and collide kernels have been minimized to support large simulations sizes in the x-direction. Table 5.6 shows the resulting maximum simulations sizes in the x-direction, due to the stream and collide kernels amount of local memory, shared memory and number of registers used per thread.

The most memory demanding parts of the implementations is the structure-of-arrays used to store the particle distribution functions, together with the array used to store the porous rocks models. Table 5.4 shows the growth in memory requirements with varying cubic lattice sizes, and the largest cubic lattice sizes that can be run with single and double precision without running out of global memory.

Table 5.6: Maximum simulation sizes in the x-direction due to register and shared memory usage per thread.

Implementation	Local Memory	Shared Memory	Registers Count	Maximum Size
GPU 32 stream	0 byte	128 bytes	13	Above 768
GPU 64 stream	0 byte	128 bytes	16	Above 768
GPU 32 collide	0 byte	144 bytes	20	Above 768
GPU 64 collide	56 bytes	144 bytes	30	Up to 576

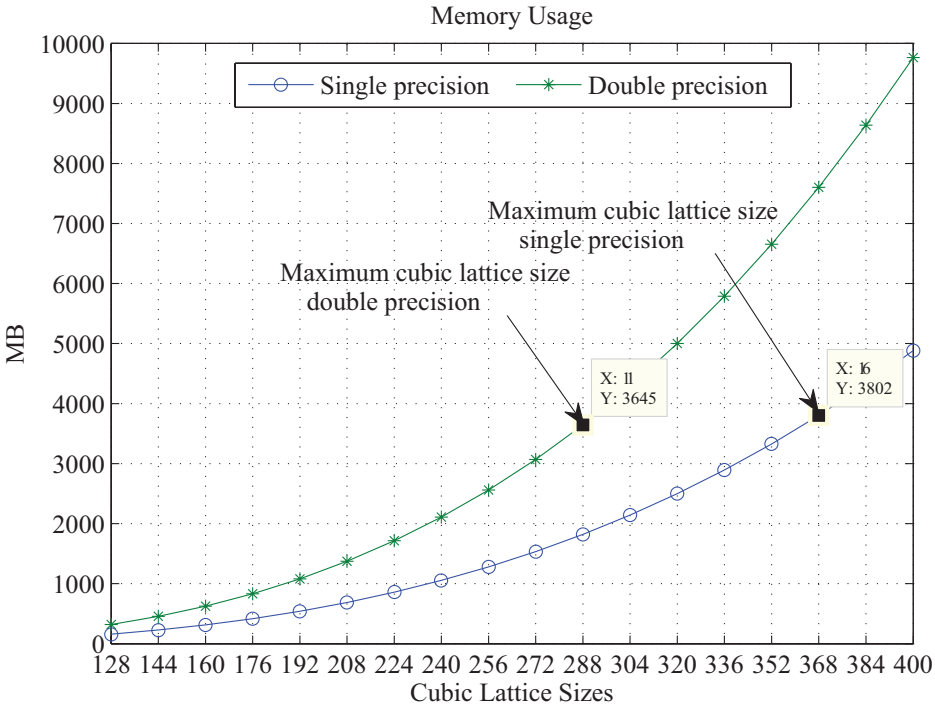


Figure 5.4: Memory requirements with varying cubic lattice sizes.

5.5 Performance Measurements

In order to measure the performance of our CPU and GPU implementation, cubic lattice sizes ranging from 8^3 up to 368^3 were used. The cubic lattices

were filed only with fluid elements, so that no extra work was required for solid-fluid interfaces. Table 5.7 and Figure 5.5 show the results of the performance measurements of our CPU and GPU implementations, which are based on the arithmetic mean over 25 iterations. Figure 5.6 shows the total time used to complete the 25 iterations.

Table 5.7: Performance results in MLUPS.

Lattice Size	CPU 32	CPU 64	GPU 32	GPU 64
8	1.59	1.37	2.38	2.06
16	1.58	1.40	15.90	12.19
32	1.42	1.22	74.40	38.92
48	1.54	1.34	125.64	53.08
64	1.27	1.15	91.99	55.67
80	0.95	1.15	83.97	60.17
96	1.27	1.15	171.40	61.25
112	1.16	1.09	171.61	61.47
128	1.24	1.14	112.32	60.41
144	1.26	1.15	178.81	62.10
160	1.25	1.13	180.34	62.54
176	1.25	1.11	178.04	63.00
192	1.34	1.13	121.86	61.97
208	1.31	1.12	178.23	62.60
224	1.36	1.13	183.08	63.35
240	1.28	1.09	149.06	60.67
256	1.31	1.14	112.05	61.27
272	1.31	1.13	181.28	61.53
288	1.35	1.14	182.79	61.87
304	1.34	-	180.35	-
320	1.34	-	115.41	-
336	1.31	-	155.83	-
352	1.33	-	184.30	-
368	1.30	-	180.19	-

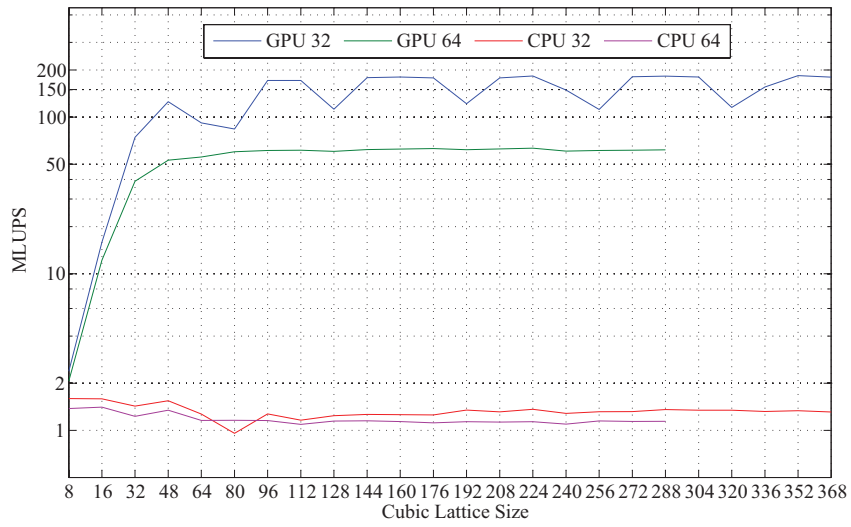


Figure 5.5: Performance results in MLUPS.

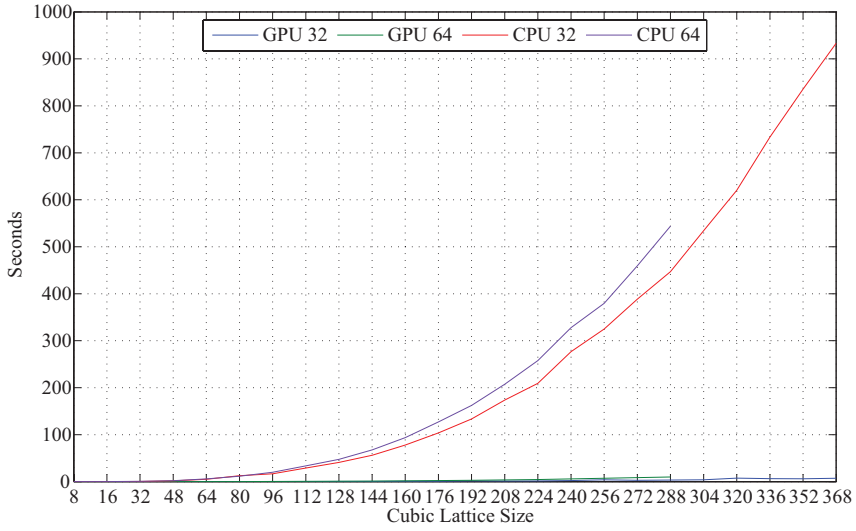


Figure 5.6: Overall execution time of performance measurements.

As we can see from Table 5.7 and Figure 5.5, the GPU implementation clearly outperforms the CPU implementation in performance. Both CPU and GPU implementations achieved their highest performance using single

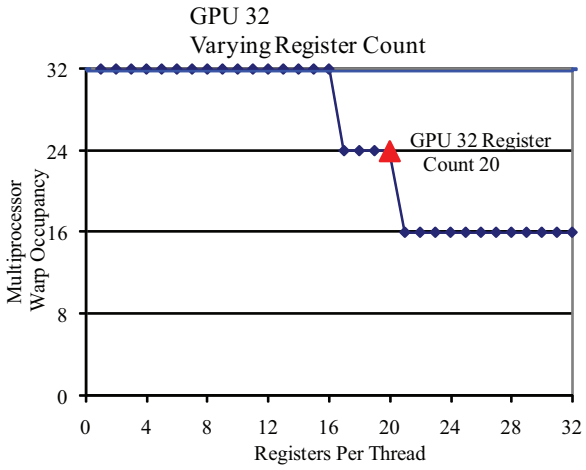


Figure 5.7: GPU 32 occupancy with varying registers count, generated with the NVIDIA CUDA occupancy calculator.

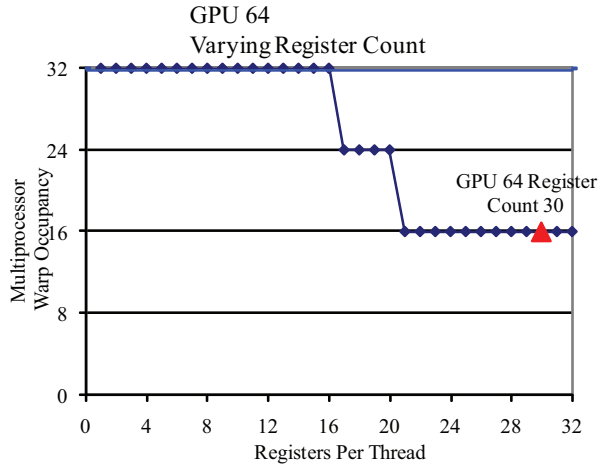


Figure 5.8: GPU 64 occupancy with varying registers count, generated with the NVIDIA CUDA occupancy calculator.

floating-point precision. CPU 32 and GPU 32 achieved the maximum performance equal to 1.59 MLUPS and 184.30 MLUPS. CPU 64 and GPU 64 achieved maximum performance equal to 1.40 MLUPS and 63.35 MLUPS. Highest performance of the CPU 32 and CPU 64 was with lattice sizes smaller

than 64^3 and 48^3, since the lattices fits into cache memory. GPU 32 and GPU 64 performance increased quickly with lattice sizes up to 96^3 and 48^3, since there was available computing capacity.

The performance difference between the GPU 32 and GPU 64 is because NVIDIA GPUs have more capacity for computing with single precision than with double precision, and because the GPU 32 and GPU 64 have some differences in occupancy. We used the NVIDIA CUDA occupancy calculator to calculate the maximum multiprocessor occupancy of the collide and stream kernels, using the number of registers, amount of shared memory and local memory used per thread as input. The GPU 32 stream kernel has maximum occupancy of 100%. The GPU 64 stream kernel has maximum occupancy of 100%. The GPU 32 collide kernel has maximum occupancy of 75%. The GPU 64 collide kernel has maximum occupancy of 50%.

Occupancy is important, because there is not enough computational work in the collision phase and the streaming phase to keep the multiprocessors busy without sufficient occupancy. With 100 % occupancy more warps are processed, which can be used to hide global memory latency. To get the highest possible occupancy, the number of registers must be restricted and shared memory usage per thread must be minimized to be able to run more thread blocks in parallel. Figure 5.9 shows the current amount of registers used by the CPU 32 and GPU 64 collide kernel, and the amount needed to get better occupancy. A better occupancy would be possible for the CPU 32 and GPU 64 collide kernel if the number of registers were reduced to 16 and 20. This would give the CPU 32 and GPU 64 collide kernel with the right block size 100% and 75% occupancy. The local memory usage of the GPU 64 should also be removed to increase performance. The turbulent performance of the GPU 32 is caused of the changes in occupancy of the collide kernel, due to changes in thread block sizes, as illustrated in Figure 5.9.

5.6 Porous Rock Measurements

In order to evaluate our implementations ability to calculate the permeability of porous rocks, three porous datasets with the known permeability provided by Numerical Rocks AS were used. Porosity that reflects only the interconnected pore spaces within the three porous datasets calculated by Numerical

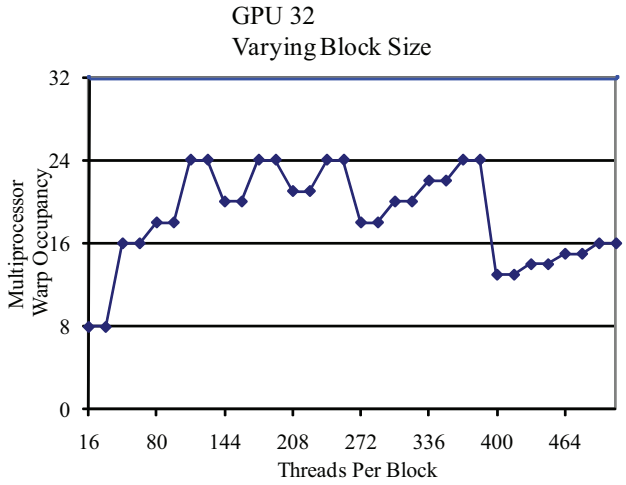


Figure 5.9: GPU 32 occupancy with varying block size, generated with the NVIDIA CUDA occupancy calculator.

Rocks AS was also used. The values of the single relaxation parameter and the external force used in the calculations of the permeability of the three porous datasets, are listed in Table 5.8.

Table 5.8: Parameter values used in the porous rocks measurements.

Parameter	Value
τ	0.65
F_x	0.00001
F_y	0.0
F_z	0.0

In the simulations, the configurations of the boundaries parallel to the flow direction were made solid, and with bounce back boundary conditions. The entry and exit boundaries were given periodic boundary conditions. There were also 3 empty layers of void space added at both the entry and exit boundaries. Simulations were run until velocity fields reached a steady state, before calculating the permeability of the three porous datasets.

5.6.1 Symmetrical Cube

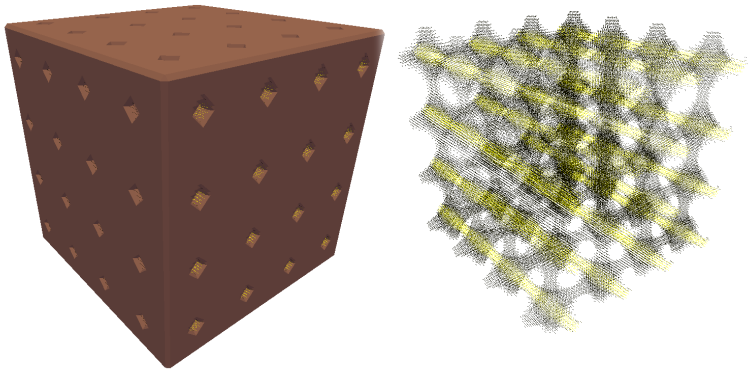


Figure 5.10: Fluid flow through the symmetrical cube.

The symmetrical cube dataset shown in Figure 5.10 has the lattice size of 80^3, with known permeability equal to 22 mD. The symmetrical cube dataset porosity is 16 %. Table 5.9 shows the results of our performance measurements and calculated permeability. Figure 5.11 compare our results in bar graphs.

Table 5.9: Symmetrical Cube performance and computed permeability results.

Implementation	Average MLUPS	Maximum MLUPS	Total Time	Number Of Iterations	Permeability Obtained
CPU 32	0.97	0.98	16.2 s	131	22.37 mD
GPU 32	28.90	30.53	0.5 s	131	22.37 mD
CPU 64	0.96	0.98	16.3 s	131	22.37 mD
GPU 64	14.55	14.98	1.0 s	131	22.37 mD

GPU 32 was the fastest of the implementations with the total simulation time equal to 0.5 seconds. All of the implementations calculated the permeability within reasonable deviation, with the relative error equal to 1.6% and the absolute error equal to 0.37. GPU 32 obtained the highest average performance at 28.90 MLUPS.

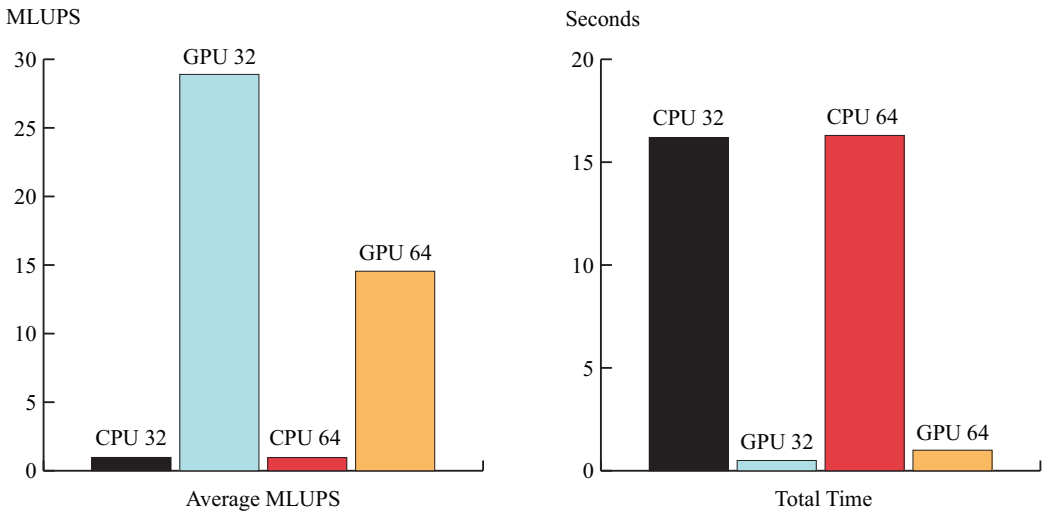


Figure 5.11: Symmetrical Cube performance and computed permeability results.

5.6.2 Square Tube

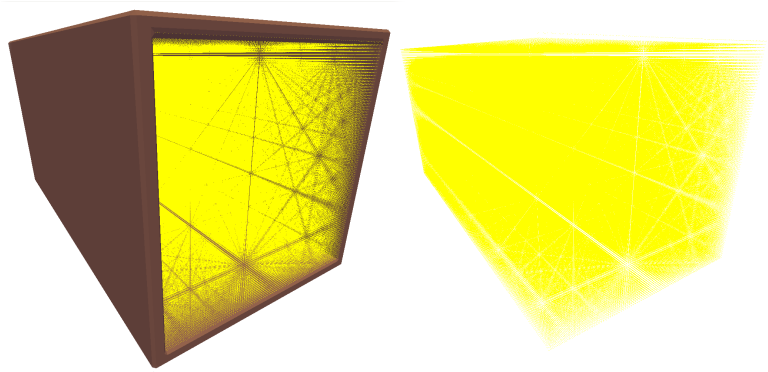


Figure 5.12: Fluid flow through the Square Tube.

The square tube dataset demonstrated handling of free fluid flow. The square tube dataset shown in Figure 5.12 has the lattice size of $200 \times 100 \times 100$, with the known permeability equal to 216 D. The square tube dataset porosity is 81 %. Table 5.10 shows the results of our performance measurements and computed permeability's. Figure 5.13 compare our results in bar graphs.

Table 5.10: Square Tube performance and computed permeability results.

Implementation	Average MLUPS	Maximum MLUPS	Total Time	Number Of Iterations	Permeability Obtained
CPU 32	1.35	1.36	1284.3 min	62221	233.52 D
GPU 32	166.14	167.58	10.4 min	61893	233.39 D
CPU 64	1.08	1.09	1597.1 min	61957	233.44 D
GPU 64	58.40	59.33	29.7 min	61957	233.44 D

GPU 32 was the fastest of the implementations, with the total simulation time equal to 10.4 minutes. All of the implementations calculated the permeability within deviation, with the maximum relative error equal to 7.1% and the absolute error equal to 15.52. GPU 32 obtained the highest average performance equal to 166.14 MLUPS. CPU 32 used 62221 iterations before it converged, which was the highest number of iterations.

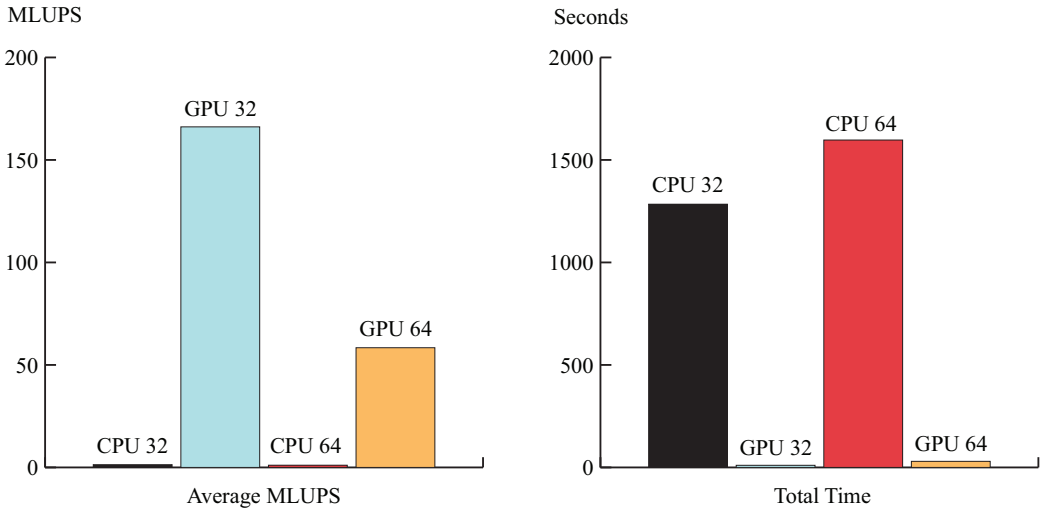


Figure 5.13: Square Tube performance and computed permeability results.

5.6.3 Fontainebleau

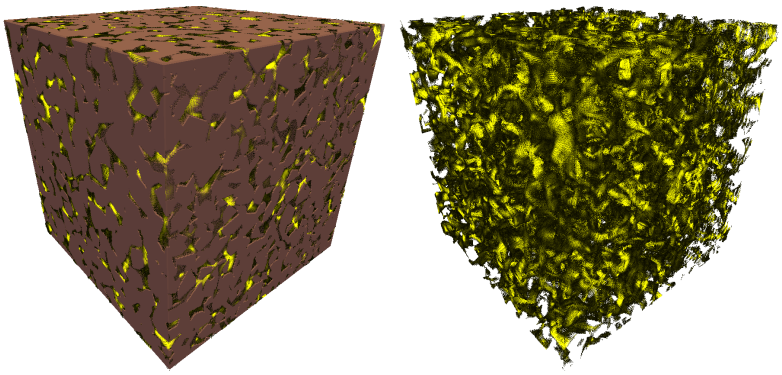


Figure 5.14: Fluid flow through the Fontainebleau.

The Fontainebleau dataset shown in Figure 5.14 has the lattice dimension of 300^3, with the known permeability equal to 1300 mD. The Fontainebleau dataset porosity is 16 %. Note that the Fontainebleau data set was too large to allocate with double precision on the GPU. Table 5.11 shows the results of our performance measurements and computed permeability. Figure 5.15 compare our results in bar graphs.

Table 5.11: Fontainebleau performance and computed permeability results.

Implementation	Average MLUPS	Maximum MLUPS	Total Time	Number Of Iterations	Permeability Obtained
CPU 32	1.03	1.04	2152 s	445	1247.80 mD
GPU 32	58.81	59.15	38.0 s	445	1247.81 mD
CPU 64	0.94	0.94	2375.4 s	445	1247.80 mD

GPU 32 was the fastest of the implementations with the total simulation time equal to 38.0 seconds. All of the implementations calculated the permeability within deviation, with the relative error equal to 4.0% and the absolute error equal to 53. GPU 32 obtained the highest average performance equal to 58.81 MLUPS.

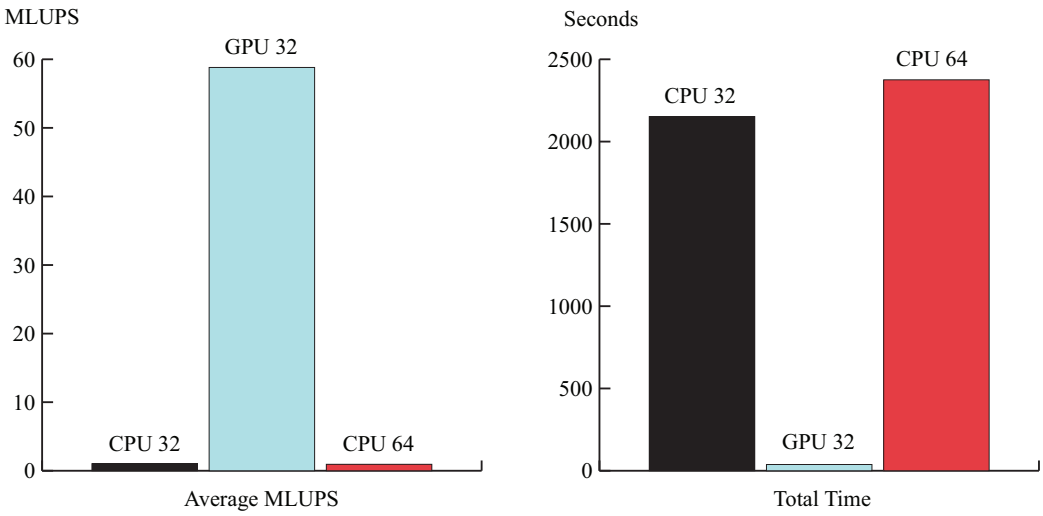


Figure 5.15: Fontainebleau performance and computed permeability results.

5.6.4 Discussion

To obtain matching calculations of porous rock permeability from the CPU and GPU implementations with single and double floating-point precision it was necessary to reduce the rounding errors that occurred in the simulation model. Figure 5.16 shows permeability progression with and without reduced rounding error for the Symmetrical Cube dataset. Figure 5.17 shows in detail the last 20 iterations of the permeability progression, to see that the CPU and GPU implementations with reduced rounding error and single precision are matching.

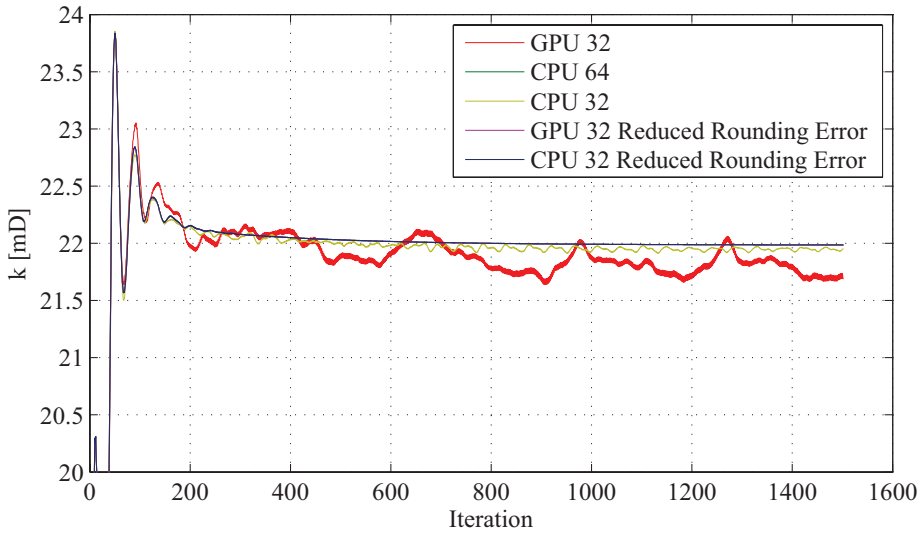


Figure 5.16: Symmetrical Cube: with and without reduced rounding error.

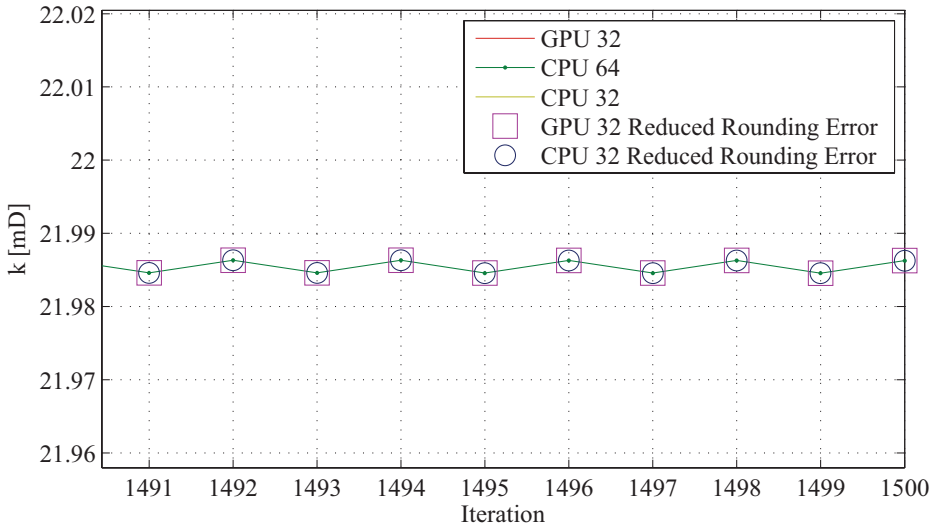


Figure 5.17: Symmetrical Cube: with and without reduced rounding error, last 20 iterations.

5.7 Visual Results

In this section, we present visual results from our implementations of the lattice Boltzmann method. The Fontainebleau is used to show the implemented visual support, with 3 extra empty layers of void space at both the entry and exit boundaries. The configurations of the boundaries parallel to the flow direction were made solid, and with bounce back boundary conditions. The entry and exit boundaries were given periodic boundary conditions. Open pores parallel to the flow direction was made visible in the visualization of the Fontainebleau. Figure 5.18 shows the fluid flow direction. Figures 5.19 and 5.20 show the first 8 iterations of fluid flows through the pore geometry of the Fontainebleau. Figures 5.19 and 5.20 show the first 8 iterations of how fluids flow inside the pore geometry.

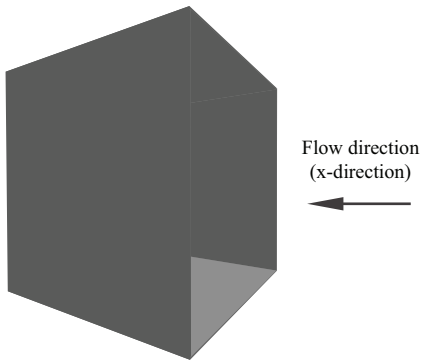


Figure 5.18: Fluid flow direction in visual results.

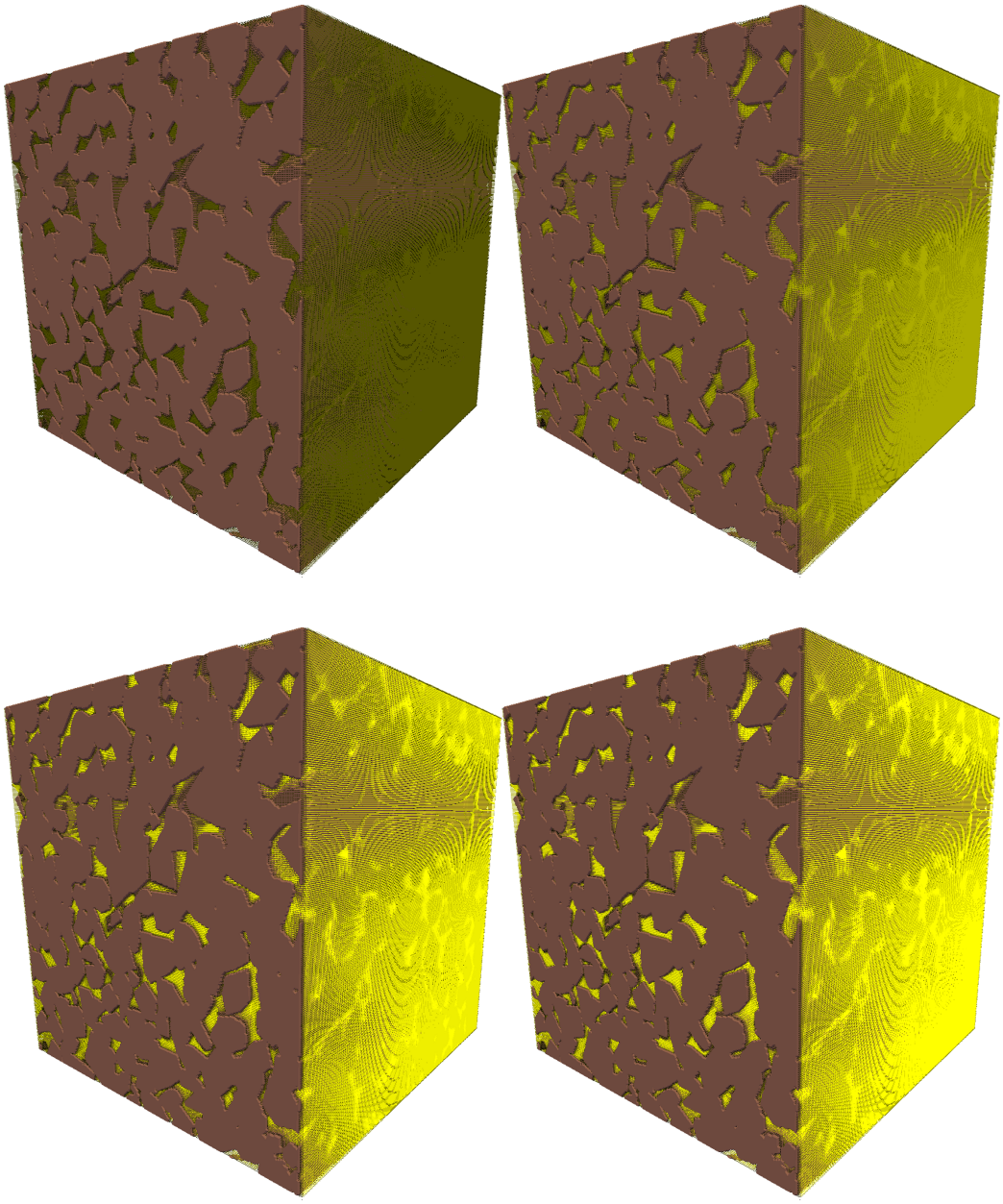


Figure 5.19: The first 4 iterations of fluid flow through Fontainebleau.

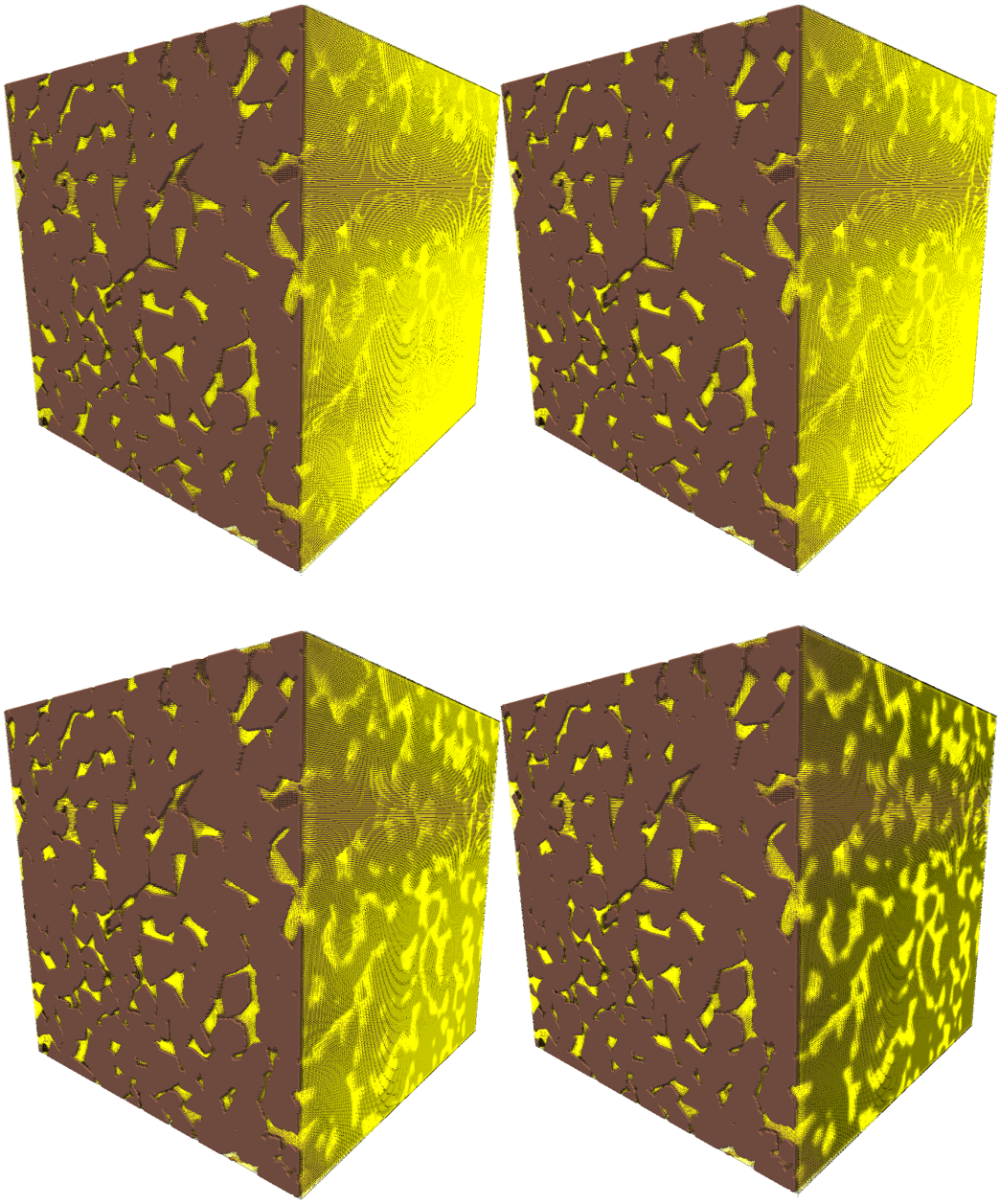


Figure 5.20: The 5th to 8th iterations of fluid flow through Fontainebleau.

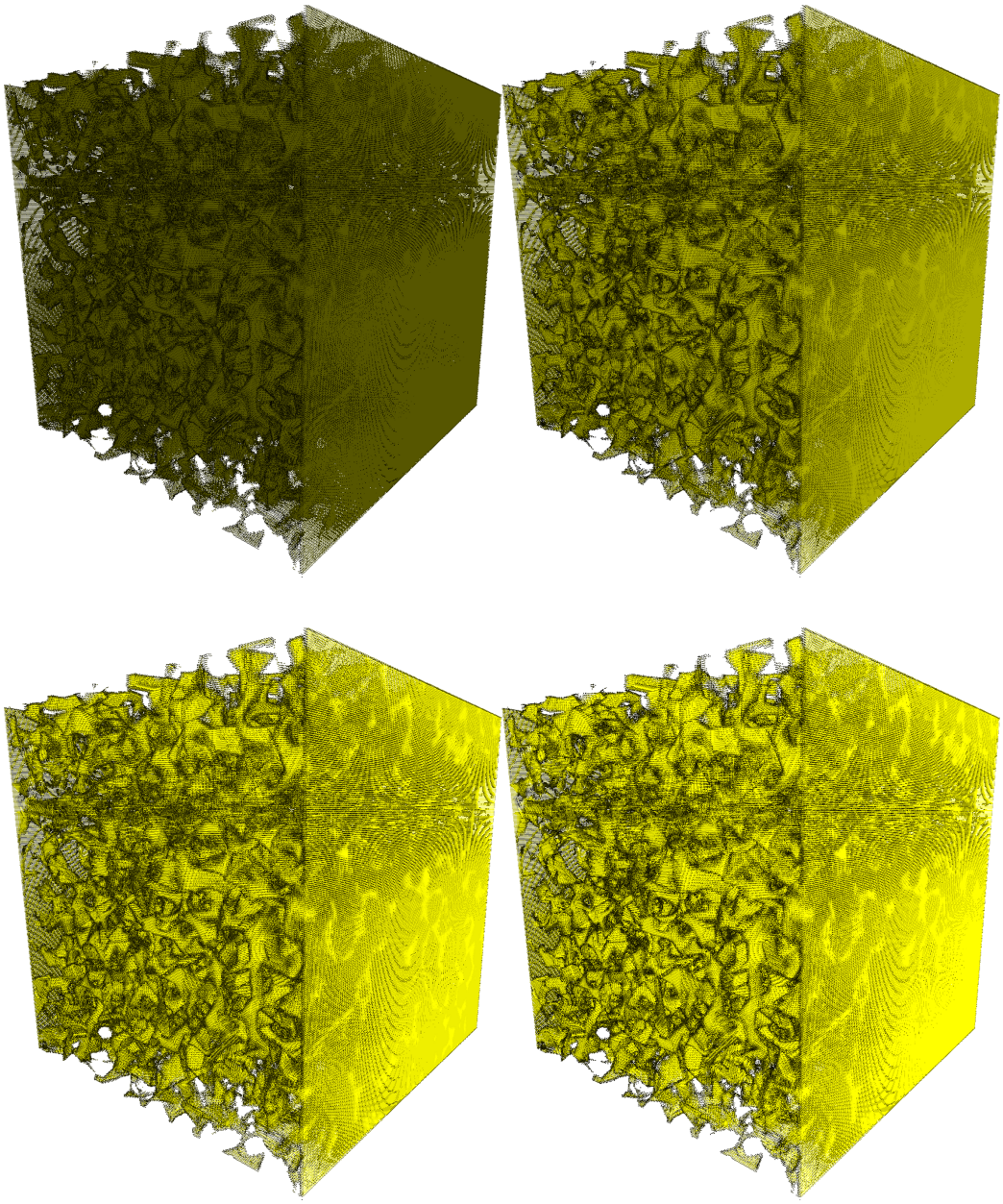


Figure 5.21: The first 4 iterations of fluid flow inside Fontainebleau.

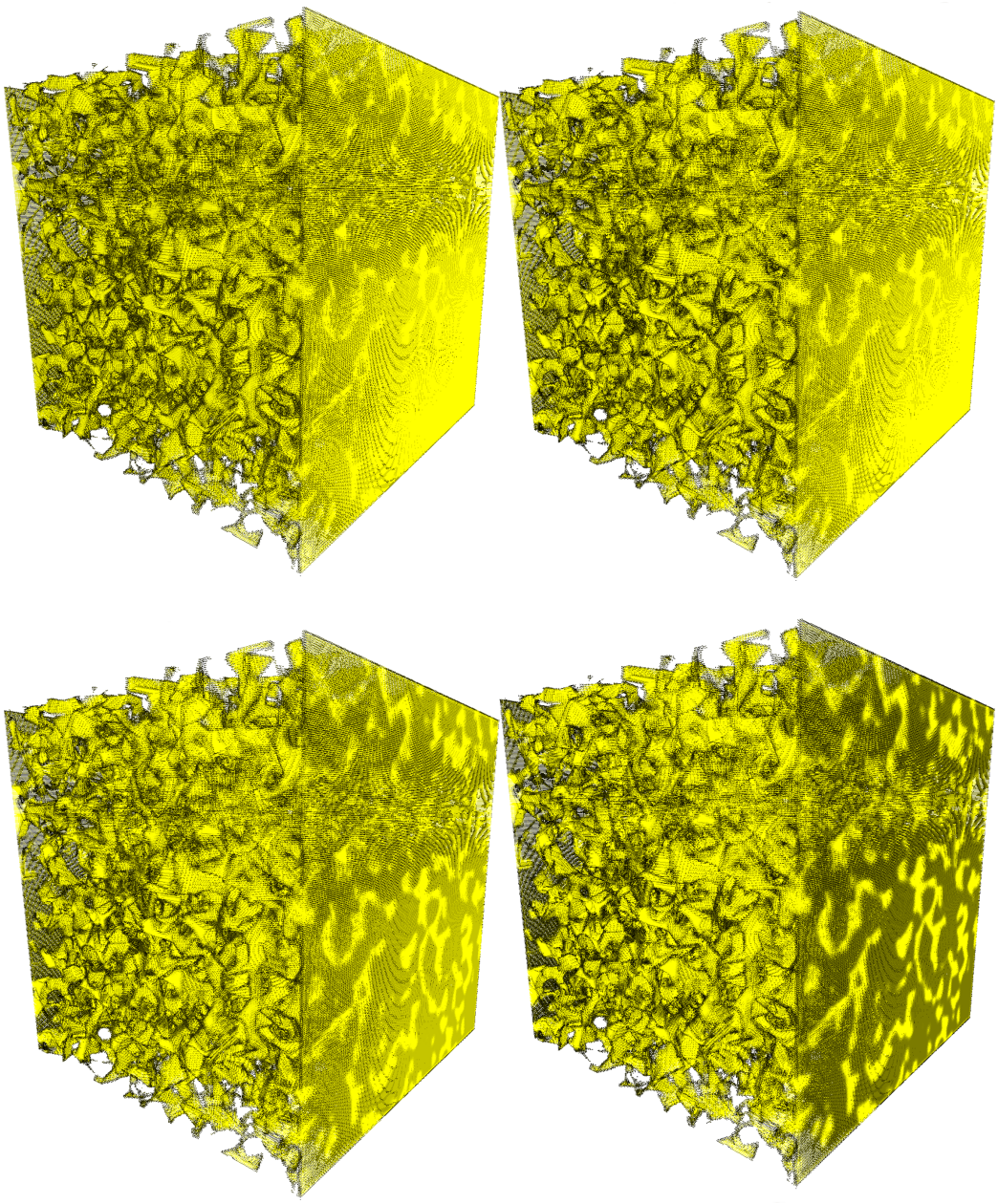


Figure 5.22: The 5th to 8th iterations of fluid flow inside Fontainebleau.

Chapter 6

Conclusions and Future Work

State-of-the-art *Graphics Processing Units* (GPUs) offer applications the potential to harvest amazing amounts of compute power, by computing several hundred instructions simultaneously. Because of this highly parallel architecture, modern GPUs are used to accelerate a wide range of scientific applications which earlier required clusters of workstations or large expensive supercomputers. Since it would be very valuable for the petroleum industry to analyze petrophysical properties of porous rocks, such as the porosity and permeability, through computer simulations, the goal of this thesis was to see if such simulations could benefit from GPU acceleration.

In this thesis, we accelerated the previously paralellized lattice Boltzmann method for the GPU, using the NVIDIA CUDA programming environment. The method is used to estimate porous rock's ability to transmit fluids. In order to better analyze our results, both parallel CPU and GPU implementations of the lattice Boltzmann method were developed and benchmarked using three porous datasets provided by Numerical Rocks AS [1] where the permeability of each dataset was known. This also allowed us to evaluate the accuracy of our results.

To be able to support as large lattice sizes, the register count and shared memory usage of the kernels were minimized. The configuration of grids and thread blocks of the kernels were properly configured. The sizes of the lattices in our simulations were limited by the memory size of the current

[1] http://www.numericalrocks.com/

GPUs. Our GPU implementation supported lattice sizes up to 368^3 (with single floating-point precision), which fit into the 4 GB memory of the NVIDIA Quadro FX 5800 card. Current GPUs have 32-bit addressing, indicating that addressing much more memory than we did for this thesis would require a major architectural change, but will inevitably happen. Future GPUs with more memory should then be able to alleviate our memory size problem. See our next section for suggestion to obtain more memory with clusters of GPUs.

Using single floating-point precision helped the memory situation some, but was also particularly advantageous for our GPU implementation, since the NVIDIA GPUs provided about 4 times more cores with single floating-point precision than with double floating-point precision. However, with single floating-point precision, our implementations were prone to round-off errors. This influenced the accuracy and stability of our simulations, and hence the quality of the permeability estimates of porous rocks. However, for the petroleum industry it is important to get permeability estimates of porous rocks as close to the real-world conditions as possible.

We reduced these round-off errors by looking at the calculations. In the collision phase of the lattice Boltzmann method, the equilibrium distribution function needed to be evaluated with typically a mixture of large and small numbers that are added and subtracted. Large round-off errors occur between two very different numbers that are added or subtracted. To reduce the round-off errors in our implementation, we hence used an approach that evaluates an equilibrium distribution function without the mixture of large and small numbers [11] [2]. This allowed us to obtain the matching estimates of porous rock's permeability with both single and double floating-point precision.

Our development efforts showed that it is possible to simulate fluid flow through the complicated geometries of porous rocks with high performance on modern GPUs. Our GPU implementations clearly outperformed our CPU implementation, in both single and double floating-point precision. Both implementations achieved their highest performances when using single floating-point precision, resulting in their maximum performance equal to 1.59 MLUPS and 184.30 MLUPS, respectively for datsets of size 8^3 by 352^3,

[2] www.lbmethod.org

where MLUPS is the measurement of million lattice nodes updates per second (indicating the number of lattice nodes that is updated in one second). Suggestions for improving these results are include in the next section.

6.1 Future Work

There are several extensions that can be made to this thesis:

- Maximum simulation lattice size in the simulations is limited by the memory capacity of the system. The 32-bit architecture of current GPUs limit the maximum size of the memory that can be referenced to 4 GB. One way to compensate for this is to use multiple GPUs. One such system is the rackmounted NVIDIA s1070 which has 4 GPUs similar to the FX8800 we tested with 16GB of mememory on each GPU.
- Use of grid refinement for the improved analysis of the fluid flow inside the narrow pore geometry of the porous rocks.
- Storing only fluid elements, this will reduce memory usage, since the porous rocks of interest often have small pore geometries.
- Extenting the lattice Boltzmann simulation model to perform multiphase fluid dynamics.

Given the importance for the petroleum industry of getting good fluid simulations of porous rocks, we expect that this will continue to be a great area of research. Flows through porous materials are also be of interest to other fields, including medicine.

Bibliography

[1] IA-32 Intel Architecture Optimization. http://www.parallel.ru/ftp/computers/intel/optimization/ia32_optimization_ref_manual.pdf Last retrived 03.04.2009, 2003. Intel corporation. Reference Manual.

[2] Urpo Aaltosalmi. Fluid Flows In Porous Media With The Lattice-Botlzmann Method. http://www.jyu.fi/static/fysiikka/vaitoskirjat/2005/urpo_aaltosalmi.pdf, 2005. University of Jyväskylä.

[3] Eirik Ola Aksnes and Henrik Hesland. GPU Techniques for Porous Rock Visualization. http://www.idi.ntnu.no/~elster/master-studs/aksnes-hesland-MSproj.pdf Last retrived 15.06.2009, January 2009. Norwegian University of Science and Technology.

[4] Usman R. Alim, Alireza Entezari, and Torsten Möller. The Lattice-Boltzmann Method on Optimal Sampling Lattices. *IEEE Transactions on Visualization and Computer Graphics*, 15(4):630–641, 2009.

[5] Mark Knackstedt Andreas Kayser and Murtaza Ziauddin. A Closer Look at Pore Geometry. *Schlumberger Oilfield Review*, 2006.

[6] 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. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html Last retrived 15.06.2009.

[7] P. Bourke. Polygonising a scalar field. http://local.wasp.uwa.edu.au/~pbourke/geometry/polygonise/ Last retrived 30.12.2008, May 1994.

[8] D. A. Caughey and M. M. Hafez, editors. *Frontiers of Computational Fluid Dynamics 2002*. World Scientific Publishing Co. Pte. Ltd., 2002.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, 2008.

[10] Hudong Chen, Shiyi Chen, and William H. Matthaeus. Recovery of the Navier-Stokes equations using a lattice-gas Boltzmann method. *Phys. Rev. A*, 45(8):R5339–R5342, Apr 1992.

[11] Bastien Chopard. How to improve the accuracy of Lattice Boltzmann calculations. http://www.lbmethod.org/ Last retrived 23.05.2009, May 2008. www.lbmethod.org.

[12] NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture. Programming Guide Version 2.0. http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf Last retrived 15.06.2009.

[13] NVIDIA Corporation. Product Overview. http://www.nvidia.com/page/products.html Last retrived 01.04.2009.

[14] NVIDIA Corporation. Readme for NVIDIA CUDA Visual Profiler Version 2.2. http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/cudaprof_1.2_readme.html Last retrived 01.04.2009.

[15] NVIDIA Corporation. Using Vertex Buffer Objects. http://developer.nvidia.com/attach/6427 Last retrived 01.04.2009, October 2003. White Paper.

[16] NVIDIA Corporation. The CUDA Compiler Driver NVCC. http://moss.csc.ncsu.edu/~mueller/cluster/nvidia/2.0/nvcc_2.0.pdf Last retrived 01.04.2009, April 2008.

[17] David Lee Davidson. The Role of Computational Fluid Dynamics in Process Industries. *The Bridge*, 32(4), 2002.

[18] Stefan Donath. On Optimized Implementations of the Lattice Boltzmann Method on Contemporary High Performance Architectures. `http://www.rrze.uni-erlangen.de/dienste/arbeiten-rechnen/hpc/Projekte/Donath.pdf` Last retrived 15.06.2009, 2004. University of Erlangen-Nuremberg.

[19] Alexander Dreweke. Implementation and Optimization of the Lattice Boltzmann Method for the Jackal DSM System. `http://www10.informatik.uni-erlangen.de/Publications/Theses/2005/Dreweke_BA_05.pdf` Last retrived 15.06.2009, 2005. Friedrich-Alexander-Universität.

[20] Robin Eidissen. Utilizing GPUs for Real-Time Visualization of Snow. `http://daim.idi.ntnu.no/masteroppgave?id=4327`, 2009. Norwegian University of Science and Technology.

[21] Simon T. Engler. Benchmarking the 2D Lattice Boltzmann BGK Model.

[22] Alphonsus Fagan. An introduction to the petroleum industry. `http://www.nr.gov.nl.ca/mines&en/education/intro.pdf` Last retrived 04.02.2009, November 1991. Goverment of newfoundland and labrador Department of Mines and Energy.

[23] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-Gas Automata for the Navier-Stokes Equation. *Phys. Rev. Lett.*, 56(14):1505–1508, Apr 1986.

[24] Dmitry B. Silin Guodong Jin, Tad W. Patzek. Direct Prediction Of The Absolute Permeability of Unconsolidated and Consolidated Reservoir Rock. *SPE 90084*, September 2004.

[25] J Habich. Performance Evaluation of Numeric Compute Kernels on nVIDIA GPUs. `http://www10.informatik.uni-erlangen.de/Publications/Theses/2008/Habich_MA08.pdf`, June 2008. Friedrich-Alexander-Universität.

[26] J. Hardy, Y. Pomeau, and O. de Pazzis. Time evolution of a two-dimensional model system. I. Invariant states and time correlation functions. *Journal of Mathematical Physics*, 14(12):1746–1759, 1973.

[27] Xiaoyi He and Li-Shi Luo. Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. *Phys. Rev. E*, 56(6):6811–6817, Dec 1997.

[28] Xiaoyi He, Xiaowen Shan, and Gary D. Doolen. Discrete boltzmann equation model for nonideal gases. *Phys. Rev. E*, 57(1):R13–R16, Jan 1998.

[29] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[30] Bruce Jacob. Cache Design for Embedded Real-Time Systems. *Electrical And Computer Engineering Department*, 1999.

[31] C. Korner, T. Pohl, U. Rude, N. Thurey, and T. Zeiser. Parallel Lattice Boltzmann Methods for CFD Applications. *Numerical Solution of Partial Differential Equations on Parallel Computers*, 51:439–465, 2006.

[32] Jonas Latt. Technical report: How to implement your DdQq dynamics with only q variables per node (instead of 2q). `http://www.lbmethod.org/openlb/downloads/olb-tr1.pdf` Last retrived 15.03.2009, 2007. Tufts University.

[33] Börje Lindh. Application Performance Optimization. `http://www.sun.com/blueprints/0302/optimize.pdf` Last retrived 01.06.2009, March 2002. Sun Microsystems AB. Sun Blueprints Online.

[34] Mats Lindh. Marching cubes. `http://www.ia.hiof.no/~borres/cgraph/explain/marching/p-march.html` Last retrived 30.12.2008, March 2003. Computer Graphics Østfold University College.

[35] E. W. Llewellin. LBflow: an extensible lattice Boltzmann framework for the simulation of geophysical flows. Part I: theory and implementation. 2006.

[36] J.L. Manferdelli, N.K. Govindaraju, and C. Crall. Challenges and Opportunities in Many-Core Computing. *Proceedings of the IEEE*, 96(5):808–815, May 2008.

[37] M. D. Mccool. Scalable Programming Models for Massively Multicore Processors. *Proceedings of the IEEE*, 96(5):816–831, 2008.

[38] 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.

[39] 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, 2008.

[40] Stuart D. C. Walsh David J. Lilja Peter Bailey, Joe Myre and Martin O. Saar. Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors. 2008.

[41] G Pontrelli, S Ubertini, and S Succi. The unstructured lattice boltzmann method for non-newtonian flows. *Journal of Statistical Mechanics: Theory and Experiment*, 2009(06):P06005 (13pp), 2009.

[42] Y. H. Qian, D. D'Humieres, and P. Lallemand. Lattice BGK Models for Navier-Stokes Equation. *EPL (Europhysics Letters)*, 17(6):479–484, 1992.

[43] Shane Ryoo, Christopher Rodrigues, Sara Baghsorkhi, Sam Stone, David Kirk, and Wen mei Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, February 2008.

[44] E. Shirani and S. Jafari. Application of LBM in Simulation of Flow in Simple Micro-Geometries and Micro Porous Media. *African Physical Review*, 1(1), 2007.

[45] Gary D. Doolen Shiyi Chen and Kenneth G. Eggert. Lattice-Boltzmann Fluid Dynamics: A Versatile Tool for Multiphase and Other Complicated Flows. *Los Alamos Science*, 22, 1994.

[46] S. Succi, E. Foti, and F. Higuera. Three-Dimensional Flows in Complex Geometries with the Lattice Boltzmann Method. *EPL (Europhysics Letters)*, 10(5):433–438, 1989.

[47] Michael C. Sukop and Daniel T. Thorne Jr. *Lattice Boltzmann Modeling, An Introduction for Geoscientists and Engineers.* Springer, Berlin, Heidelberg, 2007.

[48] David Tarjan and Kevin Skadron. Multithreading vs. Streaming. *MSPC'08*, 2, March 2008.

[49] Nils Thurey. A single-phase free-surface Lattice Boltzmann Method. `http://www10.informatik.uni-erlangen.de/~sinithue/public/nthuerey_030602_da.pdf` Last retrived 15.06.2009, 2002. Friedrich-Alexander-Universität.

[50] J Tolke. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Computing and Visualization in Science*, July 2008.

[51] J. Tolke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *Int. J. Comput. Fluid Dyn.*, 22(7):443–456, 2008.

[52] Charles F. Van Loan. *Introduction to scientific computing: a matrix-vector approach using MATLAB.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2000.

[53] G. Wellein, T. Zeiser, G. Hager, and S. Donath. On the single processor performance of simple lattice Boltzmann kernels. *Computers and Fluids*, 35(8-9):910 – 919, 2006. Proceedings of the First International Conference for Mesoscopic Methods in Engineering and Science.

[54] 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.

[55] Dieter A. Wolf-Gladrow. *Lattice-Gas, Cellular Automata and Lattice Boltzmann Models, An Introduction.* Lecture Notes in Mathematics. Springer, Heidelberg, Berlin, 2000.

Appendix A

D3Q19 Lattice

A D3Q19 lattice was used for both the implementations, with the configuration of the weight factors w_i and the discrete velocities e_i shown in Equation A.1 and Table A.1, taken from [32].

$$w_i = \begin{cases} 1/3 & i = 0, \\ 1/18 & i = 1-3, 10-12, \\ 1/36 & i = 4-9, 13-18. \end{cases} \tag{A.1}$$

Table A.1: The discrete velocities e_i for the D3Q19 lattice that was used

e_0=(0,0,0)		
e_1=(-1,0,0)	e_2=(0,-1,0)	e_3=(0,0,-1)
e_4=(-1,-1,0)	e_5=(-1,1,0)	e_6=(-1,0,-1)
e_7=(-1,0,1)	e_8=(0,-1,-1)	e_9=(0,-1,1)
e_{10}=(1,0,0)	e_{11}=(0,1,0)	e_{12}=(0,0,1)
e_{13}=(1,1,0)	e_{14}=(1,-1,0)	e_{15}=(1,0,1)
e_{16}=(1,0,-1)	e_{17}=(0,1,1)	e_{18}=(0,1,-1)

Appendix B

Annotated Citations

B.1 GPU and GPGPU

[12] describes how to implement NVIDIA CUDA applications, which can be executed on NVIDIA GPUs. It contains several fundamental optimization guidelines for high performance using NVIDIA CUDA.

[9] describes several important implementations considerations that must be met for high performance using NVIDIA CUDA.

[39] gives an introduction to both the history and the current of GPGPU, by describing the background, hardware, and programming models.

B.2 Lattice Boltzmann Method

[45] gives a very accurate and brief introduction to the lattice Boltzmann method. It presents a lot of valuable insight into some of the benefits with the method, also concerning fluid flow through complicated geometries.

[40] and [25] describes D3Q19 model of the Lattice Boltzmann method offloaded to NVIDIA GPUs using NVIDIA CUDA, with several important optimizing considerations for high performance.

[32] describes how to reduce the memory requirements by 50 %, by swapping

instead of duplicating the particle distribution functions to temporary storage in the streaming phase.

[11] describes an approach that can be used to reduce the rounding errors of the lattice Boltzmann calculations to improve the accuracy.

[50] describes D2Q9 model of the lattice Boltzmann method offloaded to NVIDIA GPUs using NVIDIA CUDA, with several important optimizing considerations for high performance.

[51] describes D3Q13 model of the lattice Boltzmann method offloaded to NVIDIA GPUs using NVIDIA CUDA, with several important optimizing considerations for high performance.

B.3 Porous Rocks

[2] presents valuable insights into simulations of fluid flow through porous rocks using the lattice Boltzmann method, and how to obtain the permeability of porous materials, and its dependence on other macroscopic material parameters.

[24] presents a method for the direct estimation of the absolute permeability of porous rocks using the lattice Boltzmann method.

Appendix C

Notur 09 Poster

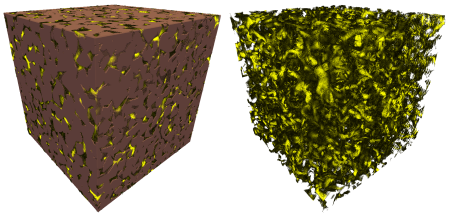
Processing of Porous Rocks on Modern GPUs

Eirik Ola Aksens, Master Student Advisor: Anne C. Elster
Norwegian University of Science and Technology, Department of Computer and Information Science

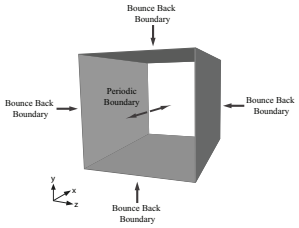
It is important for the petroleum industry to be able to quantify petro-physical properties of porous rocks to gain improved understanding of conditions that affect oil production.

Modern GPUs (Graphical Processing Units) offer with their several hundred cores, applications the potential to harvest amazing amounts of compute power.

In our work, we implement the Lattice Boltzmann Method (LBM) on the GPU. This method is used to estimate the porous rocks' ability to transmit fluid (permeability).



Simulation Setup



The fluids are set in motion with some constant body force. Boundaries parallel to the flow direction are set to solids and with bounce back boundary condition applied, and with the entry and exit boundaries applied with periodic boundary condition.

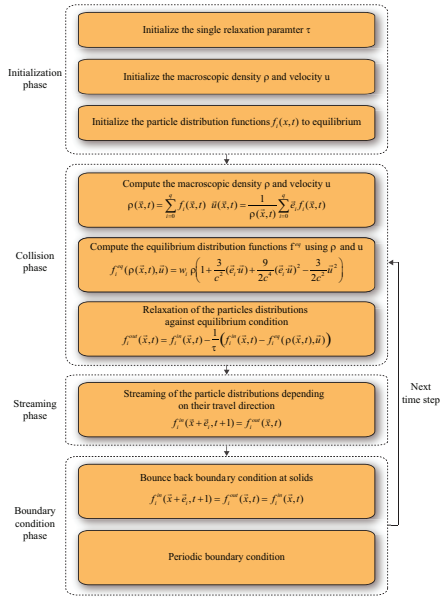
Permeability Calculation

The permeability of the porous rocks can be obtained directly from the generated velocity fields of the LBM, together with using the Darcy's law for the flow of fluids through porous media.

$$q = -\frac{k}{\rho v}\frac{P_b - P_a}{L}$$

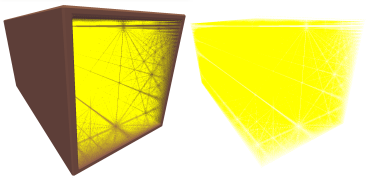
Simulation Model

Fluid flows are simulated by two main operations, the streaming and collision of fluids particles within the lattices, together with some boundary conditions that must be satisfied.



Test Case - Square Tube

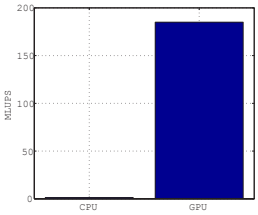
Dimension of lattice: 200x100x100



Analytical permeability: 216 D Numerical permeability: 233 D

Performance characteristic

MLUPS (million lattice nodes updates per second)



Acknowledgements: This projects is done in collaborations with Numerical Rocks AS and the Dept. of Petroleum Engineering (IPT) at NTNU. We would like to thank NVIDIA for providing several of the graphics cards used in this project through Dr. Elster's membership in their Professor Afiliates Program.