TDT4590
Complex Computer Systems,
Specialization Project

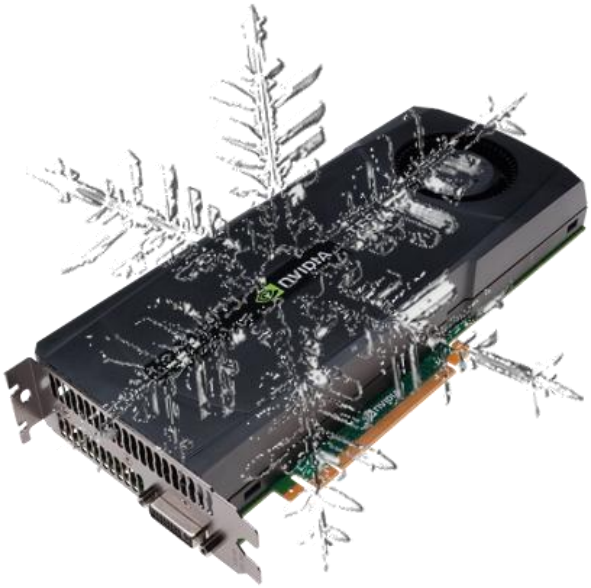
NTNU
Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

Joel Chelliah

The NTNU HPC Snow Simulator on the Fermi GPU

Advisor: Dr. Anne C. Elster

Trondheim, Norway, December 21, 2010



NTNU
Norwegian University of Science and Technology

Problem Description

Earlier master students at NTNU have implemented and continued work on a snow simulator, which is capable of rendering up to two millions snowflakes in real-time, as part of their Master's thesis or specialization projects. Each snowflake is a particle that follows its own independent path, affected by a wind field approximated using either an SOR solver or an LBM solver, and contributes to snow build-up on the ground. The current implementation is a parallel solution that utilizes the GPU for all highly intensive and parallel computations, achieving a considerably extensive and realistic simulation.

This project builds upon the work done previously by Gjermundsen [1] where an LBM fluid solver was introduced to the NTNU snow simulator. In this project, possibilities for optimizing the snow simulator, as well as the LBM solver, for the NVIDIA Fermi GPU architecture will be investigated. Results will be compared with the current version of the snow simulator to measure the performance gain that is achieved from the various optimizations performed on different parts of the code.

Abstract

The NTNU Snow Simulator utilizes the parallel computing powers offered by modern GPUs. The latest version looks at both SOR and LBM solvers for calculating the wind field and was implemented on the NVIDIA GT200 series using CUDA. The snow simulator consists of many intense and highly parallel calculations. This makes the choice of upgrading to the NVIDIA Fermi architecture very attractive as it would provide a lot more compute power for its massively parallel computations, as well as open up new possibilities for expanding the simulation.

In this project we optimize the NTNU Snow Simulator for the NVIDIA Fermi architecture and provide some general optimizations based on best practice methods to maximize performance. We focus only on optimizing the version that uses the LBM solver implemented by Aleksander Gjermundsen, which has been shown to be useful in many applications such as for seismic processing.

Various optimization strategies and techniques on how to best exploit the new features available in Fermi are investigated. Several optimizations based on these are then performed through a number of experiments on both Fermi based GPUs and older GPUs. The results are documented, compared and discussed to evaluate the strength and weaknesses of each optimization. The ideas and techniques that are performed include Fermi-specific optimizations and CUDA best practice methods suggested by NVIDIA, and some general code optimizations suggested by various sources.

We are able to achieve a considerable speedup for the most compute intensive kernels; specifically the collision kernel of the LBM solver achieves a 20%-25% speedup for large problem sizes (256x32x256). The overall performance gain for both single and double precision are close to 40% on small problem sizes (32x4x32) but converge towards 0 as the problem size increases. For the default problem size (128x16x128) the single precision run achieves a performance boost of 18% and we obtain roughly 8.5% for the double precision simulation.The simulation does however gain a huge performance gain from just running on a Fermi device compared to an older device even before any optimizations are done, which shows that it greatly benefits from the increased compute power the new generation has to offer. Taking advantage of the new Fermi specific features that are not available on older GPUs we are able to further optimize the code even after such a large performance gain is achieved.

Acknowledgements

This report is the result of the specialization project done as part of the course TDT4595 - Complex Computer Systems. It was written at the Department of Computer and Information Science at the Norwegian University of Science and Technology.

I would like to thank Dr. Anne C. Elster for providing the idea for this project and her invaluable assistance throughout the project by providing relevant reading material, moral support and helpful guidelines in the structuring of this report. Secondly, I would also like to thank Alexander Gjermundsen for taking the time to help me get familiar with the code of the previous snow simulator and the LBM solver. I would also like to thank NVIDIA for providing several GPUs to Dr. Anne Elster and her HPC-lab through her membership in the NVIDIA's professor partnership program. Last but not least, I would like to thank my fellow students at the HPC lab for many interesting and entertaining discussions of technical topics and motivation throughout the semester.

Trondheim, Norway, December 21 2010

Joel Chelliah

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

The NVIDIA Fermi GPU architecture introduces a wide range of new possibilities for parallel computing that was earlier not present on modern GPUs, while still providing good means of portability from prior generation GPUs. This makes the Fermi GPUs very attractive for massively parallel applications that require a large amount of computation.

When a GPU based solution for the NTNU snow simulator was first created by Eidissen [3] it was using the NVIDIA GT200 series. It consisted of many intense computations that need to be run in parallel to achieve realistic and real-time results. The LBM solver which was added to the simulator by Gjermundsen [1] in a later project, also contained many highly parallel functions that required utilizing the GPU. This makes the Fermi architecture a very important next step for the NTNU snow simulator as well as the LBM solver, to be able to do more intense calculations, explore more advanced methods and achieve a much more precise and realistic simulation.

In this chapter we describe the main motivations behind this project, state the goals that we want to achieve and how we wish to approach them, and provide a short outline of the structure of this report.

1.1 Motivation

The choice of optimizing the NTNU snow simulator for Fermi GPUs, and focusing mainly on the LBM solver, stems from two main motivations. Firstly, the snow simulator is a massively parallel application that will greatly benefit from the increased raw horsepower as well as the newly available features of the Fermi architecture. Should there occur an interest in using this simulator in any future applications (e.g. prediction of snowfall and build-up) it is important that it is adapted to utilize and the increased capabilities provided by newer generations of GPUs to their

full potential. Secondly, the Lattice Boltzmann Method is a powerful technique for the computational modeling and simulation of a wide variety of complex fluid flow problems and is useful in many applications such as seismic processing. The LBM solver, which was originally implemented by Aksnes in his Master's thesis [2], and introduced to snow simulator by Gjermundsen [1], also shows to have a lot of potential on the GPU. The research done on optimizing the LBM solver for the Fermi Architecture in this project can open up new possibilities for further development and more advanced work using this solver in both the NTNU snow simulator as well as other applications.

1.2 Goals

This project focuses on optimizing the NTNU Snow Simulator and the LBM fluid solver for the NVIDIA Fermi GPU Architecture as well as providing general optimization based on best practice methods to maximize performance. Research will be done on the current NTNU snow simulator as well as GPGPU programming and the Fermi GPU architecture to find out what possibilities there are for optimization. A number of experiments will then be performed on various parts of the code, and the results will be documented and compared to evaluate the strengths and weaknesses of the different optimizations.

1.3 Outline

The rest of the report is structured as follows:

Chapter 2: Background provides background information on GPUs, a short history on general purpose GPU programming, the NVIDIA CUDA framework and programming model, and finally the Fermi GPU architecture and its new features that are relevant to this project.

Chapter 3: Current Snow Simulator presents an overview of the current snow simulator, including a short history, a description of the functionalities, the main steps of the simulation, and finally the LBM fluid solver.

Chapter 4: Optimization describes the different kernel functions that will be optimized and provides a detailed list of both general and Fermi specific optimizations that are performed.

Chapter 4: Results and Discussion contains benchmarks of the old and optimized versions of the snow simulator, and provides discussion of the obtained results and an evaluation of the different optimizations that were performed on the most relevant kernels as well as the entire snow simulator.

Chapter 5: Conclusions and further work summarizes what was achieved

during this project and draws conclusions based on the results that were achieved during the final benchmarking phase; followed by some thoughts on possible future work regarding the snow simulator and the LBM solver.

CHAPTER 2

Background

This chapter provides background information on various subjects within GPUs that are relevant to this project, NVIDIA's CUDA framework and the Fermi GPU architecture.

First, Section 2.1 gives an introduction to modern GPUs including some history of their evolution and their use in general purpose programming today. Then, Section 2.2 presents the CUDA framework and programming model from NVIDIA. Finally, Section 2.3 provides a description of the new NVIDIA Fermi architecture and some of the new features that it provides.

2.1 The GPU

The Graphics Processing Unit (GPU) is a specialized processing unit that accelerates the rendering of 2D and 3D graphics. As the task of graphics processing is often a very compute-intensive and parallel problem, this is what GPUs are mainly specialized for and they are designed such that more transistors are devoted to processing the data rather than caching and flow control such as it is on the CPU. This is depicted in Figure 2.1. The design of the CPU has to account for many different things, such as branching, performing memory accesses and extracting instruction level parallelism. The GPU however has a very coherent memory access pattern, a simple flow control and is focused around processing large groups of data in parallel. The primary focus around the development of GPUs has mostly been about achieving improved performance and realistic graphics for both computer and console games. It also has a highly parallel structure meaning that it is capable of performing several instructions at the same time. This makes the GPUs very useful not only in the fields of computer graphics rendering but also for general purpose programs, specifically within the areas of High Performance Computing (HPC). Modern GPUs are now starting to focus on this new branch of applicability so that their power can more easily be exploited for highly intense and parallel computations.

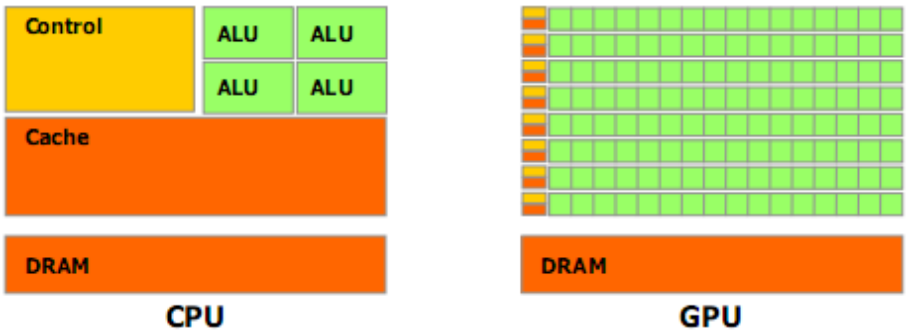


Figure 2.1: The GPU devotes more transistors to processing data, taken from [6] with permission from NVIDIA.

2.1.1 General Purpose GPU (GPGPU)

In the recent years, the development of GPUs has become influenced by the idea of using them for general purpose programming. In the early days of GPGPU doing general purpose programming on a GPU was regarded as a very difficult task with a very steep learning curve; the main reason being that programmers had to write their programs using graphics APIs. For this they needed to first master the graphical languages and then figure out a way to write their general purpose programs using the tools and limitations set by the graphical APIs.

Writing such programs using the graphics API included using shaders. Shaders are the programmable parts of the graphics pipeline that perform specific tasks such as generating vertices, drawing lines between these vertices, creating polygons and then coloring them. These tasks are done by the vertex-, pixel-, and geometry-shaders; and together with of the rest of the elements of the pipeline 3D images are created from the input data. Creating general purpose programs using these shaders was quite a challenge.

In the beginning, languages that were used to program these shaders were mostly High Level Shader Language (HLSL) or OpenGL Shading Language (GLSL), or an extension to these like NVIDIA Cg. As the GPGPU trend slowly grew, improvements were made to the GPUs to expand the programmability beyond these shaders and new programming models were created to simplify the task of such programming and hide the overheads from graphical APIs. In DirectX 10 unified shaders were introduced allowing all three shaders mentioned above to share the same type of operations. This resulted in further developments that favored general purpose programming, which lead to the creation of a more GPGPU friendly frameworks. On such framework is the NVIDIA's CUDA framework.

2.2 CUDA

Compute Unified Device Architecture (CUDA) is a general purpose parallel computing architecture introduced by NVIDIA. The purpose of CUDA is to let programmers be able to write programs in languages such as C, Fortran and OpenCL and have

the code translated into bytecode that can be run on NVIDIA GPUs. This is to relieve the burden of having to write general programs using graphical APIs, which was the case for most traditional shader languages as their main purpose was to render graphics.

Typically a CUDA application will consist of some sequential code that is to be run on the CPU, which we call host code, and some code that is to be run in parallel on the GPU. The language used for writing CUDA is called CUDA C, which is an extension of the programming language C. This extension allows the programmer to write code that is run in parallel across a large number of threads on the GPU. During the call to a kernel, which is a function run on the GPU, the programmer can set up a hierarchical ordering of how the kernel should be executed across several groups of threads.

2.2.1 Kernel Functions

Kernels are data-parallel functions that run in parallel on many threads on the GPU. The kernel is defined using special syntax that denotes the hierarchy of threads it will be running on. The programmer will also need to specify a special classifier that indicates how the kernel is called and where it is supposed to execute, e.g using a __global__ declaration specifier, means that this kernel is run on the GPU and can only be called from host functions. There are also other such classifiers as, __device__ , for code that is run on the GPU and can only be invoked by other kernels, and __host__ defines functions that can only be executed on the host. This is also the default classifier and can be omitted in most cases.

2.2.2 Thread Hierarchy

The concept of threads on the GPU is quite similar to threads on the CPU. A thread is the most basic unit that executes on the GPU and it can have its own variables and control flow independent from all other threads. Threads are grouped into a hierarchy of thread blocks which contain several threads, and grids which are arrays of several thread blocks. This three level hierarchy is illustrated in Figure 2.2. When a kernel function is called, it executes as a grid of thread blocks, where each grid can be a one- or two dimensional array, and each thread block can have up to three dimensions. The threads within each thread block have several built-in identification variables that can be used to determine their unique locations at both the grid and thread block level. When a kernel is finished executing there is an implicit synchronization of all the threads, however threads within a thread block can synchronize during a kernel execution by calling the __syncthreads() function.

2.2.3 Memory Hierarchy

As mentioned above, kernels are executed in parallel across multiple threads on the GPU. These threads have several memory spaces to access data from during their execution.

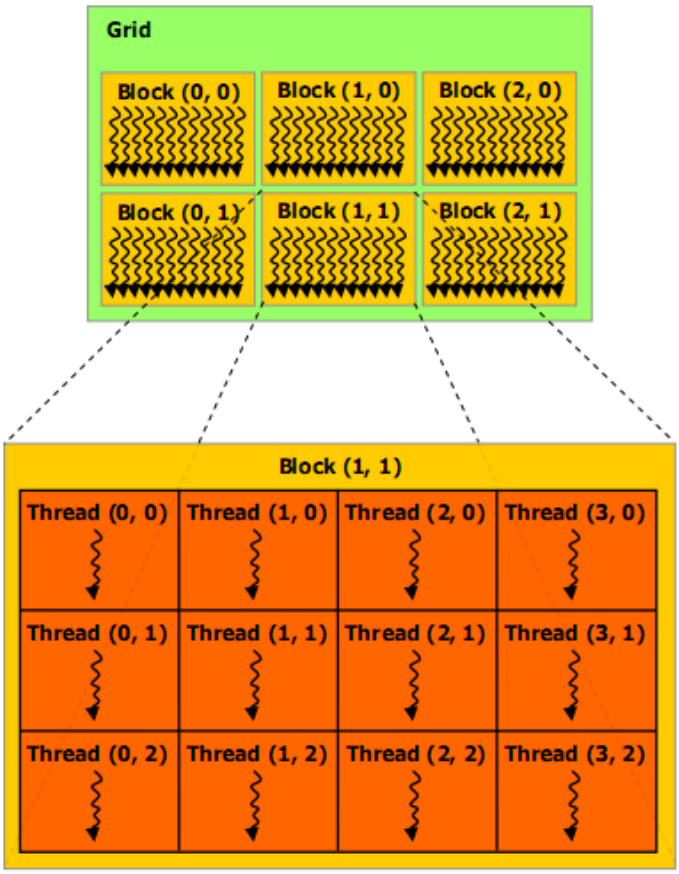


Figure 2.2: CUDA thread hierarchy, taken from [5] with permission from NVIDIA.

Registers and Local Memory

Each thread has its own private local memory and also a set of registers. The registers are naturally the fastest data storage but the total amount of registers available to each thread block is limited. The local memory for each thread is actually stored on the global memory meaning that it has high latency and registers should be used instead whenever possible. If a kernel uses more registers than there are available, register spilling occurs. This means that the data gets put into local memory instead.

Shared Memory

Each thread block has a shared memory portion that all the threads within can use to share data. The lifetime of shared memory variables are the same as the thread block's. Access to shared memory has higher latency than for registers but is still a lot faster than global memory accesses.

Global Memory

All threads can access the global memory which is very large but has high access latency. When memory transfer is taking place to and from the GPU, memory is moved in and out of global memory using different CUDA API calls. It's impor-

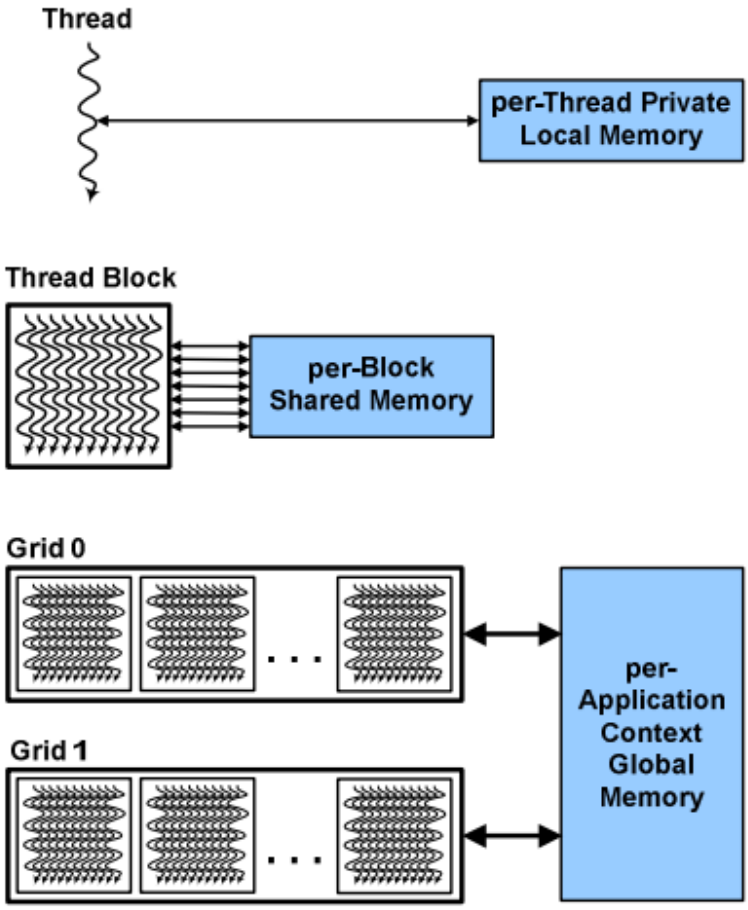


Figure 2.3: CUDA memory hierarchy, taken from (cite here) with permission fron NVIDIA.

tant that memory that is read and written from and to global memory should be coalesced. This means that each thread in a warp accesses the corresponding word segment (i.e. thread k accesses the k-th word in a segment). Threads can additionally also access two other memory spaces which also reside in global memory, but can be accessed a lot faster under certain conditions. These are texture memory and constant memory.

Constant and Texture Memory

These are read-only, which means that they cannot be altered from inside the kernels. Any variables that are to be placed in texture or constant memory must be defined in the host and copied over to the GPU before the kernel is called. They are both cached on the SMs and repeated lookups can be much faster than accessing regular global memory. Constant memory is specifically used for storing constants and texture memory is used to bind a section of memory as a cached texture.

2.3 NVIDIA Fermi Architecture

This section will give a brief overview of the NVIDIA Fermi GPU and also cover the various changes and additional features in the new architecture that are the most relevant to this project. Features that are considered not to be so relevant will be skipped. The information provided in this section is based on NVIDIA's Fermi white paper [5]. We refer to this if there is any interest in reading about all the available features and detailed specifications of Fermi. The latest generations of NVIDIA's

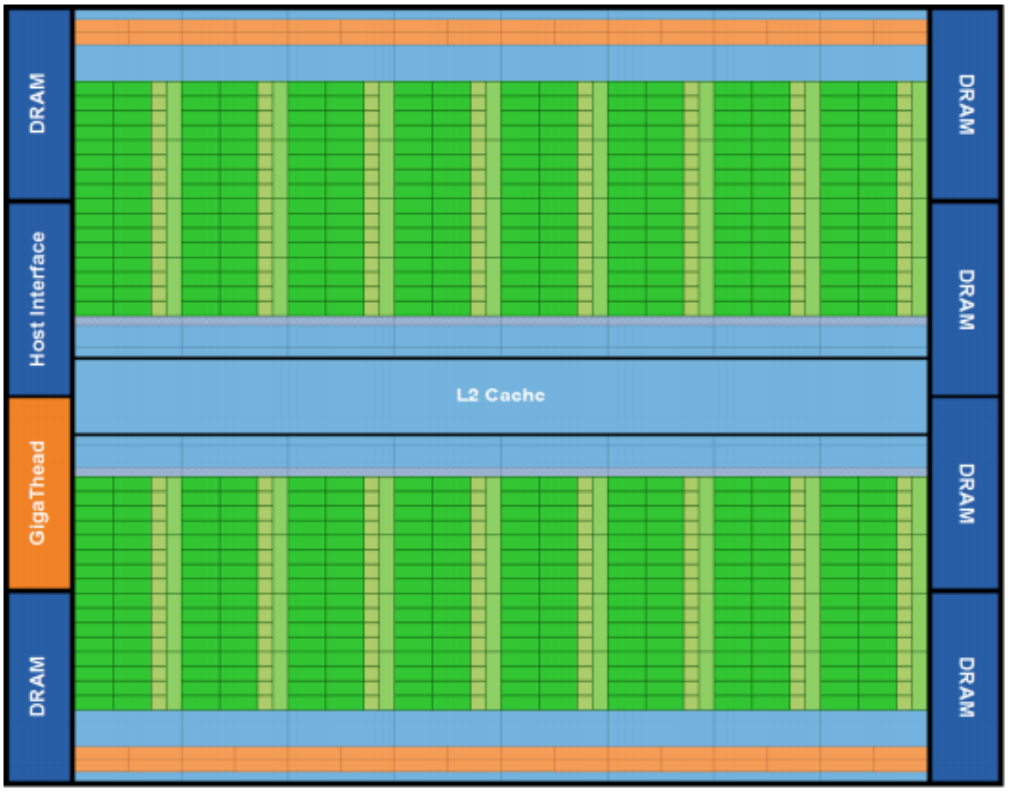


Figure 2.4: Fermi Architecture, taken from [5] with permission from NVIDIA.

GPUs are based on the NVIDIA Fermi architecture. It consists of 16 streaming multiprocessors (SM) that contain 32 CUDA cores each, and each core is able to execute a floating point or integer instruction per clock for a thread. As we can see in Figure 2.4, the 16 SMs, which are depicted as rectangular blocks, are placed around a common L2 cache. Each SM consists of a thread scheduler, execution units and L1 cache, denoted by the orange, green and blue portions respectively. The GigaThread global scheduler is responsible for distribution of thread blocks to each SM's thread scheduler; the Host Interface manages the connection between the GPU and the CPU, and the remaining space is dedicated to DRAM memory.

2.3.1 The Streaming Multiprocessor

Each SM on the Fermi GPU features four times as many CUDA cores as the previous generation, each of them consisting of a pipelined arithmetic logic unit (ALU) and floating point unit (FPU) as we can see in Figure 2.5. The SMs also provide

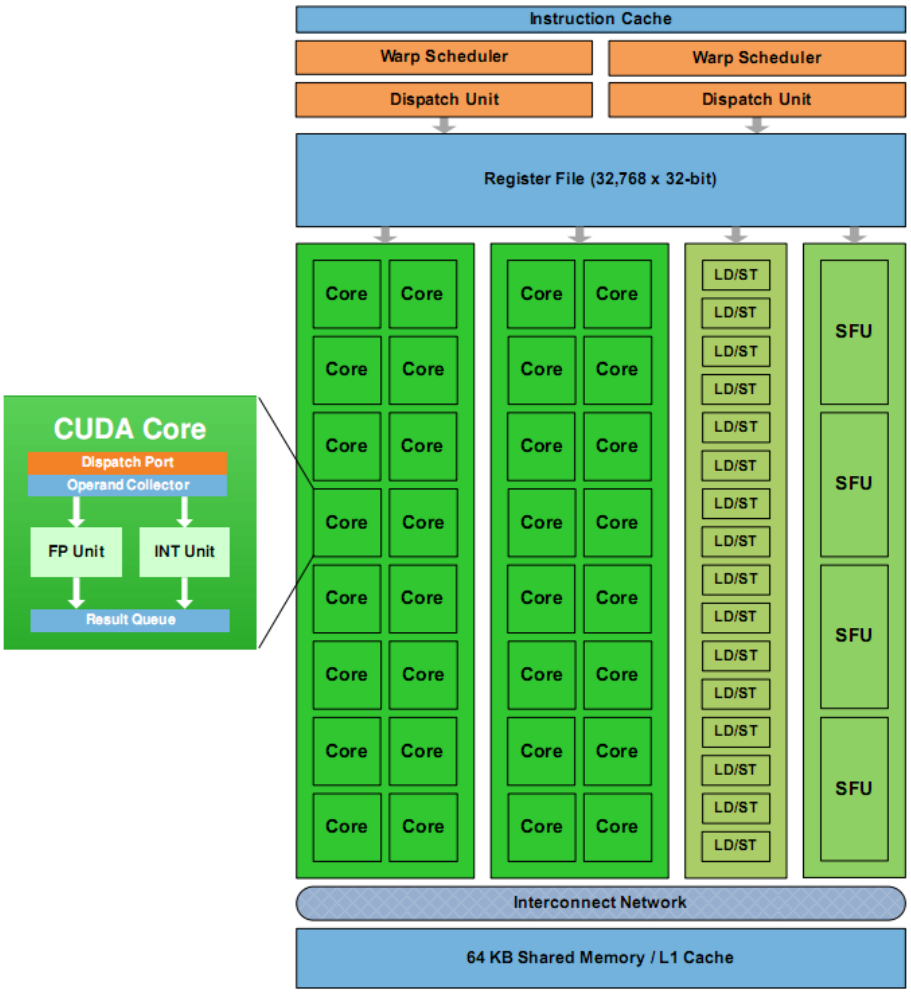


Figure 2.5: Fermi streaming multiprocessor, taken from [5] with permission from NVIDIA.

fused-multiply-add (FMA) instructions, which are floating point multiply and add operations performed in a single step, for both single and double precision. Another improvement that's worth noting is that the ALU now supports 32-bit precision for all instructions, while in prior designs the ALU was limited to only 24-bit precision multiply operations. In addition to the 32 CUDA cores, each SM also has 16 load/store units for calculating the addresses for load or store operations of 16 threads per clock cycle. It also has four special function units (SFU) to handle transcendental instructions, such as taking the square root or finding the reciprocal.

Threads are scheduled in groups of 32 called warps. As we can also see in Figure 2.5, each SM has 2 warp schedulers and instruction dispatch units, making it possible for two warps to be executed concurrently. This was not possible in previous generations as they only featured one warp scheduler. The dual warp scheduler in Fermi simultaneously issues instructions from two warps to the CUDA cores, the load/store units or the SFUs. Most instructions, apart from double precision ones, can be dispatched this way.

2.3.2 Fermi Memory Hierarchy

The new architecture takes into consideration that some algorithms work well with a lot of shared memory while others are more dependent on having a lot of cache to get good performance. To be able to adapt to both situations, the Fermi architecture is designed so that each SM has a 64KB memory on which can be configured to be either 48KB of shared memory and 16KB of L1 cache, or 46KB of L1 cache and 16KB of shared memory. There is also an additional unified L2 cache that is common for all SMs. This differs from the previous generation where the memory architecture consisted of a fixed shared memory for each SM and no explicit cache.

2.3.3 Concurrent Kernel Execution

Fermi supports running several small kernels concurrently as long as there are enough thread blocks available. Earlier generation devices ran all kernels in a serialized order, which meant that a kernel that only needed a few number of threads would occupy the entire device while the other kernels had to wait their turn. The only way to utilize all multiprocessors in earlier devices was to launch a single kernel with as many thread blocks as there were multiprocessors available. This is displayed on the left part of Figure 2.6. The image of the concurrent kernel execution on the right side shows how these kernels would run on a Fermi device assuming that they are all independent kernels that do not need to be executed in any particular order.

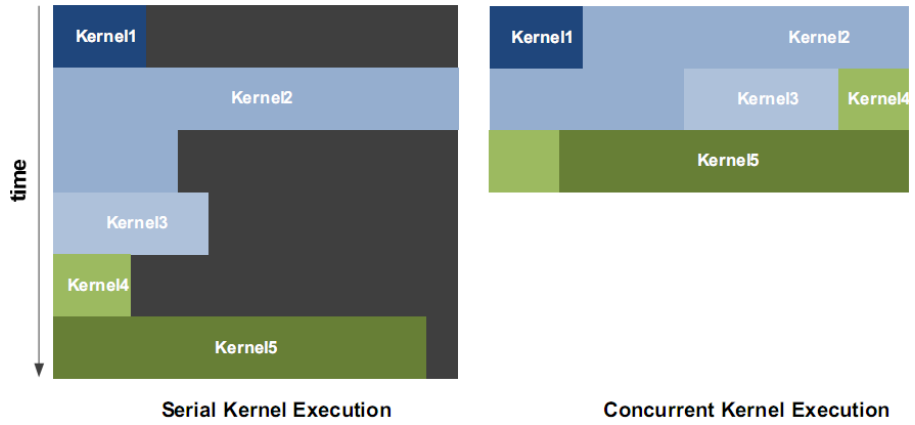


Figure 2.6: Serial and concurrent execution of the same kernels, taken from [5] with permission from NVIDIA.

CHAPTER 3

Current Snow Simulator

This Chapter provides a detailed overview of the current NTNU Snow Simulator and the LBM fluid solver.

We start by providing some history and functional overview in Section 3.1. Then we move on to a step by step description of the main loop in Section 3.2.2. Finally we take a look at the algorithm behind the LBM fluid solver, as well as a step by step walkthrough of how this solver is used in the simulation, in Section 3.3.

3.1 Application Overview

The NTNU Snow Simulator is an application that renders realistic simulation of snowfall in 3D graphics. Up to two million snow particles can be simulated in real-time on modern GPUs, and each particle follows its own independent path in a slight spiral curve, but is also affected by a number of outside forces such as gravity, wind and pressure. The realistic simulation of the wind field can be achieved using two different fluid solvers; the SOR solver and the LBM solver. The resolutions of the wind field can be adjusted to find a good tradeoff between performance and realistic motion. As we run the simulation, the scene starts off with a bird's-eye-view of a mountainous, green terrain as snowflakes slowly fall from the sky. Over time, the snow gradually builds up on the ground covering the entire area with a steadily rising blanket of snow. Naturally, the wind will affect which areas get covered first and how the snow build-up is distributed across the terrain. Each time a snow particle reaches the ground, a new one is spawned at a random position in the sky. In Figures 3.1 and 3.2 we can see how the scene looks at the very beginning compared to 5 minutes into the simulation. Here we are only looking at a small part of the terrain.

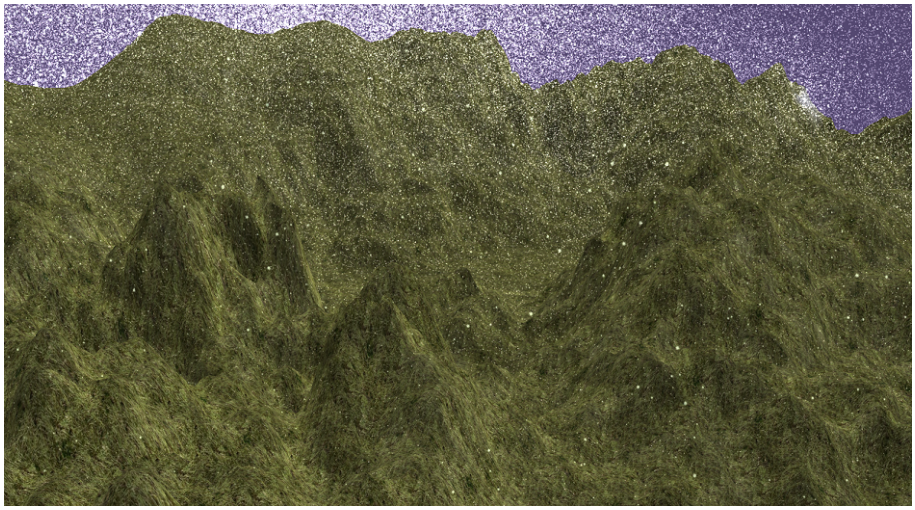


Figure 3.1: Screenshot of the snow simulator right after starting the application.

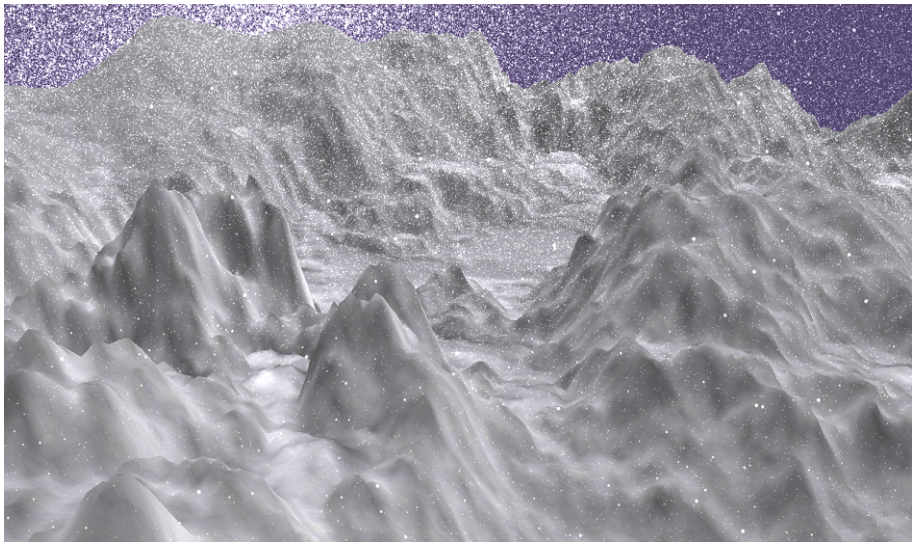


Figure 3.2: Screenshot of the snow simulator, after 5 minutes of snowfall.

3.1.1 Breif History

The snow simulator was first created by Ingar Saltvik for his Master's thesis in 2006 [4]. His thesis focused on achieving near real-time realistic simulation of snow by optimizing and parallelizing a solver for the Navier-Stokes equations, on a dual-core processor. Since then, two other students have worked on this application providing various enhancements. The first was Robin Eidissen [3] who developed a GPU-based solution to achieve a more realistic and large-scale simulation, and then Aleksander Gjermundsen [1] who introduced an LBM solver for the wind simulation and compared it to the previously used SOR solver.

3.1.2 Additional Functionalities

In addition to being able to simulate a large amount of particles the application also provides some other functionality for the purpose of debugging and enhancing

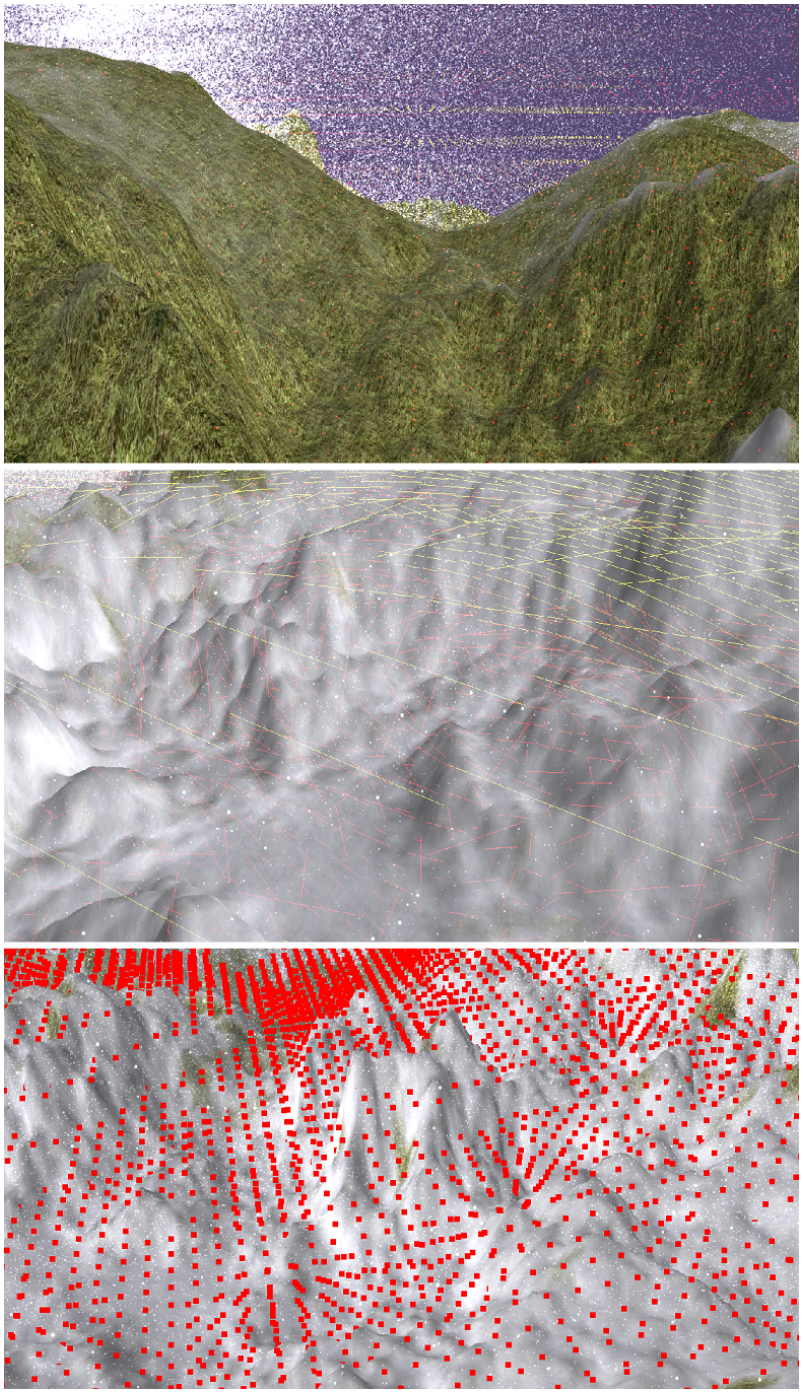


Figure 3.3: Screenshot of the snow simulator with the three debug rendering modes enabled. 1) Wind velocity vectors, 2) Pressure field, 3) Obstacle field.

the visual experience. While the simulation is running, the user is able to navigate around the scene using the arrow keys and the mouse and observe the snowfall from all directions and angles.

The application also features several debug rendering modes which the user can switch between during the simulation. These are displayed in Figure 3.3. The first one draws the wind velocity vectors that show the direction the wind is blowing,

the second one draws the pressure field and the last one displays the obstacle field for the wind. There are also several keyboard shortcuts for enabling and disabling parts of the simulation, like the snowfall, snow build-up, wind and terrain.

Additionally, it is possible to view the entire simulation in quad-buffered stereoscopic 3D. This makes it possible to view the simulation in 3D on monitors that have 120Hz refresh rate using LCD shutter glasses, but this is only supported by a few graphic cards like the NVIDIA Quadro cards. However it's also possible to view the simulation on regular screens using passive glasses.

3.2 The Main Code Overview

The code for the NTNU Snow Simulator is mostly written in C++ and CUDA C and it uses OpenGL for rendering. Additionally some open-source libraries are used for window handling, uploading textures and some various OpenGL operations. These are, among others, GLFW, GLEW and SOIL. In addition to these there are also a bunch of CUDA libraries provided by the CUDA SDK that are used.

3.2.1 Setup and Initialization

The main code starts by loading some configuration settings from a text file. In the configuration file it's possible to specify certain options such as the number of particles, snow build-up rate, which solver to use and the resolution of the wind field. Then some window handling is done before initializing the rendering and initializing CUDA on the device. Thereafter the terrain is initialized by loading the height map and normal map according to the dimensions specified in the configuration file. Then the wind simulation is initialized using either the SOR solver or the LBM solver, finally followed by the initialization of the snow particles. All these initialization methods mostly consist of allocating memory in the CPU and GPU for the oncoming simulation. Then the actual simulation can commence as the program enters the main loop.

3.2.2 The Main Loop

For each frame the application runs through a loop performing a set of computations and updates. The steps of this loop are displayed in Figure 3.4 and explained in the following list.

1. Update counters for maintaining the time and number of frames that have passed, and calculate the current fps (frames per second) rate and print this to the console.

2. Update the obstacle field based on the number of new particles that have landed on the terrain. Areas that have been hit by snow particles need to be heightened. The obstacle field is updated on the CPU and the result is then sent to the GPU to be used in the wind simulation.

3. Check for keyboard input to see if the user has moved the camera or changed any of the settings that are configurable during runtime.

4. Simulate the wind field. Computations are done based on which solver that is currently enabled and the current state of the obstacle field. The updated wind field is then kept in a 3D texture so that it can easily be looked up by the particles.

5. Update all the snow particles. Their new positions are calculated based on all the forces that affect them, including the newly updated wind field. This is also the step where any particle that hits the terrain respawns at a random location in the sky.

6. Render the scene. This is done using OpenGL, with most of the rendering done using shader programs. If stereoscopic rendering is enabled, two frames with different camera positions will be rendered for each frame.

The application jumps out of the main loop and terminates when given an exit command or if it has passed a certain time or frame limit that was set prior to execution. During the termination phase allocated memory is freed from the CPU and GPU.

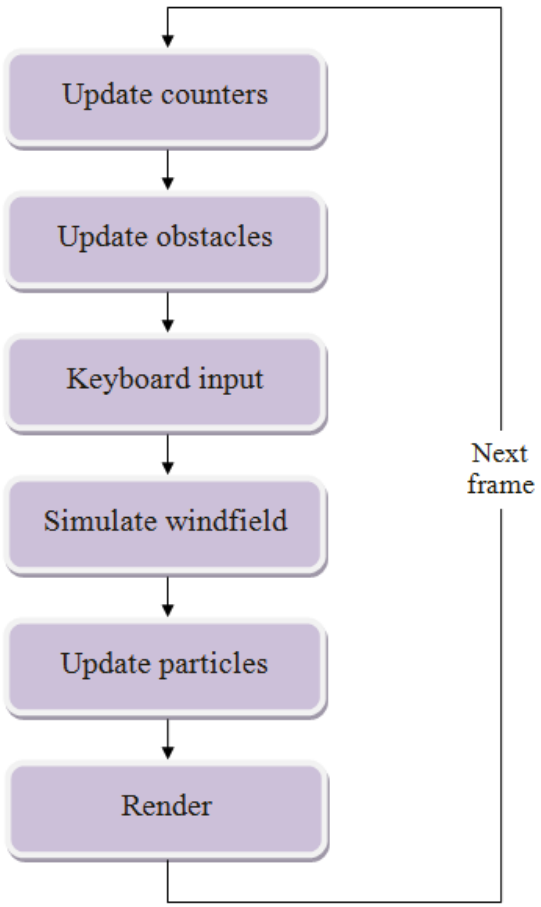


Figure 3.4: The main loop of the snow simulator.

3.3 Fluid Simulation

The application can be run using two different fluid solvers for the wind field. One is the Successive Over-Relaxation (SOR) solver that is based on iteratively solving the Navier-Stokes equations, and the other is based on the Lattice Boltzmann Methods (LBM). A short overview of the LBM solver will be given in the following sections, but we will not be covering the SOR solver as it is not relevant to this project. There is extensive information regarding the Navier-Stokes equations and the details of the fluid simulation using the SOR solver in the reports of Saltvik [4], Eidissen [3] and Gjermundsen [1].

3.3.1 Overview of the LBM Solver

The LBM implementation that is used in the snow simulator is based on the one made by Aksnes for his Master's thesis on fluid flow through porous rocks[2]. It was then introduced to the snow simulator by Gjermundsen [1] who adapted the solver so that it could be used for the wind simulation.

Lattice Boltzmann Methods

The Lattice Boltzmann methods is based on partitioning the domain into discrete lattice nodes, with each node having a set of directions. Each direction has a corresponding distribution function that describes the probability of particles moving along this direction. The combined result of all the nodes simulates a fluid. We describe the lattice of the nodes by the form DxQy. In the previous project involving the snow simulator [1], Gjermundsen used D3Q19 lattices to represent the domain. This means that each lattice node is a 3 dimensional cuboid with 19 directions (particle distribution functions); 18 of them pointing to other nodes and one pointing to itself.

The LBM Algorithm

The idea behind the LBM algorithm is that fluid flow is simulated by interacting particles within a lattice. These particles stream from one lattice node to another at each discrete time step where they collide with other arriving particles. This affects the evolution of the particle distribution functions, and the fluid flow is described by how these distribution functions evolve throughout each time step.

Solving a fluid system can be broken down to four main steps as shown in Figure 3.5:

1. **Initialization phase**. The particle distribution functions and various properties of the fluid are initialized.

2. **Collision phase**. The local particle distributions are updated based on the surrounding distributions.

3. **Streaming phase**. The updated local distributions are sent to the surrounding lattice nodes.

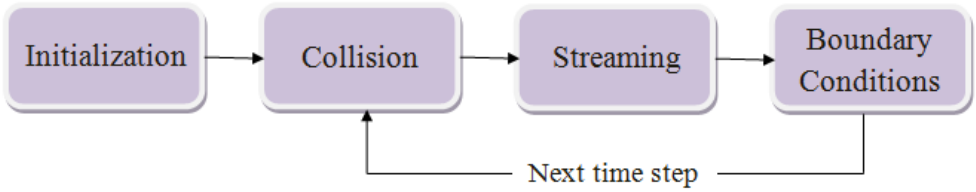


Figure 3.5: The phases of the LBM fluid solver.

4. **Boundary conditions**. Apply boundary values to solid-fluid interfaces to account for particles reaching the solid boundaries.

3.3.2 Fluid Simulation with the LBM Solver

Step four of the main loop calls a number of functions that are responsible for simulating the wind field. The main steps of the LBM fluid simulation routine are displayed in Figure 3.6 and explained in the following list.

1. Calculate the new velocities of the domain boundaries and scale the values relatively to the lattice dimensions.

2. Perform the collision step to calculate new local values of the particle distribution functions.

3. Perform the streaming step to distribute the new vales to the neighboring lattice nodes.

4. Apply new velocities to the outer boundary cells. This has to be done for all six planes of the cuboid shaped domain.

5. Create a velocity field that contains the macroscopic velocity vectors of all lattice nodes.

6. Map the velocity field to a texture array.

Each of these steps except for steps one, four and six corresponds to a kernel function. Step four is performed by three kernels, where each kernel applies velocities to different outer boundary cells. Once all these steps have been executed, the main loop moves on to the next step where it applies the new velocities calculated here to update the particle positions.

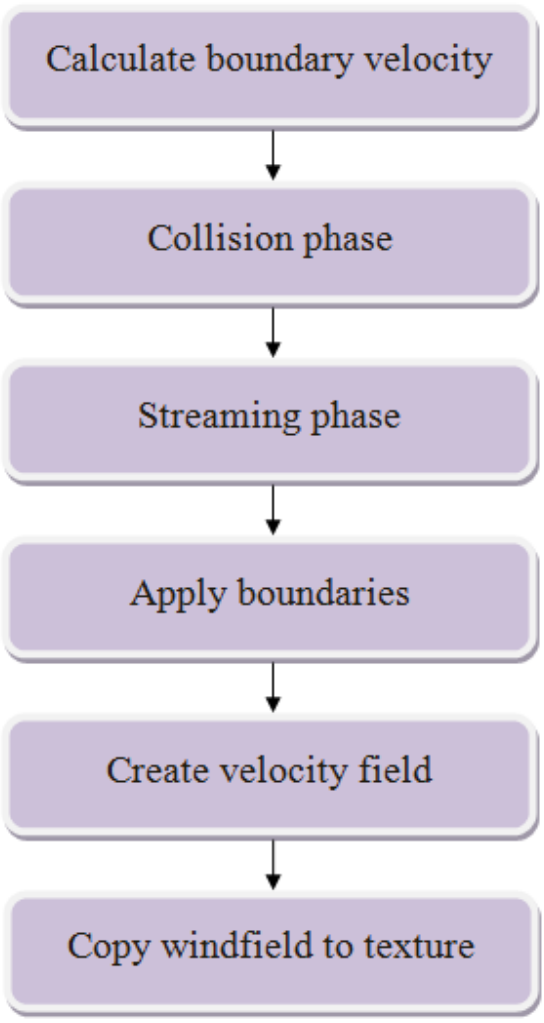


Figure 3.6: The steps of the LBM fluid solver.

CHAPTER 4

Optimization

Prior to doing any kinds of optimizations it's important to first get a good grasp of the different optimization techniques, why they are relevant for the given task, and in which ways they would provide increased performance. This chapter gives a description of all optimizations which are investigated, considered and performed on the snow simulator.

Firstly, Section 4.1 gives a brief overview of the different kernel functions that will be optimized. Secondly, Section 4.2 displays kernel profiling results of the current code to establish a starting point for the optimizations. Thirdly, Section 4.3 investigates possibilities for general optimization of these kernels. Finally Section 4.4 look into Fermi-specific optimization strategies.

All the ideas listed in the sections below are tried out on the current snow simulator throughout several experiments. The results are documented and provided in Chapter 5 followed by some analysis and discussion. Note that this chapter does not go deeply into any implementation details regarding the various optimization strategies. Instead, an overall idea of each strategy is provided here with some explanation to how it is relevant. As the results of the experiments are discussed in Chapter 5, more in-depth explanations of the implementation are revealed wherever they are deemed necessary for clarification.

4.1 Overview of the Kernel Functions

In the following sections we look at the kernels responsible for the various parts of the snow simulator and the LBM solver.

4.1.1 Particle Simulation Kernels

These kernels are responsible for updating the snow particles and the terrain.

- part_update<<< ... >>> (...)
- smooth_ground<<< ... >>> (...)

The *part_update* kernel has the task of calculating the new positions of the snow particles based on the various forces the particles are subject to. The *smooth_ground* kernel smoothes the terrain as the snow builds up on top of it.

4.1.2 LBM Kernels

These kernels are responsible for calculating and updating the wind field.

- collideAndSwapGPU_RegularBad<<< ... >>> (...)
- collideAndSwapGPU_kernelShared3<<< ... >>> (...)
- collideAndSwapGPU_kernelPrecise2<<< ... >>> (...)
- collideAndSwapGPU_kernel5Double<<< ... >>> (...)
- streamAndSwapGPU_kernel<<< ... >>> (...)
- applyOuterBoundariesGPU_kernelYZ<<< ... >>> (...)
- applyOuterBoundariesGPU_kernelXY<<< ... >>> (...)
- applyOuterBoundariesGPU_kernelXZ<<< ... >>> (...)
- createVelocityField_kernel2<<< ... >>> (...)

The kernels listed here are called during the different phases of the fluid simulation loop described in Section 3.3.2. The *collideAndSwap* kernel performs the collision phase. Four different variants of this kernel exist: Regular, shared, precise and double. They all perform the same task but are implemented with regards to different levels precision, use of shared memory and caching. The *streamAndSwap* kernel executes the streaming phase, followed by three *applyOuterBoundariesGPU* kernels that apply the new outer boundary values to all 6 planes enclosing the fluid domain. The *createVelocityField* kernel is responsible for copying the wind field to a texture.

4.1.3 Other Kernels

The remaining kernels which have not been mentioned yet are mostly initialization functions for allocating memory on the GPU for the particles, terrain and wind field. These kernels are not investigated any further as they are only called once and their individual performance rates quickly become irrelevant after just a few frames into the simulation. The kernels responsible for the different debug rendering modes mentioned in 3.1.2 are also not investigated any further.

4.2 Profiling

To gain a more detailed overview of each kernel's execution and resource usage, the simulator is run through the NVIDIA Compute Visual Profiler. Once the simulation is executed for a given number of frames the profiler outputs a range of information for each kernel, such as grid size, block size, shared memory, registers per thread, and occupancy. Additionally the profiler is also capable of creating different plots based on the execution time of kernels and memory transfer operations.

Profiling the current version of the snow simulator, with a wind resoltion of $192x24x192$ and 262144 snow particles, the Compute Visual Profiler gives the following output listed in table 4.1. The table shows warp occupancy, block dimensions, dynamic shared memory and registers per thread for the kernels we will be focusing on.

Kernel	**Occupancy**	**Block size**	**Shared**	**Reg./thread**
collideAndSwap (Regualar)	0.75	[192 1 1]	0	26
collideAndSwap (Shared)	0.167	[192 1 1]	1824	63
collideAndSwap (Precise)	0.875	[192 1 1]	0	21
collideAndSwap (Double)	0.375	[192 1 1]	0	48
streamAndSwap	1	[192 1 1]	0	15
applyBoundaries (YZ)	0.167	[24 1 1]	0	23
applyBoundaries (XY)	0.875	[192 1 1]	0	23
applyBoundaries (XZ)	0.875	[192 1 1]	0	23
createVelocityField	0.875	[192 1 1]	0	23
part_update	1	[256 1 1]	0	20
smooth_ground	0.5	[16 16 1]	5184	16

Table 4.1: Profiling results of the current snow simulator

These profiler results are used as a starting point for the optimization strategies considered in the following sections. The profiler also outputs the execution time of each kernel. These will be used to determine whether the optimization results in a performance gain or not.

4.3 General Optimizations

Before considering Fermi-specific optimization ideas, the existing code is inspected to see if any general optimizations are possible in the form of restructuring the code. This is to make sure that the control flow is designed to allow parallel execution of data whenever possible, and that thread scheduling and memory management are handled in a correct and most desirable way. This is done by first checking if the code follows the basic programming guidelines provided in the CUDA Best Practice Guide [7], and then fixing code that seems to deviate from these guidelines. The high priority points given in the guide are:

- Find ways to parallelize sequential code.
- Minimize data transfer between host memory and device memory.

- Select appropriate launch configurations to maximize device utilization.
- Make sure that all global memory accesses are coalesced.
- Use shared memory whenever possible.
- Avoid branching within the same warp.

Most of these points were of course taken into account by the Master students who previously worked on the snow simulator, so naturally the code is in fairly good condition and already follows most of the points listed above. However, after a thorough inspection of the code it shows that there is room for a bit more optimization in the areas of maximizing warp occupancy, and avoiding branching within warps for some kernels.

4.3.1 Maximizing Warp Occupancy

Maintaining high warp occupancy is essential for good kernel code performance. When a kernel has high warp occupancy, this means that there will always be warps available for execution and an SP will never have to waste time waiting to receive a warp. According to [10] there are three basic ways to accomplish this:

- By having sequences of independent instructions within a warp so that it can always make forward progress.
- By placing many threads per thread block so that at least one warp can execute while others are stalled on long latency operations.
- The hardware can assign up to eight independent thread blocks to an SM.

Although these points provide valuable information on maximizing occupancy, we are bound by the limitations of the kernel we are trying to optimize. For example, if the number of threads per block is dependent on several other factors, we cannot simply increase this number to gain higher occupancy. It should also be noted that optimization affects several aspects, so following one of the points above does not guarantee a performance increase. An optimization may increase instruction count or resource usage, making it not much of an optimization after all. So it is crucial to be aware this tradeoff when considering such strategies.

The CUDA occupancy calculator is an xls sheet that lets you easily figure out which configurations of thread count, registers and shared memory should be used to get high occupancy. Two of the graphs that it provides can be seen in Figure 4.1. On the figure to the left we see that the current per-block thread count that the kernel uses is displayed as a red triangle on the graph, and we can see for which values it is possible to achieve maximum occupancy. The figure to the right shows how many registers can be used by each thread before it will start having any negative impact on the warp occupancy given the current settings. The NVIDIA Compute Visual Profiler is also used in this area as it can easily output the warp occupancy and other essential information of all the kernels in the application. This tool is very useful to quickly determine the outcome of optimization experiments and, together with the occupancy calculator, is used as a starting point to finding the optimal settings for maximizing occupancy for each kernel.

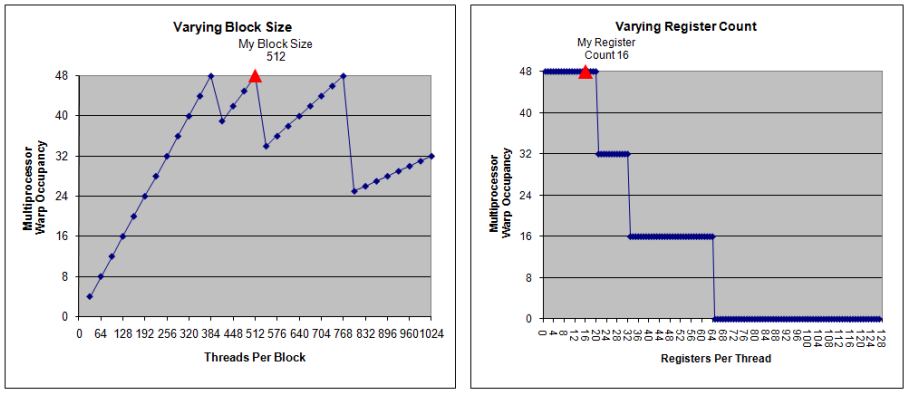


Figure 4.1: Screenshot of Cuda occupancy calculator. 1) Shows warp occupancy for different groupings of threads per block. 2) Shows Warp occupancy in relation to registers per thread.

4.3.2 Minimizing Branching

Divergent execution paths within the same warp can lead to these paths being serialized and run sequentially. This can occur as a result of branching (e.g. flow instructions such as *if* or *switch* within a kernel that tell some threads to do one thing and others to do something else). If this happens to threads within the same warp we end up with two diverging paths. After both paths have been executed in a serialized fashion, the threads will converge back to the same execution path. There are basically two ways to avoid divergence of execution paths: Try to avoid branching that depends on the id of the threads, or make sure that the branching does not split up the 32 threads within the same warp.

The kernels we will specifically be looking at regarding this issue are the three *applyOuterBoundariesGPU* kernels. Each of them updates the boundary values of two of the bounding planes of the cuboid shaped fluid domain. This is done by splitting the threads into two groups so that each group can calculate boundary values for each of the two planes. Such a strategy may work perfectly for a given set of wind resolutions, but it is obvious that we cannot guarantee that the execution path will not diverge for any wind resolution. Experiments will be done on rewriting the branching conditions to minimize divergent paths, or removing them completely to avoid divergent paths.

4.4 Optimizing for Fermi

Prior to looking at Fermi based optimization strategies, it's necessary to ensure that the snow simulator is compatible with Fermi. The main points in enabling Fermi support is to build the application using Cuda Toolkit version 3.0 or later, and making sure that the nvcc compiler creates code that targets the Fermi architecture by setting some additional flags . The details of this process are given in the NVIDIA Fermi Compatibility Guide [8].

The majority of Fermi-specific optimization ideas and possibilities were taken from

the NVIDIA Fermi Tuning Guide [9]. The following sections will give a brief explanation of these techniques and look at how they are relevant to optimizing the snow simulator.

4.4.1 Multiple Concurrent Kernel Launches

In Section 2.3.3 we saw that Fermi devices are capable of executing multiple kernels concurrently. For the snow simulator this opens possibilities for running the three *applyOuterBoundariesGPU* kernels at the same time. As these kernels are independent of each other, the application can be asked to fill the device with these kernels instead of running them one by one. This is done by placing each of the kernels in a separate stream by using CUDA streams. The application will try to run these streams as concurrently as possible based on the availability of the multiprocessors.

4.4.2 L1 Cache and Shared Memory

On Fermi devices the same on-chip memory is used for both L1 and shared memory, and it is possible to specify for each kernel whether it should dedicate more memory to L1 caching or shared memory. This feature is discussed in more detail in Section 2.3.2. Most of the current kernels don't utilize any shared memory, so for these we would naturally want to set the preference to L1 cache. Two of the kernels use shared memory. These are the *smooth_ground* kernel and the shared memory version of the *collideAndSwap* kernel. For the *collideAndSwap* kernel the choice is not clear whether to set preference for L1 or shared memory, since it depends on the resolution of the fluid domain which governs the amount of dynamic shared memory used by the kernel. This choice is made based on experiments. The *smooth_ground* uses a lot of shared memory so the memory configuration for this kernel is obvious.

There are also some other possibilities for controlling the L1 cache. Global memory caching in L1 can be disabled at compile time so that only the L2 cache is used for this. This has to be done using a flag for the nvcc compiler so it will affect all kernels. Local memory caching cannot be disabled; however it can be controlled by limiting the number of registers (using the -maxrregcount compiler flag), thus increasing the likelihood of register spilling.

4.4.3 Global Memory Access

Global memory accesses are processed per warp on Fermi devices. It is therefore recommended in [9] that kernel launch settings should be adjusted to account for this, by making sure that two-dimensional thread blocks have their x-dimension be a multiple of the warp size. The current *smooth_ground* kernel is launched with thread blocks of 16x16 threads, so changing this to 32x32 might lead to a performance increase. However this is not guaranteed as this kernel does use a lot of shared memory to avoid expensive global memory access.

4.4.4 32-Bit Integer Multiplication

On GPUs prior to Fermi, 32-bit integer multiplication is not natively supported and it is more common to use the __mul24 intrinsic for better performance. On Fermi, 32-bit integer multiplication is natively supported while 24-bit is not, therefore __mul24 should not be used. In the current code all integer multiplication are done using the __mul24 intrinsic. Changing this to 32-bit is expected provide a small overall performance gain.

4.4.5 Reduced Precision

Fermi devices perform calculations that are close to the IEEE floating point standard than previous generations (e.g. addition and multiply are combined into FFMA instructions instead of FMAD). This ensures more accurate results for single precision floating point than in older devices but may affect performance. There are some nvcc compiler options which can be set to make the compiler generate code that is closer to the ones generated by devices of the previous generation. More information on these options can be found in [6].

CHAPTER 5

Results and Discussion

In this chapter, the various experiments done in the previous chapter are reviewed and their effects on the different kernels as well as the overall performance are discussed.

In Section 5.1 the test environments used throughout the project are described. Section 5.2 looks at the performance gain that was achieved by running the old code on Fermi without any optimizations. Finally Section 5.3 evaluates the results of all the optimizations done on the individual kernels as well as the overall performance.

5.1 The Test Environment

In his report, Gjermundssen [1] mentions that he used a Tesla C1060 to run benchmarks on the snow simulator and both fluid solvers. He also states that the he tested that his algorithms work correctly on other GPUs. These include the GTX280 and the Quadro FX 5800. These three cards are all based on the GT200 series and have approximately the same number of CUDA cores. Their main differences lie in some additional features and the total amount of global memory.

During this project, the snow simulator has been tested on a range of Fermi-based GPUs, including the GTX460, TeslaC2070 and the Quadro 5000. Similarly to the ones in the GT200 series, the main differences between these new cards are the size of the memory, clock frequencies and some additional features (e.g. only the Quadro 5000 supported quad-buffered stereoscopic rendering). Two different machines were mainly used throughout the project. Their specifications can be seen in Table 5.1. The system housing the GTX460 was used mostly during the different stages of implementation, while the TeslaC2070 was not used until the final stages of benchmarking.

Hardware (system 1)	
CPU	Intel Core i7-950
CPU clockspeed	3.06 GHz
Memory size	6 GB
Graphics card #1	NVIDIA Tesla C1060
Graphics card #1 memory	4 GB
Graphics card #2	NVIDIA GeForce GTX460
Graphics card #2 memory	2 GB
Hardware (system 2)	
CPU	Intel Core i7-970
CPU clockspeed	3.20 GHz
Memory size	24 GB
Graphics card	NVIDIA Tesla C2070
Graphics card memory	6 GB
Software (same for both systems)	
OS	Windows 7
Visual Studio ver.	2008, with SP1
NVIDIA graphics driver ver.	263.06
CUDA toolkit ver.	3.2 RC

Table 5.1: Specifications of benchmarking systems

5.2 Running the Old Code on Fermi

Prior to testing the optimized version, benchmarks were performed on the old snow simulator code on a Fermi device. This was to measure the amount of performance gain achieved without any optimizations at all. The performance increase was quite large which we can see in Figures 5.1 and 5.2. These charts display the total GPU execution time distribution of the different kernels when run on a Tesla C1060 and a Tesla C2070. The Tesla C1060 has the same device used by Gjermundsen for

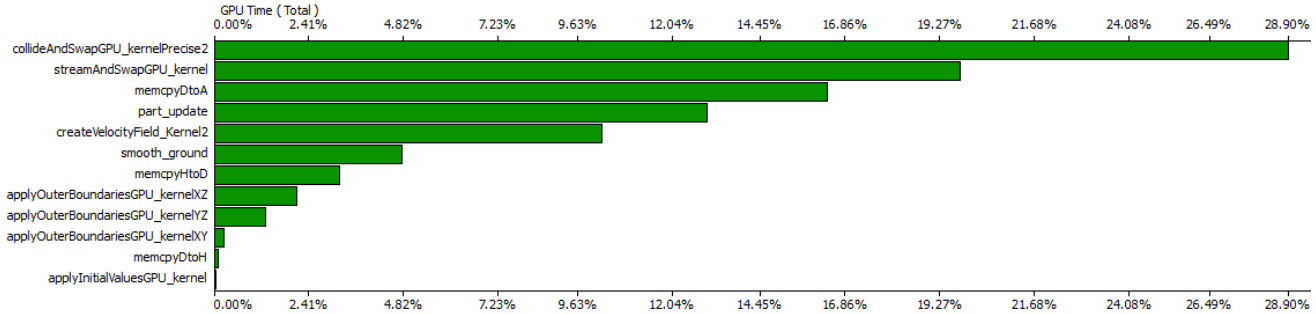


Figure 5.1: The GPU time distribution of the old code run on a Tesla C1060.

benchmarking the LBM simulation in [1], and Figure 5.1 uses the exact same configurations and setting for the snow simulator that Gjermundsen used to achieve the results he displayed in his report. The Tesla C2070 is the Fermi based device used for benchmarking in this project and is also running the snow simulation with exactly the same configurations and with no changes to the code to achieve the

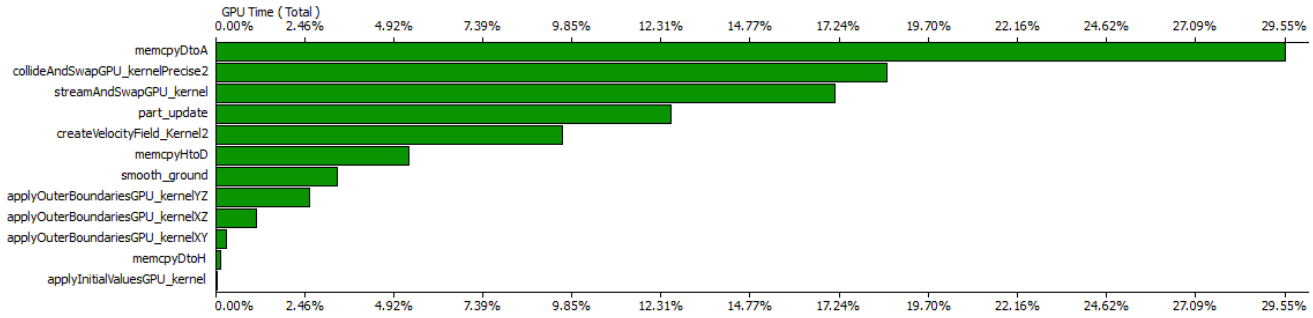


Figure 5.2: The GPU time distribution of the old code run on a Tesla C2070.

results in 5.2. These results are obtained by running the first 1000 frames of the simulation through the Compute Visual Profiler. The simulation was run using a wind resolution of 128x16x128 and 131072 snow particles.

Note that while these charts only show the percentage of time spent on each kernel compared to the total GPU time, the difference between the average execution time of the *memcpyDtoA* operations is very small (647.92μs and 611.28μs on the C1060 and C2070 respectively). Taking this into account we see that the measured time differences of all other kernels are a lot bigger. On the Tesla C1060 the *memcpyDtoA* operation only makes up 15%-16% of the total GPU time which is far below the time spent by the collision and streaming kernels of the LBM solver, but on the Tesla C2070 this has suddenly become the bottleneck. We see that on the Fermi device *memcpyDtoA* makes up almost a third of the total run time (29.55%). This entails that all the kernels, especially the LBM streaming and collision kernels have benefitted from a fairly large performance gain by simply running the simulation on a Fermi device. For the purpose of a more in-depth comparison, the average execution times of these kernels run on both devices are listed in Table 5.2. We see that the "precise" collision kernel has 3x speedup while most other kernels take at most only half the time when running on the Fermi device.

	Tesla C1060	**Tesla C2070**
collideAndSwap (Precise)	1143.59μs	383.07μs
streamAndSwap	781.917μs	353.697μs
part_update	518.537μs	259.897μs
createVelocityField	405.017μs	197.657μs
smooth_ground	188.527μs	69.27μs
applyBoundaries (YZ)	50.487μs	53.317μs
applyBoundaries (XZ)	82.637μs	22.957μs
applyBoundaries (XY)	9.317μs	5.827μs
applyInitialValues	626.627μs	176.167μs

Table 5.2: Comparing the execution time of the old snow simulator code on Tesla C1060 on the Tesla C2070

5.3 Testing the Fermi-Optimized Version

Several optimization ideas that seemed relevant for the LBM kernels as well as the particle simulation kernels were stated and discussed in the previous chapter. Experiments were done based on these ideas and the resulting code was tested with several varying parameters. Some of these proved effective and lead to a performance gain, others showed no effects at all while a few of these techniques even lead to a slight decrease in performance.

The results of these experiments are displayed in the following sections followed by some discussion about the different outcomes.

5.3.1 Routines for Testing

The benchmarking procedure used throughout this phase consisted of running the old code and the Fermi-optimized code for different fluid domain sizes and comparing the outcomes. The number of particles remain the same (262144) throughout all of the tests and the dimensions of the terrain is always 256x256 unless stated otherwise.

For benchmarking the individual kernels, both the old and optimized simulations will be run through the Compute Visual Profiler, for a duration of 10000 frames, and the average GPU execution times will be compared with each other. For benchmarking the overall performance, the same procedure will be applied but the applications will be run for 300 seconds instead, and the frame rate achieved over various resolutions will then be compared. This is to prevent simulations that run at lower domain sizes (which give them a tremendous frame rate) from terminating quickly before a stable frame rate is achieved. The general optimizations mentioned in Chapter 4 are also regarded as a part of the Fermi optimized code.

5.3.2 LBM Kernels

The most relevant kernels to individually test are the collision kernels and the streaming kernel. These two kernels together make up over one third of the total execution time used by the GPU. The collision kernels have also been tested by Gjermundsen in [1] so we refer to these results if there is any interest in comparing these results with results obtained on pre-Fermi devices.

Collission Kernels

As described earlier in Section 4.1.2 there are four variants of the collision kernel. The “regular” kernel seemed to have the same performance as the “precise” kernel and all optimizations performed on these two kernels seemed to also give very similar results. Because of this the “regular” kernel has been excluded from further experiments. The benchmarks of the remaining three kernels can be seen in Table 5.3 and Figure 5.3

Both the “precise” and “double” kernels benefitted from many of the optimization techniques. The biggest performance gain was achieved by setting the memory preference to L1 cache. Some changes were made to the code to make the kernels use

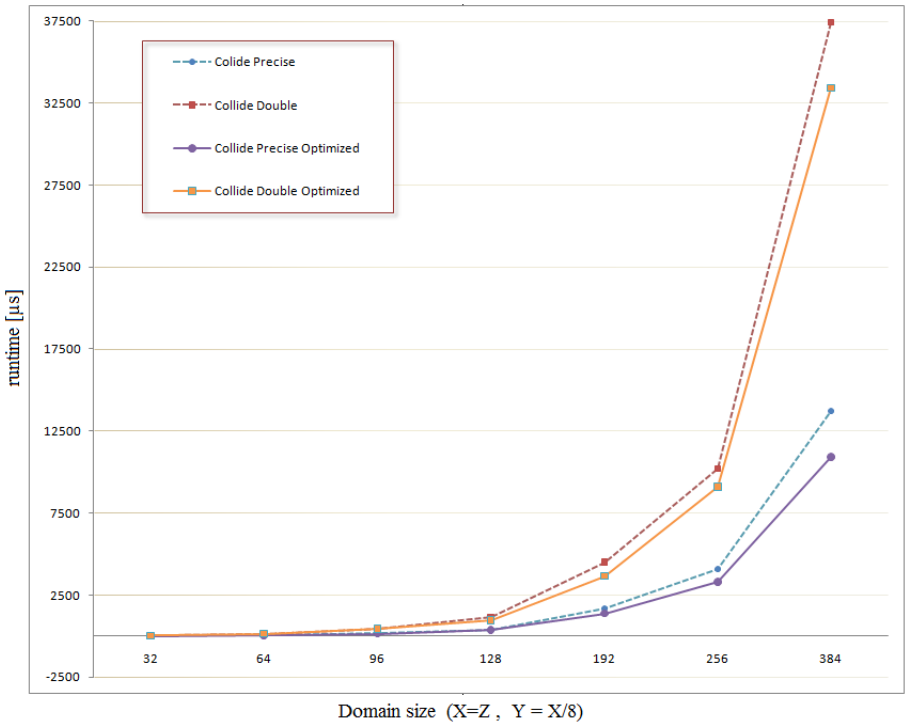


Figure 5.3: Execution times of the old and optimized collision kernels for different domain sizes.

fewer registers but the performance gain from this was very small for most domain sizes. The "double" kernel uses a lot of registers compared to the "precise" kernel; mainly because it needs twice as many registers to store all the double precision data. Experiments were performed on trying to limit the number of available registers of the "double" kernel through compiler options. Although this increased the occupancy it did not lead to any speedup.

The "precise" kernel seemed to show a stable speedup of 20%-25% once we exceed a certain domain size. The results of the "double" kernel indicate varying speedups for the different domain sizes and it was difficult to make up any notion of a general speedup from these results. The "shared" kernel didn't achieve any performance gain from any of the experiments. Even giving it more shared memory had no effect. The execution times of the "shared" kernels can be seen in Table5.3 but they are not included in Figure5.3 as both graphs would just completely overlap. No strong conclusions are made on why it did not respond well to any of the optimizations, although it should be noted that this kernel is also the most unstable of collision kernel. This can be a possible reason why no positive results were achieved.

Streaming Kernel

The streaming kernel was also benchmarked in the same fashion. Like the collision kernels, the execution time of the streaming kernel is also mainly dependent on the domain size; however the calculations performed by this kernel are a lot simpler and mostly consists of swapping some pre-calculated data. The results of the benchmarks can be found in Table 5.4 and Figure 5.4.

	Precise	**Precise Opt.**	**Double**	**Double Opt.**	**Shared**	**Shared Opt.**
32x4x32	21.28μs	8.83μs	26.05μs	25.95μs	11.91μs	11.74μs
64x8x64	55.91μs	46.19μs	146.29μs	144.58μs	48.42μs	48.31μs
96x12x96	161.66μs	145.82μs	438.62μs	432.95μs	113.66μs	113.27μs
128x16x128	386.49μs	367.44μs	1138.81μs	950.70μs	206.93μs	206.53μs
192x24x192	1678.48μs	1378.41μs	4505.42μs	3657.49μs	445.61μs	444.02μs
256x32x256	4100.97μs	3292.56μs	10195.11μs	9097.92μs	843.21μs	841.42μs
384x48x384	13712.20μs	10928.20μs	37422.10μs	33426.60μs	2759.49μs	2742.39μs

Table 5.3: Execution times of the old and optimized collision kernels for different domain sizes.

	Stream	Stream Opt.	Stream(D)	Stream(D) Opt.
32x4x32	28.9619μs	15.0342μs	16.2473μs	15.3969μs
64x8x64	68.0791μs	63.7176μs	75.6419μs	71.8232μs
96x12x96	170.121μs	138.533μs	260.668μs	209.252μs
128x16x128	348.158μs	223.532μs	597.797μs	399.783μs
192x24x192	1122.35μs	922.39μs	2372.03μs	1885.18μs
256x32x256	2661.97μs	2330.07μs	5554.95μs	4561.67μs
384x48x384	9099.27μs	8088.14μs	18117.3μs	150082.5μs

Table 5.4: Execution times of the old and optimized streaming kernel for different domain sizes (for both single and double precision).

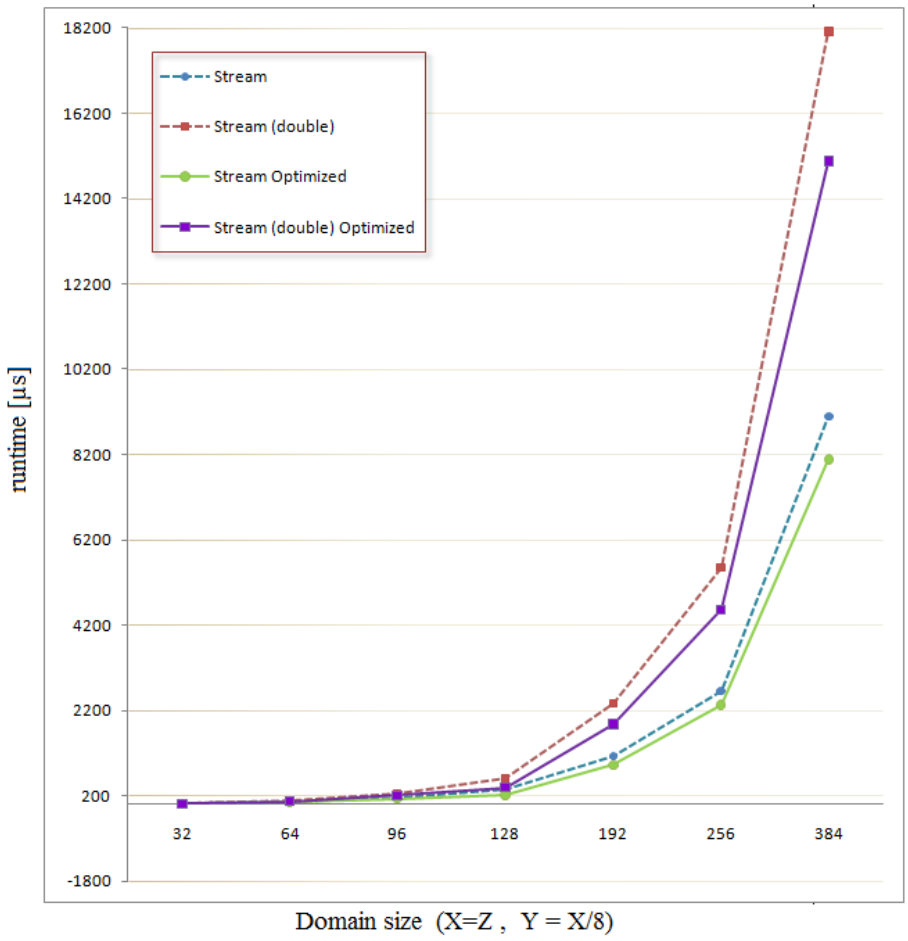


Figure 5.4: Execution times of the old and optimized streaming kernel for different domain sizes (for both single and double precision).

Here the streaming kernel was benchmarked both for single and double precision. Increased L1 cache configuration, showed positive results on this kernel, but most other optimization strategies that were specific for this kernel gave no performance gain at all. Increasing warp occupancy only lead to a slow down, most likely due to increased resource usage.

Remaining LBM Kernels

The remaining LBM kernels, which are the *createVelocityField* kernel and the three *applyOuterBoundaries* kernels were not benchmarked individually. One of the reasons being that there were very little optimization possibilities available for the *createVelocityField* kernel since its task was very short and simple. The kernels responsible for the boundary conditions did have several experiments performed upon however they made up such a small part of the GPU total execution time that looking at the result of each individual boundary kernel would not be very insightful. Another reason is that one of the most rewarding optimization was placing the three boundary kernels into streams so that they could be executed concurrently. This performance gain was not visible when looking at the individual kernel times, and can only be seen when looking at the overall performance, which is done later in Section 5.3.4. Other experiments performed on the *applyOuterBoundaries* kernels included maximizing warp occupancy and trying to avoid branching by splitting up the three kernels into six kernels that applied boundary conditions to only on plane each. These experiments did not result in any performance gain.

5.3.3 Particle Simulation Kernels

The kernels that we classified as particle simulation kernels in Section 4.1 were the *part_update* kernel for updating the snow particles and the *smooth_ground* kernel for updating the terrain.

Particle Update Kernel

For the *part_update* kernel there were not so many optimizations possible. It was one of the few kernels that already had full warp occupancy and it's also mentioned in the reports of Gjermundsen [1] and Eidissen [3] that this kernel has been very finely optimized. Setting the memory configurations to prefer L1 cache provided a small boost to the performance but this gain was barely noticeable.

Smooth Ground Kernel

The *smooth_ground* kernel used a lot of dynamic shared memory. So setting the memory configurations to prefer shared memory over L1 provided a reasonable performance gain. Having more shared memory available also lead to the kernel having full warp occupancy while previously the occupancy was at 50%. The results of these optimizations can be seen in Table 5.5. This kernel was benchmarked by

	smooth_ground	**smooth_ground Opt.**
256x256	69.242μs	55.956μs
512x512	244.722μs	201.301μs
1024x1024	932.94μs	760.195μs

Table 5.5: Execution times of the old and optimized smooth_ground kernel for different terrain dimensions.

running the simulation with a fixed fluid domain and particle count, but varying

the dimensions of the terrain. Only three maps (of dimensions 256x256, 512x512 and 1024x1024) were available for testing. Each simulation was run for 10000 frames.

Experiments based on adjusting the kernel launch configurations to assume per-warp accesses did not give any positive results for the *smooth_ground* kernel. This may be due to the fact that the kernel uses a lot of shared memory and does not access global memory that often. It could also be due to some errors in the optimized version as not much time was spent on this particular experiment in comparison to others.

5.3.4 Overall Performance

To measure the overall performance, the entire snow simulator was run over different domain sizes. The average frame rates over the length of 5 minute simulations were recorded for both the old and the optimized versions, for both single and double precision. These measurements can be seen in Table 5.6 and Figure 5.5.

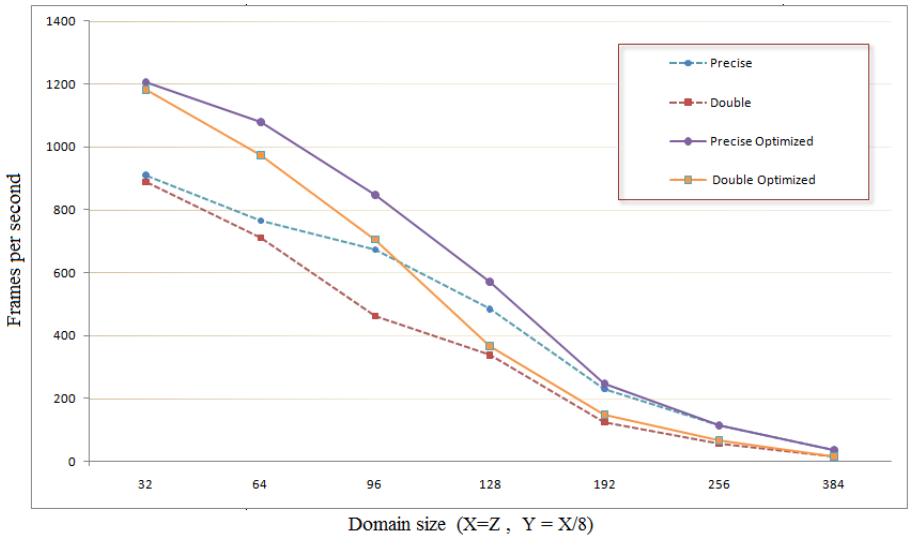


Figure 5.5: Frame rate comparison of old and optimized snow simulation for different domain sizes (for both single and double precision).

	Precise	**Double**	**Precise Opt.**	**Double Opt.**
32x4x32	910.23μs	887.44μs	1206.15μs	1180.52μs
64x8x64	765.53μs	711.56μs	1078.39μs	972.96μs
96x12x96	672.78μs	462.85μs	846.56μs	705.71μs
128x16x128	484.70μs	337.98μs	572.07μs	366.73μs
192x24x192	230.36μs	124.11μs	247.63μs	149.68μs
256x32x256	113.76μs	55.77μs	115.45μs	66.00μs
384x48x384	34.96μs	14.86μs	36.15μs	14.99μs

Table 5.6: Frame rate comparison of old and optimized snow simulation for different domain sizes (for both single and double precision).

The rendering was disabled during the benchmarking phase to be able to measure

the average frame rate with regards to the optimizations that were performed. As we saw earlier in Figure 5.2, the biggest bottleneck by far was the *memcpyDtoA*. However, no Fermi-based optimization techniques were found that could provide any improvements to this operation. The rendering was therefore disabled during benchmarking to avoid having the performance of this operation dominate the outcome of the overall results. This would have made it impossible to compare the optimizations that were actually performed on the snow simulator.

In addition to the individual speedups achieved by the largest kernels, the increased overall performance was also due to some more general optimizations that affected all parts of the code. These include the use of 32-bit integer multiplication instead of 24-bit, using compiler options to generate code using some less precise but faster operation and running multiple kernels concurrently. Exactly how much performance gain was achieved by each of these optimizations is hard to put into words and numbers as it depends on many different factors. But we did confirm that they all contributed to a speedup, whether big or small, during the testing phase.

The optimized code has a very high performance gain at lower fluid resolutions, but as the domain size increases we see that it converges towards the frame rate of the old code. The speedup at a domain size of 32x4x32 was around 40% for both single and double precision, but for more relevant domain sizes (128x16x128) the single precision performance gain was 18% while the double precision runachieved roughly 8.5% speedup. This is most likely due to the fact that we are reaching the limits of the device at such large numbers. The largest domain size available to run the simulation on without running out of memory (with the current settings), was 384x48x384. So given the fact that we are at the limit of what can be run on the device, we conclude that this is the main reason our optimizations do not provide any performance gain for large domain sizes.

CHAPTER 6

Conclusions and Future Work

The introduction of the NVIDIA Fermi architecture made way for new possibilities and ideas within the fields of GPGPU. The NTNU Snow Simulator which was previously designed for the earlier generation GPUs, is a highly parallel application that performs many intense computations. This makes the idea of upgrading to the Fermi architecture very attractive, as this can open up several potential paths for expansion of the snow simulator.

6.1 Conclusions

The goal of this project was to optimize the snow simulator and the LBM solver for the NVIDIA Fermi GPU architecture. This included a study of the new Fermi architecture, followed by research on different techniques and ideas on how to take advantage of the higher compute capability as well as exploiting the new features available on Fermi devices. Various experiments were then performed under varying parameters and levels of details to test which of these optimizations proved beneficial and under which conditions. The results of the experiments were documented, compared and discussed to understand the effects they had on individual kernel runtimes as well as the overall performance.

Based on the outcome of the different experiments we conclude that there were many optimizations that gave positive results for snow simulator. These also provided a reasonable amount of speedup to the overall performance. However, compared to the speedup gained from running the old code on a Fermi device straight away, this is only a tiny fraction. One reason for this is that the most time consuming operation when running on a Fermi device was the *memcpyDtoA* operation. This makes the application memory bound and difficult to optimize further. Another possible reason for this is that the previous snow simulator code was already in relatively good condition and coded according to good practice. This enabled the code to take advantage of most of the higher horsepower and improved compute capabilities provided by Fermi even before any optimizations were applied. The optimizations

that were added as a part of this project focused mostly Fermi-specific features. As these features weren't available on previous generation GPUs, the performance gain they provide, can only be achieved by explicitly enabling them. This is the reason it was possible to further optimize the code even after the huge performance gain obtained by running the previous code on a Fermi device.

6.2 Future work

Several advances are frequently being made in the fields of parallel computing and GPGPU programming, as well as improvements to current fluid solvers and research on new ones. Some of these may be suitable as upcoming projects for the NTNU snow simulator. This section will list some of these ideas for possible future work on the snow simulator and the SOR and LBM fluid solvers.

6.2.1 More realistic LBM simulation

The wind simulation currently done by the LBM solver is far from realistic. It gives the snow a very pleasing visual appearance and adds a lot of variation in the falling pattern which at first glance makes the scene looks very natural. However the wind simulation is not physically correct, as simulating the wind field with low enough density to achieve realism using the LBM solver is not a simple task. First of all, this would require a lot more compute power with the way the current solver is implemented; and in the results of the overall performance in Section 5.3.4 we saw that the frame rate drops rapidly as we increase the fluid resolution. Although the performance gain from using a Fermi device was quite large it is not enough to achieve real-time on large enough fluid resolutions that might provide a realistic simulation.

In addition, it should be noted that several tweaks are done in the existing LBM solver by [1] to maintain stability in the wind simulation and give visually appealing results. If the snow simulation is to be used in possible future applications for any kind of prediction of snow fall and build up, then we cannot rely on simplifications such as these as they could lead to unrealistic results.

The existing LBM solver either needs a lot more compute power to be able to run on larger domains, or designed differently to provide more accurate results even with across smaller domain sizes. Some possibilities that are worth looking into are: Executing the solver on multiple GPUs, allowing for a larger range of parameters and allowing for more complex flows. There are several active research topics within these fields and they are discussed in more detail in the "future work" chapter of Gjermundsen's report[1].

6.2.2 Optimizing the SOR-solver for Fermi

This project only focused on optimizing the LBM fluid solver for the new GPU architecture. So, it is still unknown how much improvement it is possible to gain from optimizing the SOR-solver, or even by simply running the current SOR solver on a

Fermi device. According to [1] SOR solver was also found to have more turbulent wind flow than the LBM solver and it also had a slightly better performance for larger domain sizes. This makes Fermi an attractive choice for this solver as well. A possible idea for a future project is to also optimize the SOR solver and compare the performance results to the ones obtained in this project.

6.2.3 Research Topics on New and Existing Solvers

The performance of a multigrid pressure Poisson equation solver running on a GPU cluster is investigated in [12]. This solver was written for 3D incompressible Navier-Stokes flow solver and may be of interest to the snow simulator in regards to running the SOR solver over multiple GPUs.

Cyclic reduction tridiagonal solvers on GPUs are investigated in [13]. The paper presents a method that applies this solver on a mixed precision multigrid. They claim that geometric multigrid solvers are in general the most efficient method for solving finely discretized partial differential equations, and evaluate the mixed precision solver with regards to iteratively solving sparse linear equations. The methods mentioned here may also be of interest to SOR solver of the snow simulator, with regards to a possible multigrid solution.

A GPU implementation of PETSc is presented in [14]. PETSc is a scalable solver library for various algebraic equations, specifically those arising from discretization of partial differential equations. It would be interesting to find out what possibilities are there for using the solvers from the PETSc library for the fluid simulation of the snow simulator.

Bibliography

[1] Alexander Gjermundsen. *LBM vs SOR solvers on GPUs for real-time snow simulations.* Specialization project, Norwegian University of Science and Technology, 2009.

[2] Eirik Ola Aksnes. *Simulation of fluid flow through porous rocks on modern GPUs.* Master's thesis, Norwegian University of Science and Technology, 2009.

[3] Robin Eidissen. *Utilizing GPUs for real-time visualization of snow.* Master's thesis, Norwegian University of Science and Technology, 2009.

[4] Ingar Saltvik. *Parallel methods for real-time visualization of snow.* Master's thesis, Norwegian University of Science and Technology, 2006.

[5] *NVIDIA's next generation Cuda compute architecture: Fermi.* White paper, NVIDIA, 2009. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[6] *NVIDIA Cuda programming guide, version 3.2.* Guide, NVIDIA, 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf

[7] *NVIDIA Cuda C best practices guide, version 3.1.* Guide, NVIDIA, 2010. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_BestPracticesGuide_3.1.pdf

[8] *Fermi compatibility guide for Cuda applications, version 1.3.* Guide, NVIDIA, 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/Fermi_Tuning_Guide.pdf

[9] *Tuning Cuda applications for Fermi, version 1.3.* Guide, NVIDIA, 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/Fermi_Tuning_Guide.pdf

[10] Shane Ryoo, et al. *Program Optimization Space Pruning for a Multithreaded GPU.* Published in CGO'08, 2008. http://www.gpucomputing.net/?q=node/921

[11] Rajib Nath, Stanimire Tomov and Jack Dongarra. *An improved MAGMA GEMM for Fermi GPUs.* Technical report, University of Tennessee, 2010. http://www.netlib.org/lapack/lawnspdf/lawn227.pdf

[12] Dana Jacobsen and Inanc Senocak. *Parallel 3D Geometric Multigrid solver on GPU clusters*. Poster, Boise State University, 2010. http://www.nvidia.com/content/GTC/posters/2010/D02-Parallel-3D-Geometric-Multigrid-Solver-on-GPU-Clusters.pdf

[13] Dominik Goddeke and Robert Strzodka. *Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid*. Technical report, 2010. http://www.mathematik.uni-dortmund.de/~goeddeke/pubs/pdf/Goeddeke_2010_CRT.pdf

[14] Victor Minden, Barry Smith and Matthew G. Knepley. *Preliminary implementation of PETSc using GPUs*. Technical report, 2010 International workshop of GPU solutions to multiscale problems, 2010. http://www.mcs.anl.gov/petsc/petsc-2/features/gpus.pdf

APPENDIX A

Super Computing '10 Poster

Real-time Snow Simulation on GPU: Current and future work

Joel Chelliah and Jarle Erdal Steinsland, Master Student Advisor: Anne C. Elster
Department of Computer and Information Science

NTNU – Trondheim
Norwegian University of Science and Technology

- Simulates up to 2 million snow particles in real-time on the GPU
- Implemented in CUDA and C++, and uses OpenGL for rendering
- Simulates particle movement, snow build-up and the wind field
- Approximates the windfield using an SOR-solver or an LBM-solver
- Combined work of several M.Sc. student projects and thesis work
- Supports (quad-buffered) stereoscopic 3D rendering
- Current work being done on the simulator:
  - Optimization for the NVIDIA Fermi GPU architecture
  - Porting to OpenCL

SOR solver

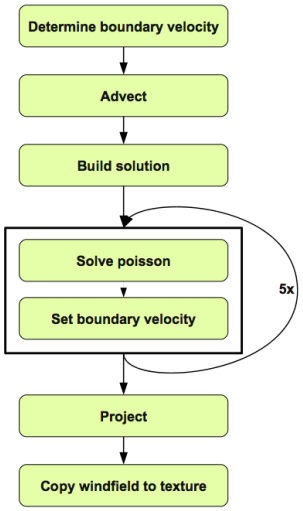
Navier-Stokes Equations:

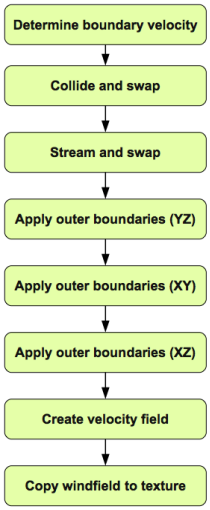
$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u}\cdot\nabla)\,\mathbf{u} - \frac{1}{\rho}\nabla p + \nu\nabla^2\mathbf{u} + \mathbf{f}$$

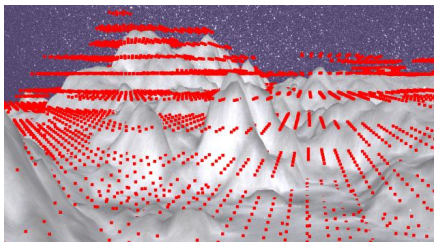
$$\nabla\cdot\mathbf{u} = 0$$

LBM solver

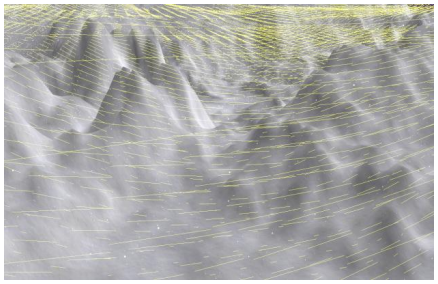
1. Initialization phase - initialize constants and macroscopic properties
2. Collision phase - check surrounding particles and compute new local distribution
3. Streaming phase - distribute new local distribution







Debug rendering which shows the obstacles for the wind field



Debug rendering which shows the wind velocity vectors

Optimizing for Fermi

To improve performance of the simulator on Fermi GPUs we will be doing research on and taking advantage of what's new and what's different in the Fermi architecture.

One of the main goals will be on the vast memory improvements, such as the increased number of registers, shared memory and CUDA cores per SM. Secondly, we will also be looking at some of the new features that are introduced in the Fermi architecture, such as the added control over L1 and L2 caching and configurable partitioning of shared memory. Thirdly, we will also ensure that the code is accounting for several changes in the architecture, such as the global memory access now being performed per warp.

Porting to OpenCL

OpenCL have greater initialization overhead than CUDA. Program source must be loaded and compiled.

CUDA and OpenCL are conceptually similar and a large part of porting kernels is keyword exchange.

OpenCL does not support loading data into constant memory at runtime. Must use global memory or load into shared memory instead in OpenCL.

No support for OpenCL images on ATI cards in Mac OS X means you have to use global memory instead.

Ackknowledgements: We would like to thank Ingar Saltvik, Robin Eidissen and Aleksander Gjermundsen for their previous work on the snow simulator. We would also like to thank NVIDIA for providing graphics cards used through Dr. Elster's membership in their Professor Afiliates Program.

HPC-Lab
Computer & Info. Science
Norwegian University of Science and Technology
http://research.idi.ntnu.no/hpc-lab