NTNU – Trondheim
Norwegian University of
Science and Technology

OpenACC-based Snow Simulation

Magnus Alvestad Mikalsen

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Anne Cathrine Elster, IDI

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

Problem Description

This projects looks at porting the HPC-Lab Snow Simulator to OpenACC and compare it to CUDA and/or OpenCL implementations. Tasks include further modularization of the code as well as testing two or more OpenACC implementation.

Assignment given: 15. January 2013
Supervisor: Dr. Anne C. Elster, IDI

Abstract

In recent years, the GPU platform has risen in popularity in high performance computing due to its cost effectiveness and high computing power offered through its many parallel cores. The GPUs computing power can be harnessed using the low-level GPGPU programming APIs CUDA and OpenCL. While both CUDA and OpenCL gives the programmer fine-grained control of a GPUs resources, they are both generally considered difficult to use and can potentially lead to complicated software design. To simplify GPGPU programming and gain more mainstream usage of GPUs, there is an increased interest in moving the complexity of GPGPU programming over to the compiler. This has lead to the development of the directive-based standard for heterogeneous computing called OpenACC, supported by NVIDIA, Cray, PGI, CAPS and others.

In this thesis, we explore using OpenACC on a high performance snow simulator code developed by the HPC-Lab at NTNU. The snow simulator consists of two main simulation components; the simulation of wind, and the simulation of snow particle movement.

Our snow simulator is made by updating and porting the current CUDA version to a sequential CPU implementation, and applying OpenACC directives to accelerate compute intensive regions in the code. The OpenACC port is also optimized by reducing datamovement between host and device using OpenACC library routines.

Due to the heterogeneous nature of OpenACC, we show that the inability to explicitly use shared memory as temporary storage and not being able to use texture memory for hardware based interpolation and 3D caching, are the largest performance bottlenecks when comparing to the CUDA version.

This is supported by the benchmarks of the OpenACC implementation which is shown to give only 40.6% performance of the CUDA version with an average speedup of 3.2x when scaling the amount of snow particles simulated and using a balanced windfield dimension. When scaling the windfield with constant snow particles 58% of the CUDA performance is reached with an average speedup of 4.84x. The best real-time performance is found at about 1.5M snow particles when using a balanced windfield with about 524K grid cells.

Using OpenACC for accelerating high performance graphical simulations can be a viable option if the goal is high code portability, however, when the goal is to achieve the best possible performance, our experience show that it is still better to use the more low-level alternatives CUDA or OpenCL.

Sammendrag

I senere årene har GPU-plattformen økt i popularitet i databehandling med høy ytelse på grunn av sin kostnadseffektivitet og høye beregningskapasitet tilbudt gjennom sine mange parallelle kjerner. GPUens datakraft kan utnyttes ved hjelp av lav-nivå GPGPU programmering APIene CUDA og OpenCL. Mens både CUDA og OpenCL gir programmereren finkornet kontroll over en GPUs ressurser, er begge generelt ansett vanskelig å bruke, og kan potensielt føre til komplisert software design. For å forenkle GPGPU programmering og få mer mainstream bruk av GPUer, er det en økt interesse i å flytte kompleksiteten i GPGPU programmering over til kompilatoren. Dette har ført til utviklingen av den direktivet-baserte standarden for heterogen databehandling kalt OpenACC, støttet av NVIDIA, Cray, PGI, CAPS og andre.

I denne avhandlingen utforsker vi bruker OpenACC på en høy ytelse snø simulator kode utviklet av HPC-Lab ved NTNU. Simulatoren består av hovedsaklig to simulering komponenter; simulering av vind, og simulering av snø partikkel bevegelse.

OpenACC versjonen av snø simulator er utviklet ved først å oppdatere den nyeste CUDA-versjonen av snø simulatoren, porting det til en sekvensiell CPU implementering og bruk OpenACC direktiver for å akselerere beregning intensive regioner i koden. OpenACC porten er også optimalisert ved å redusere data overføringer mellom vert og GPU enhet ved hjelp OpenACC bibliotek rutiner.

På grunn av den heterogene naturen OpenACC, viser vi at manglende evne til å eksplisitt bruke shared memory til midlertidig lagring og ikke være i stand til å bruke tekstur minne for hardware baserte interpolering og 3D-caching, er de største flaskehalser når man sammenligner med CUDA-versjonen.

Dette støttes av bechmarks av OpenACC implementering som viser å bare gi 40,6% av ytelse til CUDA-versjonen, med en gjennomsnittlig speedup på 3.2x ved skalering av antall snø partikler simulert med en balansert vindfelt dimensjon. Når vi skalere vindfeltet med konstant snø partikler oppnås 58% av CUDA ytelsen med en gjennomsnittlig speedup på 4.84x. Den beste real-time ytelse er funnet ved 1,5M snø partikler med et balansert vindfelt med 524K grid celler.

Å bruke OpenACC for akselerering av høy ytelse grafiske simuleringer kan være et levedyktig alternativ hvis målet er høy portabilitet av koden, men når målet er å oppnå best mulig ytelse, viser vår erfaring at det fortsatt er bedre å bruke de mer lavnivås alternativene CUDA eller OpenCL.

Acknowledgement

I would like to thank Dr. Anne C. Elster for supervising this thesis and for the work she does for the HPC-Lab at NTNU. I would also like to thank my fellow master students at the lab for creating a enjoyable environment, with special thanks to Andreas Nordahl for working with me to update the current snow simulator, and Lars Kirkholt Melhus for providing C/C++ programming expertise.

Lastly i also wish to thank NVIDIA for their support of the HPC-Lab through their CUDA Research Center and CUDA Teaching Center programs.

Trondheim, June 2013
Magnus Mikalsen

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

GPGPU General Purpose Graphics Processing Unit

CC Compute Capability

HPC High Performance Computing

GPU Graphics Processing Unit

CPU Central Processing Unit

API Application Programming Interface

CUDA Compute Unified Device Architecture

OpenCL Open Computing Language

OpenMP Open Multiprocessing

FPGA Field Programmable Gate Array

ALU Arithmetic Logic Unit

FPU Floating Point Unit

DSP Digital Signal Processor

SM Streaming Multiprocessor

SP Streaming Processor

SIMD Single Instruction Multiple Data

MIMD Multiple Instruction Multiple Data

SIMT Single Instruction Multiple Threads

ECC Error Correcting Code

PCI-E Peripheral Component Interconnect Express

LLVM Low Level Virtual Machine

FPS Frames per second

VBO Vertex buffer object

Chapter 1

Introduction

The graphics processing unit (GPU) was originally intended for acceleration of real-time graphics in games and in graphical applications. However, it was discovered that the GPUs parallel computing power could also be used for non-graphical applications like scientific computations, through the use of high-level shading languages and APIs designed for graphical programming such as DirectX, OpenGL, and Cg. Such computing became known as general-purpose computing on GPUs (GPGPU). To facilitate this further NVIDIA released in 2006 a dedicated programming model, hardware architecture, and API for GPGPU computing called Compute Unified Device Architecture (CUDA). CUDA allows for the creation of GPU computing programs using the C and C++ programming languages, enabling the GPU to become a more accessible computing device.

Even though programming general-purpose programs for the GPU is made more accessible by using CUDA, it still has some drawbacks. Mainly that CUDA is a rather low-level API making programming with CUDA difficult, potentially leading to more complicated software design, and that CUDA can only be used with NVIDIA devices. As an alternative to the proprietary nature of CUDA a standard called OpenCL was released enabling computing on heterogeneous devices. However, OpenCL also suffers from a similar programming complexity as CUDA.

With today's microprocessors hitting the "power wall"[1], the computing power given by the GPUs parallel nature is even more attractive. High Performance Computing (HPC) architectures are therefore starting to lean towards heterogeneous computer systems that combines general purpose processors with accelerators devices such as GPUs.

This increased popularity of GPU devices coupled with the fact that the existing GPU programming models such as CUDA and OpenCL are hard to use, has lead to the increased interest in moving the complexity to the compiler by using directive-based approaches to GPGPU programming. Many such approaches have been suggested like

[1] http://www.edn.com/design/systems-design/4368858/Future-of-computers–Part-2-The-Power-Walll

the PGI Accelerator Model [1], hiCUDA [2], and OpenHMPP [3], and in November 2011 a standard for GPGPU programming was formed, called OpenACC [4].

The OpenACC standard was originally founded by the companies NVIDIA, Cray, CAPS, and PGI, and is currently maintained by the non-profit OpenACC corporation. It describes a high-level API that includes compiler directives, library routines and environment variables that can be used in combination with the C, C++, and Fortran programming languages. OpenACC aims to improve productivity, simplify code maintenance, and increase GPGPU programmings appealing to the mainstream by allowing easier programming of GPUs for non-CUDA/OpenCL experts. Since OpenACC is a standard it is also portable across operating systems, host CPUs, and accelerator devices.

One of the fields where GPUs are often used as a cost efficient solution to increase performance is in simulations of real-world phenomena. At the NTNU HPC-Lab in 2006 Ingar Saltvik [31] developed a real-time[2] snow simulator for multi-core CPUs, modeling falling snow as particles and the wind as fluid using computational fluid dynamics. Saltvik's simulator was later expanded by Robin Eidissen [19] to run of the GPU platform with CUDA resulting in in a dramatic increase in performance. Enabling the snow particle count to go from tens of thousands to millions. Eidissen's GPU implementation has since been the basis of many master's thesis and projects at the HPC-Lab looking to introduce new methods for simulation, using different accelerator devices, or expanding the simulators features and visuals.

In this project we use the latest version of the snow simulator and accelerate it on a GPU using OpenACC to evaluate the applicability of using OpenACC for high performance graphical simulation code.

1.1 Problem Description

The assignment for this thesis is suggested by the student and formulated by the students supervisor Dr. Anne C. Elster as the following:

"*This projects looks at porting the HPC-Lab Snow Simulator to OpenACC and compare it to CUDA and/or OpenCL implementations. Tasks include further modularization of the code as well as testing two or more OpenACC implementation.* "

Based on this problem description the assignment is further narrowed and made into concrete goals by the student with referrals with the supervisor.

The main objective of this assignment is to create a version of the HPC-Lab snow simulator that uses the directive-based programming model OpenACC to accelerate the simulation components on a GPU, and compare the OpenACC versions performance with the most current CUDA version. The secondary objectives are to compile and test the OpenACC implementation using two OpenACC compilers and also attempt to make the snow simulator code more modular if possible.

[2]Real-time is often defined as 25 FPS in interactive media and entertainment

The purpose of the main objective is to find out if a compiler directive-based approach would be suitable for high performance simulation code such as the snow simulator, and to highlight eventual considerations needed to be taken into account when using such an approach.

1.2 Goals

The goals or tasks for this thesis are defined by the student with referrals with the supervisor Dr. Anne C. Elster, and is set as a measurement of the success of the project.

The following goals are outlined:

1. Update and improve the code quality of the current CUDA version of the snow simulator to use as a basis of comparison for the OpenACC implementation. Also attempt to improve the modularity of the code structure if possible.

2. Create a sequential CPU port of the updated CUDA version of the snow simulator to work as a basis for the OpenACC port and also to create a basis for performance measuring. CPU version should be accelerated to use multiple CPU cores.

3. Accelerate the sequential CPU snow simulator port using OpenACC directives. Attempt to optimize the OpenACC port as much as possible without affecting the simulators modularity.

4. Attempt to compile and test the OpenACC version of the snow simulator using different OpenACC compilers. Compilers that should be tested are the commercial PGI compiler and the open source accULL compiler.

5. Compare performance of OpenACC, CPU, and CUDA versions of the snow simulator.

6. Evaluate the applicability of using OpenACC with real-time graphical simulation applications.

As stated in goal 4 the OpenACC implementation should be tested using the PGI and accULL OpenACC compilers. This goal was however not reached since it was found that the accULL compiler is currently lacking support for the OpenACC directive clause called `deviceptr`. This directive clause was found to be essential in the OpenACC snow simulator version and therefore it was decided to only perform testing using the PGI compiler.

1.3 Thesis Outline

Chapter 1 presents a introduction to the project along with a description problem presented and the goals that are to be met.

In Chapter 2 a introduction is given to the parallel computing using graphical processing units (GPU). The underlying hardware architecture of the GPU is explored and programming models for the GPU is presented. The programming models presented are first the tradtional models such as CUDA and OpenCL, and then the newer compiler directive-based model OpenACCC.

Chapter 3 describes the current CUDA version HPC-Lab snow simulator developed by previous students and updated for this thesis. This description includes the structure of the snow simulator, the theoretical background behind of the simulation components, and some important implementation details.

The process of porting the snow simulator to OpenACC is explained in chapter 4, with details of the problems encountered and the final solution reached.

In Chapter 5 the OpenACC version of the snow simulator is benchmarked and its performance is compared to the current CUDA version and a parallel CPU version using OpenMP. The results are also discussed along with the the experiences of using OpenACC.

Finally, in Chapter 6 the results are summarized and conclusions are reached along with a a outlining of how the work done in this thesis can be further developed in the future.

All the code for the OpenACC snow simulator implementation described in this thesis can be found in a zip file attachment delivered alongside this report. The code that is most relevant for the OpenACC implementation is also included in Appendix A.

Chapter 2

Background

This chapter gives an introduction to GPU computing by describing the GPU architecture and the GPGPU programming models CUDA, OpenCL, and the directive-based model OpenACC.

The sections of this chapter pertaining to GPU architecture and the GPU programming models CUDA, OpenCL, and OpenACC v1.0 were taken in part from the authors specialization project written in the fall of 2012 [23].

2.1 GPU Hardware Architecture

The graphics processing unit (GPU) was popularized by NVIDIA in 1999 [5] as a co-processor which could run alongside a CPU to handle real-time graphics in games and graphics applications, typically on a dedicated extension cards connected to the system through a bus.

Displaying graphics is a computational intensive task and usually requires floating-point operations to be performed per pixel on the displayed image. The GPUs focus was therefore to enable accelerated floating-point performance and provide a large memory bandwidth through a specialized, and eventually heavily multicored, processor architecture and memory model. These features eventually made the GPU an attractive alternative to traditional CPUs for accelerating computational heavy and data parallel applications such as those often found in the fields of science and engineering.

The GPU was first exploited for non-graphical application through the use of high-level shading languages designed for graphics programming such as DirectX, OpenGL, and Cg. The applications developed to run on the GPU using these graphics APIs came to be known as general purpose GPU or GPGPU programs. However, the fact that the graphics APIs and GPUs were not originally designed for programming and running non-graphical applications meant the hardware features and the programming model was restricted, making developing applications difficult.

In November 2006, NVIDIA introduced the first GPU designed for enabling GPGPU programming called the G80 [5]. This GPU came with a new unified graphics and compute architecture, and also a parallel software and hardware architecture named Compute Unified Device Architecture or CUDA. Using this architecture NVIDIA also released, alongside its consumer aimed GeForce GPU series, a new GPU series called Tesla [5] that was specifically aimed at GPGPU computing.

With the G80 NVIDIA also introduced a version scheme to indicate the capabilities the architecture supports, called CUDA compute capability [5], starting at the version number 1.0. The scheme consists of a major and a minor revision number, where the major number identifies the core architecture and the minor indicates smaller revisions of the same architecture. In June 2008 NVIDIA also added support for double-precision floating-point operations in a major revision of the G80 architecture called the GT200 (v1.3).

In 2009 NVIDIA released the CUDA compute and graphics architecture code-named "Fermi" [5] used in the GeForce 400 and Tesla series (Compute capability 2.x). The Fermi architecture is an improvement over the GT200 with a major increase in processor cores, also called CUDA cores, per streaming multiprocessor (SM). It also introduced a true cache hierarchy, a increase in shared memory, increasing the number of warp schedulers from one to two, and large improvement in double-precision floating-point performance.

In March 2012 NVIDIA released its latest GPU architecture to date, called "Kepler" [6] used by the GeForce 600 series (Compute capability 3.0) and Tesla K series. In addition to improved hardware specifications, the Kepler architecture also introduces a feature called Dynamic Parallelism allowing the GPU to launch nested kernels without the need for communication with the CPU. Figure 2.1.1 shows the NVIDIA performance scale with newer GPU models compared to Intel CPUs for single and double precision.

As NVIDIAs major competitor AMD has also released its own GPU computing architecture with the Radeon series of graphics cards. These cards utilizes the OpenCL programming model. The architecture used by AMD is however outside the scope of this project as it focuses on NVIDIA GPUs only.

2.1.1 Hardware Architecture Overview

The main components in the GPU architecture is the Streaming Multiprocessor (SM), shown in figure 2, and the GPU memory. Every GPU consists of a multiple SMs with the amount of SMs depending on the GPU model. Each SM has a Multiple Instruction Stream Multiple Data Stream (MIMD) parallel architecture which means that each SM can operate upon separate data concurrently. All the SMs share a common global, constant, and texture memory, but because of the MIMD architecture there is no inherent means of synchronization across SMs without the use of a barrier or transferring data to the CPU and back again.

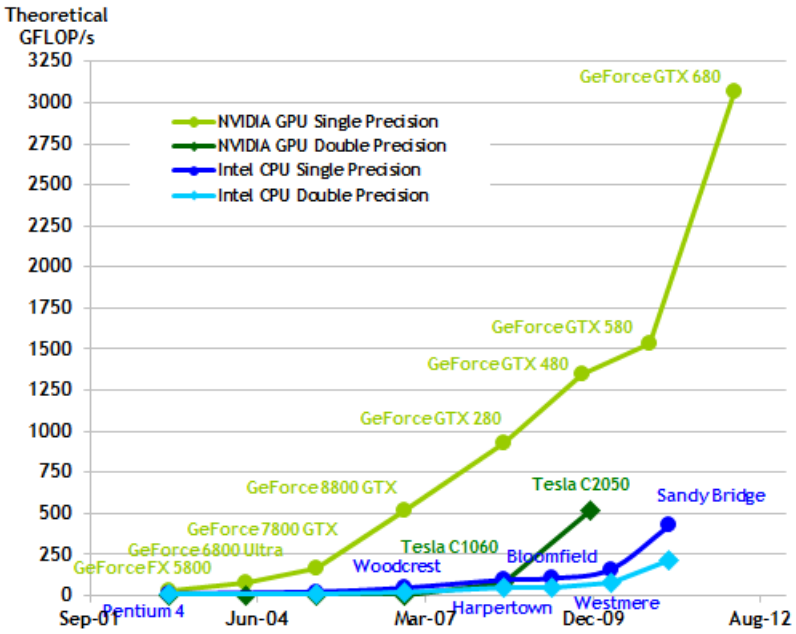


Figure 2.1.1: NVIDIA GPU single and double precision performance scale compared to Intel CPUs (With permission from NVIDIA) [33].

On the Fermi architecture [5] each SM contains 32 Streaming Processors (SP) also referred to as CUDA cores. Each of these CUDA cores has a integer arithmetic logic unit (ALU), a floating point unit (FPU), and uses a Single Instruction Stream Multiple Thread (SIMT) parallel architecture which is similar to a Single Instruction Stream Multiple Data (SIMD) parallel architecture.

The CUDA core architecture is termed SIMT because compute programs or kernels are executed as threads grouped into blocks which are distributed to SMs. Once a thread block is distributed to a SM the threads are divided into groups of up to 32 threads called warps. These warps are then scheduled for execution by the warp scheduler and executed on the CUDA cores or other execution units. The Fermi architecture also includes an additional warp scheduler enabling the SM to execute threads and instructions from two warps on up to 16 CUDA cores at a time.

The threads executed in parallel starts off with the same instruction, but the threads may branch differently forcing them to be serialized until all the threads return to the same execution path. This serialization combined with a long pipeline can have a negative impact on the performance and it is therefore important to avoid using code with a high level of branching in the GPU program [7]. For synchronization and communication the CUDA cores and thereby the executed threads can access up to 64 KB configurable shared memory and L1 cache on each SM. In addition the the CUDA cores each SM also has 16 Load/Store units, a pool of registers available for the executing threads, and four dedicated special function units (SPUs) used for calculating sine, cosine, reciprocal,

and square root instructions. The Fermi architecture also includes a L2 cache and up to total of 6 GB of ECC enabled GDDR 5 DRAM device memory shared among all the SMs.

The GPU is connected to a host using a Peripheral Component Interconnect Express (PCI-E) bus with a peak throughput of 16 GBytes/s, but has higher latency then the memory bus and is commonly considered the largest bottleneck in GPU computing [7].

2.2 GPGPU Programming Models

A GPU programming model is a parallel programming model enabling the programmer to write and execute programs according to a defined execution and memory model that takes advantage of the underlying massively parallel hardware architecture the GPU provides. This involves how a program and its data should be decomposed into smaller bits that can be mapped onto the processing elements and memory hierarchy on the GPU to maximize throughput and thereby performance.

GPU programming models can differ in granularity and functionality but usually supports programming in a high-level language such as C/C++ or Fortran and adds functionality through libraries, compiler directives, or extensions to the languages. The programming model can also be tied to a specific vendor and device or be formulated as an open standard describing only the behavior of the model and not the device dependent implementation.

The two most common GPU programming models are CUDA and OpenCL. CUDA is developed and released by NVIDIA and is can only be used on NVIDIA GPUs, while OpenCL is an open standard maintained by the Khronos Group and is aimed at a heterogeneous computing platforms in general. Since CUDA and OpenCL are both fairly low-level programming models more high-level approaches has also been released in the last few years including compiler directive-based programming models like OpenACC.

Even though the GPU programming models might be different in some ways they still adhere to the underlying GPU hardware architecture, therefore creating high-performance GPU programs generally conforms to the following three rules [8].

- Transfer the data across the PCI-E bus onto the device and keep it there as long as it is needed.
- Give the device enough work to do. Try to exploit the full width of the parallel architecture.
- Focus on data reuse within the GPU(s) to avoid memory bandwidth bottlenecks.

The following sections gives a short introduction to the CUDA and OpenCL programming models also a more thorough description of the new directive-based programming model OpenACC.

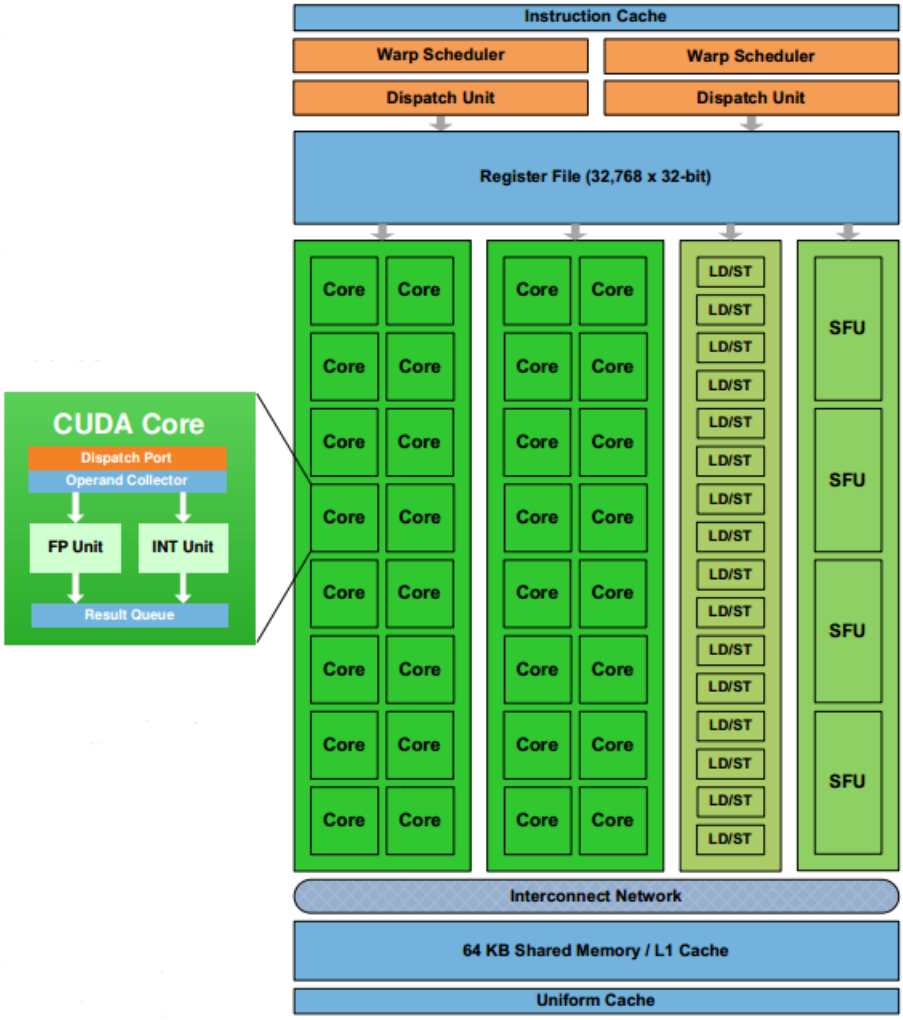


Figure 2.1.2: The hardware layout of a CUDA Steaming Multiprocessor (With permission from NVIDIA) [5].

Qualifiers	Callable From	Executes On
`__device__`	Device	Device
`__global__`	Host	Device
`__host__`	Host	Host

Table 2.2.1: CUDA Function Qualifiers.

2.2.1 CUDA

Compute Unified Device Architecture (CUDA) [7] is a parallel programming model created by NVIDIA aimed specifically at creating computing programs for the NVIDIA GPU architecture. CUDA was first introduced in November 2006 with the aim to make it easier to create GPU computing programs by avoiding using traditional shader languages meant for graphics programming such as DirectX, OpenGL, and Cg. Instead CUDA uses the C/C++ programming language with syntax extensions and library functions to express parallelism, data locality, and thread cooperation mapping to the underlying hardware architecture. NVIDIA recently released CUDA version 5.0 which includes the CUDA toolkit, SDK samples, and a unified CUDA enabled driver for NVIDIA devices. CUDA programs are compiled using the NVIDIA LLVM-based C/C++ compiler called «nvcc». The two most important concepts of the CUDA programming model is its execution and memory model.

Execution Model

A GPU computing program written in CUDA consists of function that can run on either the CPU, also called the host, or the GPU which is called the device. The functions executed on the GPU are commonly referred to as CUDA kernels and can be executed in parallel across threads on the device and also asynchronous from the host. This parallel execution is not only bound to the execution of a specific kernel, but also allows the execution of a multiple of different kernels at once, where the number of kernels that can be executed is limited by the amount of SMs on the GPU. To specify if a function should be executed on the host or the device the qualifiers shown in Table 2.2.1 are used. Table 2.2.1 also shows from where the functions with the specific qualifiers are callable from. It should also be noted that the `__host__` qualifier is usually omitted since it is the default if no other qualifier is specified.

As mentioned in the section describing the GPU architecture (Section 2.1), a kernel is executed in parallel across a set of parallel threads, which are grouped into parallel thread blocks and in each block the threads are scheduled for execution in groups called warps. When a kernel is launched it is initiated on the GPU as a grid of parallel blocks with each thread within a thread block executing an instance of the kernel, as shown in Figure 2.2.1. This thread grouping is called the CUDA hierarchy of thread, blocks, and grids [7]. Each thread has a thread ID within its thread block, program counter,

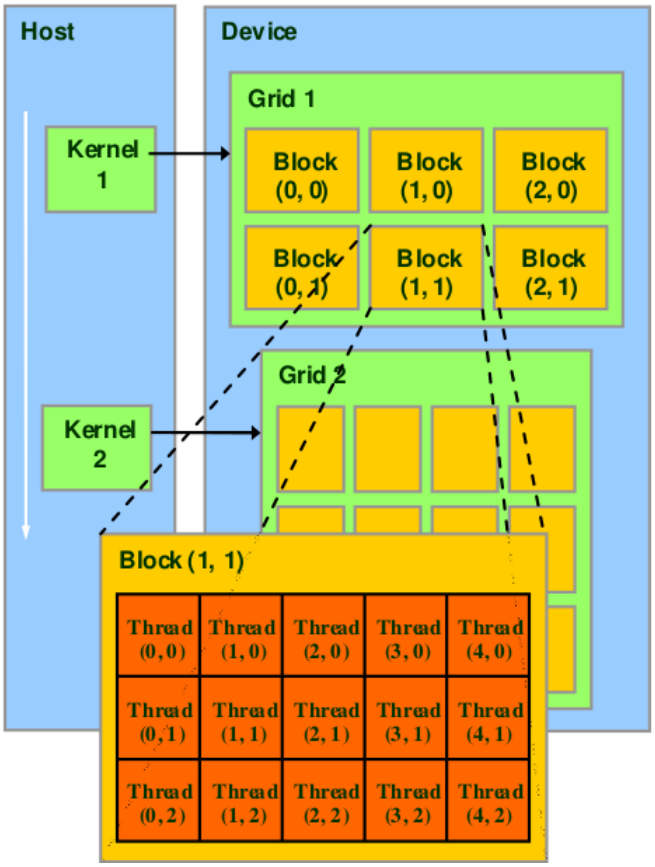


Figure 2.2.1: CUDA grid, block, and thread parallelism in connection with kernels, host, and device (With permission from NVIDIA) [33].

registers, per-thread private memory, and inputs and output results. Each thread block consists of multiple concurrently running threads that can cooperate through shared memory and barrier synchronization, and has a block id within its containing grid. And finally a grid is an array of thread blocks that executes the same kernel, reads data from global memory, and synchronize between dependent kernel calls.

Since kernels are executed across parallel threads special syntax is needed to specify the execution configuration parameters on kernel invocations. The configuration parameters are specified using two arguments inside triple angular brackets «`<<<arg1, arg2>>>`», where the first argument is the number of thread blocks (grid dimension) and the second is the number of threads per block (block dimension). Both the grid of thread blocks and the threads in a thread block may be partitioned in a 1D, 2D, or 3D domain by using a structure of dim3 type. To exploit the parallel nature of the GPU the grid dimension should be set to cover the entire computational domain of the executing kernel, and the block dimension should be match the GPU architecture [7]. The reason for considering the GPU architecture when selecting block dimension is, as previously mentioned, that

only warps of up to 32 threads are executed simultaneously within a block. Its therefore important to select a block dimension that fills the warps entirely with threads to utilize all the CUDA cores during warp execution.

The following example of a kernel invocation where a image is the computational domain show how the grid and blocks dimensions are set in CUDA:

```
dim3 grid(image_x/block_x, image_y/block_y, 1);
dim3 block(block_x, block_y, 1);
someKernel<<grid, block>>(param1, param2);
```

Within a kernel the number of blocks contained in a grid can be determined by using a built-in CUDA variable «`gridDim`», and the number of threads within a block can be determined with the built-in CUDA variable «`blockDim`». It is also possible to get the unique id or location of the thread block within the grid by the built-in CUDA variable «`blockIdx`», as well as the unique id of the thread within the thread block by the built-in CUDA variable «`threadIdx`». Since grids and blocks can be defined in 1D to 3D, these variables also have a x, y, and z component. These variables are used by the executing threads to access the correct section of the computational domain the thread is assigned too for example calculating a corresponding array index.

The following example show how `blockDim`, `blockIdx`, and `threadIdx` can be used to access a array within a kernel:

```
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
int index = x + y * image_x; int value = a[index];
```

Memory Model

In addition to the CUDA execution model the CUDA memory model, describing the GPU memory hierarchy and memory access, is also very important to consider when creating GPU computing programs. With traditional CPUs data is stored in a large main memory that has a high-latency memory access. This high-latency memory access can become a performance constraint when executing programs where the computational problem is memory and not compute bound. To hide this latency the data is moved closer to the CPU with the use of fast and very low-latency on-chip caches. The use of these caches is transparent and all memory access is uniform, which means that the programmer does not need to explicitly handle the use of caches. It can however be beneficial to limit the workloads of the program to the size of the caches since this will decrease cache misses and access to the main memory.

Similar to the CPU the GPU also has a large high-latency main memory called the global memory, caches that are called local memory (compute capability 2.0 and above), and registers. However, due to the number of computation cores on a GPU there is a very limited space for caches on the chip, and therefore the hiding of latency must instead

be achieved through high bandwidth utilization by concurrently executing threads. The achievable bandwidth utilization is determined by restrictions to the memory access pattern of thread warps which enables the coalescing of memory access into as few accesses transactions as possible. Because of parallel access to memory the CUDA memory mode is also said to have a relaxed memory consistency, which means synchronization is needed to share data between thread warps in different SMs.

In addition the the global memory, caches, and registers, the GPU also has a shared memory, constant memory and a texture memory. The shared memory is a small and fast low-latency on-chip memory available on each SM that has its own address space, and can be accessed by all threads in a thread block allocated to the SM. The shared memory is divided into banks which can be accessed simultaneously by as many threads as it has banks. However, if multiple threads try to access the same banks at the same time bank conflicts can occur, forcing the bank access to be serialized and thereby much slower. Such bank conflicts can be avoided by block synchronization using the `__synchthreads()` barrier function within a kernel. If there are no bank conflicts the shared memory can have a similar performance as registers. Shared memory must be explicitly declared by the programmer and it is commonly used for communication and sharing frequently used data between threads. The constant and texture memory is located on the device (off-chip) alongside the global, local, and constant memory. The constant memory is used to store constant global variables, and the texture memory is traditionally used for graphics textures, but can also be used to store more general structures. The texture memory also has the added benefit of hardware units that perform interpolation when accessing the stored data, clamping and wrapping memory accesses to prevent out-of-bounds faults, and spatial caching [33].

Figure 2.2.2 shows an illustration of the CUDA memory model and the connection between memory and GPU grid, thread blocks, threads and the CPU host. As the figure shows the CPU can access the GPUs global, constant, and texture memory. This access is done through the PCI-Express bus and is often viewed as a major bottleneck in GPU computing because of the latency involved. It is therefore recommended to limit the transfer of data between the host and the device, and if possible only at the initialization and end of a GPU programs execution. Figure 2.2.2 also shows that global, constant and texture memory can be accessed by all the threads in every threads block, but shared memory is restricted to all threads within a thread block and that both local memory and registers are private for each thread. Table 2.2.2 shows an summary of the memories including their location, if they are cached or not, their access permissions, scope, and lifetime within the GPU program.

Similar to the kernel qualifiers also has a set of qualifiers to indicate what type of memory a variable is allocated to. For variables declared in the global scope (outside the scope of a kernel) the qualifier `__device__` is used, for variables in shared memory the qualifier `__shared__`, and for constant variables the `__constant__` qualifier.

Allocation of memory on the device from the host is possible by using the CUDA function `cudaMalloc()`, and freed by using `cudaFree()`. Memory can also be copied by using

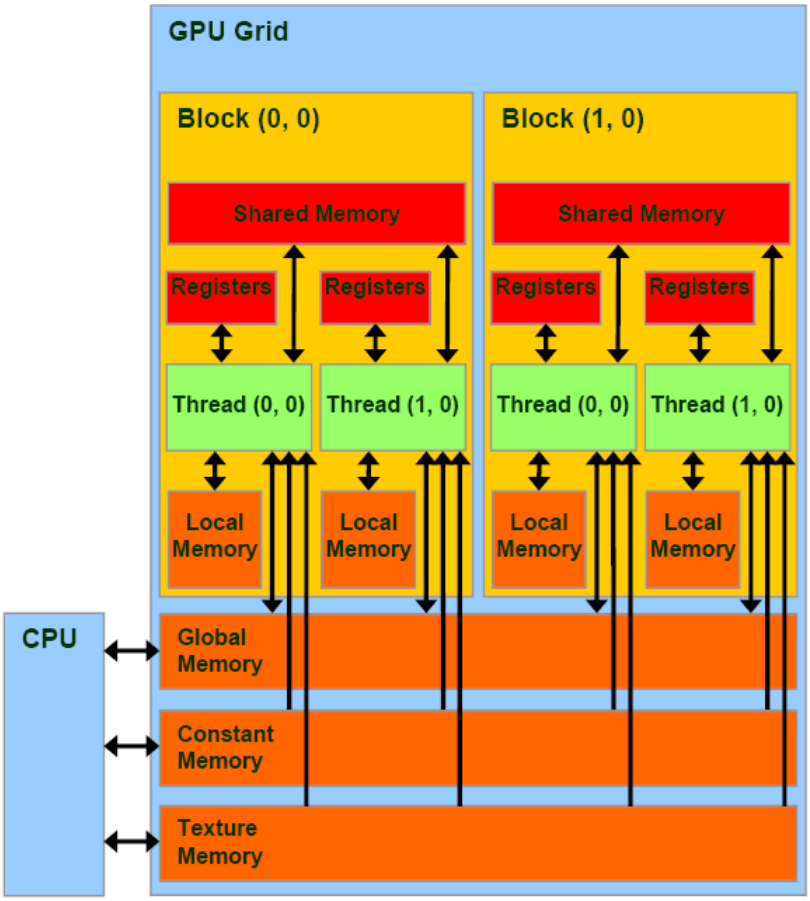


Figure 2.2.2: The CUDA memory model (With permission from NVIDIA) [33].

Memory	Located	Cached	Access	Scope	Lifetime
Register	Cache (on-chip)	N/A	Host:none Kernel:RW	Thread	Thread
Local	Device (off-chip)	Yes	Host:none Kernel:RW	Thread	Thread
Shared	Cache (on-chip)	N/A	Host:none Kernel:RW	Block	Block
Global	Device (off-chip)	Yes	Host:RW Kernel:RW	Program	Program
Constant	Device (off-chip)	Yes	Host:RW Kernel:R	Program	Program
Texture	Device (off-chip)	Yes	Host:RW Kernel:R	Program	Program

Table 2.2.2: CUDA memory properties [9].

CUDA	OpenCL	Description
SM	Compute Unit	A SIMD Core
Streaming Processor	Processing Element	Executing a single work-item
Global Memory	Global Memory	RAM
Shared Memory	Local Memory	Shared within a work-group
Local Memory	Private Memory	Thread memory
Kernel	Kernel	Parallel code
Thread	Work-item	Single instance of a kernel
Thread Block	Work-group	Group of work-items/threads
Grid (of thread blocks)	NDRange	Array of threads blocks

Table 2.2.3: CUDA vs. OpenCL terminology.

the CUDA function `cudaMemcpy()` with a keyword defining if the data should be copied from host to host, host to device, device to host, or device to device.

2.2.2 OpenCL

OpenCL (Open Computing Language) [25] is a open, royalty-free industry standard describing an API for heterogeneous computing. The OpenCL standard is maintained by the non-profit technology consortium Khronos Group and was first released in November 2008. OpenCLs focus is to enable parallel programming on any hardware that supports parallel execution, including GPUs, CPUs, DSPs, and FPGAs. This means OpenCL is not only limited to traditional computers, and is also used in some mobile devices. Since OpenCL is only a standard it is up to each vendor to implement its own API for their devices. Currently OpenCL has been adopted and implemented by the vendors Intel, AMD, NVIDIA, ARM, and IBM. OpenCL uses the C99 programming language with some limitations and additions, and the vendors also supply specific features, like double floating-point support, added to OpenCL through extensions to the API. Because of OpenCLs independent nature OpenCL code is portable across various target devices with a guaranteed correctness, but performance of a given kernel can not be guaranteed since different devices might perform better or worse on different tasks.

OpenCL and CUDA are similar in its execution and memory model when used with a GPU, but differ in how kernels are created. In OpenCL kernels are created in separate program files that are collected in program objects that are compiled at runtime into kernel objects which are passed to the device driver for optimization. The OpenCL syntax is also generally a bit more verbose then the CUDA syntax and uses different terminology for execution constructs and components. The different terminology used in CUDA an OpenCL can be viewed in Table 2.2.3.

2.2.3 OpenACC Version 1.0 - A Directive-Based Approach

Both the CUDA and OpenCL programming models feature a function rich and low-level API for GPU programming, giving the programmer fine grained control over the execution and memory management of a GPU program. Even though this fine grain control can enable the programmer to create very fast and efficient programs, it can also makes it difficult to program, complicate software design, and tie code the a specific device or vendor. As a solution to this, recent approaches try to improve programmability with an abstraction to GPU programming by making the compiler responsible for many of the low-level programming tasks through compiler directive-based high-level APIs such as OpenACC.

OpenACC is a open programming standard developed by NVIDIA, The Portland Group, CAPS, and Cray, for parallel computing on heterogeneous systems using compiler directives. The OpenACC standard was first released as version 1.0 [4] in November 2011 and it describes a API for GPU programming that includes compiler directives, library routines and environment variables. In March 2013 draft for OpenACC version 2.0 [34] was released and is currently open for public comment. The OpenACC API and its directives can currently be used in combination with the C, C++, and Fortran programming languages and aims to be portable across operating systems, host CPUs and accelerators. The OpenACC standard is maintained by the non-profit OpenACC corporation, which in addition to the founding members, has as of May 2013 expanded to also include the organizations: Allinea, Georgia Tech, University of Houston, Oak Ridge National Lab, Rogue Wave, Sandia National Laboratory, Swiss National Supercomputer Center, and Technical University Dresden.

The primary goal off OpenACC is to simplify programming applications on heterogeneous systems that are composed of general purpose processors, such as CPUs, with attached accelerator devices, such as GPUs. This simplification is done by using directives to indicate what loops or region of code should be executed on the accelerator in parallel and letting the compiler create the needed target code. This gives the programmer a simple start to GPU programming without the need to think as much about the execution and memory model, but the OpenACC API also has features for a more explicit control of the program execution. It is also important to note that the use of directives does not make parallel programming easier in general, as it is still up to the programmer to identify parallel regions or modify existing code to make it suitable for parallelism.

OpenACCs use of directives for parallel computing also makes it similar to the OpenMP standard for shared memory multiprocessing programming. This is no coincidence since many of the developers of OpenACC standard are also on the OpenMP Architecture Review Board, and that the long term aim of OpenACC is to gather experiences that will help drive the effort of OpenMP for accelerators.

Listings 2.1 and 2.2 contains examples, written in the C programming language, of how OpenMP and OpenACC uses directives to parallelize a region of code. Both OpenMP

```
int main() {
    double pi = 0.0;
    #pragma omp parallel for reduction(+:pi)
    for(long i=0; i<N; i++) {
        double t = (double)((i+0.5)/N);
        pi += 4.0/(1.0+t*t);
    }
}
```

Listing 2.1: Parallel loop using OpenMP with reduction clause.

```
int main() {
    double pi = 0.0;
    #pragma acc kernels
    for(long i=0; i<N; i++) {
        double t = (double)((i+0.5)/N);
        pi += 4.0/(1.0+t*t);
    }
}
```

Listing 2.2: Parallel loop using OpenACC.

and OpenACC uses the `#pragma` keyword to indicate where there is a compiler specific directive. The #pragma is followed by another keyword identifying the parallelism library. For OpenMP this identifier is `omp` and for OpenACC it is `acc`. Following the identifier the specific directive is stated along with any clauses that defines additional conditions or behavior for the parallel region.

Since OpenACC is only a standard describing the API, it is up to developers to create compilers with OpenACC support and decide how the GPU program should be created. A common approach is to perform a source-to-source translation of the regions marked by the OpenACC directives to CUDA or OpenCL target code built for the appropriate architecture.

Currently there are commercial OpenACC compilers available from The Portland Group[1], CAPS[2] and Cray[3], but there is also a open source compiler developed at the University of Laguna in Spain, called accULL[4] [10]. The accULL compiler is also the first OpenACC compiler to support both CUDA and OpenCL targets and implements a runtime system for memory management and kernel execution.

OpenACC is a fairly new standard and therefore only a few publications have been released addressing the OpenACC programming model. Reyes et al. [42] performs a preliminary evaluation of the compilers from PGI, CAPS and their own compiler accULL, using selected kernels from the Rodinia benchmark suite. Reyes shows that the accULL

[1] http://www.pgroup.com
[2] http://www.caps-entreprise.com
[3] http://www.cray.com
[4] http://accull.wordpress.com

compiler performs similar or better to the other compilers, while still only ranging from 5% to 67% of the performance when using CUDA. It should however be noted that multiple new versions of the tested compilers have since been released. Wienke et al. [41] evaluates OpenACCs performance on real-world applications compared to the PGI accelerator model and OpenCL, and also considers the tradeoff between programmability, productivity and performance. It is found that the best-case performance of a moderately complex kernel in the testing application using OpenACC is about 80% of the OpenCL version, and that more complex kernels reaches about 40%.

In addition to OpenACC other high-level GPGPU programming models also exists, such as PGI Accelerator Model, CAPS HMPP, hiCUDA, OpenMPC, and OmpSs.

Execution Model

OpenACC is generally aimed at heterogeneous systems that has a host processor, such as a CPU, and an attached accelerator device, such as a GPU. On such systems a program is usually executed on the host until a compute intensive region that can be executed in parallel is reached. Such regions are typically called parallel regions and usually contains one or more work-sharing loops that can be executed as a kernel. When such a region is reached the host can offload the region to the accelerator for fast parallel execution. After the execution is finished the results are transfer back to the host again. Because of this behavior OpenACC has a host-directed execution model.

The host must allocate device memory, initiate data transfer, send code to the device, pass needed arguments to the parallel region, wait for the execution to complete, transfer results back again, and deallocate the device memory. Most of this process is done by the OpenACC compiler, but it can also be specified in more detail by the programmer.

In OpenACC accelerators executes parallel regions that are similar to CUDA kernels, which typically contains single or nested work-sharing loops. These regions are indicated by the use of directives `#pragma acc kernels` or `#pragma acc parallel`, that can either be executed synchronously or asynchronously.

The parallel regions can be executed with three levels of parallelism, called gang, worker, and vector. Generally these are mapped to an architecture that is a collection of processing elements, where there is one or more processing element per node, each processing element is multithreaded, and each processing element can execute vector instructions. The mapping between gang, worker, and vector is implementation-dependent, but typical mappings to a CPU might be that gang is the number of CPUs, worker is the number of CPU cores, and vector is the number of SIMD instructions per core. On CUDA a typical mapping is that gang corresponds to the number of thread blocks in a grid, vector is the number of threads per thread block, and worker is locked to the warp size. If no specific number of gangs, workers, and vectors is specified the compiler will select values that matches the size of the computing domain and the underlying hardware architecture. However, there are no guarantees that the values selected by the compiler result is

the optimal choice, and in some cases it might be wise to tune the code to fit particular target architecture.

Memory Model

Since OpenACC is mainly targeting heterogeneous systems with a host CPU and a attached accelerator device, such as an GPU, the memory of the host and accelerator is usually completely separated. All data allocation and data movement between host memory and accelerator device memory must be performed by the host through runtime library calls. Besides the host and device memory OpenACC also has the notion of private and shared memory, where the private memory is usually hardware-managed caches and the shared memory is software-managed cache like the shared memory on CUDA. All of these memory concepts are implicit and managed by the compiler, based on compiler directives declared by the programmer.

The directives can describe the allocation and data movement by telling the compiler that memory should be allocated in device memory, data should be copied from the host to the device and back again, only copied from the host to the device, only copied to the host from the device, or that the data already exists in the device memory.

In a typical parallel region data is copied to the device from the host at the start of the region, and copied back to the host when the region ends. However, when there are multiple parallel regions that all work on the same data in sequence the copying of data back and forth between each region might slow down performance. It is therefore possible to use the OpenACC runtime API to allocate memory at the start of the program and pass the device memory pointers to the parallel regions, allowing the data to stay on the device through the programs execution.

In addition the compiler also creates barrier constructs to prevent simultaneous access to the same memory that might result in memory coherence issues.

OpenACC Constructs

In the C and C++ programming language OpenACC directives are specified by using the `#pragma` mechanism combined with the `acc` keyword, and the directive name. Directives can also be combined with clauses to define a more detailed behavior of the directive. Directives combined with clauses are called constructs in OpenACC and can be categorized as compute, data, host_data, and loop constructs. The compute construct is also further split into parallel and kernel constructs. The general form of a construct is as follows:

```
#pragma acc directive-name [clause ...]
    structured block
```

All constructs can use two general clauses:

```
if ( condition )
async (expression)
```

Where the if clause sets a condition on the execution of the region, and async clause enables the region to be executed asynchronous.

Data Construct

The data construct defines the allocation and movement of data within a defined region. This includes if memory should be allocated on the accelerator, if data should be copied from the host to the accelerator upon region entry, or if data should be copied from the accelerator to the host when exiting the region. The data handled by the data construct can either be scalars, arrays or subarrays. The data construct can typically be used to contain multiple kernel regions to remove the need for copying data from host to every kernel.

```
#pragma acc data [clause ...]
    structured block
```

The data construct has, in addition to the general clauses, some clauses that define how data movement and memory allocation.

- `copy(list)` - Allocate memory on accelerator and copy data from host to accelerator when entering region. Copy data back when exiting region.
- `copyin(list)` - Allocate memory on accelerator and copy data from host to accelerator when entering region, but not back again.
- `copyout(list)` - Allocate memory on accelerator and copy data to the host when exiting region.
- `create(list)` - Allocate memory on accelerator for use in the region only.
- `present(list)` - Indicates that data is already copied to accelerator in a containing data region.
- `deviceptr(list)` - Data is a pointer to accelerator memory meaning it is already allocated on accelerator. Typically used for CUDA/OpenCL pointers or data allocated using acc_malloc.
- `present_or_copy[in|out]` - Copy data in or out of region if data is not already present.
- `present_or_create(list)` - Create data on device if no already present.

Kernel Construct

The kernel construct identifies loops or loops nests within its defining region and converts them to separate kernels that can be executed in parallel on the accelerator. The generated kernels will be executed in order on the accelerator, but may have different configuration of gangs, workers, and vectors.

```
#pragma acc kernels [clause ...]
    structured block
```

The kernel construct can use the general clauses and any of the data clauses described under the data construct. If the async clause is not present, then there will be an implicit barrier at the end of the kernel region forcing the host program to wait until all kernels finished executing.

Parallel Construct

The parallel construct defines a region where upon entering parallel gangs of workers are created to execute the parallel region of the construct. One worker in each gang executes the region until a work-sharing construct is reached, then the work is split across workers or gangs. In a parallel constructs region the number of gangs and workers in each gang remains constant.

```
#pragma acc parallel [clause ...]
    structured block
```

The parallel construct can use the general clauses, data clauses, and be combined with the loop construct.

Loop Construct

A loop construct instructs the worksharing of the loop, immediately following the directive, among the threads on the accelerator. The loops parallelism can either selected by the compiler or set by the gang, worker, and vector clauses. The loop construct also has clauses for indicate loop-private data and reduction operations.

```
#pragma acc loop [clause ...]
    loop
```

The loop construct can be combined with the parallel and kernel construct and thereby also the clauses offered by those construct. The loop construct also has its own clauses some of which have different behavior depending on whether the loop construct is combined with the parallel or the kernel construct [4].

- `collaps(n)` - Indicates that directive should be applied to the following n nested loops.

- `seq` - Execute the following loop sequentially on the accelerator.
- `reduction(operator:list)` - Combine private variables across iterations of the loop.
- `gang(number)` - Iterations are to be executed in parallel across gangs.
- `worker(number)` - Iterations are to be executed in parallel by distributing them among the workers within a single gang.
- `vector(number)` - Iterations of the loop or loops are to be executed in vector or SIMD mode.
- `independent` - Tells the compiler that there are no data dependencies between iterations of the loop. Can alternatively use the restrict keyword on pointers in C.
- `private(list)` - Create private copy of data for each iteration of the loop.

Other Directives and Constructs

In addition to the directives and constructs already listed, OpenACC has some more specific directives and constructs which can be used.

- `host_data` - Construct that makes the address of accelerator data available on the host. Can be used for initial data transferal.
- `cache` - Directive that caches data in fast software-managed memory (Shared memory on CUDA).
- `update` - Directive that updates all or part of a host memory array with values from the corresponding array in accelerator memory.
- `wait` - Directive that causes program to wait for asynchronous accelerator activity to complete, such as parallel regions, kernel regions, or update directive.
- `declare` - Directive specifying that data is to be allocated in device memory for the duration of the implicit data region or function, subroutine or program. Also specify data movement between host an device.

Runtime Library Routines

OpenACC also has runtime library routines that can be used by the programmer and are made available through the inclusion of the header file named *openacc.h*. These library routines are shown in Table 2.2.4.

Library Routine	Description
`acc_get_num_devices()`	Return the number of accelerators devices of given type connected to host.
`acc_[set|get]_device_type()`	Set or get the accelerator device type for the parallel of kernel region.
`acc_[set|get]_device_num()`	Set or get which accelerator device to use.
`acc_async_test[_all]()`	Test for completion of associated or all asynchronous activities.
`acc_async_wait[_all]()`	Wait for completion of associated or all asynchronous activities.
`acc_[init|shutdown]()`	Initialize or shutdown the runtime for specified accelerator type.
`acc_[malloc|free]()`	Allocate data to accelerator global memory or free the memory.

Table 2.2.4: OpenACC 1.0 runtime API functions.

2.2.4 OpenACC Version 2.0 (Draft)

In March 2013 a draft for OpenACC version 2.0 [34] was announced by the OpenACC corporation and released to the public. The purpose of this draft is to get comments and feedback to the purposed changes made to the API and incorporate them before a final version is developed. The draft for OpenACC version 2.0 features new capabilities and expanded functionality and has the aim to increase the performance and ease of parallelizing code, and improve developers experiences using OpenACC.

The increase in performance comes from new controls over data movement including better handling of unstructured data and improvements in support for non-contiguous memory, and also features such as dynamic parallelism by allowing nested parallel/kernel regions. The developers experiences is simplified by adding support for explicit function calls and separate compilation, allowing the creation and reuse of libraries and accelerated code. The draft also includes some clarifications to sections of the 1.0 standard.

As the proposed version 2.0 of OpenACC is still a draft, some sections are still under discussion and may change for the final version. The topics that are most likely to change are [34]: the default attribute for scalar variables in parallel and kernels constructs. The spelling for statically scheduling loop iterations onto gangs. The spelling for making global data available for separate compilation. Extending features to include more support for Fortran contained procedures. Additional API routines.

Changes in Directives and Constructs from Version 1.0 to 2.0

The following changes has been added to the draft of OpenACC version 2.0 [34]:

- A `default(none)` clause added to `parallel` and `kernels` directives - Clause preventing the compiler implicitly determining data attributes for variables that are referenced in the compute construct that do not appear in data clause.
- Added `acc_async_sync` and `acc_async_noval` values for `async` clauses.
- Clarified allowable loop nesting - Gang may not appear inside worker, which may not appear within vector.
- `wait` clause on parallel, kernels, update and wait directives - Wait for completion of asynchronous operations.
- `enter data` and `exit data` directives - Enter data define scalars, arrays and subarrays to be allocated in device memory for the remaining duration of the program or until exit data directive is used. Can also specify that data should be copied in and out using clauses.
- `link` clause for the `declare` directives - Specifies that a named variable should be allocated in accelerator memory only when they appear in a data clause for a data construct, compute construct or `enter data` directive.
- `tile` and `auto` loop clauses - tile specifies that the implementation should split each loop in the loop nest into two loops, with a outer set of tiles loops and inner set of element loops. Also possible to specify tile sizes. auto clause specifies if the implementation should select to apply gang, worker or vector parallelism to the loop.
- `update local` introduced as a synonym for `update host`.
- `routine` directive, support for separate compilation - Tells the compiler to compile a given procedure for an accelerator as well as the host.
- `device_type` clause and support for multiple device types.
- Nested parallelism using parallel or kernels region containing another parallel or kernels region.
- New concepts: gang-redundant, gang-partitioned; worker-single, worker-partitioned; vector-single, vector-partitioned; thread - Concepts describing parallel execution modes.
- Defined behavior with multiple host threads, such as with OpenMP.
- Recommendations for specific implementations - recommendations for standard names and extensions to use for implementations for specific targets and target platforms to promote portability.

Library Routine	Description
`acc_[present_or]_copyin()`	Allocate and copy in memory to the accelerator device. present_or version checks if data is already located on device first.
`acc_[present_or]_create()`	Allocate memory on accelerator device. present_or version checks if data is already allocated on device first.
`acc_[copyout|delete]()`	copyout copies data from device memory to corresponding local memory and deallocates memory on device. delete allocates memory on device without any copy operation.
`acc_[deviceptr|hostptr]()`	Return the device/host pointer associated with a specific device/host address.
`acc_[un]map_data()`	Map/unmap preallocated device data to the specified host data.
`acc_is_present()`	Test if a host variable or array region is present on device.
`acc_wait_async()`	Enqueue a wait operation on one async queue for the operations previously enqueued on all other async queues.
`acc_memcpy_[from|to]_device()`	Copy data from device to host or to device from host.
`acc_update_[device|local]()`	Update device/local copy of data from the corresponding device/local memory.

Table 2.2.5: New library routines in OpenACC v 2.0.

New Library Routines

As shown in Table 2.2.5 OpenACC version 2.0 includes a number of new runtime library routines. Most of these new routines deals with the transferral of data between host and device and making it possible to control the device memory directly from the host. In addition to these new library routines the names for the routine `acc_async_wait` and `acc_aync_wait_all` defined in OpenACC version 1.0 have been replace with the names `acc_wait` and `acc_wait_all`.

As with OpenACC version 1.0 the library routines are made available by including the `openacc.h` headerfile.

2.2.5 Other Directive-Based Programming Models

In addition to OpenACC there exists other compiler directive-based programming models for accelerators and heterogeneous systems. All of these models are similar in the

way they use directives to indicate parallel regions of code, although they may differ in scope and features.

PGI Accelerator Model

The PGI Accelerator Model [1] is a directive-based parallel programming model developed by the Portland Group and is the model that OpenACC is based on. It has similar functionality and scope to OpenACC by that it targets host systems with attached accelerator devices, and it is also one of the models that OpenACC is based upon. It has a different syntax to OpenACC and there are some differences in functionality [1]. Parallel kernels are declared using the `#pragma acc region` directive. PGI Accelerator Model has only two levels of parallelism unlike the three levels available with OpenACC. To compile programs the PGI Accelerator Models uses the PGI C and Fortran compiler for source-to-source compilation to intermediate CUDA code before a binary is created.

Since PGI is one of the founders of the OpenACC standard, the PGI Accelerator model will in the future be a superset of OpenACC [1].

hiCUDA

hiCUDA [2] is a high-level approach towards GPU programming using compiler directives to create CUDA target code. Its directives translate to CUDA low-level API functions, by parsing the original source code using GNU-3 frontend, extracting the loops and injecting the CUDA runtime calls. To create the CUDA target code it uses the C code generator with extended CUDA syntax. When the target code is created the NVIDIA CUDA compiler is used to create the binary. The main difference between hiCUDA and other directive-based GPU programming models is that hiCUDA follows the CUDA programming model very closely which requires more knowledge about the underlying hardware architecture.

OmpSs

OmpSs [12] is a compiler directive-based approach, developed by the Barcelona Supercomputing Center in Spain, aimed at heterogeneous computing systems like clusters with distributed accelerator devices. OmpSs combines the StarSs programming model with OpenMPs task based programming model to create a "node" level programming model, where StarSs allows for runtime analysis between tasks, and automatic data transfers. The programming languages currently supported by OmpSs is C, Fortran and Java.

OpenMPC

OpenMPC [13] is developed by the School of Electrical and Computer Engineering at the University of Purdue. OpenMPC aims to combine OpenMP directives with GPGPU

programming on the CUDA platform. OpenMPC has two main focuses; programmability and tunability. The tunability is achieved through a fully automatic compilation and user-assisted tuning system to create optimized CUDA programs.

OpenHMPP

Hybrid Multicore Parallel Programming or OpenHMPP [3] is a programming standard for heterogeneous computing created by CAPS in 2007. Its aim is similar to OpenACC and the PGI Accelerator Model by using compiler directives to program hardware accelerators. OpenHMPP supports the C and Fortran languages and is supported by compilers from CAPS and PathScale. Unlike OpenACC, OpenHMPP is based on the concept of codelets, that are functions to be executed remotely on the hardware accelerator controlled by a runtime system. OpenHMPP also focuses heavily on optimization of data transfer between the CPU main memory and the accelerator device memory.

CAPS is one of the founding members of the OpenACC standard, and therefore OpenHMPP can be viewed as a superset of the OpenACC API with some advanced features. Those features include the management of multiple devices, the use of accelerated libraries with directives, and the integration of hand-written CUDA device kernels.

Chapter 3

The Current HPC-Lab Snow Simulator

This chapter gives a description of the current CUDA snow simulator developed by the NTNU HPC-Lab is the foundation for this thesis. First a brief history of the snow simulator is given followed by a description of the simulators structure and control-flow. Afterwards an overview of the theoretical methods used in the snow particle and wind simulation are presented along with some implementation details.

3.1 Brief History of the Snow Simulator

The snow simulator was first originally created by Ingar Saltvik and presented in his master's thesis [31] in 2006, also published as a paper with the collaboration of Anne C. Elster and Henrik R. Nagel [24]. Saltvik's model for the real-time animation of the simulation of snow is a combination of methods that handle the movement of snowflakes or snow particles through the air, the simulation of wind, the accumulation of snow on objects such as terrain, and rendering the result using 3D visualization techniques.

Saltvik's snow simulator was designed to run on systems with multiple CPUs by using threads to parallelize the simulation steps and was able to reach real-time performance using a few tens of thousands snow particles. The data and task parallelization scheme used by Saltvik gave a speedup of 1.99x when using a system with two CPUs [31]. Since 2006 many master and specialization projects at the HPC-Lab have been based on the snow simulator created by Saltvik. Most of these projects have mainly focused on improving the simulation performance of the snow simulator, though some have also focused on using alternative methods in the simulation steps or improving the visual representation.

In 2008 the snow simulator was greatly improved in a master's thesis by Robin Eidissen [19]. Eidissen improved the snow simulator by performing the simulation steps in parallel

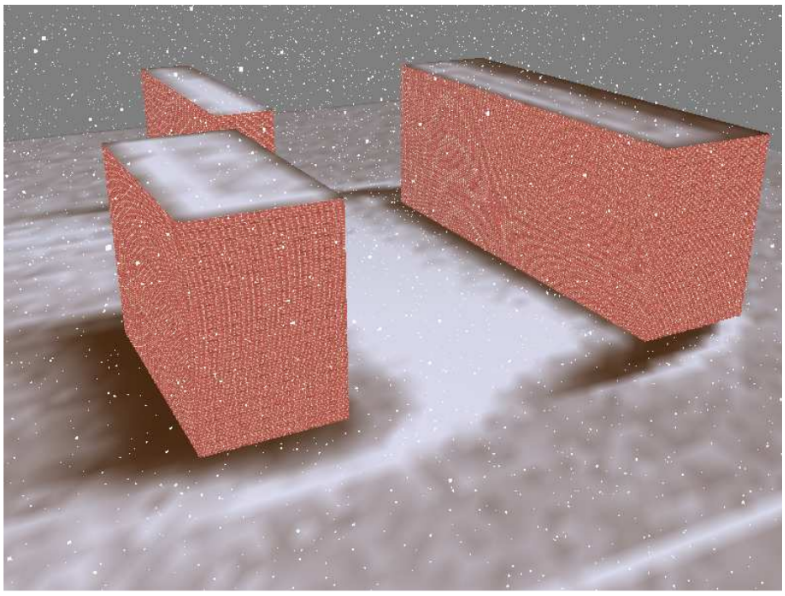


Figure 3.1.1: Screenshot of the original snow simulator created by Saltvik.

on a GPU using CUDA. Using Eidissen's version of the snow simulator millions of snow particles are simulated and animated in real-time using current graphic cards. Eidissen also greatly improved the visuals and rendering performance of the snow simulator by creating terrains using heightmaps and improving the geometry creation. The GPU snow simulator was further developed in 2009 by Gjermundsen[35] with the inclusion of the Lattice-Boltzmann method in addition to the Computational Fluids Dynamics method using the Navier-Stokes equations in the wind simulation.

In a specialization project by Chelliah [20] in 2010 the simulator was also optimized for the NVIDIA Fermi GPU architecture providing a speedup of 1.5x over the previous version. Later the same year in a master thesis by Steinsland [21] the snow simulator was also ported to OpenCL to support devices other then NVIDIA GPUs. The functionality was improved further by Lien[36] in 2011 to include the possibility to import real terrain data into the simulator and render a optimally placed road through the given terrain. In 2012 the simulator was ported to OpenCL for mobile devices by Vestre [22] and the visuals were further enhanced by Babington[37]. Master's thesis by Rovde [43] and Krog [44] were also done in the context of the snow simulator, but has not been incorporated into the codebase. These thesis explore the use of smoothed-particle hydrodynamics for more accurate simulation of fluid flow and the simulation of snow avalanches.

During the spring of 2013 there are two master's thesis, including this, based on the snow simulator. The other thesis is done by Andreas Nordahl and consists of rewriting much of the rendering system and enhance the visuals by using techniques such as perlin noise for more realistic texturing of the terrain and snowflakes, and also the addition of visual features such as fog. The work done by Nordahl on the visuals for the simulator

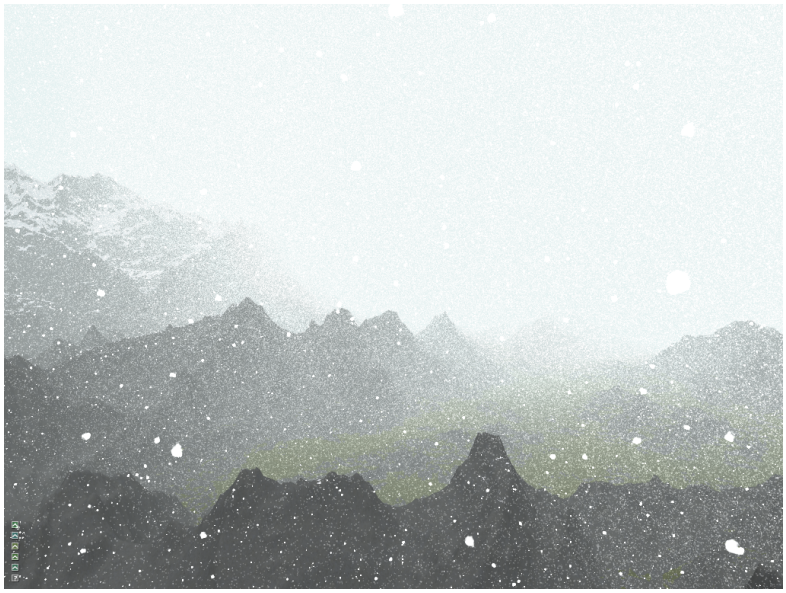


Figure 3.1.2: Screenshot of the current CUDA based snow simulator with new visuals by Andreas Nordahl.

is also included in the implementation presented in this report.

3.2 Simulator Organization Overview

At the start of this project it was decided to update the latest version of the CUDA simulator [37] as the first step to porting the simulator to OpenACC. The updating work was done in cooperation with fellow master's student Andreas Nordahl, with Nordahl responsible for the graphics system and the author responsible for the simulation system. The work done is mainly cleaning up the code by removing redundant functions, out commented sections, and unused variables, but also to achieve higher cohesion and modularity within the codebase to make it easier to develop the code in future projects. The cleanup process took quite a bit of effort since the simulation code is largely uncommented and heavily optimized making it difficult to read and understand by someone without much experience with CUDA programming. Updating the simulator did however give good insight into how the simulator works and made the porting process easier.

The simulator code is written in a combination of the programming languages C, C++, and CUDA. The core logic of the application and its supporting classes, shown in Figure 3.2.1, are written using object-oriented programming with C++11. The class diagram also shows the functions contained within each class, but has datafields omitted for brevity. The section of the class diagram labeled *ParticleSystem* is the system responsible

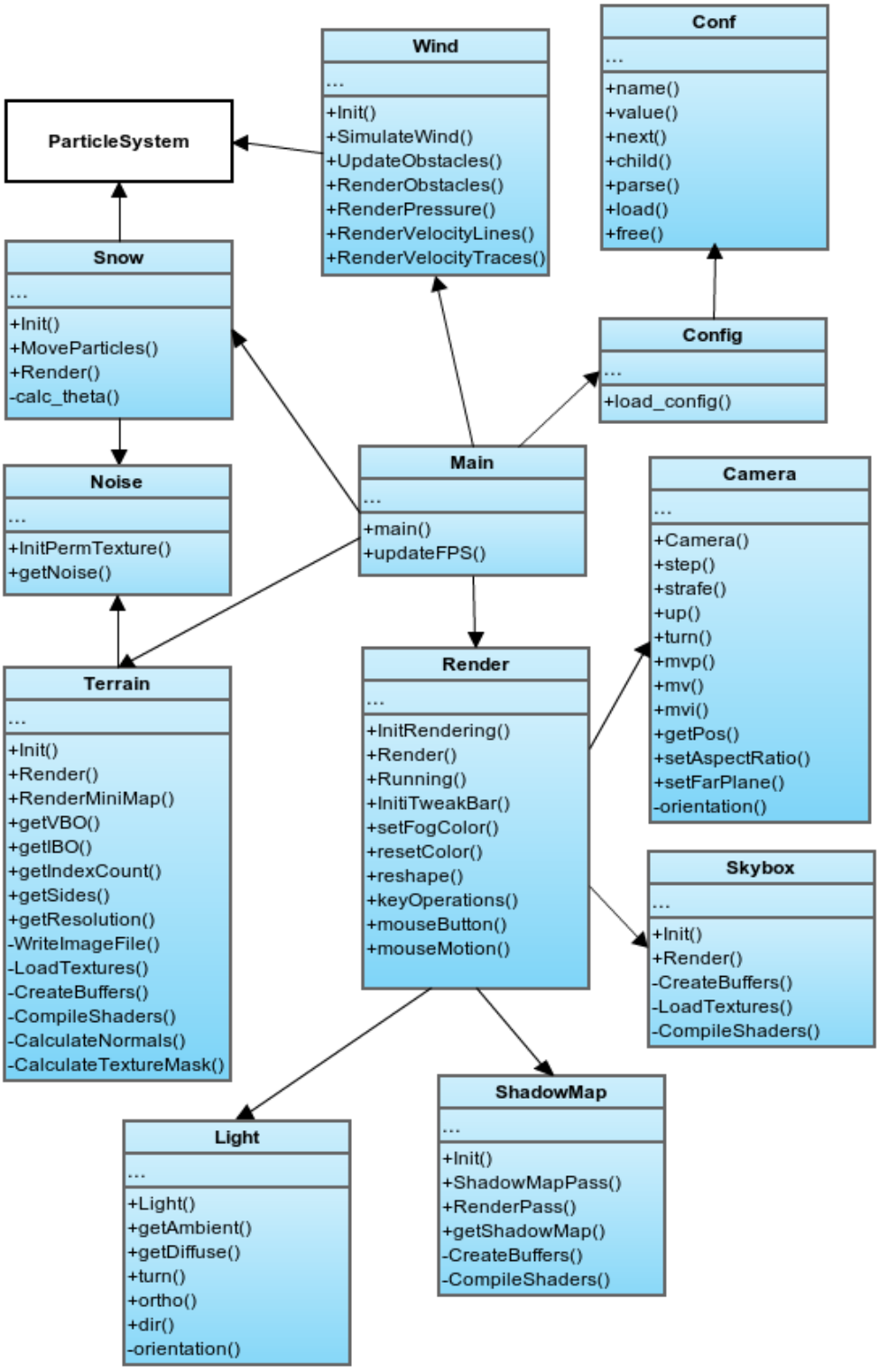


Figure 3.2.1: Class diagram of the new CUDA snow simulator. Figure is created in collaboration with Andreas Nordahl.

for performing the snow particle and wind simulations. The simulation system is logically abstracted from the other parts of the application to enable plugging in simulation systems written using different GPGPU APIs. The simulation system is written using the CUDA 5.0 toolkit and is therefore compiled separately using the NVCC compiler. Interfacing with the particle system is done through the classes *Wind* and *Snow*, which are responsible for initializing and managing the simulation.

The rendering system uses the OpenGL 4.0 graphics programming API in combination with shaders written in OpenGL Shader Language (GLSL) to enable a programmable graphics pipeline. Its responsibilities include managing graphical features such as windowing, lighting, shadows, skybox, and camera position, but also handles user input. The rendering system also performs calls to graphical functions contained within other classes such as the terrain rendering in the *Terrain* class, snow particle rendering in the *Snow* class, and windfield debug rendering in the *Wind* class. The rendering system also uses the graphics libraries GLEW[1] version 1.9 to load OpenGL extensions, and and GLFW[2] version 2.7 for window and input management.

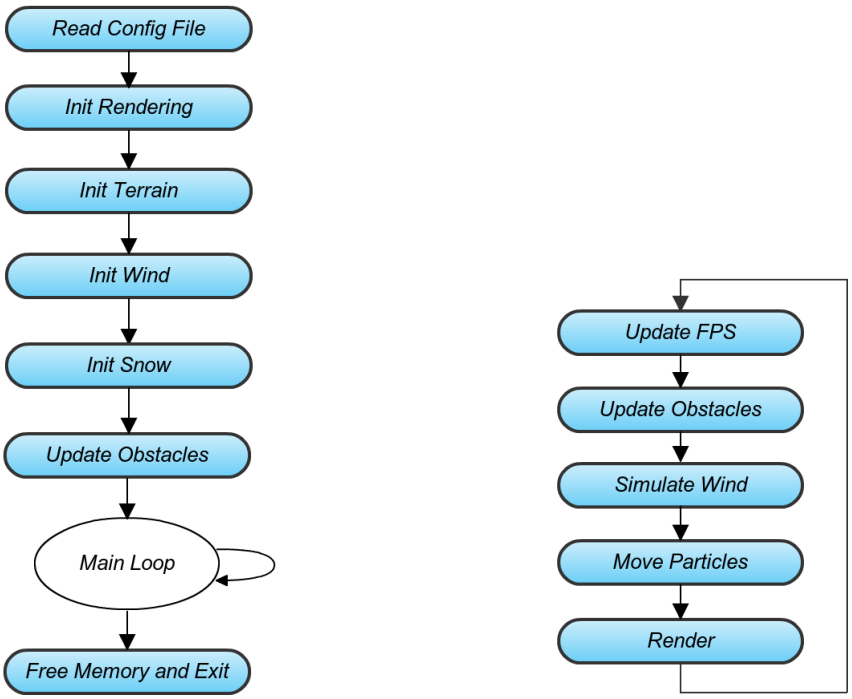
Some features available in previous versions of the simulator are left out of the new updated version. These are the features: stereo rendering [19], road generation [36], and Lattice-Boltzmann fluid simulation [35]. It was decided to leave these features out because of time constraints and to keep the codebase as simple as possible. However, these features should be fairly easy to incorporate in the future.

Initialization and Main Loop

The *Main* class contains the main function which is the entry point of the application. The main function is responsible for initializing the various systems of the application and executing it by running the main loop. As shown in a) in Figure 3.2.2 the main function starts by reading a configuration file for the execution of the program followed by initialization of the rendering, terrain, wind, and snow systems. After the initialization it runs a function in the wind simulation system that creates a initial obstacle field from the terrain to be used by the windfield simulation. After the initialization steps are performed the main loop is executed. When the main loop is exited the main function will free all allocated memory and exit the application.

The main loop, also called a game loop in graphical applications, is a continuous loop running until a termination event is raised. The purpose of the main loop is to keep the systems of the application running and updating the applications state without needing to wait for user input. The main loop is shown in b) in Figure 3.2.2 and consists of five stages resulting in a new output frame to the screen. The first stage in the loop is to update the number of frames per seconds the application is running at. The second stage is to update the obstacle field used in the windfield simulation. The obstacle field

[1] http://glew.sourceforge.net/
[2] http://www.glfw.org/



a) Program execution. b) Main loop.

Figure 3.2.2: a) The snow simulators program execution steps. b) The operations executed during one timestep or frame in the main loop.

Figure 3.2.3: Screenshot of the snow simulators menu systems available at runtime.

is increased by snow accumulating on the ground. Since snow growth is typically a fairly slow process the obstacle field does not need to be updated very often. The obstacle updating is therefore only performed every 1000 frames. The third stage is to run the wind simulation followed by the snow particle simulation in the fourth stage. In the last stage of the loop rendering is performed. The rendering system creates the new frame and also checks for user input.

Configuration and Menu Systems

Execution parameters of the simulator can be configured before application execution or during runtime. The parameters that can be adjusted include: The number of snowflakes to simulate, size of windfield, and rendering techniques. Execution parameters can be entered using three different methods. The first method is editing the configuration source file directly. The second method is to use a configuration text file called *config.txt* located in the applications *data* folder. And the last method is to use an on-screen menu system shown when starting the application. Adjusting parameters during runtime can be done using the menu system shown in Figure 3.2.3. The menu system is created using AntTweakBar[3] version 1.16.

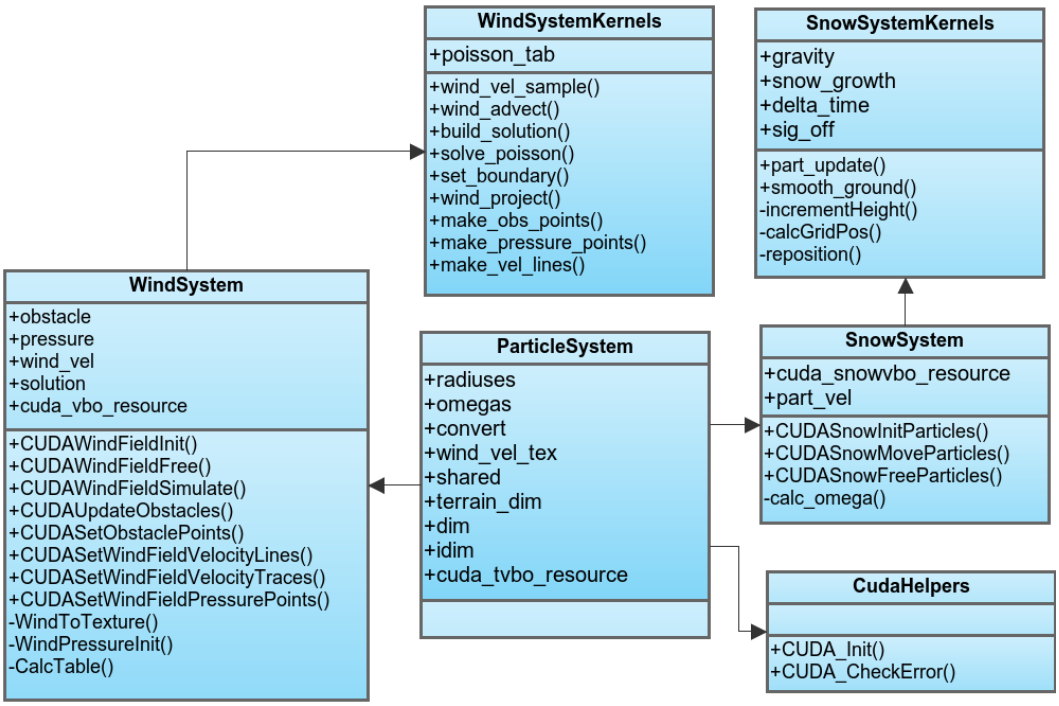


Figure 3.2.4: Class diagram for the snow simulators simulation system.

Particle System

The particle system is coded in C and CUDA and is logically divided into four files: *SnowSystem*, *SnowSystemKernels*, *WindSystem*, and *WindSystemKernels* (Figure 3.2.4). The *SnowSystem* and *WindSystem* files contains functions for allocating memory and running the simulation kernels located in the *SnowSystemKernels* and *WindSystemKernel* files. The code for these files are combined in the file *ParticleSystem* at compiletime, creating in essence one long file for the whole particle system. The reason behind this design is that shared global resources in CUDA are only available in filescope, and therefore can not be declared and shared using header files. Because of this limitation the organization of the simulator has not been changed much from previous versions except to update the CUDA API calls to CUDA 5.0, and cleaning up the code.

3.3 Snow Particle Simulation

As mentioned in the application description the simulation system is comprised of two parts. One part for simulating the motion of snowflakes (also referred to as snow particles) falling through the air, and another part for simulating the flow of air influencing

[3]http://anttweakbar.sourceforge.net

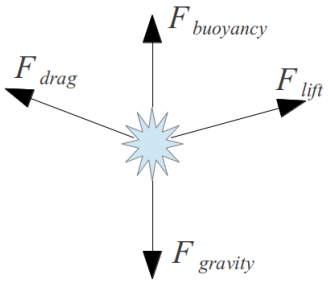


Figure 3.3.1: The forces working on a snowflake.

Symbol	Description	Initialization
P	Position (x,y,z)	Randomly within the scene boundaries
V_{snow}	Velocity	$([-1,1],[-1,1],V_{max,z})$
R	Radius of Circular Movement	(0,2)
ω	Angular velocity of circular movement	$[-\frac{\pi}{4},-\frac{\pi}{3}]$or $[\frac{\pi}{4},\frac{\pi}{3}]$
$V_{max,z}$	Vertical terminal velocity	[0.5, 1.5]

Table 3.3.1: Properties of a snowflake [19].

the snowflakes movements. This section gives a short description of how falling snow is modeled theoretically, and an overview of the snow particle simulation implementation.

3.3.1 Snow Modeling

The model for falling snowflakes used by Saltvik [31] is based on [30] and describes the forces acting on the snowflakes which determines their movement. Using this model combined with a particle system approach an algorithm can be developed for calculating a snowflakes position and velocity at incremental timesteps. The forces influencing a snowflake are shown in Figure 3.3.1.

$\mathbf{F}_{gravity}$ is a constant gravitational force for each snowflake. It is determined by a snowflakes mass m and the gravitational constant g. $\mathbf{F}_{bouyancy}$ is the force that is caused by density differences between objects and the surrounding fluid, and is ignored in the snow simulator because of its low value. $\mathbf{F}_{lift}$ is the force caused by vortices created when a snowflakes falls through the air, giving the snowflakes chaotic motion. And finally the $\mathbf{F}_{drag}$ is force that is the result from the differences in velocity between a snowflake and the surrounding air.

To calculate a realistic movements for the snowflakes they need to have some properties. Table 3.3.1 shows the properties of a snowflake as described by Eidissen [19] this set of properties is reduced compared to the properties outlined by Saltvik [31], mainly in the omission of the diameter and density of snowflakes. Some of these properties are

constant throughout the simulation, such as ω and the R. While the position P and start velocity V_{snow} are initialized at the start of the simulation and updated for every timestep. Saltvik's method for updating the state of a snowflake is shown in Listing 3.1. For a more detailed description of the snow modeling refer to [31, 19].

```
1 Initialize R, P^0, ω, V_max,z and F_gravity
2 while new timestep is needed do
3     Interpolate V^t_wind from windfield
4     V^t_circ = |V^t_fluid|/|V^t_sowflake| · ωR[−sinωt, cosωt, 0]
5     F_drag = ((V^t_fluid)^2 · m_snow · g)/(V^2_max,z) in the direction of V^t_fluid
6     a = (F_gravity + F_drag)/m_snow
7     P^{t+Δt} = P^2 + (V^t_snowflake + V^t_circ) · Δt + 1/2 · a · Δt^2
8     V^{t+Δt}_snowflake = a · Δt + V^t_snowflake
9 end while
```

Listing 3.1: Algorithm for simulating a snowflake motion [31].

3.3.2 Snow Accumulation and Ground Smoothing

To give the snow particle simulation a more realistic feel it allows for the buildup of snow on the terrain as it is hit by falling snowflakes. Snow buildup was first implemented by Saltvik [31] by adding overlaying triangle matrices representing the snow cover to planar rectangular surfaces, and later reimplemented by Eidissen [19].

In Eidissen's approach the geometry of the scene is represented by a terrain heightmap, which simplifies the accumulation of snow. Using a heightmap simplifies the collision detection between the snowflakes and the terrain to an lookup in the terrain vertex array, and enables snow buildup to be represented as a height value that are also stored within the terrain vertex array. It also allows for easier scaling of the terrain resolution, and simple manipulation of the distribution of snow height values. Adding height values directly to the terrain can also give the undesirable side effect of creating large differences in the snow height, displayed as spikes in the terrain. These spikes can be caused by a large snow growth factor or the wind causing a non-uniform snow distribution. Snow spikes can be prevented by smoothing the snow cover after adding it to the terrain. Representing the terrain as heightmaps has the additional benefit of enabling the use of real-world map data by the snow simulator, as shown by Lien [36].

3.3.3 Snow Particle Simulation Implementation

As shown in Figure 3.2.4 the simulation of snow particles consists of two kernels, `part_update` and `smooth_ground`. The `part_update` kernel is the main kernel for snow simulation and is responsible for updating the snow particles position at every timestep as described in section 3.3.1, and accumulating snow on the terrain. The `smooth_ground` kernel prevents large height differences being generated after snow is accumulated on the terrain, as described in the previous section. The smoothing kernel works using a

5-point stencil on the terrain heightmap, it decreases the height at the current location if the surrounding height values are lower, and increasing the height if they are higher.

As described in section 3.3.1 the state of the snow particle is given by its position, velocity, rotational radius, rotational velocity (ω), and its terminal velocity. During the initialization of the snow particle simulation the snow particles are given random starting positions that are copied into an OpenGL storage buffer called a Vertex Buffer Object (VBO) used for rendering the snowflakes.

The CUDA kernels uses the VBO directly to avoid unnecessary copy operations. This is done by employing the CUDA/OpenGL interoperability functionality, by registering the OpenGL resource in CUDA and mapping it through a pointer. This approach is used both for the VBO containing the snow particles positions and the VBO containing the terrain data. The registration and unregistration of the OpenGL resource is only done at application initialization and termination, while mapping and unmapping the resource to a pointer is done at every timestep before and after the CUDA kernel is called. The mapping operation has to be performed at every instance where the VBO is used, as OpenGL might alter the memory location of the VBO for various reasons.

Random values are also assigned to the initial particle velocity, the radius's, and the omegas. To limit the memory usage the rotational radius's and velocities are set to 32 values as it results in enough randomness for a pleasing visual result. All the data structures used by the snow simulation are located on the device for the duration of the applications lifetime.

The simulation of snow particles is simplified by ignoring the possibility of snowflakes colliding with each other, resulting in no dependencies between any of of the snowflakes reducing the simulation into an "*embarrassingly parallel problem*". Even though the simulation is simple it has a some performance limitations, mainly the reading of velocity from the windfield at the particles position. The windfield is located in a 3D texture that is cached based on spatial locality and supports hardware interpolation. However, since every particle has a random position coordinate, windfield lookup can result in cache misses and non-coalesced memory accesses if the whole windfield is not small enough to fit entirely into cache. Attempts have been made to improve this by sorting the snow particles by spatial locality, but has resulted in negligible performance gains as the cost of sorting equals the increased performance [22].

As mentioned in section 3.3.1 the particles are repositioned when they collide with objects or the terrain. The repositioning applies random x and z coordinates and a y coordinate at the top of the simulation volume. The current method for randomizing the particle repositioning values was developed by Eidissen [19] and employs the inherent noise in the floating point representation of particle coordinates. Eidissen explains that the method was developed using empirical testing and gives a uniform and pleasing result. The main benefit of this technique is that no additional memory storage such as seed values are required.

Execution of the `part_update` kernel is divided into the following 1D decomposition,

typically using a blocksize of 256:

```
dim3 block(PART_BLOCK, 1, 1);
dim3 grid(snow_count / block.x, 1, 1);
```

The `smooth_ground` kernel is executed in a 2D decomposition, with a typical block size of 16x16. Each threadblock has a width of a certain stride and copies the terrain data it is working on into shared memory to reduce reads from global memory.

```
dim3 block2(SG_DIM, SG_DIM, 1);
dim3 grid2(terrain_dim/SG_DIM, terrain_dim/SG_DIM, 1);
```

3.4 Wind Simulation

Instead of using a static windfield to influence the movement of the snowflakes in the simulation of snow particles, the windfield is simulated using a branch of fluid mechanics called Computational Fluid Dynamics . This section gives a introduction to Computational Fluid Dynamics, describes how CFD is used in the windfield simulation, and highlights some implementation details.

3.4.1 Computational Fluid Dynamics

A common way to describe wind would be as a flow of fluids. Fluid motion are often described using mathematical models, but as calculating such models can be complex it is often desirable to use methods for solving them numerically on computers by discretizing the problem domain and finding approximate solutions. Using these methods it is possible to realistically portray real-world phenomenon that involve fluid motion in interactive graphics applications such as games or in real-world simulators.

To simulate fluid using Computational Fluid Dynamics (CFD) [28] there are two approaches commonly used; The Navier-Stokes Equations (NSE) [26], and the Lattice Boltzmann Model (LBM) [27]. The NSE approach is based on using numerical methods and algorithms for finding solutions to a set of equations describing any single phase of the fluid flow. he LBM approach on the other hand uses the concept of lattice gas cellular automaton to view the fluid flow as particles moving along a discrete lattice. Below follows a brief description of the Navier-Stokes equations for incompressible flow.

The Navier-Stokes Equation for Incompressible Flow

The Navier-Stokes equations [26] (3.4.1) and (3.4.2) are physics equations that describes the motion of a fluid based on Newton's second law. For simplification it is often common to assume that the fluid in question also is an incompressible, homogeneous fluid. A fluid is incompressible if the volume of the fluid is constant over time, and it is homogeneous

if its density is constant in space. The Navier-Stokes equations consists of two equations; Equation (3.4.1) called the momentum equation describing the single-phase state of the fluid, and equation (3.4.2) is called the continuity equation that states the velocity field has to be divergence-free. Scalar quantities are shown in italic and vector quantities shown in bold.

$$\frac{\partial u}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + v\nabla^2\mathbf{u} + \mathbf{F} \tag{3.4.1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{3.4.2}$$

Where $\mathbf{u}$ is the velocity vector, ρ is the constant fluid density, p is the pressure, v is the kinematic viscosity, and $\mathbf{F} = (f_x, f_y)$ are any external forces that influence the fluid. Since $\mathbf{u}$ is a vector quantity equation (3.4.1) is comprised of two equations. To find a solution to the Navier-Stokes equations it must be solves for the quantities $\mathbf{u}$, v, and p. To simplify the equation it can be broken down into four different segments called advection, pressure, diffusion, and external forces.

Advection The first part of equation (3.4.1) is called the advection term. This states that the velocity in a fluid causes the fluid to transport quantities along with the flow, including the velocity itself. When the velocity is advecting itself it is called self-advection and is represented in equation (2.1) as $-(\mathbf{u} \cdot \nabla)\mathbf{u}$.

Pressure When force is applied to a fluid the molecules close to the force pushes on those farther away, and pressure builds up. This leads the acceleration of the fluid from high to low pressure areas, also leading to the velocity moving along the gradient of the pressure field. The second term $-\frac{1}{\rho}\nabla p$ of equation (3.4.1) represents this pressure acceleration.

Diffusion Some fluids are «thicker» then others because they have a higher kinematic viscosity. This viscosity is a measure of how resistant a fluid is to flow. The result from the resistance to flow is that the diffusion of momentum, and thereby of velocity. The third term $v\nabla^2\mathbf{u}$ of equation (3.4.1) represents this diffusion.

Forces The last term $\mathbf{F} = (f_x, f_y)$ of equation (3.4.1) is the acceleration that is due to forces acting on the fluid. This could be forces such as gravity, fan blowing air, or a water stream.

3.4.2 Solving the Navier-Stokes Equations

The windfield simulation used in the simulator developed by Saltvik [31] is based on the CFD model described in the previous section. The CFD approach to fluid simulation involves solving the Navier-Stokes equations (3.4.1,3.4.2). These equations can be simplified by assuming that the fluid has a viscosity of zero and a density of one, reducing

the Navier-Stokes equations to the Euler equations (3.4.3,3.4.4). In the snow simulator the force part $\boldsymbol{F}$ of the Navier-Stokes equation is also omitted since no external forces are modeled.

$$\frac{\partial u}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \nabla p \tag{3.4.3}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{3.4.4}$$

The Navier-Stokes equations provides the solution for the fluid state of a single phase or timestep, but when performing simulations we are interested in the fluids motion over time, meaning an incremental numerical solution is needed. A commonly used method for solving the Navier-Stokes equations incrementally and numerically is called the "*Stable fluids*" method described by Stam in 1999 [16]. Stam improved on the work by Foster et al. [29] making the method unconditionally stable, using a semi-Lagrangian scheme instead of a finite difference scheme when solving the advection part of the Navier-Stokes equations. The semi-Lagrangian scheme works by tracing fluid particles backwards in time to find its previous location, sampling the velocity in the previous location and using the sampled value when calculating the particles current velocity. In [32] Stam also showed how the Eulerian scheme can be used, which performs the calculation on fixed grid points instead of particles.

Stam's approach is aimed at simulating fluids in computer graphics and is therefore based on the fact that for many applications the fluid flow only has to look real, and not necessarily be completely physically accurate. Stam also optimized the approach in 2003 to enable real-time fluid dynamics to be used in computer games [17]. The approach originally described by Stam was designed to run on a regular CPU, and in 2004 an implementation of the approach on the GPU was described by Harris [15] using Shaders. Below is a brief description of the steps involved in the wind simulation as performed in the snow simulator based on the method developed by Stam.

Advection

The first step is the advection step. Advection is the process where velocity in a fluid causes the fluid to transport quantities along with the flow, including the velocity itself, called self-advection. Advection is done by tracing a particle that is passing a grid point back in time to find the velocity at the particles position. To trace the particles back in time Stam showed [17] that a Eulerian integration scheme can be used as shown in equation (3.4.5), which is also the scheme used in the snow simulator. It should be noted that an Eulerian integration scheme becomes unstable when using large timestep. This is seldom a problem since short timesteps are usually desirable.

$$\mathbf{u}^*(\mathbf{x}, t + \delta t) = \mathbf{u}(\mathbf{x} - \mathbf{u}(\mathbf{x}, t)\delta t, t) \tag{3.4.5}$$

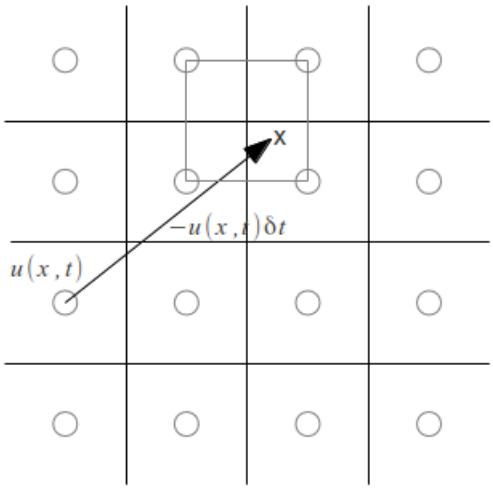


Figure 3.4.1: Tracing a particle crossing a grid point backwards in time to find its origin. This is the Eulerian integration scheme used in the self-advection step.

Figure 3.4.1 shows how this advection is performed when the velocity field $\mathbf{u}$ is self-advected. When the previous point is found it needs to be linear, bilinear or trilinear interpolated, depending on the dimensions of the grid, to find the correct corresponding grid values. After the velocity step is finished a temporary non-divergence free velocity field $\mathbf{u}^*$ is calculated. To make the velocity field divergence-free and satisfy the continuity equation (3.4.2) it needs to be made divergence-free in the Poisson and projection steps.

Solve Poisson

The next step is the Poisson solver step, which is part of the projection step. After the temporary velocity field $\mathbf{u}^*$ is created by the advection step it needs to be made into a divergence free velocity field $\mathbf{u}$ to satisfy the continuity equation (3.4.2). This can be done by using the Helmholtz-Hodge decomposition [28] which states that a vector field $\mathbf{w}$ can be decomposed into the sum of two other vector fields; a divergence-free vector field $\mathbf{u}$, and the gradient of a scalar field ∇p as shown in equation (3.4.6).

$$\mathbf{w} = \mathbf{u} + \nabla p \tag{3.4.6}$$

By rearranging the equation as shown in equation (3.4.7) and inserting the non-divergence free velocity field $\mathbf{u}^*$ for $\mathbf{w}$ we get that the divergence free velocity field $\mathbf{u}$ can be found by subtracting the gradient of the resulting pressure field p.

$$\mathbf{u} = \mathbf{u}^* - \nabla p \tag{3.4.7}$$

If we then apply the divergence operator ($\nabla\cdot$) (3.4.8) on both side of the equation we get the Poisson equation for pressure (3.4.9).

$$\nabla \cdot \mathbf{u} = \nabla \cdot (\mathbf{u}^* - \nabla p) = \nabla \cdot \mathbf{u}^* - \nabla^2 p \tag{3.4.8}$$

$$\nabla^2 p = \nabla \cdot \mathbf{u}^* \tag{3.4.9}$$

The Poisson equation (3.4.9) is a partial differential equation and can be solved numerically using finite difference. Thus we can represent the Poisson equation as a system of linear equations as shown in (3.4.10) where A is a sparse coefficient matrix, p is the pressure field, and b is the solution vector.

$$Ap = b \tag{3.4.10}$$

To solve this system of equations we first need to find the solution vector $b = \nabla \cdot \mathbf{u}^*$ by computing the divergence of the velocity field $\mathbf{u}^*$ as shown in equation (3.4.11) .

$$(\nabla \cdot \mathbf{u})_{i,j,k} = \frac{(x_{i+1,j,k} - x_{i-1,j,k}) + (y_{i,j+1,k} - y_{i,j-1,k}) + (z_{i,j,k-1} - z_{i,j,k-1})}{h} \tag{3.4.11}$$

When we have the solution vector b we can compute the Poisson equation. The linear system of equations representing the Poisson equation for pressure can be solved using a Poisson solver. In the snow simulator an iterative method for Poisson solving, called Successive Over-Relaxation (SOR) [40] is used. SOR is based on the Gauss-Seidel method and solves the Poisson equation by using an initial guess to the solution (usually zero) and incrementally improve on the guess by sampling all the neighboring approximations using a 7-point stencil (3D grid) for each iteration until convergence is reached. To reduce the amount of iterations needed to reach convergence the SOR method uses the new values in the grid as soon as they are computed. SOR also uses a relaxation factor ω between 0 and 2 to weight the new and old values sampled from the grid. The SOR equation is shown in equations (3.4.12 , 3.4.13).

$$p_{i,j,k}^{n+1} = (1-\omega)p_{i,j,k}^{n} + \omega p_{i,j,k}^{*} \tag{3.4.12}$$

$$p_{i,j,k}^{*} = \frac{p_{i+1,j,k}^{n} + p_{i,j+1,k}^{n} + p_{i,j,k+1}^{n} + p_{i-1,j,k}^{n+1} + p_{i,j-1,k}^{n+1} + p_{i,j,k-1}^{n+1} - h^2 b_{i,j,k}}{6} \tag{3.4.13}$$

Projection

The last step is the projection step where the pressure field is used to make the temporary velocity field $\mathbf{u}^*$ divergence-free and create the velocity field $\mathbf{u}$ representing the current timestep. The velocity field is made divergence-free by computing the pressure gradient and subtracting it from the non divergence-free velocity field. The pressure gradient can be computed as seen in equation (3.4.14).

$$(\nabla p)_{i,j,k} = \frac{1}{2h}(p_{i+1,j,k} - p_{i-1,j,k}, p_{i,j+1,k} - p_{i,j-1,k}, p_{i,j,k+1} - p_{i,j,k-1}) \tag{3.4.14}$$

Boundary Conditions

In Stam's approach [16, 14, 32] the boundary conditions are set to deny any flow from entering or exiting the simulation domain. This is done by setting the velocity component normal to the boundary surrounding the domain or objects contained within the volume to zero. To add a flow to the simulation an emitter is placed in the scene is often used (e.g. cigarette or fan). However, when simulating a natural phenomenon such as wind it is not possible to use an emitter since wind does not flow from any central point.

Instead the simulation uses *Dirichlet boundary conditions* for the wind velocity field where the values of the domain boundary are set to a specific value to give the simulation an initial wind velocity. This wind velocity at the boundary is sampled in the advection step and will propagate through the domain. The boundary conditions against objects contained within the domain are handled as described in Stam's approach.

For the pressure field the Von Neumann condition is used, which states that the pressure gradient is is zero across the boundaries. This is done by setting the pressure value on the boundary equal to the neighboring pressure values [31].

3.4.3 Wind Simulation Implementation

The windfield simulation is the largest part of the simulator containing five kernels used for simulation and three kernels used for visualization of the windfield. The simulation kernels uses the following data structures to calculate the windfield:

- **Wind velocity field** - A 3D array of float4 values representing the wind velocity vector field in the simulation volume.
- **Wind velocity field texture** - A 3D texture containing the resulting divergence-free velocity field at the end of each timestep. Its contents is updated as the last step of the wind simulation.
- **Pressure field** - A 3D array of floats containing scalar field with pressure values used in the projection step.

- **Solution vector** - A 3D array of floats representing the b-vector used in solving the Poisson equation.
- **Obstacle map** - A 3D array of integers covering the domain where the bits indicates if the current grid cell or the neighboring cells contains obstacles.

Wind Simulation Kernels

As shown in the class diagram in Figure 3.2.4 in section 3.2 the wind simulation contains the following kernel: `wind_advect`, `build_solution`, `solve_poisson`, `set_boundary`, and `wind_project`.

Each of these kernels are responsible for the steps outlined in section 3.4.2. The `wind_advect` kernel performs the self-advection step, but also adds an initial velocity to the boundary of the simulation volume, and sets the pressure field values to zero to ready it for the Poisson solving step. The `build_solution` kernel creates the b-vector that is the right-hand side of the Poisson equation. The `solve_poisson` kernel solves the Poisson equation using the SOR method and five iterations. Each iteration uses a different relaxation factor to prevent numerical instability in the pressure field [19]. In each iteration after the `solve_poisson` kernel is executed the `set_boundary` kernel is also used to preserve the boundary conditions of the pressure field. Lastly the `wind_project` kernel is responsible to create a divergence-free velocity field by subtracting the pressure gradient from the intermediate velocity field created by the `wind_advect` kernel. After all the steps in the simulation are finished the wind velocity vector field is copied to the wind velocity texture to be used in the snow particle simulation and when calculating the windfield for the next timestep.

Debug Kernels

For visualizing the windfield for debugging purposes the following kernels are used: `make_vel_lines`, `make_pressure_points`, and `make_obs_points`. Each of these kernels takes a OpenGL VBO as input and sets values in it from the corresponding data structures used in the simulation. The `make_vel_lines` kernel is used for visualizing the windfield velocity vectors by creating a line to represent the velocity direction, and a color for magnitude. The `make_pressure_points` kernel sets scalar pressure values that are colormapped to indicate low or high pressure. Low pressure is colored yellow, high pressure is colored red. The `make_obs_points` kernel visualizes the obstacle field created from the terrain.

Domain Decomposition

All the simulation kernels except `set_boundary` works by sampling voxel values (grid points) from 3D arrays in the neighboring x, y, and x directions. In memory these 3D

arrays are however stored in 1D in the (x,z,y) order to match the OpenGL coordinate system, which has to be reflected in the domain decomposition to get efficient memory accessing pattern.

To account for this the threadblocks are allocated to fill a single (x,z) plane in the domain, with rows of the x axis split along the threads and read into shared memory, and the z axis split over threadblocks and accessed concurrently by different SMs. The threadblocks then iterate vertically in the y direction from the bottom to the top of the volume, ensuring that the data furthest apart in memory is accessed least frequently. This decomposition also keeps preceding voxel values in registers, and ensures good exploitation of caches.

Update Obstacle Field

All the wind simulation kernels use a obstacle field representing object and geometry in the simulation scene as a mask to prevent performing needed calculation and to satisfy internal boundary conditions. The obstacle field is created by the function `CUDAUpdateObstacles` which runs on the CPU. This function reads the values from the terrain VBO and sets the first bit value to one in the corresponding obstacle field integer if it is part of the terrain, and to zero otherwise. Each obstacle field integer also stores bit information about its neighbors in the x, y, and z direction to reduce reads from memory when the obstacle field is used.

the obstacle field updating kernel need to run periodically to reflect snow accumulation on the terrain, but not often since snow accumulation is typically a slow process. In the previous snow simulator implementation the obstacle field updating kernel was split into steps performed each 0.1 seconds resulting in a new complete obstacle field every couple of seconds. This has been changed to updating the obstacle field every 1000 frames in this version of the snow simulator.

3.5 Simulation Flow Summary

The relationship and flow between the snow and wind simulation used in the particle simulation for each timestep can be summarized as shown in Figure 3.5.1. As shown the windfield is computed using the methods outlined in section 3.4 resulting in a windfield stored in texture memory. This windfield is sampled in the particle update kernel when calculating a new position for the snow particles, which also adds snow to the terrain. The newly added snow cover is then smoothed by the ground smoothing kernel before the updated terrain and the snowflakes at their new positions are rendered by the rendering system.

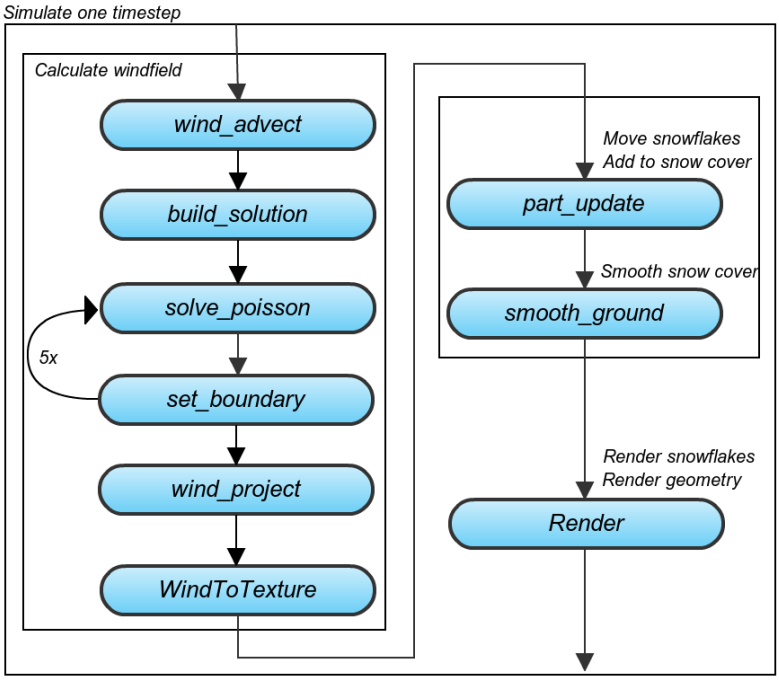


Figure 3.5.1: The simulation flow for a single timestep or frame.

Chapter 4

Porting The Snow Simulator To OpenACC

This chapter describes the process of porting the snow simulator to OpenACC and details about the final OpenACC implementation. The porting process is divided into three steps. The first step is to create a sequential CPU implementation of the snow simulator. The second step is to accelerate the sequential implementation using the GPU and OpenACC directives. And the third step is to optimize the OpenACC implementation by reducing the movement of data between the host and the device.

Since the OpenACC version of the snow simulator is based upon the CUDA version it has the same exact functionality. A description of the snow simulators theoretical basis and functionality is found in chapter 3.

4.1 Development Platform Setup

As shown in section 3.2 the snow simulator has a modular design where the simulation system is interchangeable from the rest of the application. The rest of the application is written in the C++11 programming language and is compiled separately using the GCC 4.6 compiler.

For the OpenACC compiler implementation it was initially intended to use both the open source accULL[1] compiler version 0.1 alpha and the commercial PGI[2] OpenACC compiler version 13.3. However, it was found that due to accULLs early development state it lacks the `deviceptr` data clause which is necessary for the OpenACC snow simulator to gain any reasonable performance. The commercial alternatives from Cray and CAPS were also considered, but ultimately the PGI compiler is chosen because it is widely used and well documented.

[1] http://accull.wordpress.com
[2] http://www.pgroup.com

The PGI compiler supports OpenACC version 1.0 and the programming languages Fortran, C and C++. It does however not yet support C++11 preventing using the PGI compiler to compile the whole application. Because of this limitation the PGI compiler is only used to compile the simulator system and during linking. The simulation system itself is written in the C programming language.

As with most of the OpenACC compiler implementations the PGI compiler does a source-to-source compilation of the code to a specified compiler target platform. The only target currently supported by PGI is CUDA, limiting its use to NVIDIA GPUs only. The CUDA version used by the PGI compiler is CUDA 5.0. The OpenACC implementation of the snow simulator also includes some CUDA code that is compiled using the CUDA NVCC compiler directly.

The snow simulator application is developed on the Ubuntu 64-bit 12.04 Linux operating system. The compilers and tools used in the implementation are however available on the Microsoft Windows and Apple OS X operating systems also.

4.2 Sequential Implementation

The first step for porting the snow simulator to OpenACC is to create a sequential CPU implementation from the CUDA version described in chapter 3. This can be viewed as a bit of a backwards process compared to how OpenACC's intended use, where typically there is already a sequential CPU code that we want accelerated by running it in parallel on a GPU. However, the only CPU implementation of the snow simulator, not counting those written using OpenCL, is the version originally created by Saltvik [31] from 2006. In the 7 years since Saltvik's version the snow simulator has undergone many alterations in the subsequent GPU versions using CUDA and OpenCL, as described in section 3.1, making it necessary to create a new sequential CPU implementation to get the best basis for comparison. A case can also be made for this process if the goals is to increase the portability and readability of a current low-level CUDA or OpenCL implementation.The sequential version was made by forking the reworked CUDA version from its version control system, enabling the sequential version to be synchronized with the CUDA version allowing easy inclusion of any new features added. The simulation system in the sequential version is written in C99 and compiled using GCC 4.6.

4.2.1 Porting the Wind Simulation

The wind simulation system is the largest and most complicated part of the simulation system. Unlike the snow particle simulation which is dependent on the wind simulation for generating a windfield to sample from when updating particle positions, the wind simulation can be viewed as an independent system. The wind simulation is therefore a good starting point for the porting process. Kernels in the wind simulation were ported

```
/* Run wind simulation */
float_4 *tmp = wind_vel0; wind_vel0 = wind_vel; wind_vel = tmp;
wind_advect(wind_vel, wind_vel0, pressure, pressure0,
                            obstacle, deltatime, dim, vec);

build_solution(wind_vel, solution, obstacle, dim,
                            0.5f / (h * deltatime));

float relax[] = { 1.7f, 1.5f, 1.2f, 1.1f, 1.1f };
for (int i = 0; i < 5; i++) {
    float *tmp = pressure0; pressure0 = pressure; pressure = tmp;
    solve_poisson(pressure, pressure0, solution,
                    obstacle, poisson_tab, dim, relax[i]);
    set_boundary(pressure, obstacle, dim, mask);
}
wind_project(wind_vel, pressure, dim, 2.0f*0.5f * deltatime/h);
```

Listing 4.1: Code showing the wind simulation kernel calls in the CPU and OpenACC snow simulator versions. These are the kernels called in a single timestep.

one at a time and the debug kernels were used to visualize the windfield to confirm correctness by comparing it to the visual output from the CUDA version.

The first part of the wind simulation code ported was the obstacle creation since this code was already being executed on the CPU, and also because most of the other wind simulation kernels use the obstacle field in their calculations. Following that the wind advection, build solution, and projection kernels were ported, and finally the SOR Poisson solver and the set boundary kernels. All CUDA specific code was removed such as the explicit use of shared memory in the kernels and the use of texture memory for storing the windfield.

With the removal of the windfield 3D texture an additional windfield 3D array is introduced to store the previous windfield state used in the wind advection kernel. To remove any copy operations the pointers to the previous and the current windfield arrays are swapped before the wind advection kernel is called, as shown in Listing 4.1. An additional 3D array for the pressure field is also used in the Poisson solver and pointers are swapped for every iteration. A separate storage is not really needed in the SOR Poisson solver when running sequentially, but is implemented to avoid race conditions when using shared memory parallelism.

Memory Accessing

All the 3D vector and scalar fields used in the wind simulation are stored 1D in memory. Therefore care has to be taken to make sure that elements are accessed based on their memory locality. The loop nests in the kernels reflect this memory locality by iterating over the y-axis elements in the outer loop, the z-axis in the middle loop, and x-axis in the inner loop. Which means that the elements along the x-axis are stored closest in

memory and y elements farthest. The memory elements are accessed using the following macro to calculate the array index:

```
#define I(X,Y,Z) ((X) + (Z)*dim.x + (Y)*dim.z*dim.x)
```

When relating to the OpenGL right-handed coordinate system, kernels work on the voxels in (x,z) planes from the bottom to the top of the volume following the positive y-direction. All data structures used in the wind simulation are also given a halo padded around them to simplify border management when performing stencil operations in the kernels.

Interpolation

As explained in section 3.4.2 the wind velocity at a current voxel in the simulation grid is computed by self-advecting the velocity backwards in time. The previous velocity value is then found by using trilinear interpolation at the backtraced location. In the CUDA version this is supplied by free with the use of a 3D texture memory on the GPU device, with integrated hardware units to perform the interpolation. On the CPU however the interpolation has to be performed in software resulting in a significantly higher cost. This cost is also coupled with the fact that the system memory is only cached using 1D spatial locality compared to texture memory which supports both 2D and 3D caching. This cost is also highlighted by Eidissen [19] as a performance bottleneck in Saltvik's [31] original implementation.

The trilinear interpolation is performed as shown in Listing 4.2 where the coordinates are first clamped to the edge of volume then interpolated in the x, y and z directions, resulting in a total of 24 memory reads.

4.2.2 Porting the Snow Particle Simulation

The snow particle simulation only contains two kernels. One for updating the snow particle movement and one for smoothing the newly added snow covering the terrain. The snow particle simulation contains less CUDA specific code and was therefore much simpler to port then the wind simulation.

In the particle update kernel each particles state is stored in 1D arrays and therefore only a single loop is needed for updating the particles. As explained in section 3.3.1 the particle movement is dependent on the windfield generated by the wind simulation. For each particle the windfield is sampled at least once per timestep. The first sampling is performed when calculating the drag affecting the particle. Subsequent windfield samplings can also occur if the snow particle needs to be repositioned when colliding with the ground or if it moves outside the bounds of the simulation volume. This can result in a maximum of two windfield lookups per particle in each timestep where software trilinear interpolation (Listing 4.2) is needed. Because of the particles random position and that the CPU only uses 1D caching, windfield lookups can often result in

```
// Clamp to edge of volume
if(x < 0.5f) x = 0.5f;
if(x > dim.x+0.5f) x = dim.x+0.5f;
if(y < 0.5f) y = 0.5f;
if(y > dim.y+0.5f) y = dim.y+0.5f;
if(z < 0.5f) z = 0.5f;
if(z > dim.z+0.5f) z = dim.z+0.5f;

int i0 = (int)x; int i1 = i0+1;
int j0 = (int)y; int j1 = j0+1;
int k0 = (int)z; int k1 = k0+1;

float s1 = x-(float)i0; float s0 = 1.0f-s1;
float t1 = y-(float)j0; float t0 = 1.0f-t1;
float u1 = z-(float)k0; float u0 = 1.0f-u1;

float_4 v;

// interpolate x
v.x = s0*( t0 * (u0 * vel[I(i0,j0,k0)].x + u1 * vel[I(i0,j0,k1)].x) +
         ( t1 * (u0 * vel[I(i0,j1,k0)].x + u1 * vel[I(i0,j1,k1)].x))) +
      s1*( t0 * (u0 * vel[I(i1,j0,k0)].x + u1 * vel[I(i1,j0,k1)].x) +
         ( t1 * (u0 * vel[I(i1,j1,k0)].x + u1 * vel[I(i1,j1,k1)].x)));

// interpolate y
v.y = s0*( t0 * (u0 * vel[I(i0,j0,k0)].y + u1 * vel[I(i0,j0,k1)].y) +
         ( t1 * (u0 * vel[I(i0,j1,k0)].y + u1 * vel[I(i0,j1,k1)].y))) +
      s1*( t0 * (u0 * vel[I(i1,j0,k0)].y + u1 * vel[I(i1,j0,k1)].y) +
         ( t1 * (u0 * vel[I(i1,j1,k0)].y + u1 * vel[I(i1,j1,k1)].y)));

// interpolate z
v.z = s0*( t0 * (u0 * vel[I(i0,j0,k0)].z + u1 * vel[I(i0,j0,k1)].z) +
         ( t1 * (u0 * vel[I(i0,j1,k0)].z + u1 * vel[I(i0,j1,k1)].z))) +
      s1*( t0 * (u0 * vel[I(i1,j0,k0)].z + u1 * vel[I(i1,j0,k1)].z) +
         ( t1 * (u0 * vel[I(i1,j1,k0)].z + u1 * vel[I(i1,j1,k1)].z)));
```

Listing 4.2: The trilinear interpolation code used in wind advection and particle updating kernels for windfield lookup in the CPU and OpenACC snow simulator versions.

cache misses and accesses to system memory, making the particle update step a major bottleneck when the number of particles increases. The ground smoothing kernel was ported by removing the shared memory usage and explicitly manage borders since there is no padding on the terrain VBO.

4.2.3 Parallelization using OpenMP

In recent years multicored CPUs have become common. Therefore it is not very realistic to measure performance using only a single CPU core when comparing against other parallel implementations. To utilize multiple cores the sequential snow simulator is accelerated using shared memory parallelism with the OpenMP standard for multithreading. This gives a performance baseline when testing the CUDA and OpenACC versions. The OpenMP implementation used is OpenMP 3.0 supplied with the GCC compiler.

Parallelization is performed by adding OpenMP pragmas to all the kernels as shown in Listing 4.3 using default thread scheduling and specifying which data structures are shared or private. In total twelve OpenMP pragmas are used in the wind simulation and four in the snow particle simulation. The number of threads is configured by adjusting the environment variable `NUM_OMP_THREADS`.

The use of shared memory parallelism does however demand some implementation changes to the SOR Poisson solver in the wind simulation. As described in section 3.4.2 the SOR solver uses the same pressure field to both read from and write to in the same iteration. Using shared memory this can lead to potential memory incoherence. To avoid this a temporary pressure field is used to store the previous state. Such memory incoherence can also occur with the ground smoothing kernel, and can be solved by making a copy of the terrain VBO to read from or to implement a border exchange scheme. None of these solutions are implemented since they would lead to diminished performance and the effects are not noticeable during visualization.

4.3 Directive-Based GPU Implementation using OpenACC

The OpenACC implementation is based directly on the sequential CPU implementation outlined in section 4.2 by off-loading computational-heavy loop nests onto the GPU with the use of OpenACC directives (Section 2.2.3). The directives are applied with the goal of making as few alterations as possible to the structure and flow of the existing code to maintain the codes cohesion and modularity, while still achieving a high performance gain.

The porting process to OpenACC was performed in the same manner as the sequential port by converting one kernel at a time. Starting with the windfield simulation, before

```
/* Wind advection kernel */
void wind_advect(float_4 *d, float_4 *d0, float *p, float *p0,
                 int *obs, float dt, dim_3 dim, float_4 boundary) {
    #pragma omp parallel for default(none) \
        shared(d,d0,p,p0,obs,dt,dim,boundary)
    for(int y = 0; y < dim.y; ++y) {
        for(int z = 0; z < dim.z; ++z) {
            for(int x = 0; x < dim.x; ++x) {
                /* code omitted */
            }
        }
    }
}
```

Listing 4.3: Code showing the wind advection accelerated on the CPU by using OpenMP.

moving onto the snow particle simulation. For correctness verification the wind simulation debug kernels were used. The code for the OpenACC kernels and simulation initialization can be viewed in Appendix A.

As described in section 2.2.3 parallel regions are defined by the `kernels` or `parallel` directives combined with data clauses to signify the movement of data in and out of the parallel region. The first option explored was therefore simply to add the `kernels` directive at the top of every loop nest, accompanied by the `copyin` and `copyout` data clauses as shown in Listing 4.4. To parallelize each of the loops in the loop nest the `loop` directive is used, combined with the `independent` clause to indicated that there are no data dependencies between the loops. This solution did however decrease the performance to below that of the sequential implementation.

The reason for the performance decrease was found to be the increase of time spent transferring data between the host and device compared to actual kernel execution. To reduce the data movement two possible solutions were considered. The first solution was to create a containing data region using the `data` directive (Section 2.2.3) and combine the `kernels` directives with the `present` data clause indicating the data already is present in device memory. This would allow for data to be initialized on the host, transferred to device memory at the start of the simulation execution, and retained in device memory for the applications lifetime. Using this solution would however require changing the applications structure since the data region would have to contain the simulators main loop (Section 3.2). Not only would this diminish the modularity and cohesion of the application, but it would also become problematic to compile since the PGI compiler currently does not support C++11 used in the rest of the snow simulator application.

The second alternative considered, and ultimately chosen, is to use the OpenACC library routines (Section 2.2.3) to allocate device memory during the simulation initialization phase. For the memory allocation the routine `acc_malloc()` is used along with the routine `acc_free()` for deallocation on application termination. To indicate that the data

```
/* Solve Poisson using SOR */
void solve_poisson(float *p, float *p0, float *b, int *obs,
                        float *poisson_tab, dim_3 dim, float w) {
#pragma acc kernels copyout(p[0:size]) \
    copyin(p0[0:size],obs[0:size],poisson_tab[0:size],dim,w)
{
    #pragma acc loop independent
    for(int y = 1; y < dim.y-1; ++y) {
        #pragma acc loop independent
        for(int z = 1; z < dim.z-1; ++z) {
            #pragma acc loop independent
            for(int x = 1; x < dim.x-1; ++x) {
                /* code omitted */
            }
        }
    }
}
}
```

Listing 4.4: Code showing a the SOR Poisson solver using copy data clauses.

structures passed to the kernels are allocated in device memory, the `kernels` directive is combined with the `deviceptr` data clause as shown in Listing 4.5. This method gives the benefit of conforming well with the simulation systems current structure, although it also has the drawback of breaking portability with the CPU code. A workaround for this is to use conditional compiling with the preprocessor macro `_OPENACC` to indicate which sections of the code should be compiled when using the OpenACC compiler flag. Conditional compiling is however not implemented here as it would introduce more code to the OpenACC implementation.

The final particle system ported to OpenACC is a bit different then the CUDA version as shown in Figure 4.3.1. The main differences are that all data structures and variables are moved to the common file *ParticleSystem* and a wrapper interface called *ParticleSystemWrapper* is added. As with the CUDA version all the code is added to a single file in the preprocessing stage in the compilation to comply with the CUDA filescope restriction for global variables.

4.3.1 Particle System Wrapper

As with the CUDA version of the simulator (Section 3.3.3 and 3.4.3), some of the simulation kernels either needs to read from or write to the snow, wind, and terrain OpenGL VBOs used for rendering. These can be accessed in CUDA by using the CUDA/OpenGL interoperability library. OpenACC does not currently supply such support in its library routines so other options needed to be explored.

The first option explored was the use of the OpenGL API directly. The OpenGL API provides the possibility to access VBOs by binding and mapping the VBOs to pointers.

```
/* Solve Poisson using SOR */
void solve_poisson(float *p, float *p0, float *b, int *obs,
                   float *poisson_tab, dim_3 dim, float w) {
#pragma acc kernels deviceptr(p,p0,obs,b,poisson_tab) copyin(w)
{
    #pragma acc loop independent
    for(int y = 1; y < dim.y-1; ++y) {
        #pragma acc loop independent
        for(int z = 1; z < dim.z-1; ++z) {
            #pragma acc loop independent
            for(int x = 1; x < dim.x-1; ++x) {
                /* code omitted */
            }
        }
    }
}
}
```

Listing 4.5: Code showing the SOR Poisson solver using deviceptr clause.

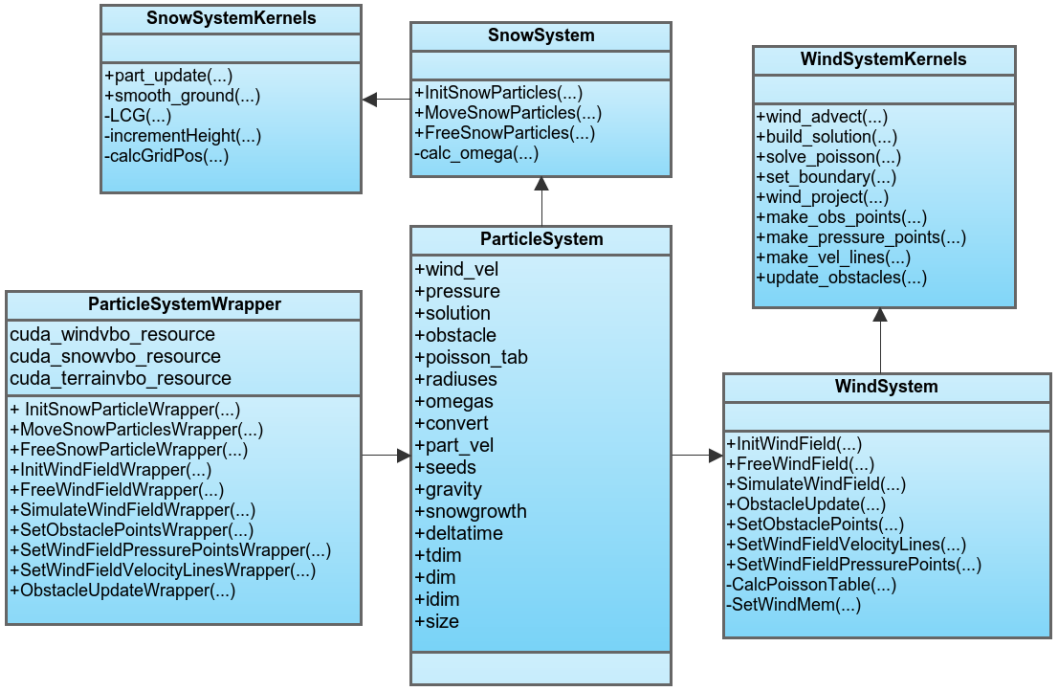


Figure 4.3.1: Class diagram of the simulation system in the OpenACC simulator version.

These pointers give access to the GPU device memory and can be used directly from host code with the OpenGL runtime system abstracting the copy operations involved. Since these pointers essentially are GPU device pointers they can also, in theory, be used in OpenACC parallel regions with the `deviceptr` data clause for direct on-device memory access. This was also the first solution attempted and worked with 12.8 version of the PGI compiler. However, due to implementation changes this solution stopped working when upgrading to version 13.3. After consulting with the the PGI support staff the reason for this was found to be the change to pinned or page-locked memory access introduced in the 13.x series of the compiler to increase performance of asynchronous data copies. Although support for changing between pinned and non-pinned memory access might be added in the future a more implementation independent solution was desirable.

The solution selected was to include the CUDA method for mapping VBOs in the OpenACC implementation. The CUDA code is contained in wrapper interface functions in the file *ParticleSystemWrapper* shown in 4.3.1. The wrapper functions use the CUDA/OpenGL interop library to register and map VBOs to CUDA device pointers, which are passed to the OpenACC parallel regions using the `deviceptr` data clause. The drawback of using CUDA code for this mapping is that the code contained in *ParticleSystemWrapper* has to be compiled using the NVCC compiler as the PGI compiler currently cannot compile both OpenACC and CUDA code. This introduces yet another compiler environment to the implementation and also contributes to breaking the portability with the CPU code.

4.3.2 OpenACC Windfield Simulation

The OpenACC implementation of the wind simulation is structured the same as the sequential CPU implementation, but uses the OpenACC library routine `acc_malloc()` to allocate data structures to device memory during initialization. Since there are no functions similar to `memset` or `memcpy` included in OpenACC v1.0 (Section 2.2.3), the initial values in the data structures has to be set manually in OpenACC kernels. These kernels are only called once in the initialization phase, and therefore do not affect the simulators performance. During application termination the device allocated data structures are deallocated using the `acc_free()` library routine. During initialization of the windfield the library routines `acc_set_device_num()` and `acc_init()` are also used. `acc_set_device_num()` specifies witch accelerator device to use and `acc_init()` powers up the accelerator device so there are no initialization costs when kernels are called.

Since OpenACC is a device independent standard it does not include support for device dependent hardware components such as texture memory in its memory model (Section 2.2.3) . This gives the OpenACC version the same limitation as the sequential version by needing to perform trilinear interpolation in software when performing windfield lookups (Section 4.2). The OpenACC version therefore has the same memory setup as the sequential version by using a temporary storage for the windfields previous state.

Similar to the sequential version the device pointers used in the OpenACC version can also be swapped (Listing 4.1), removing any need to perform copy operations.

The fact that OpenACC is a standard also means that shared memory cannot be explicitly allocated and used for sharing data between threads in threadblocks, as frequently done in the CUDA version of the snow simulator. This gives the OpenACC version a significant performance limitation compared to the CUDA version as more reads has to be made directly from global memory. Shared memory can however be used explicitly as cache in OpenACC by using the `cache` directive. This was attempted, but no discernible performance gain was observed, and it is unclear if the cache directive was used at all since the compiler gave no feedback about it. Similarly to the sequential version using OpenMP (Section 4.2.3) the OpenACC versions SOR Poisson solver kernel is susceptible to memory incoherence. Therefore the same solution is implemented by using a temporary pressure field to store the previous state.

Using preallocated device pointers, CUDA mapped VBO pointers, and pointer swapping, all large data structures used in the wind simulation are contained fully on the device without the need to perform data copying. Small constants such as grid dimension size and relaxation factors are however copied into the parallel regions every kernel execution. These transfers are so small that they do not affect the simulation performance.

Domain Decomposition

The OpenACC wind simulation uses the same memory accessing scheme as the sequential version (Section 4.2) with a triple-nested loop where the inner loop iterates over elements located sequentially in memory. As shown in Listing 4.5 each of the loops use the `loop` directive to indicate it should be executed in parallel.

Parallel decomposition in OpenACC is determined in two possible ways. Either the compiler automatically selects a decomposition or it can be determined by using the `gang`, `worker`, and `vector` clauses on the `loop` directive, where the `worker` clause is set to the warp size on NVIDIA devices. Generally it is recommended to let the compiler determine the decomposition since it usually chooses a good solution based on heuristics. Manually specifying the decomposition can also lead to degraded performance when executing the code on different accelerator devices.

When letting the compiler choose the decomposition it selects to split the z-axis across gangs and the x-axis across gangs and vector with a vector size of 128 while looping over the y-axis. Which decomposition the compiler chooses can be viewed by compiling with the `Minfo` flag, giving the output shown in Listing 4.6. The gang and vector dimensions used during execution can also be viewed by enabling debugging mode setting the environment variable `PGI_ACC_TIME` to 1, giving the output shown in Listing 4.7.

Manually setting the gang and vector configuration was also attempted but no configuration was found that gave any significant performance gain over the configuration selected by the compiler. Selecting a good decomposition configuration can be difficult and it is

```
solve_poisson:
    107, Generating present_or_copyin(w)
         Generating compute capability 2.0 binary
    110, Loop is parallelizable
    112, Loop is parallelizable
    114, Loop is parallelizable
         Accelerator kernel generated
         112, #pragma acc loop gang /* blockIdx.y */
         114, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x
              */
         CC 2.0 : 42 registers; 0 shared, 108 constant, 0 local memory
             bytes
```

Listing 4.6: PGI compiler output shown by using Minfo compiler flag. The output shows how the kernel is parallelized by the compiler.

```
solve_poisson
    107: region entered 2180 times
         time(us): total=430,034 init=133 region=429,901
             kernels=273,166 data=24,805
         w/o init: total=429,901 max=554 min=159 avg=197
         114: kernel launched 2180 times
             grid: [1x128]  block: [128]
             time(us): total=273,166 max=286 min=121 avg=125
```

Listing 4.7: PGI compiler timing output. Profiling information showing kernel runtime statistics such as kernel execution number and time distribution.

hard to say if the best configuration has been found. Work has however been done to create frameworks that auto-tunes the mapping configuration for a given program and input sizes that result in the upwards to 4.8x performance gain over the configuration chosen by the compiler [39].

Update Obstacle Field

Like in the CUDA version of the snow simulator the updating the obstacle field is performed on the host and the results are copied over to device memory. Attempts were made to move the obstacle creation kernel over to the GPU with the use of OpenACC directives, but a conditional loop in the code prevented it from being parallelizable. Since updating the obstacle field is seldom performed (every 1000 frames) it has no significant impact on performance and therefore priority was not given to find a working GPU solution. The copy operation is performed manually in a parallel region using the `kernels` directive and the `deviceptr` data clause.

4.3.3 OpenACC Snow Particle Simulation

Similarly to the wind simulation the snow particle simulation also allocates device memory by using the `acc_malloc()` routine and frees it by using `acc_free`(). The data is also initialized using parallel regions executed only once.

The snow particle simulation contains only two kernels and were therefore much simpler to convert to OpenACC. Still some considerations had to be made. The snow particle kernel responsible for updating the particles position performs calls to small functions when converting from particle coordinates to terrain coordinates and also when adding snow to the terrain. These functions have to be declared inlined since the PGI compiler does not currently support calling functions from within a parallel region. However, support for external function calls are planned to be implemented in a future release. The trilinear interpolation (Listing 4.2) code was also attempted separated to its own inlined function, but resulted in incorrect values being produced. The trilinear interpolation code is therefore included in particle update kernel increasing the length of the code.

As with the sequential version, the OpenACC version also suffers from the need to perform software interpolation when sampling velocity from the windfield. This penalty is made worse by the random nature of the lookups preventing coalesced memory reads and the potentially high number of cache misses from the 1D cache limitation.

Domain Decomposition

The particle update kernel uses only a single loop to update the particles and consequently the compiler chooses a 1D decomposition configuration where the loop is split along gangs and vectors with a vector size of 128. The smooth ground kernel uses a 2D

loop nest to loop in the y- and x-directions of the terrain. When allowing the compiler to select the configuration the y-axis is split along gangs, and the x-axis is split along gangs and vectors with a vector size of 128.

Particle Repositioning

When a snow particle either hits an obstacles or the terrain it is repositioned at a random x and z coordinate at the top of the simulation volume. In the CUDA version this is performed by using a repositioning function developed by Eidissen as explained in section 3.3.3. When attempting to implement this method in the OpenACC version it resulted in an invalid floating point operation making the snow particles disappear from the scene. The reason behind this was discovered to be related to how the compiler translated the code to CUDA, but a working fix was not achieved. Problems using this function was also encountered by Vestre [22] who reached the following conclusion "*No fully working replacement was found, but a half way working replacement was made using float, int-casts and modulo operations*" [22] (page 37).

Instead of using the solution by Vestre a pseudo random number generator called a Linear Congruential Generator (LCG) [38] is implemented. The LCG is among the oldest and most common random number generators and gives a good enough distribution for the snow simulator since high-quality randomness is not critical. The implementation of the LCG is shown in Listing 4.8. Each particle is given its own initial random floating point seed number generated with a system calls during the snow particle initialization. The seed number is used in the LCG function when creating new coordinates when repositioning resulting in a new seed number for the next repositioning as shown in Listing 4.9.

```
inline int LCG(int seed) {
    seed = (1103515245 * seed + 12345) % 2147483648;
    return seed;
}
```

Listing 4.8: Linear Congruential Generator used for particle repositioning.

```
int s = abs(LCG(seeds[i]));
pos.x = (float)s*(1.0f/RANDMAX)*SCENE_X;
s = abs(LCG(s));
pos.z = (float)s*(1.0f/RANDMAX)*SCENE_Z;
seeds[i] = s;
pos.y = SCENE_Y-2.0f;
```

Listing 4.9: The particle repositioning code used in the particle update kernel. .

The drawback of using this method, apart for its low-quality randomness, is the extra memory required to store seed values. For every snow particle an extra float has to be allocated, resulting in about 7.7 MB of extra data when using a snow particle count of 2 million. The benefits from this method is its speed and that it should be portable across devices and compilers.

Chapter 5

Results and Discussion

In this chapter we evaluate the OpenACC implementation of the snow simulator by comparing it with the current CUDA version and the sequential CPU version using OpenMP. The evaluation is performed by reviewing the program profiling information, and running benchmarks of the snow simulator using different configurations of snow particles and windfield dimension. At the end of the chapter the visual results are also presented along with a discussion of the experiences using OpenACC in this project.

5.1 Test Platform and Setup

All tests are performed using hardware and software available at the HPC-Lab at NTNU as shown in Table 5.1.1. As a baseline for the performance evaluation a OpenMP version of the snow simulator is tested on a Intel i7 CPU running with a thread for each of its logical cores resulting in a total of 8 threads. Since the PGI compiler only currently supports CUDA target code, three graphics card with NVIDIA GPUs are used when testing the OpenACC and CUDA versions of the snow simulator. These cards consist of a consumer oriented Geforce GTX 480, and the two Tesla cards C2070 and K20c, aimed at the HPC market.

The GTX 480 was also used during development, while the two Tesla cards were only used for testing. Since the GTX 480 was used for development, performance is tuned against its Fermi GPU architecture. The C2070 is also based on the Fermi architecture, while the K20c is based on the newer Kepler architecture. Results gathered using the K20c might therefore not necessarily reflect its true performance as the implementations are not design with its architecture in mind. During testing it was discovered that version 12.8 of the PGI compiler gave better performance then the newer 13.3 version. Tests using the GTX 480 and Tesla C2070 are therefore performed using the 12.8 versions, while the tests on the Tesla K20c are performed on version 13.3 since the 12.8 version does not support the Kepler architecture.

Test Computer	
CPU	Intel Core i7-930
CPU cores	8 logical
CPU clockspeed	2.8 Ghz
CPU cache size	8 MB
System memory	16 GB

Software	
Operating system	Ubuntu 12.04 64-bit
NVIDIA driver	310.32
CUDA toolkit	5.0
PGI compiler	12.8 and 13.3
GCC compiler	4.6

Graphic Cards						
Card name	Memory	SMs	Cores	SM Clock rate	Architecture	CC
NVIDIA GTX 480	1.5 GB	15	480	1401 MHz	Fermi	2.0
NVIDIA Tesla C2070	6 GB	14	448	1147 MHz	Fermi	2.0
NVIDIA Tesla K20c	5 GB	13	2496	705 MHz	Kepler	3.5

Table 5.1.1: The test computer, graphic cards, and software specifications.

Both the Tesla C2070 and the GTX 480 cards have video output and are used as both the compute and rendering device during testing. The K20c on the other hand is a pure compute card and is used in conjunction with the GTX 480 to provide display output. This can put some limitation to performance to the K20c when testing the snow particle simulation as data has to be transferred over the PCI-E bus between the two devices. Both Tesla cards had the ECC memory setting turned off during testing.

The tests are performed by measuring the frames per seconds (FPS) of the program during a timeframe of 15 seconds for each configuration, without including the time needed for initialization of the program and the simulation. For the test configurations that showed variation in the results, multiple runs were performed and the average taken. The test configurations chosen are similar to those used by Eidissen [19]. The snow particle simulation is tested by scaling the snow particle count without wind simulation and with wind simulation using a balanced windfield. The wind simulation is tested by scaling the windfields voxel count without snow particle simulation and with a balanced snow particle count. Both the snow particle count and the windfield voxel count are scaled by the power of two for best conformity with the CUDA hardware architecture (warp size of 32). The snow particle count varies in the range from about 1M to 4.7M, and the windfield voxel count ranges from 130K to 8.3M.

5.2 Profiling Analysis

Profiling of both the OpenACC and CUDA version of the simulation is performed to view the kernel time distribution, kernel occupancy, and register usage. Profiling is carried out by using the NVIDIA Visual Profiler using the Mount St. Helens terrain with a resolution of 768x768, 2097152 (2048*1024) snow particles, and 2097152 (256x32x256) voxels in the windfield.

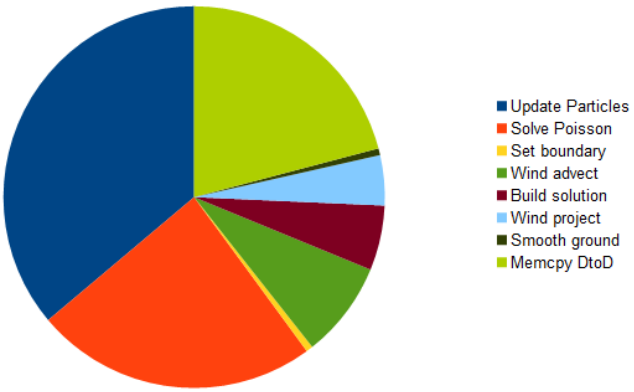


Figure 5.2.1: Kernel time distribution for the CUDA snow simulator.

The CUDA versions time distribution can be viewed in Figure 5.2.1 where we can see that the particle update kernel is by far the most time-consuming, followed by the Poisson solver. This is not surprising since the particle update kernel must perform a large amount of lookups in both the windfield texture and terrain VBO. Also interesting is that the third most time-consuming operation is a device-to-device (DtoD) memory copy operation. The memory copy is performed after each wind simulation timestep to update the windfield texture. We can also see that the set boundary and smooth ground kernel use a minimal amount of time compared to the other kernels.

Figure 5.2.2 shows the time distribution of the OpenACC version and right away we can see a significant difference from the CUDA version. In the OpenACC version over 80% of the time is spent on the particle update kernel, which reflects the added penalty the OpenACC version suffers from software interpolation and 1D caching when sampling from the windfield. Among the wind simulation kernels the wind advection kernel surprisingly uses more time then the Poisson solver. This can be explained by the wind advection kernel is also performing software interpolation making it slower then the CUDA version. It is also noteworthy that the kernel responsible for setting the pressure boundaries uses about the same time as the Poisson solver kernel, which should not be the case. This might indicate that there are some inefficiencies within the implementation of this kernel and it should be altered. The OpenACC version also only uses a negligible amount of time performing copy operation since it only transfers small constant values.

The occupancy and register usage for both the CUDA and the OpenACC versions are shown in Table 5.2.1. The CUDA version uses fewer registers then the OpenACC version because of its explicit allocation of shared memory. The high number of registers used by the OpenACC version also lowers its occupancy percentage as most of the kernels become register constrained.

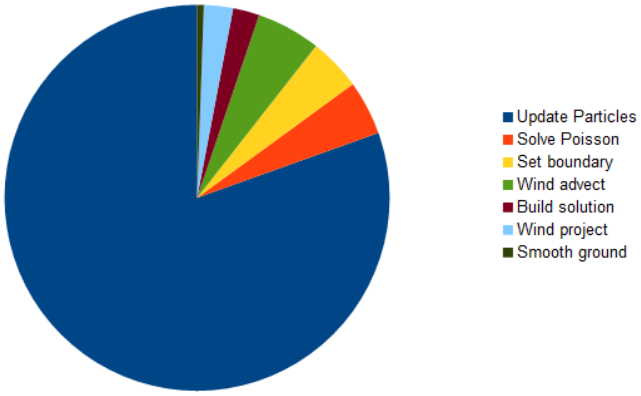


Figure 5.2.2: Kernel time distribution for the OpenACC snow simulator.

Occupancy		
Kernels	CUDA	OpenACC
part_update	83.3%	41.7%
solve_poisson	56.2%	50.0%
set_boundary	93.8%	66.7%
wind_advect	75.0%	41.7%
build_solution	56.2%	66.7%
wind_project	75.0%	66.7 %
smooth_ground	100%	66.7%

Register usage		
Kernels	CUDA	OpenACC
part_update	24	46
solve_poisson	30	42
set_boundary	17	25
wind_advect	22	46
build_solution	32	29
wind_project	22	24
smooth_ground	19	23

Table 5.2.1: Kernel occupancy and register usage in the CUDA and OpenACC snow simulator versions.

5.3 Performance Evaluation

As mentioned in the test setup description, the simulator versions are tested by modifying the snow particle count and the windfield dimension with and without snow and wind simulation. Most of the scaling tests are peformed without any rendering enabled and using all the graphic cards, but a test is also performed using rendering only on the GTX 480. Performance is compared against an OpenMP version in all tests using 8 parallel threads. Any speedup mentioned is defined as GPU_{fps}/CPU_{fps} if not otherwise specified.

5.3.1 Scaling Snow Particles

First we test scaling the snow particle count. Scaling the snow particle count is tested without running the wind simulation, and with the wind simulation and a balanced windfield dimension. All the tests are performed without rendering enabled.

No Windfield Simulation

When scaling the snow particle count and not performing wind simulation we still need a static windfield for the particle update kernel to sample from. The static windfield is set to a size of 64x32x64 and is initialized to only values of zero. Using zero wind velocity values lets the snow particles fall straight down towards the ground as they are only being affected by rotational velocity and gravity.

The results for this test using the GTX 480 card is shown in Figure 5.3.1. We see that the CUDA version performs quite a bit better then the OpenACC version, although the OpenACC version still gets a good speedup compared to the OpenMP version. The difference between the CUDA and OpenACC versions is largest for the lowest amount of snow particles and decreases as the snow particle count is increased.The CUDA version gains an average speedup of 18.1x, while the OpenACC version only gains 3.52x, resulting in only 19.45% of the CUDA versions performance in average.

The resulst from the Tesla cards C2070 and K20c are shown in Figure 5.3.2 and 5.3.3. Both the Tesla cards show a decrease in performance with both the CUDA and the OpenACC versions. The C2070 shows a similar trend as with the GTX 480, while on the K20c we see the performance of the CUDA version decreases more then with the OpenACC version. A likely reason for this is that performance becomes limited by the PCI-E bus when using the K20c card. For every frame, terrain and snow VBO data has to be transferred from the display device to the K20c.

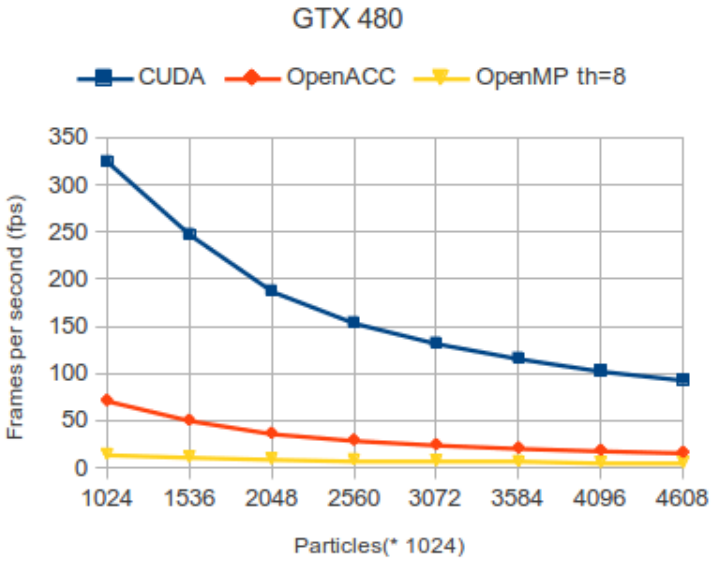


Figure 5.3.1: Scaling snow particle count without wind simulation on GTX 480.

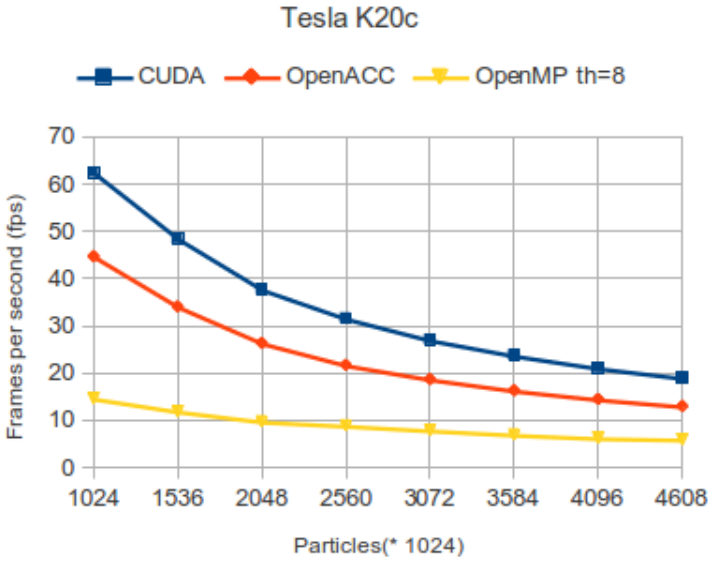


Figure 5.3.2: Scaling snow particle count without wind simulation on Tesla C2070

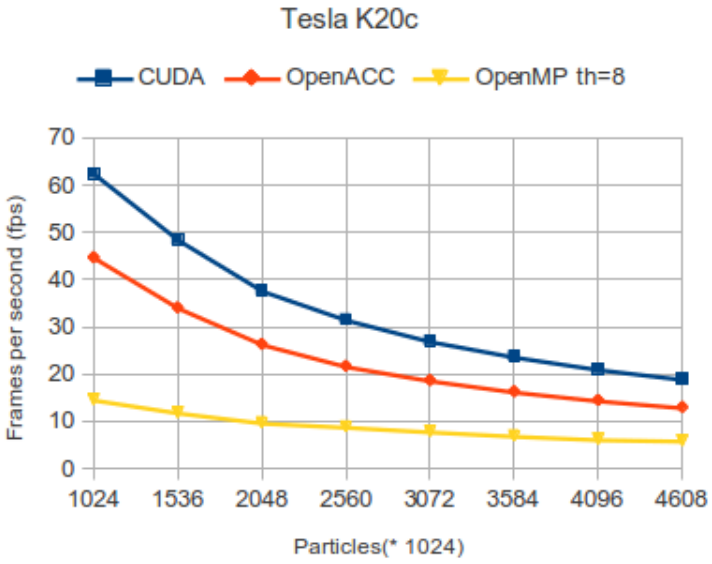


Figure 5.3.3: Scaling snow particle count without wind simulation on Tesla K20c.

Balanced Windfield

The second configuration is to scale the snow particle count with wind simulation and a balanced windfield dimension. The windfield dimension is set to a size of 128x32x128 equaling about 524K voxels. This dimension is chosen because it provides a high enough windfield resolution to affect the movement of the snow particles in a realistic manner. It also allows for real-time performance with the CUDA version and 4M snow particles.

The results for the GTX480, as shown in Figure 5.3.4, follows the same trend as with the previous test only with a slightly less declining curve. With a balanced windfield the CUDA version gains an average speedup of 13.9x, and the OpenACC version 3.3x. This equals to the OpenACC version reaching 23.7% of the CUDA versions performance on average.

Again the same trend is also shown by the Tesla cards in Figure 5.3.5 and 5.3.6. The overall performance is lower and it is the lowest with the K20c card. On the C2070 the CUDA version gains a speedup of 11.7x with 4.7M snow particles and the OpenACC version gains a speedup of 2.4x. On the K20c the CUDA versions speedup is 4.1x while the OpenACC versions speedup actually increases to 2.7x with 4.7M snow particles. The increased speedup of the OpenACC version in contrast to the decreased speedup of the CUDA version can be attributed to CUDA version being hand optimized for the Fermi architecture, while with the OpenACC version the compiler optimizes for the specified architecture. The speedup for 1M, 2.6M, and 4.7M snow particles is shown in Table 5.3.1.

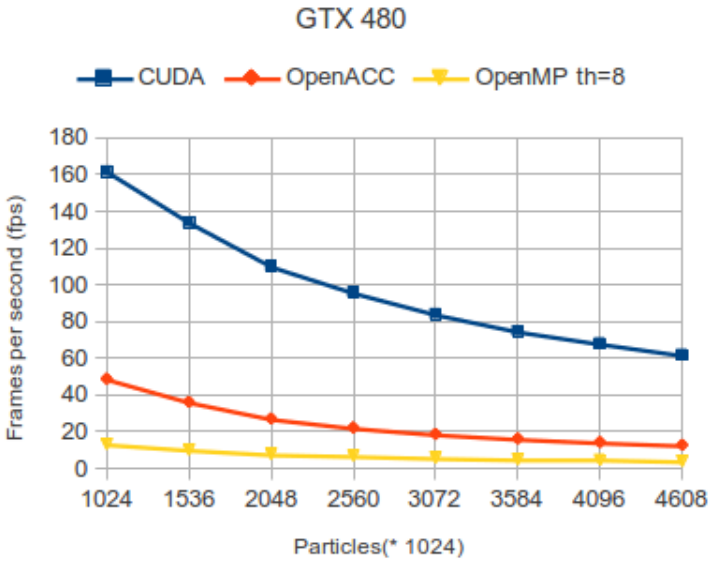


Figure 5.3.4: Scaling snow particle count with wind simulation and balanced windfield on GTX 480.

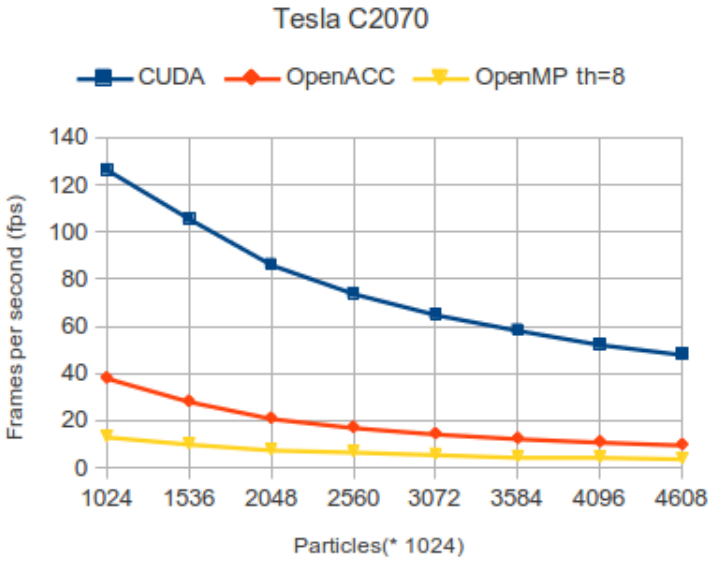


Figure 5.3.5: Scaling snow particle count with wind simulation and balanced windfield on Tesla C2070

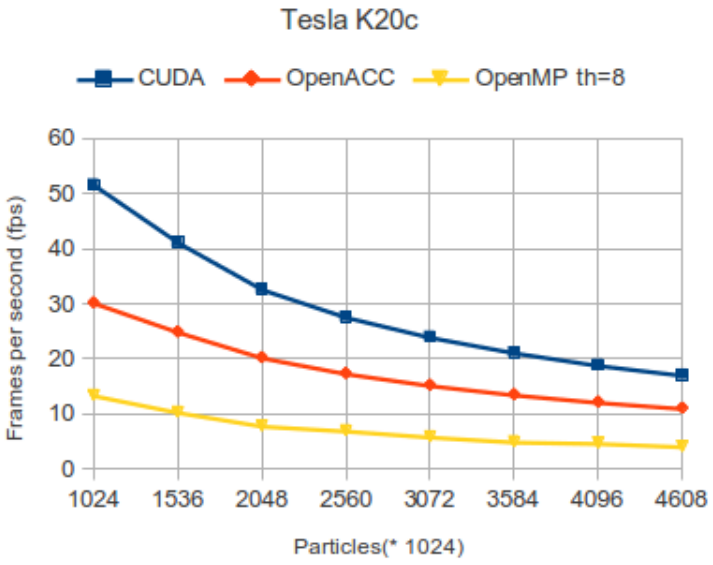


Figure 5.3.6: Scaling snow particle count with wind simulation and balanced windfield on Tesla K20c.

Speedup						
	GTX 480		C2070		K20c	
Particles	CUDA	OpenACC	CUDA	OpenACC	CUDA	OpenACC
1024	12.1	3.6	9.4	2.9	3.9	2.3
2560	13.7	3.2	10.6	2.5	4.0	2.5
4608	15.0	3.1	11.7	2.4	4.1	2.7

Table 5.3.1: Speedup when scaling snow particles with balanced windfield.

5.3.2 Scaling Windfield

Similar to the snow particle scaling test the windfield is scaled and tested with both no snow simulation and with snow simulation and using a balanced snow particle count. These tests are also performed without rendering enabled.

No Snow Particles Simulation

First we test the windfield scaling without any snow particle simulation. The windfield is scaled in four configurations ranging from 130K to 8.3M voxels.

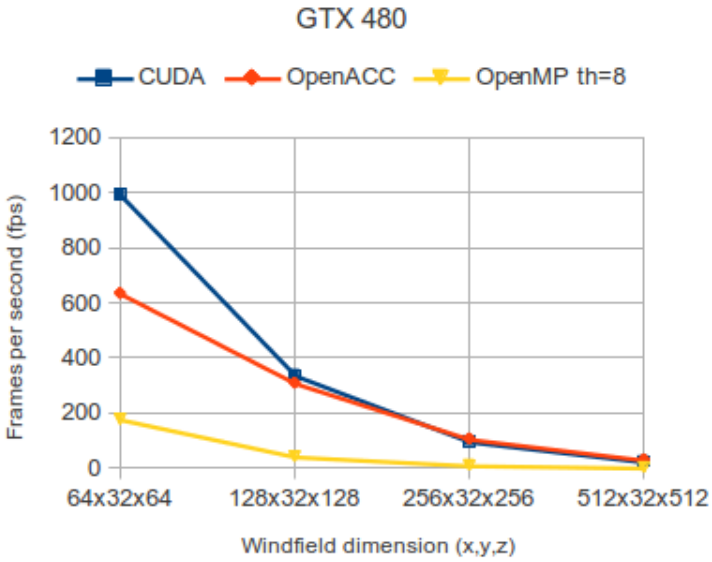


Figure 5.3.7: Scaling windfield dimension without snow particle simulation on the GTX 480.

The results from the GTX 480 is shown in Figure 5.3.7. From the graph we can see that the CUDA and the OpenACC performance is quite similar for all the grid dimensions except for the smallest. For the smallest dimension we would expect all the simulator versions perform well as much of the windfield can be contained directly within the cache. The CUDA version gets the best performance on the smallest dimension because of its high kernel occupancy and shared memory usage, but the performance greatly diminishes as the dimension increases because of the copy operation from global to texture memory after each timestep. Since the OpenACC version uses pointer swapping no copy operations are needed and the performance actually exceeds the CUDA version for the next largest and largest windfield dimensions with a speedup of 15.7x compared to the CUDA speedup of 12.2x for the largest dimension. When looking at the average

speedup the OpenACC also gets ahead. The OpenACC version gains an average speedup of 9.25x while the CUDA version gains 8.83x.

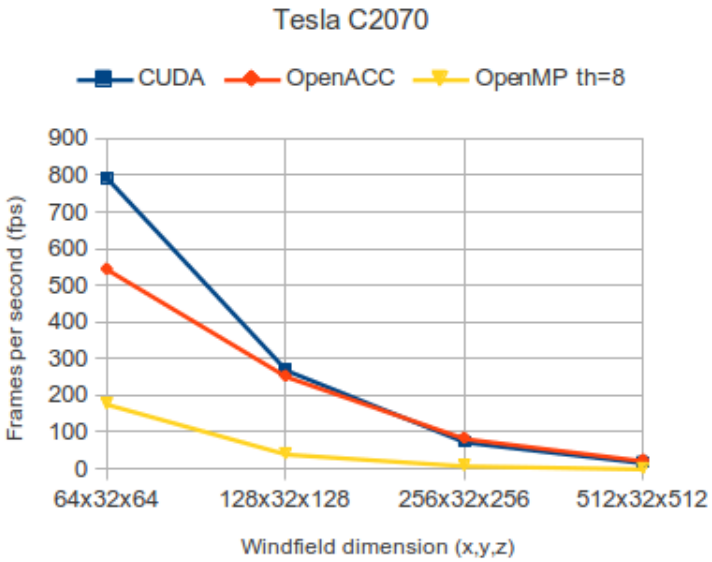


Figure 5.3.8: Scaling windfield dimension without snow particle simulation on the Tesla C2070

On the C2070 as shown in Figure 5.3.8 and 5.3.9 we can see similar results as with the GTX 480, only with a lower overall performance. On the K20c the OpenACC performance is extremely low for the smallest windfield dimension only gaining a 0.6x speedup compared to the 4.3x speedup gained by CUDA. At the larger dimension the OpenACC version closes in on the CUDA version and gains a speedup of 10.4x compared to the CUDA speedup of 14x.

Balanced Snow Particle Count

the last configuration is scaling the windfield and performing snow particle simulation with a constant snow particle count. The snow particle count selected is 1024*1536 = 1572864, which gives a good visual result and real-time performance on the OpenACC version with a reasonable large windfield dimension (128x32x128).

When scaling the windfield with snow particle simulation on the GTX 480 (Figure 5.3.10) we see the same trend as with the previous windfield scaling test. the CUDA version far exceeds the performance of the OpenACC version for smaller windfield dimension, but the gap is narrower for the larger dimensions. We also see the OpenACC performance

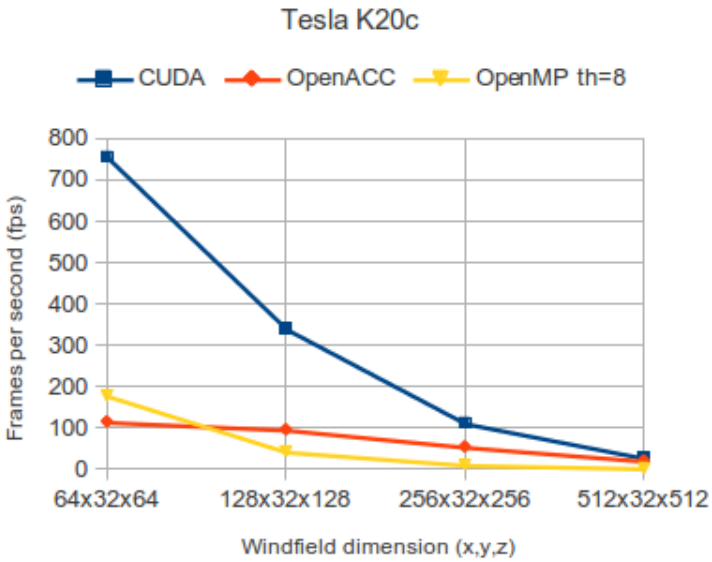


Figure 5.3.9: Scaling windfield dimension without snow particle simulation on the Tesla C2070 and K20c.

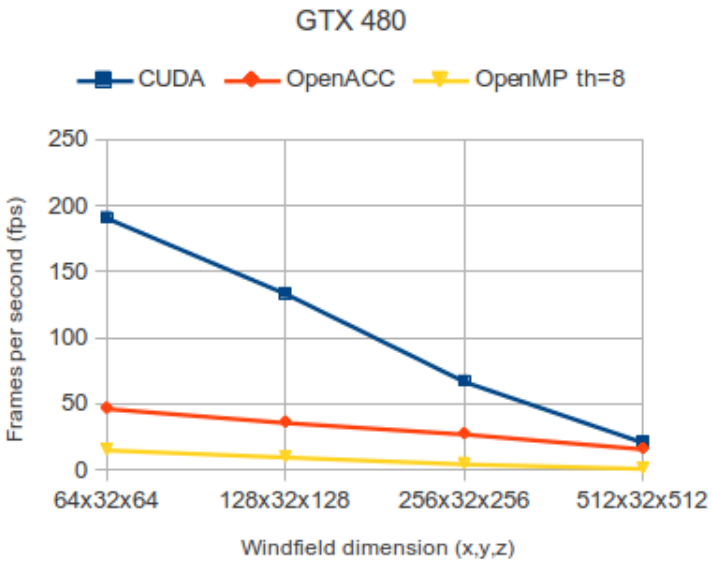


Figure 5.3.10: Scaling windfield size with constant snow particle count on GTX 480.

graph is much more flat then before, indicating that simulating snow is lowering performance overall. The OpenACC version gains an average speedup of 5.47x and the CUDA version a speedup of 12.82x, giving the OpenACC version on average about 42.67% of the CUDA versions performance.

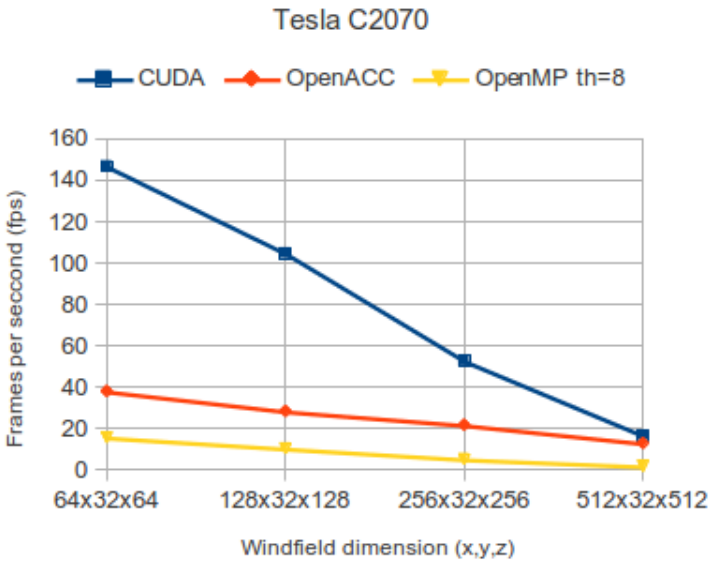


Figure 5.3.11: Scaling windfield size with constant snow particle count on Tesla C2070

On the C2070 (Figure 5.3.11) we get similar results with a overall drop in performance. This trend is also followed on the K20c (Figure 5.3.12), but with a more rounded graph. The speedups from this test are shown in Table 5.3.2. Here we see that the OpenACC performance reaches 77.3% of the CUDA performance using the GTX 480 with the largest windfield dimension.

Speedup						
	GTX 480		C2070		K20c	
Sizes	CUDA	OpenACC	CUDA	OpenACC	CUDA	OpenACC
64x32x64	12.2	3.0	9.4	2.4	2.9	1.7
128x32x128	13.0	3.5	10.2	2.8	4.0	2.4
256x32x256	13.3	5.4	10.4	4.3	6.5	4.0
512x32x512	12.8	9.9	10.0	7.7	10.3	7.4

Table 5.3.2: The speedup when scaling windfield with balanced snow particle count (1.5M).

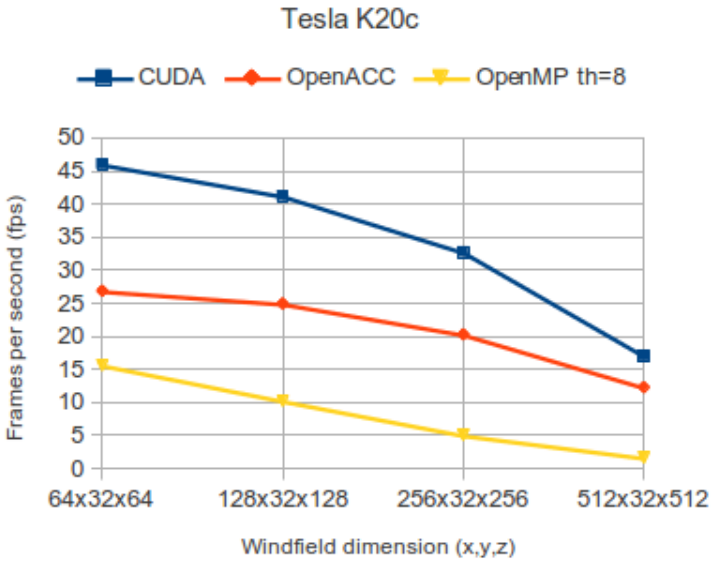


Figure 5.3.12: Scaling windfield size with constant snow particle count on Tesla K20c.

5.3.3 Performance with Rendering

All the previous tests are performed without the rendering system enabled, but it is also interesting to see the performance while using rendering. In this test the windfield and the snow count are scaled with terrain and snow particle rendering. The windfield scaling is performed with a balanced snow particle count of 1.5M, and the snow particle scaling is performed with a balanced windfield of 128x32x128. The terrain is generated from a heightmap of Mount St. Helens with a resolution of 768x768, and the snow particles are rendered as sprites with perlin noise. All the other graphics options are turned down to the lowest setting, which means that no supersampling is enabled and that only the simplest shaders are used.

The results from these tests are shown Figure 5.3.13 and 5.3.14. In Figure 5.3.13 we see the results from scaling snow particle count. Here the OpenACC version gains real-time performance at between 1.5M and 2.1M particles, while the CUDA version easily runs in real-time for particle counts above 4.7M. When averaging the speedups from all the particle counts the OpenACC version has a speedup of 3.2x and the CUDA version a speedup of 7.88x. Using the average speedup the OpenACC reaches about 40.6% of the CUDA performance in this test. When scaling the windfield with balanced snow particle count as shown in Figure 5.3.14, we see that the CUDA and OpenACC converge for larger windfield dimensions as in previous tests. The average speedup of the OpenACC version for this test is 4.84x and for the CUDA version it is 8.25x. This means the OpenACC versions reaches about 58% of the CUDA versions performance on average.

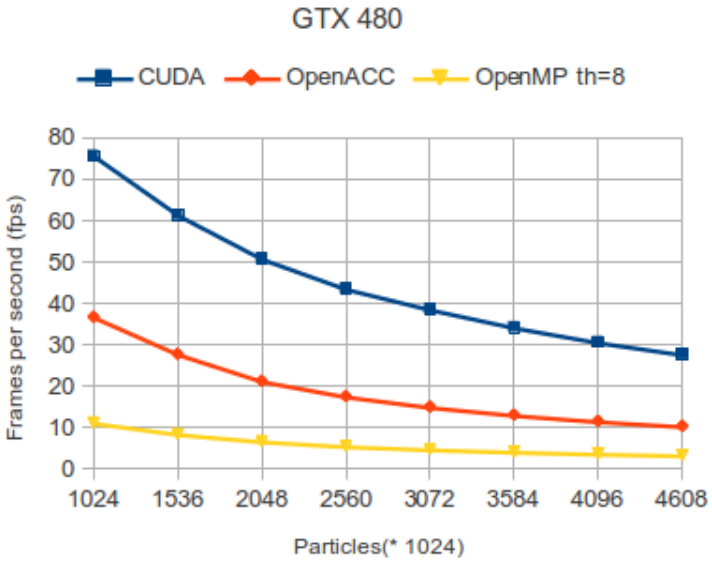


Figure 5.3.13: Scaling snow particle count with balanced windfield and rendering enabled on GTX 480.

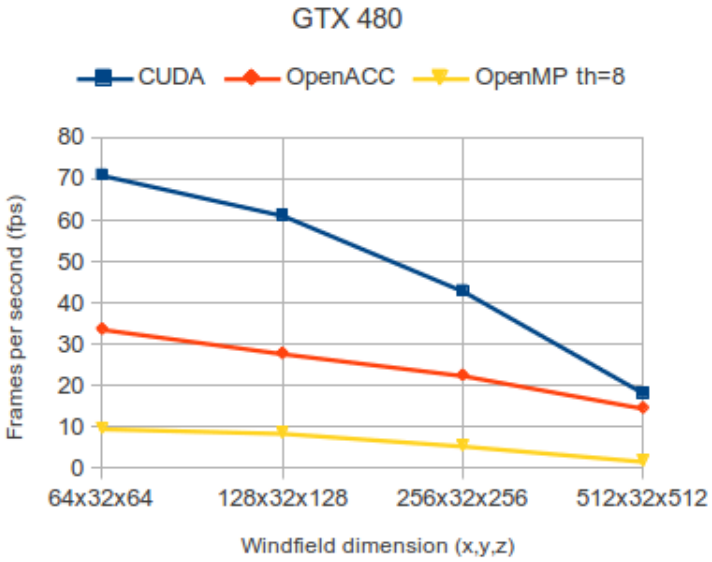


Figure 5.3.14: Scaling windfield with balanced snow particle count and rendering enabled on GTX 480.

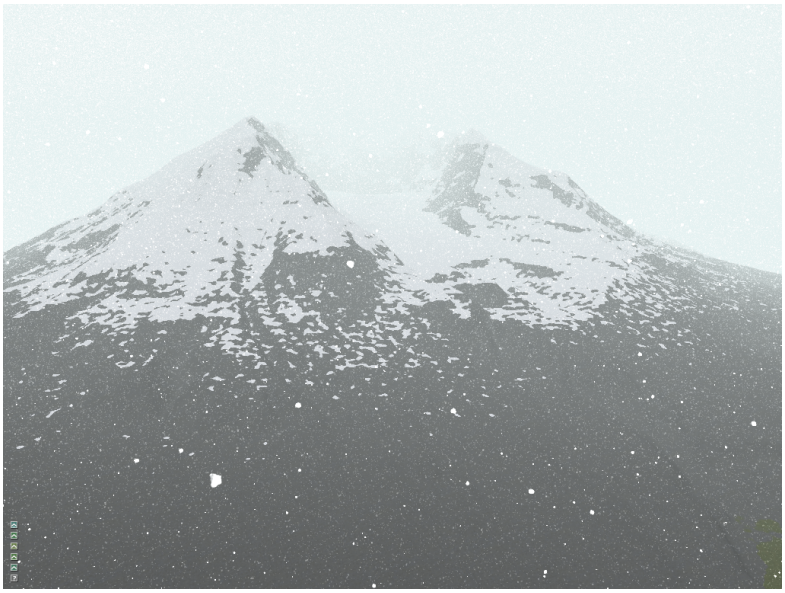


Figure 5.4.1: Mount St. Helens with approximately 1M snow particles and fog effect.

5.4 Visual Results

The visual results of the OpenACC snow simulator are seemingly identical to the CUDA version indicating that it produces the correct results. The new snow particle repositioning method used (Section 4.3.1) also seems to give a distribution similar to the method implemented by Eidissen in the CUDA version.

In Figure 5.4.1 a screenshot of the snow simulator is shown using 1M snow particles and fog providing a very good visual effect. It should be noted that the snow on the terrain in the illustration is not part of the snow buildup functionality of the simulator, but applied as a texture using perlin noise created by Andreas Nordahl. The terrain shown in the image is also created from real-world heightmap data and represents the volcano Mount St. Helens located in the United States.

Figure 5.4.2 shows the debugging kernel used for visualizing the obstacle field used in the wind simulation as a filter in the simulation kernels. Figure 5.4.3 shows a visualization of the pressure field where yellow cells indicate low pressure and red cells high pressure. The map used in this illustration is not based on real-world data, but is automatically generated using noise.

In Figure 5.4.4 the last of the debugging kernels are shown. Here the windfield is visualized using lines to indicate the wind velocity direction and color to indicate the velocity magnitude.

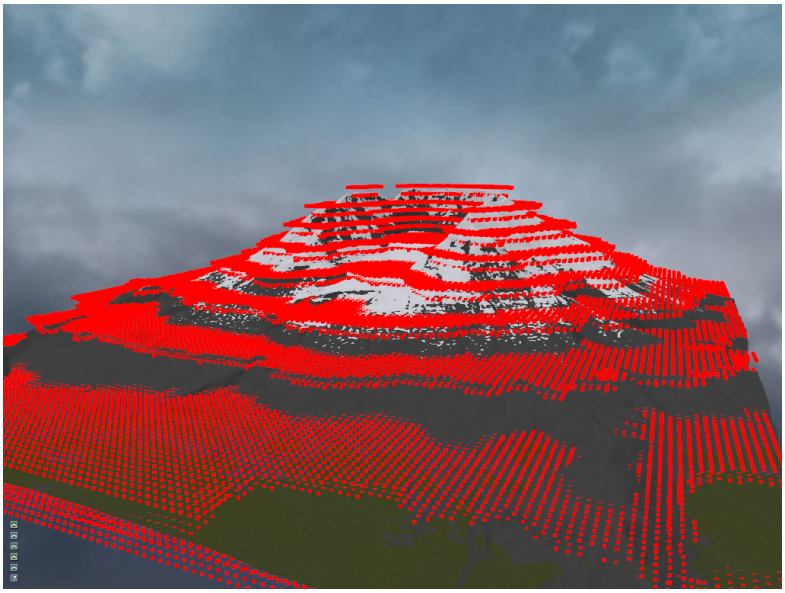


Figure 5.4.2: Mount St. Helens with visualization of obstacle field generated from terrain.

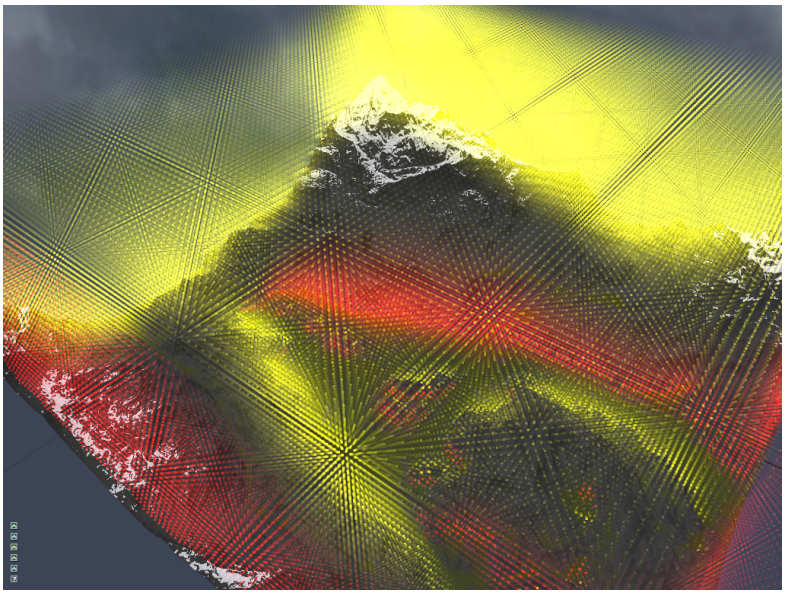


Figure 5.4.3: Noise generated map with visualization of pressure field. Yellow indicates low pressure and red indicates high pressure.

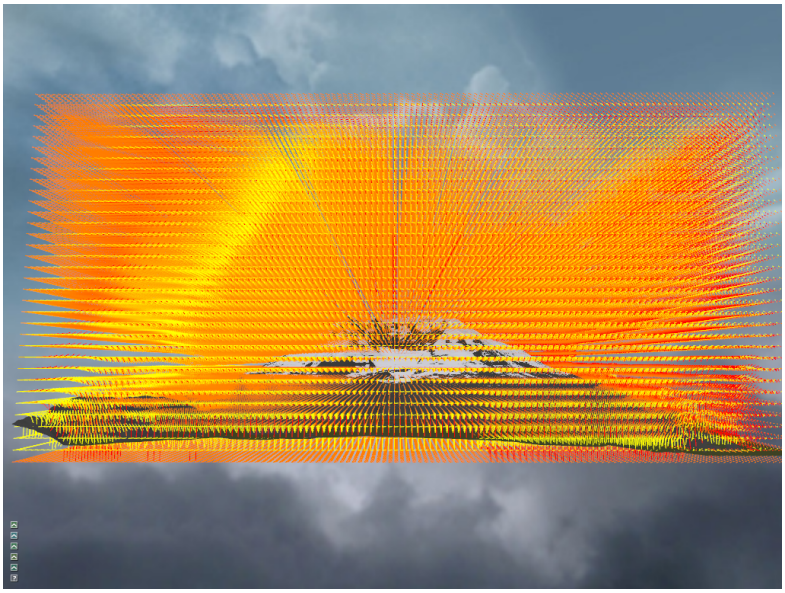


Figure 5.4.4: Mount St. Helens with visualization of wind velocity field.

5.5 Experiences using OpenACC

Even though the benchmarks show that the OpenACC has some performance limitations compared to CUDA there are still some important benefits to using OpenACC. These are mainly increased programmability and readability of the code. This is also reflected in that the most difficult part of this project was porting the CUDA code to a sequential implementation, that accelerating the code with OpenACC directives was fairly simple, and the resulting code is highly readable.

Concerning programmability OpenACC is fairly similar to OpenMP with the exception to the memory model. This is also where the main challenges to using OpenACC lies, as the locality of data greatly influences performance. Fortunately, in the snow simulator all the data can be contained on the device for the applications lifetime, as long as the windfield dimension and number of snow particles is limited to fit in device memory. Since we do not need to synchronize data between the host and device during program execution the actual number of OpenACC directives needed was fairly few, since many of them concern data movement.

Using the OpenACC library routines for allocating and deallocating device memory prevented any need to change the structure of the code, with the exception to some extra data initialization kernels. With the proposed additions in the OpenACC version 2.0 draft (Section 2.2.4) these kernels could be replaced with the `acc_memcpy` routine. Using these routines does have the unfortunate side effect of breaking portability with the CPU code. A solution to this would be to introduce conditional compiling to indicate which part of the code to compile when OpenACC is enabled. However, this will also

lead to more cluttered code. The version 2.0 draft introduces the `enter data` and `exit data` directives that enables the allocation of device memory for the duration of the program without needing a containing data region. Using these directives in the snow simulator could prevent the need to use library routines and enable code portability.

The main drawback to using OpenACC was found to be the lack of explicitly using shared memory, mainly for reducing the register usage in the wind simulation kernels. This is of course a natural effect of OpenACC being a standard aimed at heterogeneous systems and device. Shared memory can, however, be explicitly used as cache with the cache directive. As mentioned in section 4.3.2 using the cache directive was found to not have any effect on performance, and it was unclear it it was working correctly. Nonetheless, the cache directive might not be so beneficial in this context as Fermi and newer NVIDIA GPUs already performs L1 hardware caching. A feature also missed was using the device texture memory. This was evident in the performance of the simulation of snow particles, where hardware interpolation and 3D caching would be an immense benefit. It would be interesting to see if this performance hit could be reduced by sorting the snow particle to reflect the 1D caching used by global memory.

It was also found very beneficial for OpenACC to have compatibility with CUDA. This was especially evident with the handling of OpenGL VBOs, where it was possible to map the VBOs to CUDA device pointers and pass them to OpenACC kernels. Without this functionality the alternative would be to copy the VBO data to host and then to copy it to OpenACC kernel, and back again. Needless to say these copy operations would dominate execution time. This was also the main reason for not using the accULL OpenACC compiler, as it does not currently support the deviceptr data clause needed to pass device pointers to OpenACC kernels. The deviceptr clause is however stated as “todo” in the source code of the compiler, indicating it will be implemented in the future.

Using the PGI OpenACC compiler is also highly recommended, as it made programming with OpenACC much easier with detailed compiler feedback indicating data flow and loop scheduling. Also useful is the built in feature to perform program execution timing and view generated CUDA code.

Chapter 6

Conclusion & Future Work

The main goal of this thesis was to update the latest CUDA version of the HPC-Lab snow simulator and to port it to the compiler directive-based approach to GPGPU programming called OpenACC. The purpose of this was to evaluate the performance achieved to examine if this new standard for GPGPU programming was applicable for accelerating such an application by comparing its performance to the CUDA version. Secondary goals included making the code more modular were possible and testing both the commercial PGI and open source accULL OpenACC compilers. Testing using the accULL compiler was not performed as it was found to be lacking support for some of the OpenACC directives needed to get reasonable performance.

6.1 Conclusions

Effort was put into cleaning up and updating the code of the CUDA snow simulator and making it more modular before starting the porting process. A modular structure was achieved with a separation of the simulation code from the rest of the application. The port was made by first creating a sequential CPU implementation of the snow simulator, applying OpenACC directives to accelerate compute intensive regions, and optimizing it by reducing data movement between the host and the device with the use of OpenACC library routines to allocate device memory.

During the porting process some potential performance bottlenecks connected to the platform independent nature of OpenACC were identified. The most important of these were the inability of explicitly using shared memory for blocking and temporary storage in kernels, and not being able to use texture memory for hardware interpolation and three dimensional caching. These limitations were also reflected when performing kernel profiling and testing as it showed that a large majority of execution time was spent performing snow particle updating.

The OpenACC snow simulator version was tested and compared against the CUDA version and a OpenMP version running on 8 CPU cores used as the performance ba-

sis. The tests were performed by scaling the snow particle count and the windfield dimensions with and without rendering and the other simulation components activated. With rendering, a balanced windfield dimension, and scaling the snow particle count the OpenACC version achieved 40.6% of the CUDA version performance with an average speedup of 3.2x compared to 7.88x achieved with CUDA. With rendering, a balanced snow particle count, and scaling the windfield dimension an average speedup of 4.84x was achieved with the OpenACC version resulting in 58% of the CUDA version performance that had a average speedup of 8.25x. For all the tests scaling the snow particle count the CUDA version was found to be consistently much faster then the OpenACC version, however, when scaling the windfield the OpenACC version was faster than the CUDA version on larger windfield dimensions. Real-time performance with the OpenACC version was found at about 1.5M snow particles when using a balanced windfield of about 524K voxels.

While the results shows that the OpenACC version performance cannot compete with an optimized CUDA implementation, it still gives a reasonable good speedup over a multicored CPU approach. Making it a viable alternative if the goal is create an implementation that can run on heterogeneous systems. However, most often the goal in simulation applications, and especially graphical simulator applications, is to be able to get the highest performance possible, making the use of more low-level programming models such as CUDA or OpenCL an better approach overall.

6.2 Future Work

To extend this work in the future the most important focus area would be to test more of the OpenACC compiler implementations, especially the accULL compiler since its open source nature makes it of most academic interest. This will however require the implementation of the `deviceptr` data clause in the accULL compiler. More testing should also be performed using cache directive and loop scheduling as these are the main courses for improving performance in an OpenACC application besides making changes to the algorithm or reducing data movement.

To improve performance it might also be interesting to investigate if the OpenACC snow simulator benefits more from sorting the snow particles than with the CUDA/OpenCL versions. This might be the case since the OpenACC version has a larger penalty when performing lookups in the windfield compared to CUDA and OpenCL. The OpenACC version of the snow simulator can also be further extended by including the Lattice Boltzmann method for simulation of fluid flow, as well as 3D rendering and road generation.

Another natural course of investigation would be to explore the use of multiple GPUs to handle the simulation. The major obstacle for this would be the need for border exchanges between the GPUs, especially concerning the snow particles as they are not bound to any specific section of the simulation volume. Effort should also be made to

optimize the CUDA version of the snow simulator for Kepler as the current performance using this architecture is lacking.

Bibliography

[1] PGI Accelerator Programming Model for Fortran and C v1.3, The Portland Group, http://www.pgroup.com, 2010.

[2] Tianyi David Han, Tarek S. Abdelrahman, hiCUDA: High-Level GPGPU Programming, IEEE Transactions on Parallel and Distributed systems 22(1), 78-90, 2011.

[3] R. Dolbeau, S. Bihan, and F. Bodin, HMPP: A Hybrid Multi-Core Parallel Programming Environment, CAPS Enterprise, www.caps-enterprise.com, 2001.

[4] CAPS Enterprise, Cray Inc., NVIDIA, and the Portland Group: The OpenACC Application Programming Interface, v1.0, www.openacc.org, 2011.

[5] NVIDIA Fermi Architecture: White paper NVIDIA's Next Generation CUDA Compute Architecture, http://www.nvidia.com, 2009.

[6] NVIDIA Kepler Architecture: White paper NVIDIA's Next Generation CUDA Compute Architecture Kepler GK110, http://www.nvidia.com, 2012.

[7] J. Sanders, E. Kandrot, CUDA By Example: An Introduction to GPGPU Programming, NVIDIA, Addison-Wesley, ISBN: 978-0-13-13-138768-3, 2011.

[8] Rob Farber, The OpenACC Execution Model, Dr. Drobbs – The world of Software Development, http://www.drdobbs.com/parallel/the-openacc-execution-model/240006334, 2012.

[9] Jeremiah van Oosten, CUDA Memory Model, 3D Game Engine Programming, http://3dgep.com/?p=2012, 2011.

[10] Ruyman Reyes, Ivan Lopez-Rodriguez, Juan J. Fumero, and Francisco de Sande, accULL: An OpenACC Implementation with CUDA and OpenCL Support, University of Laguna, Spain, 2012.

[11] Michael Wolfe, The PGI Accelerator Compilers with OpenACC, http://www.pgroup.com/lit/articles/insider/v4n1a1.htm, 2012.

[12] A. Duran, E. Ayguade, R. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas: OmpSs: A Proposal for Programming Heterogeneous Multicore Architectures, Parallel Process. Lett, 21(2), 173-193, 2011.

[13] S. Lee, R. Eigenmann, OpenMPC: Extended OpenMP Programming and Tuning for GPUs, IEEE, SC10, 2010.

[14] Jos Stam. Interacting With Smoke and Fire in Real-Time. Commun. ACM, 43(7):76-83, 2000.

[15] Mark J. Harris, GPU Gems Chapter 38. Fast Fluid Dynamics Simulation on the GPU, University of North Carolina, 2004.

[16] Jos Stam. Stable Fluids. In SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques, pages 121-128, New York, USA, 1999, ACM Pres/Addison-Wesley Publishing Co.

[17] Jos Stam. Real-Time Fluid Dynamics for Games. 2003.

[19] Robin Eidissen, Utilizing GPUs for real-time visualization of snow, Master Thesis, IDI, NTNU, 2009.

[20] Joel Chelliah, The NTNU HPC Snow Simulator on the Fermi GPU, IDI, NTNU, 2010.

[21] Jarle Erdal Steinsland, Porting the OpenCL Snow simulator to opencl for mobile applications, Master Thesis, IDI, NTNU, 2010.

[22] Frederik Magnus Johansen Vestre, Enhancing and Porting the HPC-Lab Snow Simulator to OpenCL on Mobile Platforms, Master Thesis, IDI, NTNU, 2012.

[23] Magnus Mikalsen, Fast Fluid Dynamics Simulation using Directive-Based GPU programming with OpenACC, IDI, NTNU, 2012.

[24] Ingar Saltvik, Anne C. Elster, and Henrik R. Nagel. Parallel methods for real-time visualization of snow. In Bo Kgstrm, Erik Elmroth, Jack Dongarra, and Jerzy Wasniewski, editors, PARA, volume 4699 of Lecture Notes in Computer Science, pages 218–227. Springer, 2006.

[25] Khronos OpenCL Working Group, The OpenCL Specification Version 1.2, Rev. 19, 2012.

[26] Chorin, A.J., and J.E. Marsden. A Mathematical Introduction to Fluid Mechanics. 3rd ed. Springer, 1993.

[27] Wei, X., Li, W., Mueller, K., Kaufman, A.E.: The lattice-boltzmann method for simulating gaseous phenomena. IEEE Transactions on Visualization and Computer Graphics 10(2), 164–176, 2004.

[28] Michael Griebel, Thomas Dornseifer, and Tilman Neunhoeffer. Numerical simulation in fluid dynamics: a practical introduction. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[29] Nick Foster and Dimitris Metaxas. Modeling the motion of a hot, turbulent gas. Computer Graphics, 31(Annual Conference Series):181–188, 1997.

[30] Michael Aagaard and Dennis Lerche. Realistic modelling of falling and accumulating snow. 2004. Master Thesis, Aalborg University.

[31] Ingar Saltvik. Parallel methods for real-time visualization of snow. 2005. Master thesis, NTNU.

[32] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In SIGGRAPH '01: Proceedings of the 28th annual conference on Computer.

[33] CUDA C Programming Guide, NVIDIA, http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2012.

[34] The OpenACC Foundation, The OpenACC Application Programming Interface, v2.0, Public Comment Draft, www.openacc.org, 2013.

[35] Alexander Gjermundsen, LBM vs SOR solvers on GPUs for real-time snow simulations, Master's thesis, IDI, Norwegian University of Technology and Science, 2009.

[36] Hallgeir Lien, Procedural Generation of Roads for use in the Snow Simulator, 2011.

[37] Kjetil Babington, Terrain Rendering Techniques for the HPC-Lab Snow Simulator, Master's thesis, IDI, Norwegian University of Technology and Science, 2012.

[38] D. E. Knuth. The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89684-2. Section 3.2.1: The Linear Congruential Method, pp. 10–26.

[39] Alberto Magni, Dominik Grewe, and Nick Johnson, Input-Aware Auto-Tuning for Directive-Based GPU Programming, ACM, 2013, ISBN: 978-1-4503-2017-7, GPGPU-6 Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, pp. 66-75.

[40] David M. Young, Iterative Solution of Large Linear Systems, Courier Dover Publications, 2003.

[41] S. Wienke, P. L. Springer, C. Terboven, and D. an Mey. OpenACC - first experiences with real-world applications. In International European Conference on Parallel and Distributed Computing, Euro-Par, 2012.

[42] R. Reyes, I. Lopez and J. J. Fumero. A preliminary evaluation of OpenACC implementations. The Journal of Supercomputing, Springer, 2013.

[43] Jan M. Rovde, Real-Time Granular Flow Simulation Using the PCISPH Method on GPGPU Devices Using CUDA, Master's Thesis, NTNU, IDI, 2012

[44] Øystein Eklund Krog, GPU-base Real-Time Snow Avalanche Simulation, Master's Thesis, NTNU, IDI, 2012

Appendix A

OpenACC Simulation Code

This sections shows the code from the OpenACC snow simulation implementation where OpenACC directives or library routines are used. The snow simulators complete code can be found in the code attachment.

A.1 Wind Simulation Initialization

```
1  /* Precalculate divergence values used by Poisson solver */
2  void CalcPoissonTable(float *p_tab) {
3  #pragma acc parallel deviceptr(p_tab)
4  {
5      p_tab[63] = 0.0f;
6      for(int i = 0; i < 63; ++i) {
7          int div = 0;
8          for(int j = 0; j < 6; ++j)
9              div += (i & (1<<j)) == 0;
10         p_tab[i] = 1.0f / (float)div;
11     }
12 }
13 }
14
15 /* Set initial values to the wind simulation data structures */
16 void SetWindMem(float_4 *wind_vel, float_4 * wind_vel0, float *solution, int *
       obstacle, int size) {
17 #pragma acc kernels deviceptr(wind_vel,wind_vel0,solution,obstacle) copyin(size)
18 {
19     #pragma acc loop independent
20     for (int i = 0; i < size; ++i) {
21         wind_vel[i] = make_float_4(0.0f, 0.0f, 0.0f, 0.0f);
22         wind_vel0[i] = make_float_4(0.0f, 0.0f, 0.0f, 0.0f);
23         solution[i] = 0.0f;
24         obstacle[i] = 0;
25     }
26 }
27 }
28
29 /* Initialize the wind simulation */
30 void InitWindField(int device_num, int dimx, int dimy, int dimz, int terdim) {
```

```
31
32      // Init device
33      acc_set_device_num(device_num,acc_device_nvidia);
34      acc_init(acc_device_nvidia);
35
36      // Add halo
37      dim.x = dimx+2;
38      dim.y = dimy+2;
39      dim.z = dimz+2;
40
41      tdim = terdim;
42
43      idim.x = 1.f/(float)dim.x;
44      idim.y = 1.f/(float)dim.y;
45      idim.z = 1.f/(float)dim.z;
46
47      size = dim.x*dim.y*dim.z;
48
49      // Allocate device memory
50      wind_vel    = (float_4 *)   acc_malloc(size*sizeof(float_4));
51      wind_vel0   = (float_4 *)   acc_malloc(size*sizeof(float_4));
52      pressure    = (float *)     acc_malloc(size*sizeof(float));
53      pressure0   = (float *)     acc_malloc(size*sizeof(float));
54      solution    = (float *)     acc_malloc(size*sizeof(float));
55      convert     = (float *)     acc_malloc(3*sizeof(float));
56      poisson_tab = (float *)     acc_malloc(64*sizeof(float));
57      obstacle    = (int *)       acc_malloc(size*sizeof(int));
58
59      CalcPoissonTable(poisson_tab);
60
61      // Initialize the simulation data
62      SetWindMem(wind_vel,wind_vel0,solution,obstacle,size);
63
64      // Initialize snow to wind conversion constants
65  #pragma acc parallel deviceptr(convert) copyin(dim)
66  {
67      convert[0] = (float)dim.x / (float)SCENE_X;
68      convert[1] = (float)dim.y / (float)SCENE_Y;
69      convert[2] = (float)dim.z / (float)SCENE_Z;
70  }
71  }
```

Listing A.1: Wind simulation initialization function. Shows the small parallel regions needed to initialize data allocated to device memory.

A.2 Wind Simulation Kernels

Wind Advection

```
1  /* Self-advect the wind velocty */
2  void wind_advect(float_4 *d, float_4 *d0, float *p, float *p0, int *obs, float dt,
       dim_3 dim, float_4 boundary) {
3  #pragma acc kernels deviceptr(d0,obs,d,p,p0) copyin(dt,dim,boundary)
4  {
5      #pragma acc loop independent
6      for(int y = 0; y < dim.y; ++y) {
7          #pragma acc loop independent
```

```
8           for(int z = 0; z < dim.z; ++z) {
9               #pragma acc loop independent
10              for(int x = 0; x < dim.x; ++x) {
11
12                  float_4 v = d0[I(x,y,z)];
13
14                  // Only perform inside halo
15                  if(x>0 && x<dim.x-1 && y>0 && y<dim.y-1 && z>0 && z<dim.z-1) {
16                      // Backtraced coordinates
17                      float vx = (float)x - dt * v.x - 0.5f;
18                      float vy = (float)y - dt * v.y - 0.5f;
19                      float vz = (float)z - dt * v.z - 0.5f;
20
21                      /* Perform trilinear interpolation. Code omitted */
22
23                      // Mask results with obstacle field
24                      int mask = obs[I(x,y,z)];
25                      if(mask & (VOX_SELF | VOX_LEFT | VOX_RIGHT)) v.x = 0.0f;
26                      if(mask & (VOX_SELF | VOX_ABOVE | VOX_BELOW)) v.y = 0.0f;
27                      if(mask & (VOX_SELF | VOX_UP | VOX_DOWN)) v.z = 0.0f;
28
29                      if(y == dim.y-2) d[I(x,y+1,z)] = v;
30
31                  } else {
32                      // Set to boundary value if at halo
33                      v = boundary;
34                  }
35                  // Set new velocity value and
36                  // also initialize pressure field to zero
37                  d[I(x,y,z)] = v;
38                  p[I(x,y,z)] = 0.0f;
39                  p0[I(x,y,z)] = 0.0f;
40
41              }
42          }
43      }
44  }
45  }
```

Listing A.2: The wind advection kernel. The trilinear interpolation code is removed for brevity.

Build Solution

```
1   /* Build the right-hand side of the poisson equation (b-vector) */
2   void build_solution(float_4 *vel, float *s, int *obs, dim_3 dim, float factor) {
3   #pragma acc kernels deviceptr(vel,s,obs) copyin(dim,factor)
4   {
5       #pragma acc loop independent
6       for(int y = 1; y < dim.y-1; y++) {
7           #pragma acc loop independent
8           for(int z = 1; z < dim.z-1; z++) {
9               #pragma acc loop independent
10              for(int x = 1; x < dim.x-1; x++) {
11
12                  // Get obstacle mask
13                  float mask = (float)(((int)obs[I(x,y,z)] & VOX_SELF) == 0);
14
```

```
15                  // Calculate divergence
16                  s[I(x,y,z)] = factor * mask *
17                              ((vel[I(x+1,y,z)].x - vel[I(x-1,y,z)].x) +
18                               (vel[I(x,y+1,z)].y - vel[I(x,y-1,z)].y) +
19                               (vel[I(x,y,z+1)].z - vel[I(x,y,z-1)].z));
20              }
21          }
22      }
23  }
24  }
```

Listing A.3: Build solution kernel for creating the right hand side b-vector in the Poisson equation.

SOR Poisson Solver

```
1   /**
2   * Poisson solver using SOR.
3   * res = pold(i-1,j,k)+pold(i+1,j,k)+pold(i,j-1,k)+pold(i,j+1,k)+ \
4   *       pold(i,j,k-1)+pold(i,j,k+1)-b(i,j,k)*w
5   * pnew(i,j,k) = w*tes+(1-w)*told(i,j,k)
6   */
7   void solve_poisson(float *p, float *p0, float *b, int *obs, float *poisson_tab,
        dim_3 dim, float w) {
8   #pragma acc kernels deviceptr(p,p0,obs,b,poisson_tab) copyin(w)
9   {
10      #pragma acc loop independent
11      for(int y = 1; y < dim.y-1; ++y) {
12          #pragma acc loop independent
13          for(int z = 1; z < dim.z-1; ++z) {
14              #pragma acc loop independent
15              for(int x = 1; x < dim.x-1; ++x) {
16
17                  // Get obstacle mask
18                  int mask = obs[I(x,y,z)] & 127;
19
20                  if ((~mask & 126) && (mask & VOX_SELF) == 0) {
21                      float res = 0.0f;
22                      res += p0[I(x-1,y,z)] * (float)((mask & VOX_LEFT) == 0);
23                      res += p0[I(x+1,y,z)] * (float)((mask & VOX_RIGHT) == 0);
24                      res += p0[I(x,y-1,z)] * (float)((mask & VOX_BELOW) == 0);
25                      res += p0[I(x,y+1,z)] * (float)((mask & VOX_ABOVE) == 0);
26                      res += p0[I(x,y,z-1)] * (float)((mask & VOX_UP) == 0);
27                      res += p0[I(x,y,z+1)] * (float)((mask & VOX_DOWN) == 0);
28                      res -= b[I(x,y,z)];
29                      res *= poisson_tab[mask>>1];
30                      res *= w;
31                      res += p0[I(x,y,z)]*(1.0f - w);
32                      p[I(x,y,z)] = res;
33                  }
34                  else {
35                      p[I(x,y,z)] = 0.0f;
36                  }
37              }
38          }
39      }
40  }
41  }
```

Listing A.4: The SOR Poisson solver used by the wind simulation to calculate the pressure field used to make the velocity field divergence-free in the projection step.

Set Pressure Boundaries

```
/* Correct the pressure fields boundary values */
void set_boundary(float *p, int *obs, dim_3 dim, int which) {
#pragma acc kernels deviceptr(p,obs) copyin(dim,which)
{
    #pragma acc loop independent
    for(int y = 1; y < dim.y-1; ++y) {
        #pragma acc loop independent
        for(int z = 1; z < dim.z-1; ++z) {
            #pragma acc loop independent
            for(int x = 1; x < dim.x-1; ++x) {

                int mask = obs[I(x,y,z)] & 127;

                if((mask & VOX_SELF) && mask != 127) {
                    if(!(mask & VOX_LEFT))
                        p[I(x,y,z)] = p[I(x-1,y,z)];
                    else if(!(mask & VOX_RIGHT))
                        p[I(x,y,z)] = p[I(x+1,y,z)];
                    else if(!(mask & VOX_UP))
                        p[I(x,y,z)] = p[I(x,y,z-1)];
                    else if(!(mask & VOX_DOWN))
                        p[I(x,y,z)] = p[I(x,y,z+1)];
                    else if(!(mask & VOX_ABOVE))
                        p[I(x,y,z)] = p[I(x,y+1,z)];
                    else if(!(mask & VOX_BELOW))
                        p[I(x,y,z)] = p[I(x,y-1,z)];
                }

                if(x == 1 && which & VOX_LEFT)
                    p[I(0,y,z)] = p[I(1,y,z)];
                if(x == dim.x-2 && which & VOX_RIGHT)
                    p[I(dim.x-1,y,z)] = p[I(dim.x-2,y,z)];

                if(z == 1 && which & VOX_UP)
                    p[I(x,y,0)] = p[I(x,y,1)];
                if(z == dim.z-2 && which & VOX_DOWN)
                    p[I(x,y,dim.z-1)] = p[I(x,y,dim.z-2)];

                if(y == 1 && which & VOX_BELOW)
                    p[I(x,0,z)] = p[I(x,1,z)];
                if(y == dim.y-2 && which & VOX_ABOVE)
                    p[I(x,dim.y-1,z)] = p[I(x,dim.y-2,z)];

            }
        }
    }
}
}
```

Listing A.5: Pressure boundary kernel in the wind simulation. Corrects the pressure boundary values to satisfy the boundary conditions.

Wind Projection

```
/* Make the velocity field divergence-free by subtracting the pressure gradient */
void wind_project(float_4 *vel, float *p, dim_3 dim, float factor) {
#pragma acc kernels deviceptr(vel,p) copyin(dim,factor)
{
    #pragma acc loop independent
    for(int y = 1; y < dim.y-1; ++y) {
        #pragma acc loop independent
        for(int z = 1; z < dim.z-1; ++z) {
            #pragma acc loop independent
            for(int x = 1; x < dim.x-1; ++x) {
                vel[I(x,y,z)].x -= factor * (p[I(x+1,y,z)] - p[I(x-1,y,z)]);
                vel[I(x,y,z)].y -= factor * (p[I(x,y+1,z)] - p[I(x,y-1,z)]);
                vel[I(x,y,z)].z -= factor * (p[I(x,y,z+1)] - p[I(x,y,z-1)]);
            }
        }
    }
}
}
```

Listing A.6: Wind projection kernel in the wind simulation.

A.3 Snow Particle Simulation Initialization

```
/**
* Initialize the snow particles, ie. velocity and spiral movement radius
*/
void InitSnowParticles(int num_parts, float _gravity, float snow_growth, float
    delta_time) {

    gravity = _gravity;
    snowgrowth = snow_growth;
    deltatime = delta_time;

    float *t_omg, *t_rad;
    int *t_seeds;
    float_4 *t_vel;

    // Allocate temporary host data to initialize with system calls
    t_omg = (float *) malloc(sizeof(float)*32);
    t_rad = (float *) malloc(sizeof(float)*32);
    t_seeds = (int *) malloc(sizeof(int)*num_parts);
    t_vel = (float_4 *) malloc(num_parts*sizeof(float_4));

    // Initialize radiuses and omegas on host using
    // system calls to get random values
    for (int i = 0; i < 32; ++i) {
        t_rad[i] = drand48()*2.0f;
        t_omg[i] = calc_omega()*1.0f;//0.1f;
    }

    // Initialize particle velocity on host using
    // system calls to get random values
    for (int i = 0; i < num_parts; ++i) {
        float maxv = drand48() + 1.0f;
        t_vel[i].x = drand48();
```

```
32          t_vel[i].y = -maxv;
33          t_vel[i].z = drand48();
34          t_vel[i].w = 1.0f / (maxv * maxv); // terminal velocity
35          t_seeds[i] = floor(drand48()*RANDMAX); // repos seed values
36      }
37
38      // Allocate device memory
39      omegas      = (float *)     acc_malloc(32*sizeof(float));
40      radiuses    = (float *)     acc_malloc(32*sizeof(float));
41      seeds       = (int *)       acc_malloc(num_parts*sizeof(int));
42      part_vel    = (float_4 *)   acc_malloc(num_parts*sizeof(float_4));
43
44      // Copy initialized host data to device memory
45  #pragma acc kernels deviceptr(omegas,radiuses,part_vel,seeds) \
46  copyin(num_parts,t_omg[0:32],t_rad[0:32],t_vel[0:num_parts],t_seed[0:num_parts])
47  {
48      #pragma acc loop independent
49      for (int i = 0; i < 32; ++i) {
50          omegas[i] = t_omg[i];
51          radiuses[i] = t_rad[i];
52      }
53      #pragma acc loop independent
54      for (int i = 0; i < num_parts; ++i) {
55          part_vel[i] = t_vel[i];
56          seeds[i] = t_seeds[i];
57      }
58  }
59
60      // Free temporary host data
61      free(t_omg);
62      free(t_rad);
63      free(t_seeds);
64      free(t_vel);
65  }
```

Listing A.7: Initialization of the snow particle simulation data using parallel regions.

A.4 Snow Particle Simulation Kernels

Particle Update

```
1   /* Update each snow particles position and velocity */
2   void part_update(float_4 *part_pos, float_4 *part_vel, float_4 *wind_vel,
3               float_4 *grid, float *omegas, float *radiuses, float gravity,
4               float snow_growth, float deltatime, int num_parts,
5               int tdim, float *convert, dim_3 dim, int *seeds) {
6
7   #pragma acc kernels \
8       deviceptr(part_vel,wind_vel,radiuses,omegas,part_pos,grid,seeds,convert) \
9       pcopyin(gravity,snow_growth,deltatime,num_parts,tdim,dim)
10  {
11      #pragma acc loop independent
12      for (int i = 0; i < num_parts; ++i) {
13          float_4 pos = part_pos[i];
14          float_4 vel = part_vel[i];
15
16          // Sample windfield to get velocity in the snow particles position
```

```
17          float_3 v_drag, tpos;
18          tpos.x = pos.x*convert[0];
19          tpos.y = pos.y*convert[1];
20          tpos.z = pos.z*convert[2];
21
22          /* Perform trilinear interpolation. Code omitted */
23
24          // Subtract previous particle velocity
25          // from current sampled drag velocity
26          v_drag.x -= vel.x;
27          v_drag.y -= vel.y;
28          v_drag.z -= vel.z;
29
30          float v_fluid_abs = sqrtf(v_drag.x*v_drag.x +
31                                    v_drag.y*v_drag.y + v_drag.z*v_drag.z);
32          v_drag.x *= v_fluid_abs*gravity*vel.w;
33          v_drag.y *= v_fluid_abs*gravity*vel.w;
34          v_drag.z *= v_fluid_abs*gravity*vel.w;
35
36          float ax = v_drag.x;
37          float ay = -gravity + v_drag.y;
38          float az = v_drag.z;
39
40          // Calculate rotation
41          float rot_omega = omegas[i & 31];
42          pos.w += rot_omega*deltatime;
43          rot_omega *= radiuses[i & 31];
44          float v_abs = fmax(sqrtf(vel.x*vel.x + vel.y*vel.y + vel.z*vel.z), 1.0f);
45          rot_omega *= v_fluid_abs / v_abs;
46          float v_circ_x = -sinf(pos.w) * rot_omega;
47          float v_circ_z = cosf(pos.w) * rot_omega;
48
49          // Mult with 0.2 to slow down the snow
50          pos.x += ((vel.x+v_circ_x)*deltatime + 0.5f*ax*deltatime*deltatime)*0.2f;
51          pos.y += (vel.y*deltatime + 0.5f*ay*deltatime*deltatime)*0.2f;
52          pos.z += ((vel.z+v_circ_z)*deltatime + 0.5f*az*deltatime*deltatime)*0.2f;
53
54          // Calculate new velocity
55          vel.x += ax*deltatime;
56          vel.y += ay*deltatime;
57          vel.z += az*deltatime;
58
59          part_vel[i] = vel;
60
61          // Reset particle y coord if its below -1.0f (below the sim scene)
62          // Also reset the position if its above the simulation scene
63          if (pos.y < -1.0f) pos.y = SCENE_Y-2.0f;
64          else if (pos.y > SCENE_Y) {
65              // Reposition
66              int s = abs(LCG(seeds[i]));
67              pos.x = (float)s*(1.0f/RANDMAX)*SCENE_X;
68              s = abs(LCG(s));
69              pos.z = (float)s*(1.0f/RANDMAX)*SCENE_Z;
70              seeds[i] = s;
71              pos.y = SCENE_Y-2.0f;
72
73              float_3 wind, tpos;
74              tpos.x = pos.x*convert[0];
75              tpos.y = pos.y*convert[1];
76              tpos.z = pos.z*convert[2];
77
78              /* Perform trilinear interpolation. Code omitted */
```

```
79
80              part_vel[i] = make_float_4(wind.x, wind.y, wind.z, vel.w);
81          }
82
83          int moved = 0;
84
85          // Check if particle has exited the scene at the X and Y edges.
86          // if it has we respawn it at the other side of the volume.
87          if (pos.x < 0.0f) {
88              pos.x += SCENE_X;
89              moved = 1;
90          }
91          else if (pos.x > SCENE_X) {
92              pos.x -= SCENE_X;
93              moved = 1;
94          }
95          if (pos.z < 0.0f) {
96              pos.z += SCENE_Z;
97              moved = 1;
98          }
99          else if (pos.z > SCENE_Z) {
100             pos.z -= SCENE_Z;
101             moved = 1;
102         }
103
104         // Get the current snow height from the terrain buffer
105         int_2 ip = calcGridPos(pos, tdim);
106         float_4 hv = grid[(tdim * ip.y) + ip.x];
107         float h = hv.y + hv.w;
108
109         // Check if y coordinate of partcile is below the
110         // terrain + snow buildup
111         if (pos.y < h) {
112             // Check to see if particle has exited the edges of the volume
113             // if it has we dont want to add snow to the terrain
114             if (moved) {
115                 // Reposition
116                 int s = abs(LCG(seeds[i]));
117                 pos.x = (float)s*(1.0f/RANDMAX)*SCENE_X;
118                 s = abs(LCG(s));
119                 pos.z = (float)s*(1.0f/RANDMAX)*SCENE_Z;
120                 seeds[i] = s;
121                 pos.y = SCENE_Y-2.0f;
122
123                 float_3 wind, tpos;
124                 tpos.x = pos.x*convert[0];
125                 tpos.y = pos.y*convert[1];
126                 tpos.z = pos.z*convert[2];
127
128                 /* Perform trilinear interpolation. Code omitted */
129
130                 part_vel[i] = make_float_4(wind.x, wind.y, wind.z, vel.w);
131             }
132             else if (ip.x != tdim-1 && ip.y != tdim-1) {
133                 // The snow particles Y coordinate is below the terrain and we
134                 // are inside the terrain dimensions. Therefore the snow particle
135                 // has hit the ground and we need to add snow to the terrain.
136
137                 incrementHeight(ip.x, ip.y, 0.15f, grid, snow_growth, tdim);
138
139                 incrementHeight(ip.x+1, ip.y, 0.095f, grid, snow_growth, tdim);
140                 incrementHeight(ip.x-1, ip.y, 0.095f, grid, snow_growth, tdim);
```

```
141                 incrementHeight(ip.x, ip.y+1, 0.095f, grid, snow_growth, tdim);
142                 incrementHeight(ip.x, ip.y-1, 0.095f, grid, snow_growth, tdim);
143
144                 incrementHeight(ip.x+1, ip.y+1, 0.059f, grid, snow_growth, tdim);
145                 incrementHeight(ip.x+1, ip.y-1, 0.059f, grid, snow_growth, tdim);
146                 incrementHeight(ip.x-1, ip.y+1, 0.059f, grid, snow_growth, tdim);
147                 incrementHeight(ip.x-1, ip.y-1, 0.059f, grid, snow_growth, tdim);
148
149                 incrementHeight(ip.x+2, ip.y, 0.026f, grid, snow_growth, tdim);
150                 incrementHeight(ip.x-2, ip.y, 0.026f, grid, snow_growth, tdim);
151                 incrementHeight(ip.x, ip.y+2, 0.026f, grid, snow_growth, tdim);
152                 incrementHeight(ip.x, ip.y-2, 0.026f, grid, snow_growth, tdim);
153
154                 incrementHeight(ip.x+2, ip.y+1, 0.015, grid, snow_growth, tdim);
155                 incrementHeight(ip.x+2, ip.y-1, 0.015, grid, snow_growth, tdim);
156                 incrementHeight(ip.x-2, ip.y+1, 0.015, grid, snow_growth, tdim);
157                 incrementHeight(ip.x-2, ip.y-1, 0.015, grid, snow_growth, tdim);
158                 incrementHeight(ip.x+1, ip.y+2, 0.015, grid, snow_growth, tdim);
159                 incrementHeight(ip.x+1, ip.y-2, 0.015, grid, snow_growth, tdim);
160                 incrementHeight(ip.x-1, ip.y+2, 0.015, grid, snow_growth, tdim);
161                 incrementHeight(ip.x-1, ip.y-2, 0.015, grid, snow_growth, tdim);
162
163                 incrementHeight(ip.x+2, ip.y+2, 0.0037, grid, snow_growth, tdim);
164                 incrementHeight(ip.x+2, ip.y-2, 0.0037, grid, snow_growth, tdim);
165                 incrementHeight(ip.x-2, ip.y+2, 0.0037, grid, snow_growth, tdim);
166                 incrementHeight(ip.x-2, ip.y-2, 0.0037, grid, snow_growth, tdim);
167
168
169                 // Reposition
170                 int s = abs(LCG(seeds[i]));
171                 pos.x = (float)s*(1.0f/RANDMAX)*SCENE_X;
172                 s = abs(LCG(s));
173                 pos.z = (float)s*(1.0f/RANDMAX)*SCENE_Z;
174                 seeds[i] = s;
175                 pos.y = SCENE_Y-2.0f;
176
177                 float_3 wind, tpos;
178                 tpos.x = pos.x*convert[0];
179                 tpos.y = pos.y*convert[1];
180                 tpos.z = pos.z*convert[2];
181
182                 /* Perform trilinear interpolation. Code omitted */
183
184                 part_vel[i] = make_float_4(wind.x, wind.y, wind.z, vel.w);
185             }
186         }
187         // set particle position for next timestep
188         part_pos[i] = pos;
189     }
190 }
191 }
```

Listing A.8: Particle movement kernel in the snow particle simulation. The code performing trilinear interpolation has been omitted for brevity.

Smooth Ground

```
1 */ Smooth snow cover using a stencil operation on the terrain VBO */
```

```
2  void smooth_ground(float_4 *grid, int tdim) {
3  #pragma acc kernels deviceptr(grid) copyin(tdim)
4  {
5      #pragma acc loop independent
6      for (int y = 1; y < tdim-1; ++y) {
7          #pragma acc loop independent
8          for (int x = 1; x < tdim-1; ++x) {
9
10             float_4 curr,xdirleft,xdirright,ydirleft,ydirright;
11
12             curr = grid[tdim*y+x];
13             xdirleft = grid[tdim*y+(x-1)];
14             xdirright = grid[tdim*y+(x+1)];
15             ydirleft = grid[tdim*(y-1)+x];
16             ydirright = grid[tdim*(y+1)+x];
17
18             float diff = 0.0f;
19             float mod = 0.0f;
20
21             // Positive xdir
22             diff = (xdirright.y + xdirright.w)-(curr.y+curr.w);
23             if (fabs(diff) > 0.05f) {
24                 if (diff < 0) {
25                     if (curr.w > 0.1f) {
26                         float snow = curr.w;
27                         if (-diff < snow)
28                             snow = -diff;
29                         mod -= 0.05f * snow;
30                     }
31                 }
32                 else {
33                     if (xdirright.w > 0.1f) {
34                         float snow = xdirright.w;
35                         if (diff < snow)
36                             snow = diff;
37                         mod += 0.05f * snow;
38                     }
39                 }
40             }
41
42             // Positve ydir
43             diff = (ydirright.y + ydirright.w)-(curr.y+curr.w);
44             if (fabs(diff) > 0.05f) {
45                 if (diff < 0) {
46                     if (curr.w > 0.1f) {
47                         float snow = curr.w;
48                         if (-diff < snow)
49                             snow = -diff;
50                         mod -= 0.05f * snow;
51                     }
52                 }
53                 else {
54                     if (ydirright.w > 0.1f) {
55                         float snow = ydirright.w;
56                         if (diff < snow)
57                             snow = diff;
58                         mod += 0.05f * snow;
59                     }
60                 }
61             }
62
63             // Negative xdir
```

```
64              diff = (xdirleft.y + xdirleft.w)-(curr.y+curr.w);
65              if (fabs(diff) > 0.05f) {
66                  if (diff < 0) {
67                      if (curr.w > 0.1f) {
68                          float snow = curr.w;
69                          if (-diff < snow)
70                              snow = -diff;
71                          mod -= 0.05f * snow;
72                      }
73                  }
74                  else {
75                      if (xdirleft.w > 0.1f) {
76                          float snow = xdirleft.w;
77                          if (diff < snow)
78                              snow = diff;
79                          mod += 0.05f * snow;
80                      }
81                  }
82              }
83
84              // Negative ydir
85              diff = (ydirleft.y + ydirleft.w)-(curr.y+curr.w);
86              if (fabs(diff) > 0.05f) {
87                  if (diff < 0) {
88                      if (curr.w > 0.1f) {
89                          float snow = curr.w;
90                          if (-diff < snow)
91                              snow = -diff;
92                          mod -= 0.05f * snow;
93                      }
94                  }
95                  else {
96                      if (ydirleft.w > 0.1f) {
97                          float snow = ydirleft.w;
98                          if (diff < snow)
99                              snow = diff;
100                         mod += 0.05f * snow;
101                     }
102                 }
103             }
104
105             grid[tdim*y+x].w += mod;
106         }
107     }
108 }
109 }
```

Listing A.9: Ground smoothing kernel.