Enhancing and Porting the HPC-Lab Snow Simulator to OpenCL on Mobile Platforms

Frederik Magnus Johansen Vestre

Master of Science in Computer Science
Submission date: June 2012
Supervisor: Anne Cathrine Elster, IDI

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

Problem description

This project builds on the snow simulator developed by current and previous graduate students at the NTNU HPC-Lab. The work will include porting the code so that it works on current and future mobile platforms. Improved rendering techniques and other enhancements will also be considered.

Abstract

English

Porting a computationally demanding CUDA application to a GPU designed for mobile phones and tablets, which supports OpenCL, is the subject of this thesis.

Significant effort is made to prepare the snow simulator of the HPC-LAB at IDI, NTNU, for porting to an OpenCL capable GPU for mobile phones, with a reasonably limited effort, when it arrives. The snow simulator is ported to OpenCL, documented, and improved by considering multiple sorting algorithms, as well as sorting the snow particles.

A thorough study of GPUs for mobile devices and high performance computing, as well as their history is conducted to serve as a background for future porting of the simulator.

The core code resulting from the OpenCL port is documented in detail to prepare for future projects on completing the port to a mobile device.

The OpenCL port of the snow simulator is tested on a range of different OpenCL implementations. The performance of GPUs designed for different use is compared, and memory management is identified as the biggest bottleneck for performance. This bottleneck is further investigated by studying the performance of the simulator when disabling certain copy operations.

OpenCL supports more than just GPU devices. CPUs, Cell processors, and other acceleration cards are also supported. To investigate OpenCL on other devices, a part of the simulation is executed on a CPU, and compared with executing it on a GPU. The CPU version perform on par with the GPU version when using a laptop GPU.

Norwegian

Porting av en CUDA-applikasjon som krever høy ytelse, til en GPU designet for mobiltelefoner og nettbrett er tema for denne oppgaven.

I oppgaven gjøres det en betydelig innsats for å legge til rette for at snøsimulatoren til tungregningslaboratoriet på IDI, NTNU, med forholdsvis begrenset innsats, kan portes til en GPU for mobiltelefoner og nettbrett som støtter OpenCL, når den blir tilgjengelig. Snøsimulatoren er tilpasset OpenCL, dokumentert og forbedret ved å vurdere ulike sorteringalgoritmer, samt å sortere snøpartiklene.

Som bakgrunn for fremtidige tilpasninger av simulatoren, er det gjennomført en inngående studie av GPUer for mobile enheter, og tungregning, samt deres historie.

Kjernekoden som kommer fra OpenCL-tilpasningen, er dokumentert i detalj for å legge til rette for fremtidige prosjekter, og forberede dem på å gjennomføre tilpasningen av simulatoren til en mobil enhet.

Tilpasningen av simulatoren til OpenCLer testet på en rekke forskjellige OpenCL-implementasjoner. Ytelsen til GPUer som er designet for forskjellig bruk er sammenlignet, og minnehåndtering er utpekt som den største flaskehalsen for ytelse. Denne flaskehalsen er undersøkt ytterligere ved å studere ytelsen til simulatoren, når visse kopieringsoperasjoner er deaktivert.

OpenCL støtter mer enn kun GPUenheter. CPUer, Cellprossesorer, og andre akselerasjonskort støttes også. For å undersøke OpenCL på andre enheter, ble en del av simuleringen kjørt på en CPU, og sammenlignet med å gjøre den på en GPU. CPU versjonen yter på samme nivå so GPU versjonen når den blir kjørt på en laptop GPU.

Acknowledgements

I would like to thank Dr. Anne C. Elster for supervising this thesis. I would like to thank Jan Rovde for looking at all sorts of strange behaviour conducted by the snow simulator and giving hints of their reasons. Many thanks go to Rune Jensen for looking at my code, helping my back on the right track during debugging and suggesting ways to optimize the simulator. I would like to thank the Norwegian demo scene community at #scene.no at the EFNet Internet Relay Chat (IRC) server for many interesting discussions and viewpoints of mobile GPU computing and the history of mobile phones in general.

I would also like to thank ARM for cooperation regarding mobile GPU access, and NVIDIA for donating the GPUs in the NTNU/IDI HPC-Lab through Dr. Elster's membership in their Professor Partnership program.

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface. 1, 2, 10, 12, 15, 16, 18, 20, 21, 27, 28, 35, 37, 50, 56, 62

CAD Computer Aided Design. 53

CPU Central Processing Unit. 1, 7, 8, 10, 16, 17, 20, 23, 49–52, 61

CTM Close To the Metal. 16

CUDA Compute Unified Device Architecture. 1–3, 15–21, 23, 25, 26, 28, 35–37, 42, 43, 51, 56, 58, 61, 62

DAG Directed Acyclic Graph. 26

DMA Direct Memory Access. 20

ES Embedded Systems. 10, 11

FPS Frames Per Second. 50, 51, 53–55

GLSL the openGL Shading Language. 12, 15, 16, 21

GPGPU General purpose graphics Processing Unit. 1–3, 7, 11, 15–18, 20, 26–28, 35, 50–55, 62

GPU Graphics Processing Unit. 1–3, 5–13, 15–21, 23, 25–28, 31, 35, 39–42, 49–51, 53, 54, 61, 62

GS Gauss-Seidel. 32

HPC High Performance Computing. 1–3, 15, 23

IDE Integrated Development Environment. xix

IDI Department of Computer and Information Science. 2

IP Intellectual Property. 7–9

IRC Internet Relay Chat. vii

LBM Lattice Boltzmann methods. 23, 25, 30

NDA	Non disclosure agreement. 12
NTNU	Norwegian University of Science and Technology. 2
SDK	Service Development Kit. 42, 50, 51
SIMD	Single Instruction Multiple Data. 17
SM	Streaming multiprocessor. 17, 35
SoC	System on a Chip. 7–9
SOR	Successive Over Relaxation. 25, 32
SPMD	Single Program Multiple Data. 17, 40
VBO	Vertex buffer object. 28, 30, 37
VCS	Version Control System. 25, 26

Glossary

AMD Advanced Micro Devices: A semiconductor company that develops computer processors and related technologies for commercial and consumer markets. In 2006 AMD bought ATI, a GPU vendor. 16, 20, 21, 42, 43, 50, 51

ARM ARM: Maker of processor IP with near monopoly for processing in advanced mobile devices. 2, 3, 7, 10, 12, 13

Direct X A graphics API from Microsoft which mainly is used for programming games on the Windows platform. 9

Git A distributed version control system for code. 2

NVIDIA A semiconductor company that develops GPUs and related technologies for commercial and consumer markets. Currently largest in both markets. 1, 8, 16, 21, 25, 42, 51–53

OpenCL Open Computing Library is a standard specification defining a cross-platform API for functions that are specialized for running on highly parralell hardware, e.g a Graphics Card. 1–3, 8, 15–21, 25, 26, 28, 34–37, 42, 43, 49–52, 56, 58, 61, 62

OpenGL Open Graphics Library is a standard specification defining a cross-language, cross-platform API for writing applications that produce 2D and 3D computer graphics. It is among other things widly used for scentific visualization. 9–12, 15, 20, 28, 30, 35, 50, 52, 53, 55, 56

Silicon Graphics Inc. SGI was a manufacturer of high-performance computing solutions, including computer hardware and software. Its initial market was 3D graphics display terminals, but its products, strategies and market positions evolved significantly over time. 15

XNA A quite high level coss platform archictecture and Integrated Development Environment (IDE) for writing games on Microsoft platforms. Originally an acronym for Xbox New Architecture, but recently "XNA's Not Acronymed". 9

Chapter 1

Introduction

High Performance Computing (HPC) has traditionally been exclusive for large organization with lots of resources. However, computing performance has always been increasing. During the last decades the consumer marked for computers have exploded. Due to the economics of scale, developing hardware for consumers gives better return on investment than developing specialized hardware for demanding professionals. Consumer technology has therefore formed the basis of HPC solutions in the last decade [DSSS05].

General purpose graphics Processing Unit (GPGPU) is the means of using a Graphics Processing Unit (GPU) to perform computation in applications traditionally handled by the Central Processing Unit (CPU). The use of GPGPUs for HPC computing became feasible about 5 years ago with the introduction of a GPGPU Application Programming Interface (API) by NVIDIA called Compute Unified Device Architecture (CUDA). New forms of HPC computing became accessible for the general public as quit cheap GPUs which were originally designed to support computationally demanding graphics in computer games could now be used for demanding physical simulations and signal processing tasks.

The popularity of smartphones and tablets has created strong demand for relatively advanced graphics on those devices too. The programming interfaces of GPUs for mobile phones are inspired by the interfaces for desktop computers, and devices that supports one of interfaces for GPGPU programming on desktops, will be available for smartphones this autumn. The devices for smartphones have different underlying architecture and performance characteristics than desktop devices, even though the programming interface is the same. To gain maximum performance out of the devices, these difference have to be taken into consideration when designing or porting programs for smartphones or tablets.

1.1 Goals

The aim of this thesis is to port a computationally demanding program to the GPU of a tablet or mobile phone which supports OpenCL. By looking into porting a program which requires high performance to a performance constrained device insight into how the mobile platforms work, how powerful the platforms are, and what kinds of quirks the they inhibit will hopefully be gained.

The HPC lab at Department of Computer and Information Science (IDI), Norwegian University of Science and Technology (NTNU) started to develop a snow simulator in 2006. The simulator as it is today, is a result of several thesis and has been used to experiment with different models for physical phenomena, and do research on how to get maximum performance from GPUs. The simulator already had been tuned for a desktop GPU. It was therefore a good candidate for porting to a mobile device, since the experiences of tuning it to the mobile device can be compared to the experiences from the desktop GPU tuning.

1.2 Problem description

> “This project builds on the snow simulator developed by current and previous graduate students at the NTNU HPC-Lab. The work will include porting the code so that it works on current and future mobile platforms. Improved rendering techniques and other enhancements will also be considered.”

Figure 1.1: Thesis assignment

The assignment for this master thesis is quoted in Figure 1.1. The goals for the assignment was suggested by the student, but the final text was written by the advisor. The main focus of the assignment is to port the snow simulator to mobile platforms, getting it working on the platforms, and gather experience from that.

Because mobile platforms which contained hardware necessary to run the snow simulator were not available for the market yet when this thesis was written cooperation with the vendors of the graphics solution was required. The IDI HPC lab at NTNU has contact with ARM which design a mobile GPU which are going to support OpenCL. This thesis, however, was not initiated by ARM. Therefore some uncertainty existed about the feasibility of the project.

1.3 Project description

One of the API which future mobile GPUs will support is OpenCL. The snow simulator was written using another GPGPU API, named CUDA. An earlier student had ported an older version the snow simulator from CUDA to OpenCL a specialization project. The ported code had not been touched in two years, and the simulator has changed much since. Therefore the first part of the project was to get the code up running and up to date with the last changes of the simulator.

A serious effort was done in to obtaining the resulting code from the OpenCL port. However no archived code was found at the HPC lab. Contacting the student who did the port was attempted, but alas no contact could be made. Since the old code was no longer available it was decided to redo the port from scratch using the most up to date simulation code.

To make sure that future students would be able to not experience these problems a proper version control system was introduced, and a group for the whole HPC-lab, including my advisor was created at the servers of the department. The Git version control system was selected to enable students to work with their assignments easily in separate branches; using local repositories if necessary. When each branch is merged at the end of the semester this will produce a common base for future students to work from, and a history where it is possible to see which projects which are included in which version.

Porting the simulator to OpenCL was done in parallel with lobbying to get access to the mobile hardware. The first step of getting access to mobile hardware supporting OpenCL was to do a survey to discover which vendors which were designing OpenCL capable hardware. After browsing the internet for press releases and product specifications 5 vendors were discovered. An application to the ZiiLabs OpenCL Early Access Program was submitted, but did not result in any response.

My advisor did also contact the graphics division of ARM which promised to try to get a Mali sample to work on. The faculty has a good relationship with ARM, which have done several thesis together with the faculty earlier, and even did some thesis in cooperation with the faculty this term. An informal agreement was made to borrow prototype hardware from ARM, but the hardware did not arrive in time to be useful for this thesis.

Even if the hardware necessary for running the snow simulator on mobile targets did not arrive porting of the simulator to OpenCL was continued. When the port reached a runnable and physically stable state attention was divided to focus on optimizing and refining the rendering of the snow simulator.

The development of video game consoles, and personal computers have influenced how high performance computing is done. The research and development cost for new processors like the CELL processor has been divided between scientific HPC use and developing the Playstation 3 game console.

Graphic cards for gaming have surpassed the computing power of the central computing unit of personal computers several years ago. In recent years it has been a revolution in the programming flexibility of graphic cards for personal computers. When tasks are computationally intensive and possible to solve in parallel, GPGPU frameworks such as CUDA and OpenCL are capable of exploiting the performance of the graphics card.

1.4 Outline

This thesis falls into three parts. In the first part, Chapters 1 to 3, an outline of the field is presented. The technologies required for the simulator and their history is examined. In Chapter two the motivation for porting the snow simulator to a tablet is presented. The programming interfaces and technical architecture of mobile GPUs are also presented. In Chapter three the history and characteristics of GPGPU is presented. An overview of the differences between CUDA, which the snow simulator is ported from, and OpenCL, which it is ported to, is given to put the port in context.

In the second part, Chapters 4 and 5, the porting, and ways of optimizing rendering is described. The description of the porting is contained in Chapter 4. It description is quite detailed to form a basis for further work by later students. Chapter 5 looks at how rendering performance can be optimized by sorting snow particles on the GPU.

In the last part, Chapters 6 and 7, the simulator is tested and the thesis is wrapped up in a conclusion. Chapter 6 concerns testing to profile the performance and visual results sported by the simulator. In this Chapter some characteristics of the simulator which could benefit from further optimization is also discovered. At last in Chapter 7 conclusions from this thesis are summarises, and ideas for future work are discussed.

Chapter 2

Mobile graphics computing units (GPUs)

This chapter gives an overview of how mobile phones and tablets have became an essential part of the computing experience for end users, and how this has affected the GPUs in these devices. Relevant standards and developments of mobile graphics are discussed. At the end of the chapter an introduction to the architectures of mobile GPUs are given.

2.1 The emergence of smartphones

The Apple iPhone revolutionized the phone marked for end users. It was an evolution of Apples digital music players. Therefore the focus of the phone was not calling but to be a mobile entertainment and communication device. The iPhone came with unlimited Internet usage. It was integrated in to the iTunes store. Small programs called Apps can be bought from the store very easily to extend the phones functionality. However the most important feature of the iPhone was its ease of use and focus on aesthetics, even if it has quite advanced futures. Many users had old accounts for iTunes used for buying music, which lowered the bar for buying Apps to the phone. Advanced Apps which pushed the hardware of the phone to the limits appeared creating demand for phones with even more performance. A web browser capable of browsing normal desktop web pages was also included.

The ubiquity of mobile devices does also open for interaction with the environment which is difficult for a fixed computer. Augmented reality is extending the real world with synthetic elements trough the use of technology. In practise this means capturing a live image of the real world on a camera, and compositing signs, images etc. on top of the image. To do this the image has to be placed in to the model of objects to be composed. This requires substantial processing power. Currently this is only possible for recognition with low frame rates [BOM11].

Mobile gaming is another driving factor for smartphone adaption. Mobiles are good devices for casual games because people always bring their smart phone with them. Smartphone games are much simpler than games for more powerful devices. Therefore they are cheaper to create. This allows more experimentation in game play because the cost of failure is lower. Game development has always been fast to push hardware to the limit, and is a central driver for more performance in mobile phones as well.

2.2 Phone inspired tablets

Traditionally tablet computers have been struggling to find a need to fill for the general consumer. They have been very expensive, and the user interface of the tablets was adapted from desktop computers which use a mouse and a keyboard as input tools. User interfaces designed to be operated by a mouse have widgets which require precise placement of the mouse pointer, such as small buttons, sliders etc. To gain the accuracy of a mouse on tablets styluses are required. Styluses however are easily lost, and cumbersome to use. As a consequence tablets based on desktop technology flopped. [JS09] describes the tablet marked, form factor and design goal as it was in 2009.

April the 3rd, 2010 the tablet marked was completely changed by the announcement of the Apple iPad [App10]. It created a mass marked for tablets by expanding the iPhone ecosystem. Instead of scaling down PC user interfaces Apple scaled up the relatively simple user interface used on their smartphones. Relatively cheap components originally made for mobile phones was used to make a more affordable tablet. The tablet was marketed as a hip fashion item.

The iPad was not the first tablet to focus on a simple interface for Internet consumption. Nokia for example released a tablet with the same use case in 2005 [Sha05]. However no integration with a similar ecosystem as the one supported by the iPhone was present.

Tablets and smart phones are also gaining markets from Nintendo GameBoy and Playstation portable, positioning themselves as a replacement for mobile game consoles for casual gaming. The games used for tablets are often quite small, cheap, and fast to make. Therefore a game can be created by one, or just a few persons. This lowers the bar of entry to the marked, and makes it possible to experiment and create innovative gameplay concepts.

Tablets and mobile are also expanding in to professional use cases. Professionals are expressing interest in using these devices in settings where personal computers are viewed as cumbersome and difficult to use. Examples include supporting doctors on a sick bed visit in an hospital, and using mobile phones to collect data from inspections of roads, railways etc.

2.3 The use cases of mobile GPUs

The mobile GPU has evolved from displaying simple graphics to do a range of heavy processing functions of visual data on the mobile phone. In this section several use cases of the modern mobile GPU is outlined.

The screen estate of mobile phones and to some extend tablets are very small. Therefore innovative user interfaces are required to use the screen space optimally. This have prompted a redesign of the user interfaces from phones from the ground. These interfaces uses fancy animations to make them attractive for the users. In the beginning the graphics accelerators for user interfaces supported acceleration of 2D graphics; creating smooth gradient, composing transparent images etc.

Movie playback has been seen as a obvious feature for smart phones, partially because it was supported on the iPod, which the iPhone evolved from. A video accelerator was required to enable video playback on a battery constrained device. This hardware has been obsoleted by the mobile GPU.

The demand for games is creating a demand for more processing power to do be able to cope with demanding graphics and effects. Simple 3D games are already supported on modern phones. This has caused mobile GPU's to focus on 3D graphics too.

Most smart phones and tablets have cameras. In the beginning images were acquired straight from the camera and transferred to social media platforms. To differentiate the applications offering these services additional features were included. Simple image processing effects were implemented, filters simulating analogue effects making images look worn have gained popularity. GPU makers have identified this as operations which may be benefited from GPU acceleration. Therefore GPGPU processing have been added to the road maps of nearly all mobile GPU vendors.

The market for tablets and smartphones has grown substantially over the last years, and is expected to grow still. The growing marked has released lots of capital for development of new devices, making tablets and smart mobile phones more powerful than desktops were merely 15 years ago. To keep up with the graphics performance expected of the devices, specialized graphics processors are included. Many different companies compete to create the best graphics processors. The graphics processors are released at horrendous speed, supporting more and more features and programming interfaces for each release.

2.4 Mobile device manufacturing ecosystem

Mobile smart phones and tablets are advanced devices which contains hardware and software produced by many different vendors. The relationships between these vendors are outlined in Figure 2.1. The relationship is split in to 4 horizontal layers along interfaces which are often interfaces between components delivered by different vendors. However some vendors handle all except the topmost layer themselves.

All modern, mobile advanced electronic devices consist of so called System on a Chip (SoC). The SoC integrate the most important functions of the device in to a single chip to reduce the distance between electronic components, and hence the power required to transfer data between the parts of the system. This chip consists of several subsystems. Designing such a subsystem is a extremely complex task requiring substantial work by highly skilled engineers. In addition software, like for example compilers and debuggers, has to be written to interface with the hardware; for example compilers and debuggers. Therefore most companies buy finished designs (called Intellectual Property (IP)), from other companies. Mobile CPU is the prime example of IP being licensed from a specialized firm. ARM is the leading company for supplying IP for mobile device CPUs, having over 95% of the market [Mor11]. Similarly some mobile GPUs can be licensed as IP, while others are only supplied as part of finished SoCs.

A mobile device manufacturer may buy finished SoCs from SoC manufacturers and integrate them into their devices (as in Figure 2.1 layer 3). The mobile device manufacturer designs the complete phone and write software for the phone. The software is based on modules licensed from the SoC manufacturer. In this layer the device producer have to take care of selecting chips and components that can be integrated with the software they want to use on the device.

The operating system for Android phones and tables is provided by Google, but the device manufacturers customise it by adding their special software, and drivers for their hardware.

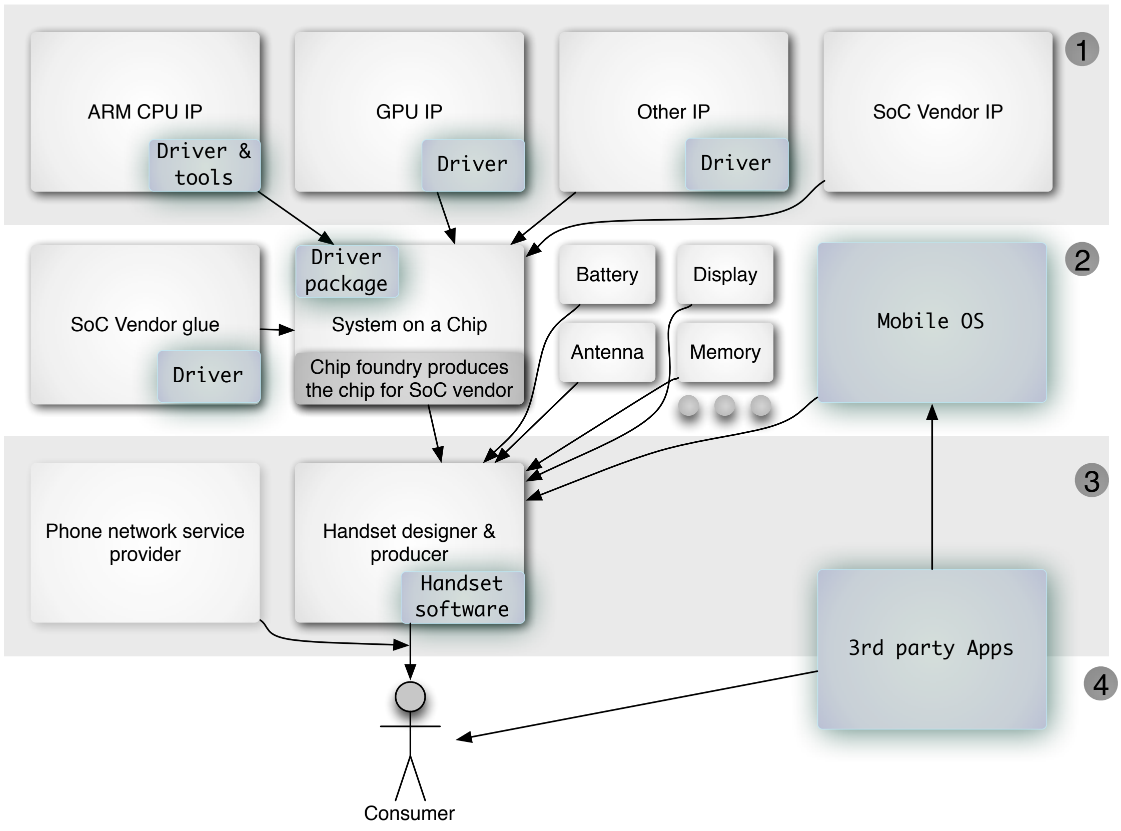


Figure 2.1: Stakeholders of mobile standardization & elements of a mobile device

App developers interface the operating system of the devices to create apps. The apps are then sold directly to the consumer in a store controlled by the operating system vendor. These developers are partly consumers themselves, partly selling to the consumer market. Therefore they are located between layer 3 and 4 in Figure 2.1.

2.5 Mobile GPU market

The mobile GPU marked is a healthy marked with four large competitors. An overview of these competitors is given in Table 2.1. A fifth competitor; Zii Labs does also produce mobile GPUs but no products for end users are available from normal outlets on the Internet. All vendors except NVIDIA have announced roadmaps which claims GPUs with OpenCL support will be released in Q4 2012 or Q1 2013. The vendors of CPUs which uses a IP based business model have finished the IP for GPU models which support OpenCL already. However no SoCs using this IP is quality assured and ready to be released in consumer products at this moment. The IP vendors will not have finished the drivers for the GPUs yet either.

[1]Tegra4 rumored to support GPGPU [Val12]

Vendor	CL model	BM	Hardware examples
AMD Mali	T604	IP	SmartIQ, Samsung galaxy: tab 7.7, phones 2 & 3
Qualcomm Adreno	3xx	SoC	HP touchpad tablet, HTC EVO phone
Power VR SGX	Series 6 [Ima12]	IP	iPhone, iPad, Nexus S
Nvidia Tegra	Tegra 4[1]	SoC	Thinkpad tablet, EEE-Pads, Galaxy tab 10.1

Table 2.1: Leading hardware GPUs for mobiles and tablets

2.6 State of standardisation

The smartphone device platform is a quite standardized platform for developers. When it comes to interfacing with the operating system of mobile phones the success of the Apple iPhone created a marked of critical size for apps. Other handset makers answered this by creating a common operating system for their phones. This lead to a big enough marked share for some apps to be released for both platforms.

Lately Microsoft have joined the marked with their Windows based phones. However the number of different platforms is still quite limited for app developers who want to reach all smartphone users. The availability of apps has a considerable influence of whether it is possible for new platforms to survive in the marked. Microsoft did actually pay several companies to create apps for their phone before it was launched, to create a critical mass of apps so that consumers would be interested in buying smartphones with their operating system.

App developers who target smartphones, encounters the same lack of formal standardization of app platforms, as developers targeting the desktop computer market. A few operating systems dominates the market and their interfaces act as de facto standards. This is a typical development of a mature marked where no standards exist. When it comes to mobile GPUs, their programming interfaces mirror the interfaces used on PC's. They are however modernized, and scaled down to fit the needs of mobile devices.

Both Google and Apple use operating systems based on OpenGL on their desktops. Therefore their phone operating systems also use OpenGL. There are several mobile GPU manufacturers, but only two big operating systems have existed on mobile phones. Therefore all GPU vendors have to implement the GPU standards required by the mobile OS manufacturers.

Microsoft however uses their own Direct X technology on the desktop. Therefore the Windows phone operating system does not use OpenGL, but a scaled down version of Direct X with the XNA framework on top to simplify the creation of games. Consequently all graphics code has to be rewritten to interface Direct X or XNA, when ported from another smartphone platform. This may stop some vendors from porting their application to Microsoft phones, if the marked for their application is not believed to be big enough. However code application can be written cross platform for all platforms controlled by Microsoft. This may provide the required marked share for porting applications by tapping into the marked share of desktop users.

Standardization is an important enabler for interoperability between different solutions, and often a sign of a mature ecosystem. Leading vendors will often try to inhibit standardisation to maintain their leading position, and to make it more difficult for competitors to compete with their products. History shows many examples of this. When graphics acceleration solutions were developed for desktop and mainframe markets several different interfaces for the systems existed. Each vendor would deliver a vertically integrated system from the hardware, with software to the user interface. For external developers several different programming interfaces had to be interfaced. The graphics computer industry continued in this state for many years until smaller vendors started to mimic bigger vendors, and one of the leading vendors (SGI) started to standardize their interface to get a head start with the upcoming standard (OpenGL) [CvDPH98].

Apple computing is the last vendor which continues to produce vertically integrated systems for personal computers. This is possible because they have targeted a niche marked and focused on exclusivity and design instead of price and performance. Apple have gained a substantial marked share among creators of creative content like movies and graphics on desktop systems. In the recent years however they have shifted towards using standardized hardware components, and then assemble finished products. Vertical integration does however limit the number of combination of finished components which simplifies software development and testing of different hardware combinations.

Apple has transfered the recipe used on desktop system to music players, and then mobile phones with the iPhone. This proved to be very successful and positioned apple as a marked leader in mobile phones. The phone did also use standardized hardware components, but the software was tightly controlled, but quite easily extendible with applications (apps). The first major competitor to the iPhone was provided by Google with the Android operating system. The Android operating system counters the tightly controlled iPhone with openness setting an unprecedented standard of openness in the mobile handset marked. This is a classical way to counter tightly controlled marked leading solutions. Several competing manufacturers were collaborating to create a common platform which made programs interoperable between vendors. To differentiate themselves, they included their own branding, applications and hardware.

The development of mobile platforms has been driven by the vendors of the complete systems, not the makers of the underlying technology. The vendors of the complete systems want to be able to change components simply and without changing their code base. For Android support of several different underlying hardware has been essential to the philosophy of their operating systems. This has led to standardization of all hardware interfaces. Hardware component vendors, who do not implement standards, do not stand a chance in the marked. The hardware vendors are forced to compete on implementing the standards as cheap as possible with the highest possible performance, instead of locking each other out with incompatible interfaces. Most of the standards of GPUs are exposed to the programmers of apps for the phones and tablets. For the app programmers being able to use standards increase the number of potential devices can run their code, and hence the marked potential of the apps.

Currently ARM has a near monopoly on the designs of the CPU architecture used in mobile phones. Therefore their instruction set has emerged as a de facto standard for CPU programs on mobile phones.

GPUs and graphics accelerators however are provided by several different vendors who implement the graphics rendering a bit differently. Therefore a common API is presented to the programmers. The first API which was produced was the OpenGL Embedded Systems (ES) [Gro04] in 2004. This API was created by the Khronos Group, which is the proprietor of OpenGL. Soon Khronos released several other specifications for other graphics and media purposes. These standards are listed in Table 2.2.

Standard name	Description of area
OpenGL ES	Embedded version of OpenGL 3D graphics API
OpenVG	For 2D Vector Graphics to accelerate Flash, SVG & 2D interfaces
OpenCL	For GPGPU processing
OpenSL ES	For music and sound playback and recording
OpenMAX	For audio/video-playback, 3 layers: AL, IL & DL

Table 2.2: Khronos standards applicable for mobile computing

Many of these standards are implemented in the majority of GPUs in mobile phones today. OpenGL ES is supported in one version or another on nearly all phones already, OpenVG does also enjoy widespread support on mobile GPU's. OpenCL is announced in future models by several vendors, and OpenMAX is going to be integrated with future Android API's.

2.7 Open source graphics drivers

Traditionally mobile phones have been rather closed devices. After the introduction of smartphones some flexibility has been introduced. However this flexibility has been confined to virtual machines with extensive sandboxing. This is very good for traditional end users which want a phone which can do some fancy stuff, but where no apps can interfere with the basic usage of the phone.

However there have always been enthusiasts who want do improve their products. Hardware vendors guard their trade secrets from competitors, and others by only shipping phones with compiled binary drivers with the phones. This stops enthusiasts from improving the lower layer of the phone operating system. Because the operating systems of smartphones are based on quite open Unix operating systems, which then are locked down some mobile devices, they have been hacked (also called rooted, or jailbraked). A hacked phone can be changed by the user. For Android devices the operating system is based on Linux, which is Open Source. The GPU drivers however are often binary modules without any source available. This hinders optimization of the drivers, and cause incompatibility with newer version of the kernel. Therefore some developers have started reverse engineering the drivers for some mobile GPUs.

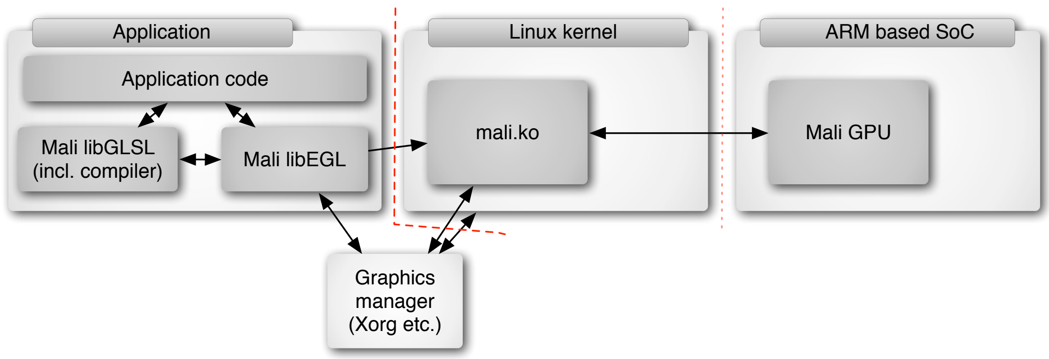


Figure 2.2: Interfaces intercepted by Lima for reverse engineering, based on info from [Ver12]

Currently two projects for reverse engineering GPUs are in development. Both projects are intercepting communication between the driver and hardware to figure out how the hardware works. The most mature project is the Lima driver for ARM Mali GPU's. The project has just demonstrated a working driver for simple OpenGL example scenes consisting of cubes. Figure 2.2 shows how a mobile GPU driver is integrated with the surrounding system on a Linux platform. The red lines symbolizes the interfaces which were monitored during the reverse engineering of the Lima driver. At the moment it looks like most of the focus of the drivers is to implement the openGL Shading Language (GLSL) support for programming the GPU.

The other project created to make open source mobile GPU drivers is named Freedreno. This driver should drive the Adreno GPUs from Qualcomm. 2D acceleration is supposed to work. The driver is created by an employee in the GPU section of Texas Instruments. The employee started his own driver because of fears concerning Non disclosure agreement (NDA)s for the Mali GPUs. Because this project still is young the success of this project is very uncertain.

Open Source Graphics driver does also benefit academic work. Details on how the GPU's work which is difficult to figure out are documented by the driver projects. Even more detail is available by reading the source code produced by the projects. This may for a basis for research in to low level graphics development outside the graphics companies.

The architecture of current open source drivers is driven by a modular approach where reuse is promoted. The common code for GPU drivers is developed as a part of the Mesa [Mes] open source OpenGL implementation by the [WMV] project. The main developers of the Gallium project is currently employed by the visualization firm VMWare which uses the drivers for their visualization solutions.

Because the development of the drivers is not done as part of the marked competition between GPU vendors, it may be possible to enhance the performance of GPUs with less efficient drivers. The reuse of modules does in effect also provide a lower level abstraction of the GPUs than the current standardized APIs. This makes it easier to experiment with other APIs and interfaces. In the Linux world several replacements for the dominating X Window System have been developed over the years. However getting good driver support has always been an issue. Binary drivers which interface with the X Window System cannot be used by replacements, and getting vendors to support experimental projects which produce no benefit for the vendors, is very difficult.

2.8 The architecture of mobile GPUs

The architecture of mobile GPUs are quite different from normal desktop and laptop GPUs. This is mainly because of the power envelope each GPU type is designed for. A mobile GPU is designed to run off a small battery and need to use as little power as possible. GPUs for mobiles use very low voltages internally to conserve power. These voltages are too small to be used to drive external buses. Therefore higher voltages have to be used for buses. This means that external transfers are little power efficient. This is the most important premise for the design of mobile GPUs. [AMS08]

Two of the most popular mobile GPUs (PoverVR & Mali) uses a tiling algorithm for rendering [AMS08] [SKP10]. This algorithm has also been used in gaming consoles like the Xbox 360 [SKP10].

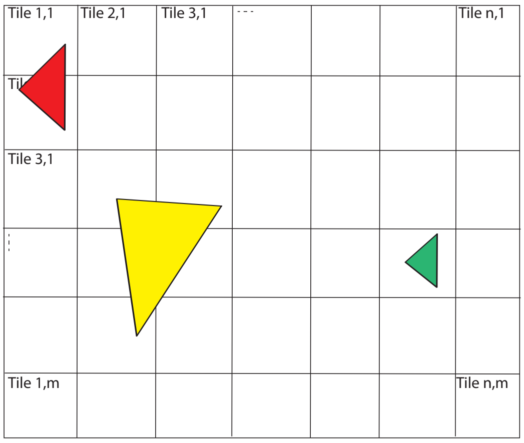


Figure 2.3: Triangles in a tile-grid

The tiling algorithms divide the screen in to tiles (of e.g. 16x16 pixels) as in Figure 2.3, with accompanying bins. Each tile is then rendered separately. The tiling algorithm requires less memory, but accesses the memory more often. The memory used in the algorithm is moved inside the graphics chip, as cache memory. This makes it power efficient, as no external bus transfers are required for access.

When a GPU scene is prepared for rendering, all triangles are transferred to the GPU. Then all triangles are transformed to their final positions. Afterwards each triangle is considered, and a pointer is added to the bin accompanying each tile the triangle intersects. The tiles can then be rendered in parallel.

2.8.1 PowerVR tile rendering

The PoverVR architecture is described in an overview [Pow09] and is used as a basis for the description of how each title is rendered.

The first step of rendering a tile is to Figure out which parts of which triangles which are visible. This is done by casting rays at each pixel, by specialized hardware. Afterwards the GPU Figures out which pixels share the same vertex to simplify shading and texturing. The shading and texturing done by PowerVR is very similar to desktop GPUs and was not described in the overview.

2.8.2 Mali tile rendering

I was not able to find detailed documentation of the ARM Mali processor architecture at the ARM web page, but the general optimization guide [ARM11] hinted of a tiling architecture similar to PoverVR. However I found a marketing article [Fal06] from Falanax, the firm which designed Mali before they were acquired by ARM. This article outlines the architecture.

The Mali architecture tries to be a hybrid between (desktop) intermediate mode rendering and tile based rendering, to exploit the advantages of each algorithm. The z-termination algorithm seems to focus more on efficiency than accuracy, and typically eliminates approximately 50% of the occluded pixels.

Chapter 3

General purpose computing on graphical computing units

This chapter concerns general purpose computing on graphical computing units. It is intended to give a basic understanding on how a GPU executes general programs, and how problems have to be formulated to exploit the power of GPUs.

First a short overview of the history of GPUs is given, then modern GPUs are described generally. At last the details of the CUDA and OpenCL APIs are described, and how they differ.

3.1 High performance computing: The road to GPGPU

For the real time graphics industry 1992 was the year of standardization. Before that each vendor had it's own proprietary interface. At the time Silicon Graphics Inc. was the leader of the computer graphics market. When the market got more competitive and other vendors started to standardize the competing programming interfaces, Silicon Graphics Inc. opened, documented and standardized it's proprietary API and renamed it OpenGL. In effect this forced all other vendors to do catch up.

Originally graphical computing units were device with a fixed hardware architecture for rendering graphics. No programming of the units where possible. In February 2001 [NVI01], graphics pipelines became much more flexible by supporting programmable shaders. Shaders were made to do advanced lightning and produce more detailed surfaces. They are executed in parallel for each pixel or vertex in the models they shade. This parallelism can be exploited for doing HPC work by rendering the objects to off screen image buffers and sending the buffers back to the CPU.

GLSL [KBR04] which was standardized in 2003 [Boa03], introduced a high level language which made it quite easy to write shaders. GLSL has a syntax inspired by C and functions for doing geometric math functions. In the early days of GPGPU computing GLSL was the language of choice for doing computations on GPUs. Writing general purpose programs using GLSL requires some effort mapping problems into a graphic domain. All information used outside the shading kernels had to be mapped in to 2- or 3-dimensional textures with one or more colour channels. GLSL shaders are split up to multiple passes of the graphics pipeline. Early programmable GPU used different hardware for each pass. Therefore it was important to split the work between all the passes for optimal performance.

After GLSL started to be used for generic computation the vendors recognized the business potential of GPGPU. In 2006 the GPU both NVIDIA and AMD released solutions adapted for GPGPU programming. NVIDIA even released specialized graphics cards without screen connectors for GPGPU.

AMD wanted to give low level access to their GPUs and released Close To the Metal (CTM) [AMD]. CTM was an assembly-like language for AMD GPUs. Because CTM was very low level it was not used very much. This language is now defunct, and AMD has gone over to supporting OpenCL. NVIDIA released CUDA which is a more high level solution modelled. The language CUDA uses for code executed on the GPU is a combination of a generalization of GLSL and the syntax and level of hardware control characteristic of the C programming language.

3.2 History and benefits of OpenCL

The OpenCL standard was developed in 2008. Apple computer wanted to use GPUs in their operating system to increase performance of their systems. Apple has good experience challenging different vendors to provide the cheapest and best components for their computers. It is believed that it is the reason why CUDA was not chosen as the sole solution for GPGPU by Apple.

To standardize OpenCL Apple took the incentive [Mar08] to form a Khronos "Compute Work Group" to standardize OpenCL. AMD and NVIDIA joined the work group from the start. The standard finished very fast, in about 6 months, to be included in Apples operating system. Other implementations, and use outside Apples echo system needed, however, 2-3 years to get reasonably stable. It is still very easy to provoke the need for a reboot of a machine by passing argument which are slightly outside the specification to the drivers.

Even if OpenCL is driven by GPGPU computing and inspired by CUDA, it is designed with more than GPUs in mind. OpenCL is designed to be independent of hardware as long as it is highly parallel, and has quite low memory latency compared to for example clusters.

CPUs for desktop, laptop and mobile computers have also became parallel to increase performance, because heat has made it prohibitive to increase the clock frequencies of the processors. Both of the leading desktop CPU manufacturers provide OpenCL drivers for their CPUs. This means that in a few years, all modern computers will contain an OpenCL capable device. Computers which does not have high end GPUs, will be able to run OpenCL code, with less performance. Therefore no alternative implementations of code optimized using OpenCL is required to run the program even on low end hardware.

The OpenCL specification is also designed to work with state of the art processor architectures designed for high performance computing like the IBM Cell Blade servers. Some enthusiasts have also managed to run OpenCL programs on the Cell processors of a Playstation 3 gaming console, but that became impossible when Sony stopped supporting custom programs on the PS3. This makes OpenCL a very scalable and accessible API available on everything from mobile phones to supercomputers.

3.3 GPGPU architecture

In this chapter OpenCL terminology will be used. More information on the terminology and OpenCL versus CUDA terminology is detailed in Section 3.5.

GPUs are designed to do high performance computations. They are not fit do to information bookkeeping etc. Therefore device management, memory management, program allocation and invocation is done at an host processor, together with work not fit for GPGPUs.

3.3.1 Programming model

Modern GPUs are highly parallel computers. They can run over 128 different executions concurrently, normal state of the art desktop CPUs can run up to 8. They focus on computations and not logic.

Most programs for CPUs are serial or task parallel programs. These program split problems in to several tasks which are run concurrently, or multiple programs are ran concurrently on different cores for multitasking.

Work groups have common memory which can be accessed faster than global memory and independent of other work groups. This minimizes the speed penalty of synchronization, compared to using global memory. An iterative differential equation solver is a good example of using this organization. The matrix can be divided in to blocks, and then only the borders between blocks need global synchronization.

Graphics synthesis is generally a data parallel problem. The same operation is executed on millions of vertices and pixels. Therefore GPUs are designed to run data parallel programs. This parallelism is exposed to GPGPU programs in the form of kernels. The main idea of kernels is to expose the parallelism of for-loops. Instead of running a loop serially for each element of a problem the looping is managed by the GPU. The inner part of the loop is programmed in a kernel and executed concurrently by the GPU. If the kernel is executed more times than the GPU supports concurrently, the GPGPU framework will schedule multiple concurrent execution passes.

GPU does however need to do some different tasks concurrently. Therefore the concurrent processors are organized in three different levels of parallelism. In the upper level different type of work can be executed with out significant performance penalty.

The upper parallel processing level consists of elements named work groups. All work groups are executed on separate Streaming multiprocessor (SM) which are responsible for executing their group. No synchronization between work groups are available, except by calling a new kernel. The next level is work items. Each work item can operate on different addresses in memory, have separate registers and variables. All work items in one work group runs the same executions (called Single Program Multiple Data (SPMD) in Flynn's taxonomy). Diverging executions, for example produced by if-tests, have to be executed by all work items of a work group. If the instructions are not supposed to be executed by the work item, the result will be thrown away. The last level of parallelism is Single Instruction Multiple Data (SIMD). All GPGPU languages support vector data types, float4 is for example 4 floats in one variable. When doing arithmetic operations on vector data types, all the elements of a vector can be computed in one instruction by parallel electronics in hardware.

3.4 Memory model

Memory is one of the largest bottlenecks in modern processors. Mass memory is often several orders of magnitudes slower than the processors. Therefore intelligent use of caching and local memory are required to keep the processor running at maximum speed. The memory model of GPUs is adapted to the programming model. It is split in to a complex hierarchy of memory, as illustrated in Figure 3.1, to try to deliver the performance required by modern graphics applications. Each type of memory has it's own characteristics strengths and weaknesses.

The largest memory is the global memory. This memory is designed to hold textures and geometry information for graphics. Therefore this memory is huge (often more than a gigabyte), but writing to this memory from the GPU is quite slow. Management and allocations in this memory has to be done by the host. The host memory is the only memory which is accessible by the host, and the only persistent memory between kernel calls.

The local memory is memory which is coupled to each work group. The size is in the magnitude of 16-64kb per work group. The memory works as scratch memory for the work groups. In OpenCL, the size of this memory has to be known before the kernel is executed. Since the memory only is writeable for the kernel, it has to be initialized from other memory. Often it works as a per work group cache for global memory.

The memory for each work item is split in to two. Each item has a number of registers (e.g. 8 or 16), which are fast, and some overflow memory which is slow. To maintain good speed it is essential to keep the number of variables in a kernel below the number of the registers.

Access to all memory shared by more than one work item is cached. To exploit the caching, programs have to be written in a manner where adjacent memory is accessed at the same time. Sometimes however this is not the case. The main memory of GPUs is divided in to multiple banks. Each banks supports only one read at a time. Therefore the most efficient way to access memory is to fill the width of the memory buses by accessing adjacent memory, but then address memory which is far away and hopefully in another bank.

3.4.1 Synchronization

Barriers are used inside a work group to make sure all work items inside the groups have reached the barrier before continuing. If the kernel does not execute the barrier, the result is undefined and most likely the program will not advance beyond the barrier.

If programmers want to synchronize kernels without the work group, they have to make a new kernel call. Barriers can also be used on the host with multiple work queues at the host ensuring that all work queues reach the barrier before continuing.

3.5 Terminology used in GPGPU

This section will describe the terminology used in the two leading programming interfaces for GPGPU; CUDA and OpenCL. Two leading programming models for GPGPU is CUDA and OpenCL. The first API to be designed was CUDA. OpenCL was created as an effort to standardize the API across multiple graphics card vendors, and to be used by other types parallel processors too. The OpenCL API is very inspired CUDA. Therefore it is quite similar organized.

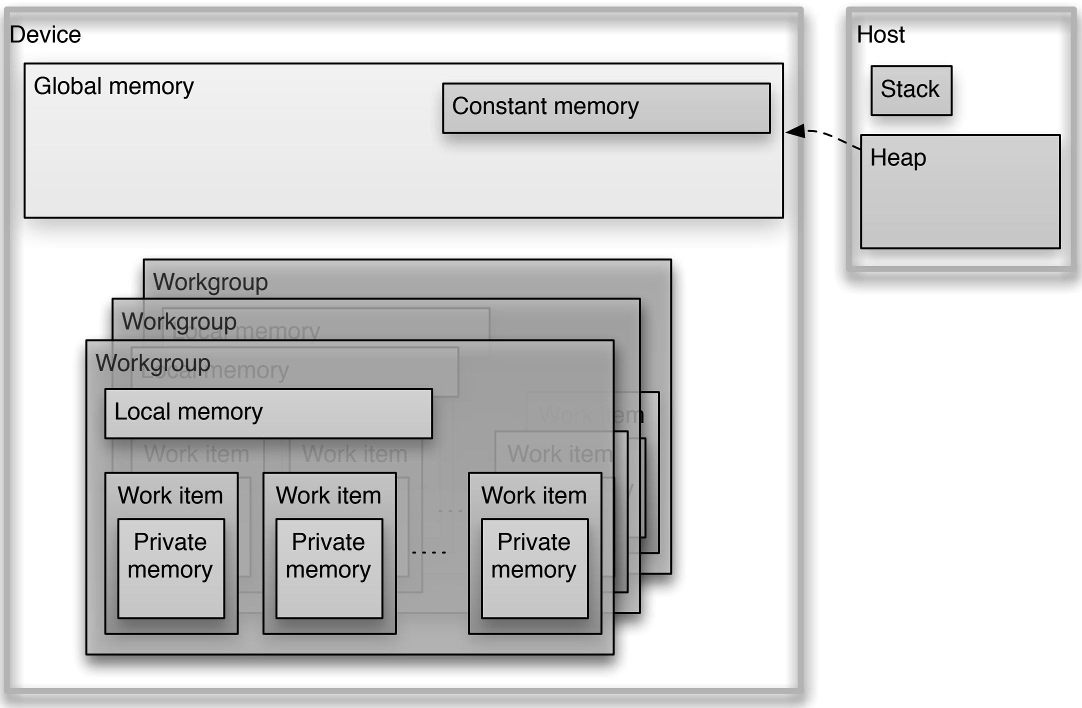


Figure 3.1: The memory model of GPUs

OpenCL	CUDA
Work-item	Thread
Work-group	Thread block
Global memory	Global memory
Constant memory	Constant memory
Local memory	Shared memory
Private memory	Local memory

Table 3.1: The most used terminology in OpenCL and CUDA

Porting from CUDA to OpenCL is mostly syntax dependent, and very little reorganization of the program is required.

3.6 Debugging

Debugging code which runs on GPUs are very difficult, mostly because of the massive parallelism of the code. Some tool kits and tools for debugging do exist. But the support for OpenCL is very new, and immature in many cases. Most of the debugging tools does only debug the outside of the kernels. This can be used to profile which kernels are bottlenecks in the application, but will not give insight into how variables are changed inside the kernels and why memory accessing errors occurs.

Some implementations have started to integrate debugging with their GPGPU API implementations. AMD has created a printf statement for it's OpenCL implementation, but this is not portable across implementations. Cross platform support of debuggers are another problem. For example Intel released a debugger supporting step by step debugging in Visual Studio. No Linux support for debugging was mentioned in the release notes. [Int12b]

3.7 OpenCL

In this section the OpenCL standard is described in more detail, outlining programming details, and how the architecture of GPUs are mapped to the OpenCL API.

3.7.1 Devices, Context & Command-queue

The syntax of the OpenCL API is inspired by the other standards created by the Khonos Group like OpenGL. The C version is quite verbose, but standardized wrappers for more high level languages exist. Multiple devices and device types are supported.

Some boilerplate code is necessary to use OpenCL. At the start of each program, all devices are enumerated and the program asks for a device which matches certain characteristics (type: GPU or CPU, available memory, number of compute units etc.). When the program decides which device it wants to use, it creates a context. The context is used to create command queues and manage memory objects.

The only way a host can order a device to do operations is to submit items to a work queue. To submit an item with the C-API, all parameters have to be pushed by a separate function call. This creates very verbose code compared to CUDA code.

Command queues can be ordered or out of order. If the queue is out of order, the device can reorder elements in the queue to make the execution more efficient. This means that other mechanisms must be used to enforce dependencies.

3.7.2 Synchronization

OpenCL features two methods for synchronizing parallel work. Events do global synchronization in a work-queue between kernel and memory-operations, while barriers manage synchronization inside a kernel.

Events are used at the host side to build a graph of dependencies between elements on a work queue. Then out of order execution may be enabled on the queue, which lets the scheduler execute elements in any order (as long as the event graph structure is fulfilled). Memory operations may for example be executed even if a kernel is running, if the Direct Memory Access (DMA) engines used for transferring memory are free.

3.7.3 Memory

OpenCL features explicit memory management in the same way as C. Memory may be allocated on a (limited) stack or managed explicitly by using buffers (or images).

Buffers are allocated on the host and passed as pointers in to kernels. Copying data from host memory to device memory and visa versa is done explicitly because it is relatively expensive.

The copying can be done when the buffers are allocated, or as an operation added to the work queue. After buffers are used they have to be freed to not leak memory.

Buffers can be allocated read only, write only and read write. Specifying the mode of the buffer can optimize the placement of the buffer in device memory. All modes are viewed from the device.

3.8 OpenCL versus CUDA

The main benefit for CUDA is that it is more mature, and supports more features because it does not depend on the bureaucracy of standardization, nor does it have to be flexible to support as many platforms as OpenCL. However OpenCL does support extensions which may be used before the features are standardized. The biggest benefit of OpenCL is that it supports more platforms and vendors, and is simpler to set up. OpenCL is supported by consumer drivers shipped by NVIDIA and AMD for Windows.

In CUDA code ran on the device, and code ran on the host can be programmed in the same file. Global variables can be defined in the host code and accessed from kernel code. The host code and the kernel code is compiled before the program is executed. All code calling GPU kernels has to be compiled together with the kernels for all targets supported by the compiler. This leads to incredibly bloated executable files. In OpenCL the code is compiled at runtime, passed as a string to the driver, like GLSL. The driver may then optimize the code for the device it is supposed to be ran for only. Debugging CUDA code is simpler than OpenCL because the code is produced by the same compiler, and because the tools are more mature.

To run multiple kernels asynchronously in CUDA, a specialized set of function calls has to be used. This is more cumbersome than the generic OpenCL APIs. The support for running kernels asynchronously has lagged behind OpenCL, but will most likely catch up pretty soon because NVIDIA uses CUDA as a basis for their OpenCL driver. This can be seen in Linux; stack traces generated when using OpenCL include calls to the CUDA library by OpenCL driver functions.

Chapter 4

Description of the snow simulator

This chapter is organized in two parts. First the current state of the snow simulator is outlined. The code and implementation of the mathematical algorithms is the focus of this part. For a more detailed discussion of the algorithms of the simulator see [Sal06]. Afterwards the porting process, and the changes to the simulator introduced during porting is described.

4.1 History of the simulator

The snow simulator has been a basis for multiple thesis and autumn projects at NTNU. Most of the thesis produced are available from the web page of the lab [Els] maintained by my advisor, or from DAIM [IDI], a repository of master thesis maintained by the faculty. Figure 4.1 provides an overview of all relevant thesis and specialization projects relevant to the snow simulator produced by the HPC lab since the first thesis about the simulator. Records on the timeline marked with a leaf are specialization projects written during the autumn semester. Records which are written with grey text concerns simulation subjects relevant for the simulator, but haven't worked with the code of the simulator. In the following paragraphs a quick overview of all thesis which have worked on the actual simulator code will be given.

The first snow simulator written was written by Saltvik in 2006, as a master thesis [Sal06]. This thesis laid the foundation for the simulator. The underlying mathematical and physical models from this thesis still forms the core of the simulator today. The simulator was implemented on a CPU using multiple threads to parallelize the simulation work. Real time simulation with an acceptable frame rate was achieved. However very few particles (a few 10'th of thousands) were simulated.

Because the simulator still was quite slow no further work on the simulator was done in two years. During autumn 2008 [Eid09] ported the snow simulator to CUDA, but the report is dated 2009 because it was delivered in February. Nearly all memory used by the simulator was kept on the GPU and extensive care was shown to optimize the program for the GPU architecture. In this thesis rendering was also rewritten and improved.

The GPU snow simulator was developed further in spring 2009 in [Gje09]. In this specialization project the simulation code for simulating fluid flow trough porous rocks using Lattice Boltz-

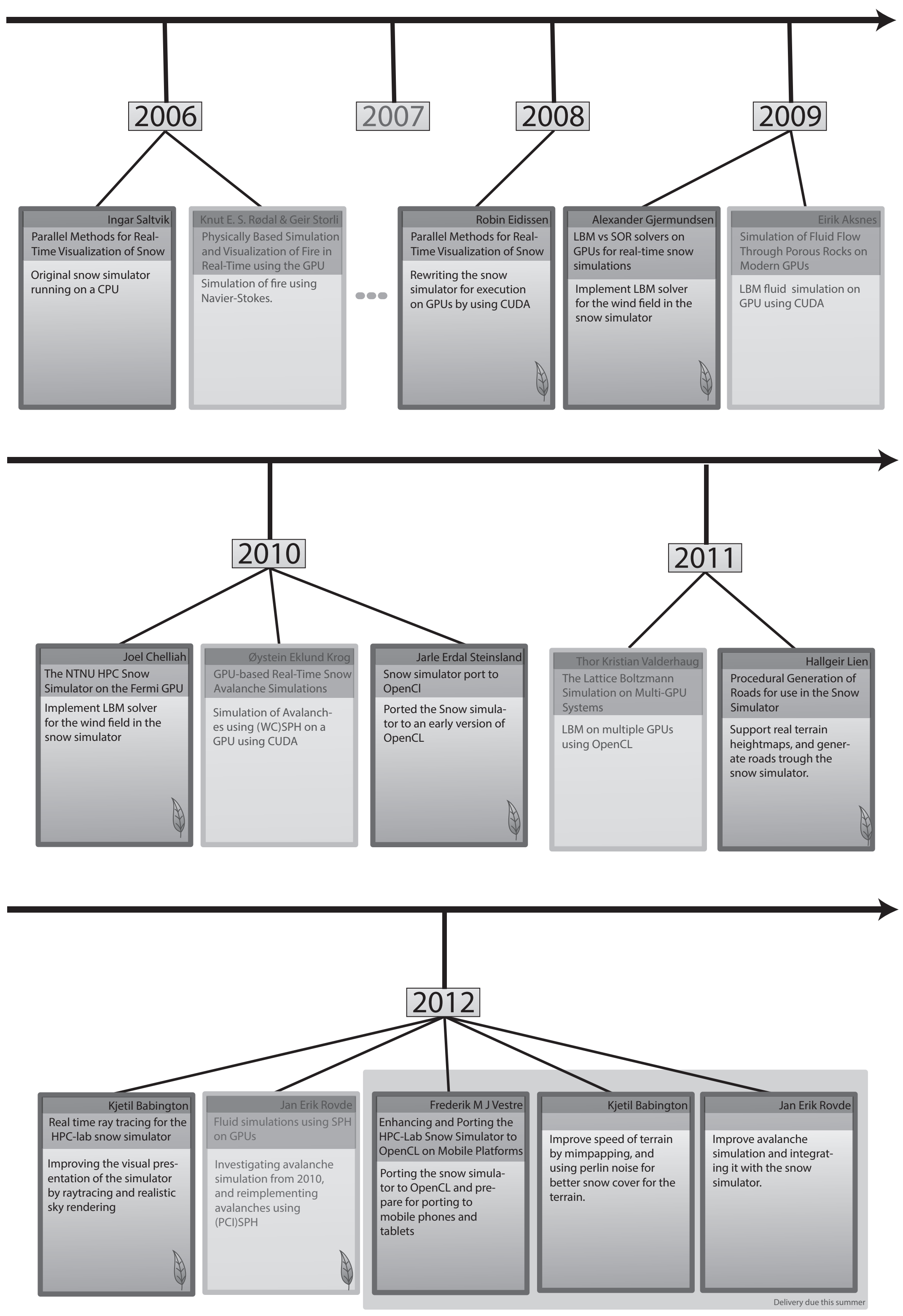

2006
Ingar Saltvik
Parallel Methods for Real-Time Visualization of Snow
Original snow simulator running on a CPU
Knut E. S. Rødal & Geir Storli
Physically Based Simulation and Visualization of Fire in Real-Time using the GPU
Simulation of fire using Navier-Stokes.
2007
2008
Robin Eidissen
Parallel Methods for Real-Time Visualization of Snow
Rewriting the snow simulator for execution on GPUs by using CUDA
2009
Alexander Gjermundsen
LBM vs SOR solvers on GPUs for real-time snow simulations
Implement LBM solver for the wind field in the snow simulator
Eirik Aksnes
Simulation of Fluid Flow Through Porous Rocks on Modern GPUs
LBM fluid simulation on GPU using CUDA
2010
Joel Chelliah
The NTNU HPC Snow Simulator on the Fermi GPU
Implement LBM solver for the wind field in the snow simulator
Øystein Eklund Krog
GPU-based Real-Time Snow Avalanche Simulations
Simulation of Avalanches using (WC)SPH on a GPU using CUDA
Jarle Erdal Steinsland
Snow simulator port to OpenCl
Ported the Snow simulator to an early version of OpenCL
2011
Thor Kristian Valderhaug
The Lattice Boltzmann Simulation on Multi-GPU Systems
LBM on multiple GPUs using OpenCL
Hallgeir Lien
Procedural Generation of Roads for use in the Snow Simulator
Support real terrain heightmaps, and generate roads trough the snow simulator.
2012
Kjetil Babington
Real time ray tracing for the HPC-lab snow simulator
Improving the visual presentation of the simulator by raytracing and realistic sky rendering
Jan Erik Rovde
Fluid simulations using SPH on GPUs
Investigating avalanche simulation from 2010, and reimplementing avalanches using (PCI)SPH
Frederik M J Vestre
Enhancing and Porting the HPC-Lab Snow Simulator to OpenCL on Mobile Platforms
Porting the snow simulator to OpenCL and prepare for porting to mobile phones and tablets
Kjetil Babington
Improve speed of terrain by mimpapping, and using perlin noise for better snow cover for the terrain.
Jan Erik Rovde
Improve avalanche simulation and integrating it with the snow simulator.
Delivery due this summer

Figure 4.1: Timeline of thesis related to the snow simulator

mann methods (LBM) [Aks09] was integrated with the snow simulator. The LBM method and the original Navier-Stokes based method was benchmarked for performance.

During autumn 2010 [Che10], the snow simulator was optimized for the NVIDIA Fermi GPU. Substantial gains were produced by optimizing memory accesses and usage to take advantage of all caches and different types of memory on the GPU. This project focused mainly on optimizing the LBM code, but the Successive Over Relaxation (SOR) code was also touched. Removal of branches and serial code was also performed to optimize the code. This resulted in a speed up of about 1.5 times the original speed for the simulator. Some of these optimizations were specific to the GPU and the compiler.

In 2010 [Ste10] ported the simulator to OpenCL in order to support more devices than NVIDIA GPUs. During autumn 2011 a module for importing real terrain was integrated in the snow simulator by Hallgeir Lien in [Lie11]. The terrain was integrated in order to do snow cover simulation for roads, and to optimize the location of the roads in the terrain of the simulator. In [Bab11] Kjetil Babington looked in to use ray tracing for visualizing the snow particles of the simulator, and different methods for improving the appearance of the sky. In [Rov11] the avalanche framework started in [Kro10] was improved in order to integrate it with the snow simulator.

During this spring three thesis concerning the snow simulator are written. The two other students working on snow simulator related thesis are: Kjetil Babington who continues to improve the visual appearance of the snow simulator by improving the terrain rendering, and Jan Rovde who continues the avalanche integration started during autumn.

4.1.1 Code history

Most of the thesis in Figure 4.1 which are not greyed out have contributed code to the simulator. Porting the simulator to OpenCL was the subject of a specialization project during autumn 2 years ago. Both my advisor and I have tried to retrieve the code and report for this project. This has unfortunately not been possible. Therefore I had to port the simulator from CUDA to OpenCL one more time. No explicit routines for managing the snow simulator code have existed. My advisor has asked all students to deliver code on memory sticks at the end of each semester. However most students are very exhausted when their thesis are delivered, and likely to forget to do this. Physical memory sticks are also easy to loose, compared to networked storage provided by an IT-department.

The lack of routines regarding the management of the snow simulator code has caused some confusion regarding the history of the development of the simulator. This is the case regarding which simulator has been used as the code base for later thesis. Development of the simulator has sometimes been done in parallel. This results in diverging code which has to be merged together. If the code is not merged it may be lost, or at least incompatible with later developments. Then the advancements of the thesis which produce the code will not be benefiting later students working on the simulator. To avoid this, the code from this thesis has been included in a Version Control System (VCS) at a centralized location managed by the faculty, and not by the students.

A VCS for code is a system which manages all source code for a programming project. The primary element of a VCS is a commit. A commit is a snapshot of a specific version of the code.

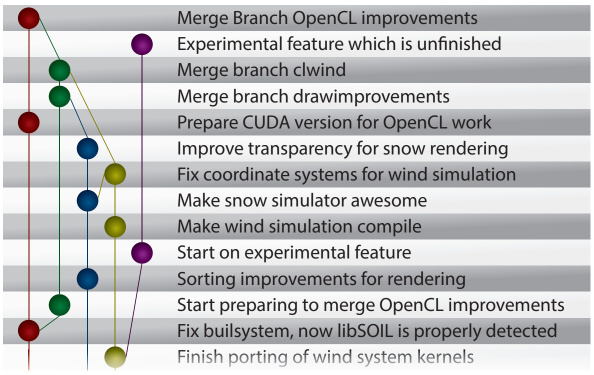

Merge Branch OpenCL improvements
Experimental feature which is unfinished
Merge branch clwind
Merge branch drawimprovements
Prepare CUDA version for OpenCL work
Improve transparency for snow rendering
Fix coordinate systems for wind simulation
Make snow simulator awesome
Make wind simulation compile
Start on experimental feature
Sorting improvements for rendering
Start preparing to merge OpenCL improvements
Fix builsystem, now libSOIL is properly detected
Finish porting of wind system kernels

Figure 4.2: Git version control history DAG example. The commit messages for the commits are listed to the right reverse ordered by date

Ideally the code should be committed every time a bug is fixed or a feature is implemented. The VCS stores the difference between each commit. Therefore old commits can be accessed, and the code can be retrieved as it was when it was committed.

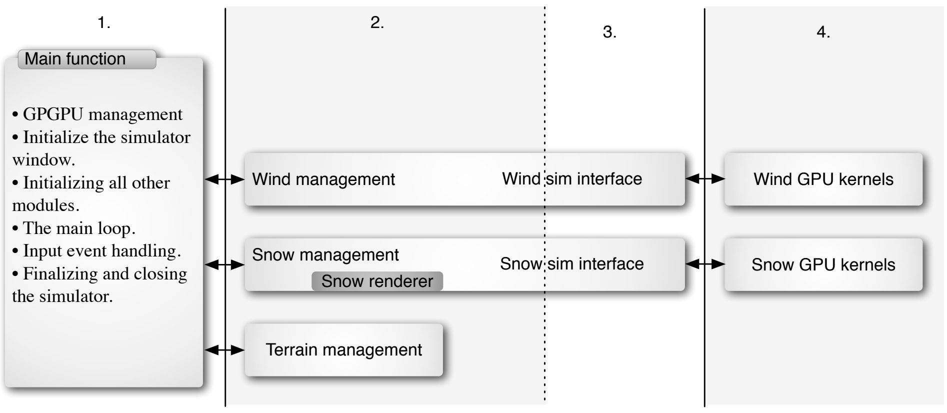
VCSs have traditionally been managed on a central server. All commits and changes have been done on the server. In the last few years distributed VCSs have become more popular, at least in the open source community. These systems works locally. Each developer has his own repository and creates his own history of commits called a branch. Over time the branches and commits will form a Directed Acyclic Graph (DAG) as shown in Figure 4.2. The branch is then pushed into a central repository, which collects the branches of multiple developers. Two branches can be merged to form a single code base for further development. Merging works by extracting the changes from a common earlier commit, and applying changes from both branches to the same base. When both branches change the same part of a file, a conflict is marked. This has to be solved manually by the user merging the histories.

Different thesis may work in their own branches, and push to their own separate branch on the server maintained by the faculty. Then at the end of the semester, the branches can be merged, and a common basis can be created for the new projects which are going to build on the code. If a student obfuscates the code when optimizing it, or introduces regressions in other ways later projects may retrieve parts of the code and use it to understand how the simulator works.

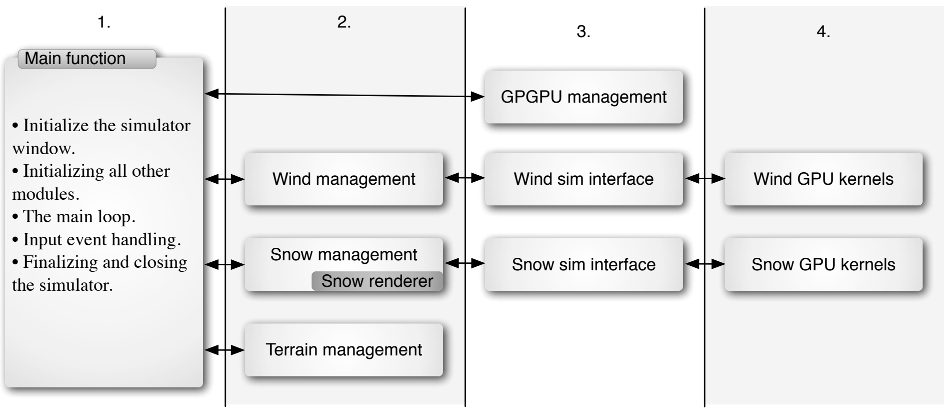
4.2 Organisation of the simulator

The simulator was originally written in C, C++ and CUDA. The simulator consists of several classes and functions. To make the snow simulator support both OpenCL and CUDA, I have introduced an abstraction layer between the GPU independent and the GPU dependent part of the simulator. The process of creating the abstraction layer is described as part of the porting process in Section 4.6.1.

A visual overview of the organization is given in Figure 4.3. Before the abstraction layer was created, the GPGPU interface was managed by the main function and the distinction between management of each sub module and the GPGPU interface was a bit fuzzy. This is symbolized



(a) Before porting



(b) After porting

Figure 4.3: Organization of the snow simulator

by the dotted line between column 2 and column 3 of Figure 4.3a.

After the abstraction 4 levels are prominent in the architecture of the simulator. This is symbolized by the 4 columns of Figure 4.3b which represents each level of detail towards the kernels run on the GPU. Column 1 and 2 are independent of the GPGPU API while column 3 & 4 are dependent on it. The interfaces between column 2 and 3 are formalized and one implementation for each GPGPU API is created.

The first column consists of a function called “main”, which is the entry point of the program. This function coordinates all the other modules and keeps the simulator running. The responsibilities of this function is listed in the figure. Because all GPGPU responsibilities are moved in to separate classes, the only difference in the main function between GPGPU APIs is which classes that are initialized. The window system and event loop handling is independent of each GPGPU system. A mobile platform however may need other interfaces with the underlying system, and then the main function may have to be rewritten completely. However this will be dependent on the mobile platform. Most likely the other modules will be usable from a rewritten main function.

The second column contains the common code for all subsystems which is not dependent on the GPGPU API in use. The Wind and Snow interface controls the API specific interfaces and manages all communication with the outside world. Most of this code is resource management and it is simple to understand their function by looking at the code.

The terrain class does not use CUDA at all, therefore no changes to this class has been done during the porting. The terrain class is not described in this chapter, as no knowledge of the class was required during the porting. The terrain implementation is however described quite detailed in Section 3.2.1 of [Eid09]. Interaction with the terrain, as a result of snow particles hitting it, is described in Section 4.5.

The rendering code in the snow particle interface accepts Vertex buffer object (VBO) (an OpenGL memory buffer) with one entry for each snow particle. This object is created on the GPU. The result of the OpenCL and the CUDA simulation is identical. Therefore no changes to the snow particle rendering was done during the porting.

Column 3 and 4 of Figure 4.3 are outlined in the next sections.

4.3 Simulator flow

The GPU code of the simulator consists of two sub simulations which are very independent: Wind simulation and simulation of snow particles. The interaction between the parts of the simulation is outlined in Figure 4.4. The figure outlines the main functions for each sub simulation; “Simulate” in the wind simulation, which is described in Section 4.4, and “MoveParticles” in the snow particle simulation, which is described in Section 4.5. The outlines list the kernels executed on the GPU, together with the number of elements they are executed for. For the move particles function each locked OpenGL texture is also listed.

Each simulation subsystem is placed on its own side of a centre dividing line. The only communication between the subsystems is the wind field, which needs to be communicated for each time step, and the obstacles of the terrain, which only needs to be communicated occasionally; e.g in the magnitude of each 10. second. This makes it possible to do the simulation of the different subsystems using different type of hardware and API’s.

4.3.1 Obstacles

Both the wind and snow particle simulation are dependent on the terrain of the simulator. A simplification of the terrain, called obstacles, is used for interaction with the terrain. One or two times a second an obstacle map is regenerated from the terrain by the host.

The obstacles are stored in an integer matrix covering all of the wind domain. The values of this matrix is initially set to 1 if the coordinate is below the terrain, or 0 if not. Then each coordinate is examined and the final value is determined based on the values of the neighbouring cells. At last the elements at the borders are updated to satisfy border conditions.

4.4 Wind simulation

The wind simulation consists of 6 steps outlined in the left part of Figure 4.4. The names used in the figure are the names of the GPU kernels implementing each step. For more information

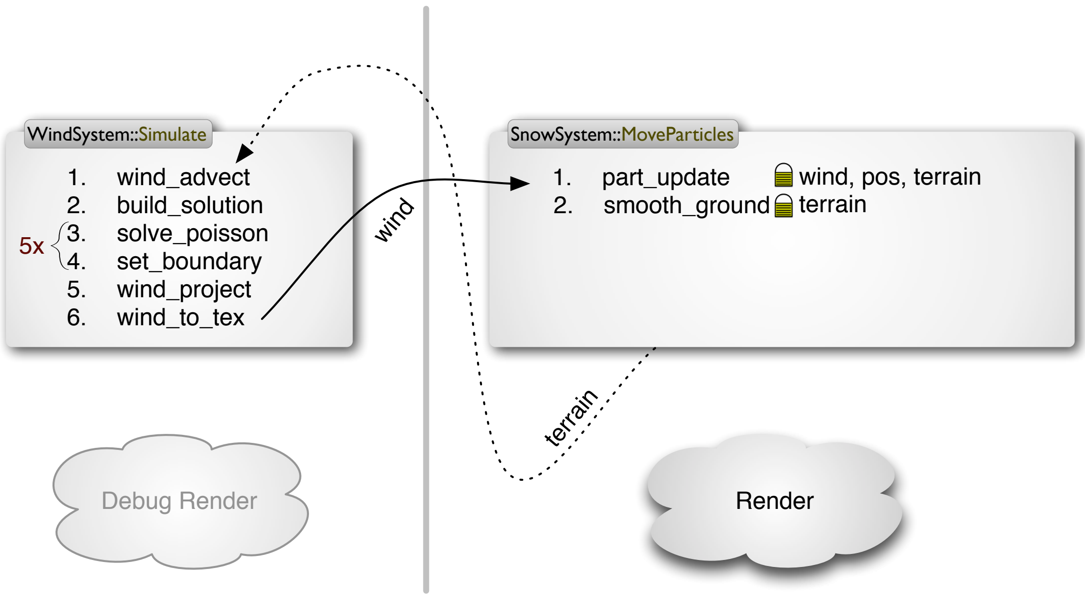


Figure 4.4: Simulation loop sequence & interaction diagram

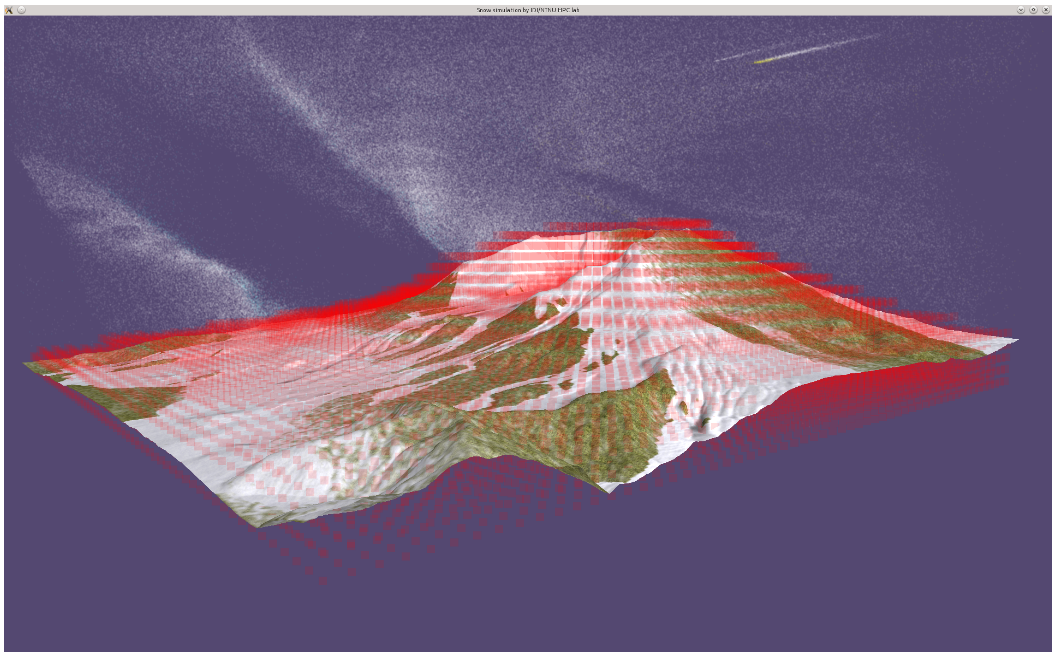


Figure 4.5: Obstacles covering the area under the terrain

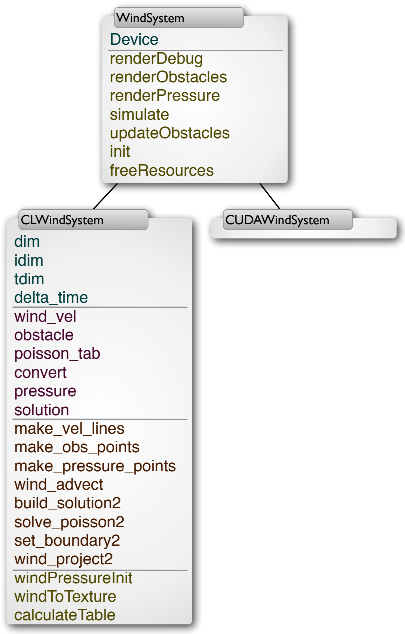


Figure 4.6: Wind system classes overview

about the mathematics and assumptions applied in the wind simulation see [Sal06], Chapter 4.2. For information about the original implementation of the CUDA simulation see [Eid09] Chapter 3.

Some more kernels are included in the wind system class. These kernels provide data for debug rendering. These kernels work by iterating trough a grid of n points evenly distributed over the wind, or pressure memory buffer and record the coordinates and the values of the buffers at these locations. These values are then written in to VBO objects and passed to OpenGL drawing functions by the host in the Wind management class in Figure 4.3. The output of this rendering is studied in Section 6.7.

The wind simulation is the largest part of the simulator. Two versions of the wind simulation are implemented. One method based on LBM, and one using the Navier-Stokes equation. Both versions model the wind as a fluid. The LBM version simulate the fluid on a microscopic level using particles, while the Navier-Stokes version look at the fluid macroscopically. I have ported and looked in to the Navier-Stokes version, which was used in the original simulator. Therefore only the Navier-Stokes based simulation is described here.

4.4.1 Navier-Stokes simulation

$$\nabla \cdot \mathbf{u} = 0 \tag{4.1}$$

$$\frac{\delta \mathbf{u}}{\delta t}\mathbf{u} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \nabla \mathbf{p} \tag{4.2}$$

The Navier-Stokes equation is an equation made to model several physic phenomena necessary to predict the flow of fluids. Some of these phenomena are not contributing much when the equation is used to model air flow in an open landscape. Therefore some simplifications has been done. The air is assumed to have has zero viscosity (4.1) and defined to have a density of one. After these simplifications the wind model can be computed using incompressible Euler equations (4.2).

The Euler equations are multi dimensional differential equations. These equations has to be solved for the entire wind domain. It is not feasible to solve the equations analytically. Therefore the equations are approximated numerically. The volume of the simulation is sampled in a 3-dimensional grid, and solved for discrete points in time and space. This is a massively parallel high performance computing problem, which is ideal to do on a GPU.

The Euler equation is solved in two steps; self-advection and projection. When implementing these steps they are further split which results in the GPU kernel calls outlined at the left side of Figure 4.4.

Three buffers of memory covering the domain are used when solving the equation. The buffer named "wind_vel" contains the velocity and direction of the wind. The buffer "pressure" contains the atmospheric pressure for each point in the domain. The buffer "solution" contains the solution of Equation 4.3. Which is used in the next step. The pressure is reset during each time step, and recalculated from the wind field. In addition an obstacle buffer is used to mask out everything which is below a terrain etc.

Self-Advection

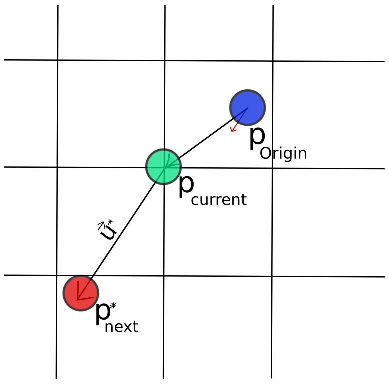


Figure 4.7: Self advection

First the self advection part $\mathbf{u}^* = -(\mathbf{u} \cdot \nabla)\mathbf{u}$ is computed by using an Eulerian interpolation as illustrated in Figure 4.7. To get the new direction of the advection the old direction is subtracted to get an origin of the movement. The magnitude and direction of the wind at the origin is sampled (using interpolation) and used as the new direction.

Poisson equation preparation

Then Equation 4.1 and the last part of Equation 4.2 is combined. Then the combined equation is transformed to a Poisson equation using the Helmholtz-Hogde decomposition. The Poisson equation can be expressed as $Ap = b$. Here p is unknown. To solve the equation A and b must be found. A is a normal diagonal matrix, which is hard coded in to the Poisson solving function. b is computed before the equation is solved in the "build_solution" kernel by solving Equation 4.3 for all points in the domain. This solution is stored in a separate buffer and used together with the pressure field in the next step.

$$(\nabla \cdot \mathbf{u})_{i,j,k} = \frac{(x_{i+1,j,k} - x_{i-1,j,k} + y_{i,j+1,k} - y_{i,j-1,k} + z_{i,j,k+1} - z_{i,j,k-1})}{h} \tag{4.3}$$

Poisson equation solving

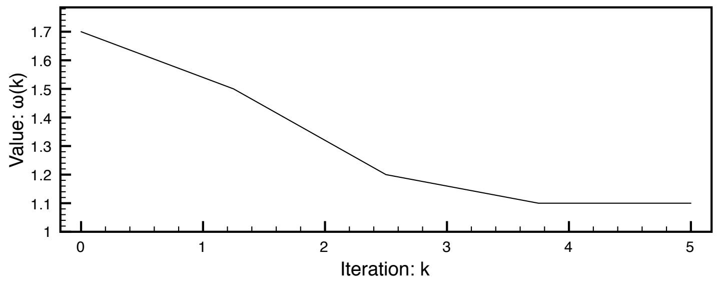


Figure 4.8: Values for $\omega(k)$ used in SOR

$$\mathbf{x}^k = (1 - \omega(k))\overline{\mathbf{x}}^k + \omega(k)\overline{\mathbf{x}}^{(k-1)} \tag{4.4}$$

The GPU kernels "solve_poisson" and "set_boundary" are responsible for solving the Poisson equation.

The POisson equation is solved by using SOR, a variant of Gauss-Seidel (GS) interpolation. SOR is an iterative equation approximation technique. Equation 4.4 is ran for multiple steps. k is the index of the current step, and $\omega(k)$ is the relaxation factor. This factor is dependent on each problem, and no way except trial and error has been found to find the optimal factor. In our implementation the factor varies with each time step as outlined in Figure 4.8, to give faster convergence. In each iteration, boundary conditions are maintained by the "set_boundary" kernel.

A significant part of the "solve_poisson" kernel manages obstacles i.e. the border between the wind-/pressure-field and the terrain. Only the elements above the surface of the terrain is managed by the algorithm. This introduces a whole new set of special boundary conditions because SOR uses the average of the neighbourhood belonging to each point as a basis for solving the equation. For each point the obstacle map is consulted, and if an obstacle is present the corresponding value from the neighbourhood is considered to be zero. This upsets the average.

Therefore a table ("poisson_tab") based on how many, and what kind of masks which are applied is factored in to the solution.

Projection

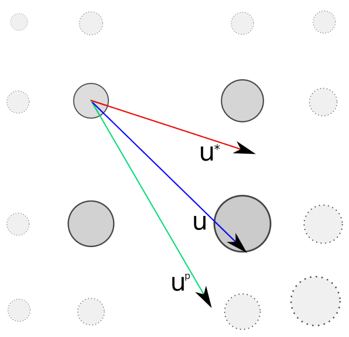


Figure 4.9: Schematic overview of wind projection

After the pressure has been computed by the POisson solver it has to be projected on to the temporary wind information created by advection ($\mathbf{u}^*$). This is done by the "wind_project" kernel and is illustrated in Figure 4.9. The projection uses two vectors as a base; $\mathbf{u}^*$ and $\mathbf{u}^p$. ($\mathbf{u}^p$) is created inside the kernel by computing the gradient of the pressure field. Then the two vectors are multiplied component-wise to form ($\mathbf{u}$) which is used as an input for the snow simulation, and for the next wind simulation step.

4.5 Snow simulation

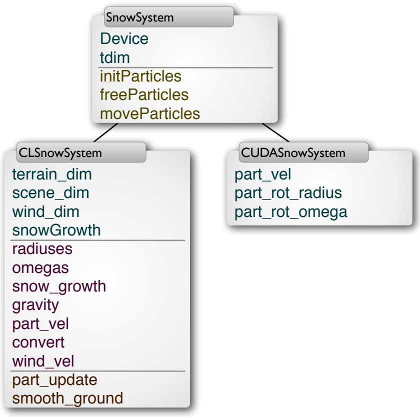


Figure 4.10: Snow system classes overview

The snow simulation is based on two kernels. One kernel to manage the particles ("part_update"), and one to smooth the ground after the particles hit it ("smooth_ground").

Particle management

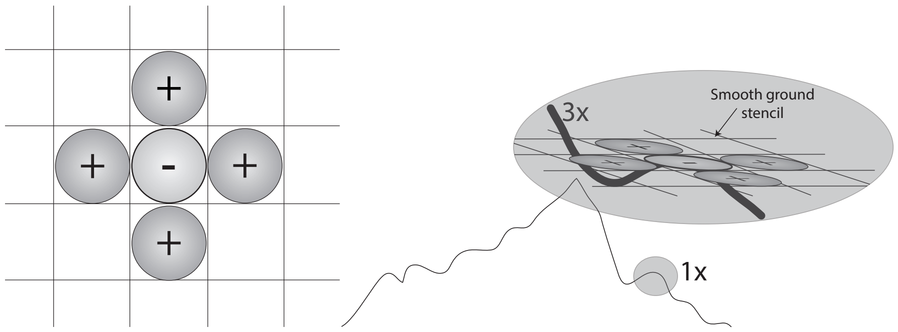
Each particle has several attributes: position, velocity, radius and rotation. The position and velocity is maintained for each particle, while the radius and rotation is initialized to random values and reused for each 31′th particle. They are treated as read only.

The particle management kernel works by computing the drag of the particles, factoring in wind and gravity. Afterwards a circular motion is added depending on the radius and rotation of the particle.

When a particle is above or below the domain. The particle is moved to the top of the domain and, repositioned in the x-z axes pseudorandomly by bit fiddling the previous position. The velocity is adapted from the wind field at the new position. If the particle is hitting the terrain the snow level of the terrain increased before the particle is repositioned. The snow level is increased for the point in the terrain where the particle hits, and some points in the vicinity.

If a particle gets out of the domain one each of the sides (in front, behind, to the left or to the right) the particle is wrapped around and enters from the other side.

Smooth ground



(a) Smooth ground per pixel stencil (b) Smooth ground operation, from the side

Figure 4.11: Smooth ground kernel operation

To simulate how snow is distributed on the terrain, a smooth ground kernel is implemented. The kernel goes trough the terrain and smooths out any spikes. It does also facilitate a very coarse approximation of avalanche management. The kernel works by looking around in a 4-point stencil as illustrated in Figure 4.11, and if the current point is the highest it decreases the hight, if not it increases the hight. The stencil is implemented by dividing the terrain in to quadratic blocks to be solved in one OpenCL work group. The width of the work group is called a stride. The terrain is then copied to local memory for each block. The local memory is one element larger than the block in each direction to be able to run the stencil for the outer elements.

4.6 Porting to OpenCL

The first thing that was done when porting the code was to get an overview of the code and try to figure out which elements had to be ported. To improve the changes of keeping the port updated a common code base supporting both GPGPU APIs was created. Creating this abstraction was a good way to get familiar with the code.

4.6.1 Abstracting API specific code

When the code was examined, a partial abstraction of the GPGPU interface was discovered. This interface was a result of the fact that the snow simulator was written in C++, and the interface to CUDA was written in C. Therefore an interface between C and C++ was required. However it was preferable to have the interface for GPGPU in C++ to be able to use class-inheritance. The CUDA specific host code was therefore ported to C++ and a minimal C interface was created to call the GPU kernels. These wrappers are generated with information from the CUDA kernel source code by a python script. The wrappers can be linked externally as normal C functions, and contains one line of code only, which calls the kernel which they wrap.

After the restructuring, each subsystem using GPGPU is represented by a (super)class, which then has a subclass with implementations for each GPGPU API as (outlined in Figure 4.6). This makes it simple to switch subsystems at compile time. During the first phase of the porting the wind simulation was running using OpenCL, while the snow simulation ran using CUDA. This is possible because the communication between the two subsystems is very limited (as outlined in Figure 4.4), and uses OpenGL textures and vertex buffers.

4.7 Calling conventions

The biggest challenge when porting the snow simulator was to get the coordinate order correct, as the order was y,z,x, and not the intuitive x, y, z. Each dimension was oriented as in Figure 4.12. CUDA does not optimize kernel calls which are called with three dimensional bounds for memory access. Therefore the snow simulator uses only one dimensional bounds. The dimensions are split along the memory hierarchy of the GPU as in Figure 4.12. The x axis is split along the threads of the GPU; one thread per row. Then each thread accesses adjacent x values, which translates to adjacent memory. This memory may be fetched in one memory access by the SM. The values along the z axis are distributed over different blocks, which means they are run concurrently, but not on the same streaming multiprocessors. However access to the global memory is kept in the same y-plane as the other SMs. This keeps the access to the global memory adjacent, exploiting caches caching global memory. At last the y axis is distributed over loop iterations. This happens most seldom, and therefore the y-planes are stored furthest apart in memory.

4.7.1 Kernel porting

The kernel porting was done one kernel at the time, until all kernels were compiled without errors. During porting, a clean up of the CUDA was done, to make the resulting OpenCL code as easy to understand and debug, as possible. None of the clean up steps should cause any performance degradation of the code, as any sane compiler should generate the same code regardless of how pointers are computed, and variables are named.

Especially the wind simulation kernel file was very messy, and showed signs of optimizations and hacks. Different versions of the same function was present, but not called from anywhere

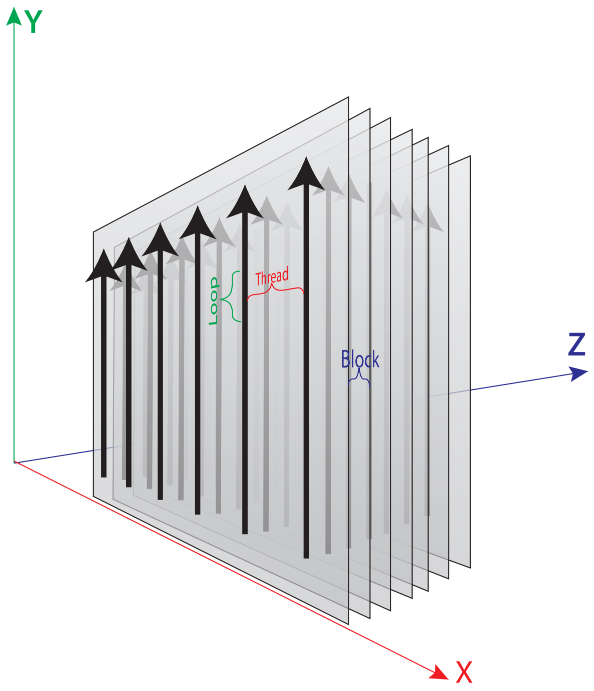


Figure 4.12: Kernel memory access strategy

in the code. Most of the variables in the functions was one letter long; most likely because equations use one letter. Pointer arithmetic are used in loops and to look up elements in arrays. The arithmetic did not seem to work properly in OpenCL, and included some assumptions on the sizes of variables used by the architecture, which may result in different behaviour on different OpenCL implementations. Because the CUDA compiler does not do very much optimization, some constants where defined as volatile variables to increase speed. This made the code difficult to read.

After the kernels compiled as OpenCL code, one kernel at the time was introduced to the simulation loop. Each time a new kernel was introduced the program crashed. Then parts of the kernel was commented out until the program ran again. When the program ran, the parts identified to make the program crash were investigated and corrected.

When all kernels were running without crashing the focus turned to getting the simulator to behave in the same way as the CUDA version. At this stage the calling conventions of Section 4.7 was a large obstacle to figure out, and then to get consistent across all kernels. Before this was figured out, random crashes, and crashes as a result of small changes were frequently experienced. After this was solved, the wind simulation progressed quite steadily to a stable state. In this state, it did not work exactly as the CUDA code, but it was stable enough to provide a base for performance benchmarks to be done, and to give meaningful information when ported to a mobile device.

The wind simulation kernels were ported by exporting the result of the wind simulation to a VBO, which was then imported in to CUDA, to do the snow simulation. When the wind simulation was stable enough, the snow simulation was ported. This code had not been optimized as heavily as the snow simulation code. Therefore it was simpler to understand and easier to port. The initial port of the snow simulation kernels took about a week. However, two issues remained.

One issue was the “smooth_ground” function which showed strange spikes at regular locations in the terrain. After some investigation it was discovered that the amount of shared memory was too small for the kernel to do it’s work. This resulted in a working kernel, until the size of the terrain changed. After further investigation, the padding of the area examined in one block was figured out. This is documented in Section 4.5. By setting the stride width correctly and providing enough memory the smooth ground works as in CUDA.

The other issue was the particle initialization code, which is originally described in the “Particle repositioning” subsection of section 3.3.1 in [Eid09]. The CUDA implementation “use inherent noise in the floating point representation of the position of each particle. The method has no specific formal basis and was developed by trial and error, but the resulting distribution appears uniform and pleasing to the eye.” [Eid09] (page 44). This algorithm is difficult to port to OpenCL and may be dependent on the device, and therefore can vary between different OpenCL implementations. No fully working replacement was found, but a half way working replacement was made using float, int-casts and modulo operations. This is discussed further in Section 6.7.

4.7.2 Implementation changes

The largest changes in the simulator results from the abstraction of API specific code outlined in Section 4.6.1. In the kernels some usage of local memory was removed to simplify the code at the time and focus on getting it correct. Reintroducing this usage of block memory would most likely improve performance.

Computation of array indexes was also rewritten to use array indexes instead of pointer arithmetic. The address of the resulting array element was then retrieved and stored in the pointer. A good compiler will compile these expressions to the same code as the pointer arithmetic. The code is however easier to read and does not reference the type name. Therefore it will work even if the variable type is changed.

Chapter 5

GPU sorting

This chapter looks as sorting on the GPUs in order to enhance the performance of snow particle rendering, and to advance the visual quality of particle rendering in the snow simulator.

5.1 Motivation

The snow simulator renders a lot of snow flakes as particles on the GPU. These particles are originally stored in a random order. In section 3.3.1 of [Eid09], sorting the particles is suggested to improve cache performance. Several techniques used in particle rendering require the particles to be sorted [MG10]. Techniques which measure densities of particles for transparent rendering and shadowing, need to look up particles at a certain position. They also require that particles are visited in a certain order dependent on the light source or camera. Rendering without transparency needs to figure out which particles are obstructing others. This is simpler and faster to do on a sorted list. Sorted lists do also aid collision detection [KSW04], which probably will be implemented in future versions of the simulator in order to do more realistic simulations. A sorted particle list may also speed up computation of the snow particle movement because accesses to the wind field which drives the particles will have better locality, and therefore will be cached better.

5.2 Traditional sequential sorting algorithms

For traditional computers, sorting has been done with sequential algorithms. The most efficient algorithms for comparison based sorting, are based on the divide and conquer principle. This may look like a very good candidate for parallelization, as the divided problems may be sorted independently. However the outermost steps have to be done as one operation, or as a parallel merge.

5.3 Requirements for GPU algorithms

The most efficient sequential algorithms for comparison based sorting, are data driven. The steps the algorithm takes depends on the data which it is sorting. This may lead to very different execution patterns depending on the characteristics of the data to be sorted. This is no problem for serial execution. For parallel execution this causes a number of problems. An algorithm which is designed for parallel execution, should meet as many as possible of the following requirements, at least partially.

1. Equal workload on all parallel processors.
2. Clear communication patterns to simplify synchronization and reduce synchronization overhead.
3. Cache friendly memory accesses.

Reasons for requirement 1

If the workload of the parallel processors is distributed unevenly, waiting will be introduced if one execution unit is finished before another. In this way the processing power of the waiting units is wasted. If the processors are guaranteed to finish at the same time, synchronization logic can can be limited completely resulting in even better performance.

Reasons for requirement 2

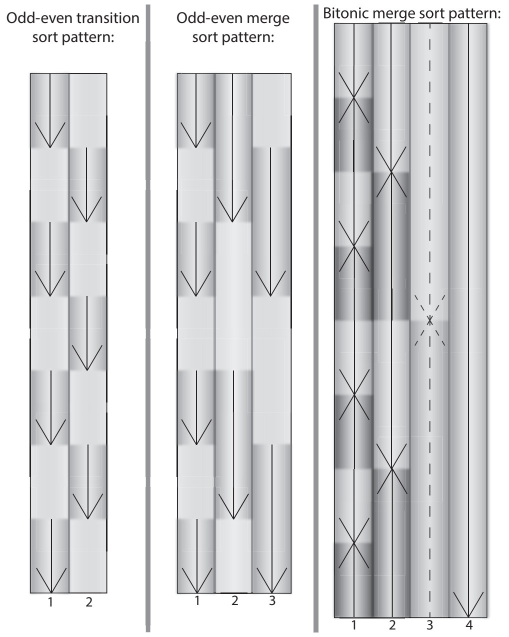
Tightly coupled parallel systems, like GPUs, use SPMD execution units. These units benefit from coordinated, predictable communication patterns. If the code does not exhibit such patterns, the GPU has to execute all instructions requested for all the data in each unit, and then throw away the results that are not applicable. This degrades performance.

Reasons for requirement 3

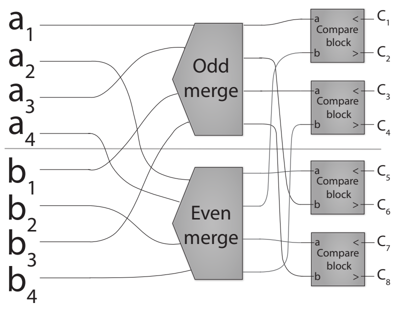
GPUs do also have requirements for memory access patterns. For example recent NVIDIA cards have a 128 byte memory bus. Therefore data accesses of up to 128 bytes of data linearly will be combined to one access. However data accesses exceeding the bandwidth of the bus will be slower, as multiple accesses can be done simultaneously, but only if they access different memory banks. Such memory accesses is much simpler to create with an algorithm with clear communication patterns, especially if the communication patterns are predictable, and independent of the data that the algorithm is processing.

5.4 GPU sorting algorithms

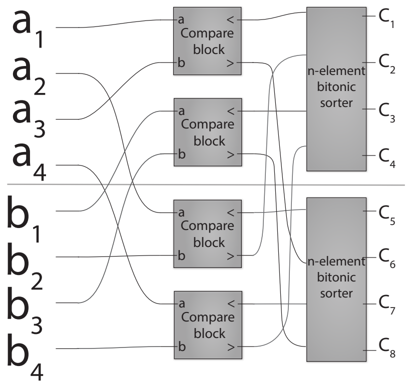
Data-independent sorting algorithms satisfy most of the requirements outlined in Section 5.3. They can be designed easily to saturate the SPMD's of the GPU and make memory access patterns, memory bandwidth usage and synchronizing points deterministic. The algorithms organize the sorting problem as a network of compare and swap operations, where the addresses and organization of the elements which are compared are independent of the data. Even though data independent sorting algorithms are considered best in theory, papers exist which concludes that data driven sorting algorithms are best for parallel use cases. For example [CT08] implements a version of quick sort tailored for GPUs. In the result section of the paper they conclude that their algorithm is the best based on their tests. Later papers however like [PSHL10] provides faster data independent sorting implementations. This may indicate that the algorithm to the GPU, making sure that all execution units, memory bandwidth etc. is saturated with useful work, seems to be more important than the algorithm in use. Additionally some algorithms may be better suited to certain initial distributions and problem sizes than others.


Odd-even transition sort pattern:
Odd-even merge sort pattern:
Bitonic merge sort pattern:
1 2
1 2 3
1 2 3 4

(a) Sorting patterns for GPU sorting algorithms



(b) Network for one odd-even merge sort iteration



(c) Network for one bitonic sort iteration

Figure 5.1: GPU sorting networks

The simplest data independent sorting network is odd-even transition sort. Odd even transition sort works in two passes, comparing all odd elements with the next one, and then all even elements with the next one. This is illustrated in the left part of Figure 5.1a. This gives an $O(n^2)$ sorting algorithm, which is not very efficient. However all intermediate passes may be used as partially sorted sequences, which may be useful if only nearly sorted data is required.

The odd-even merge sort algorithm improves on the odd-even transition sort by sorting the data into two separate sequences recursively, and then merging the sequences to form one sequence. A sorting network implementing the merge step is visualized in Figure 5.1b. The resulting sorting pattern is outlined in the middle of Figure 5.1a. By moving from a linear reduction step to a recursive reduction step, the runtime is shortened from $O(n^2)$ to $O(n \log^2(n) + \log(n))$. This is slower than quick sort's average case ($O(n \log(n))$), but in practice it can outperform quick sort for a specific range of n, because the algorithm is better suited to the processing device. If the sequences are nearly sorted already, quick sort will get near its worst case performance, which favours odd-even merge sort even more.

5.5 Bitonic sort

Bitonic sort is a sorting algorithm which was designed in 1968 by Ken Batcher [Bat68] for parallel hardware implementation using circuits with wires for implementing the network as connected in Figure 5.1c. Figure 5.2 is a schematic drawing of a circuit implementing bitonic sort for 16 elements. Because the algorithm is designed for fixed wires, its memory access is predictable. If the implementation is done correctly, all the requirements for GPU sorting outlined in Section 5.3 are fulfilled.

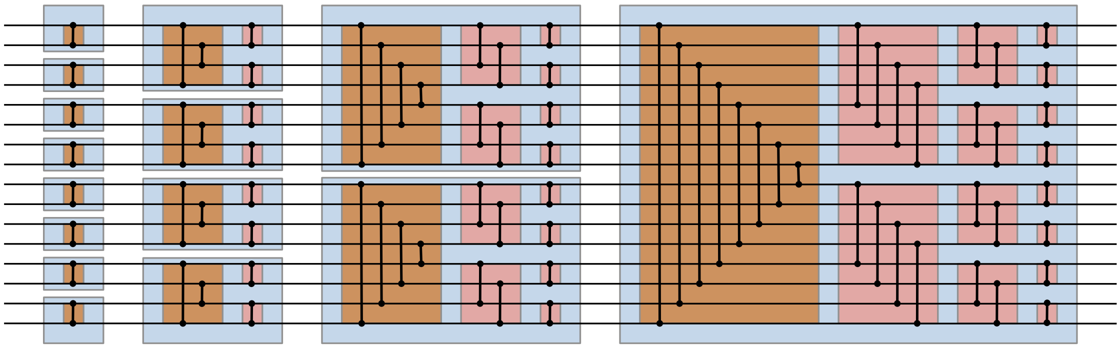


Figure 5.2: Bitonic sort network for 16 elements, figure in public domain from Wikipedia

Bitonic sort can be regarded as an extension of odd-even merge sort, where the merge and compare passes are combined to reduce the amount of work. It works by arranging bitonic sequences recursively for 2 elements into a sequence, expanding the sequence until it occupies all elements to be sorted. [Bat68] defines a bitonic sequence as "the juxtaposition of two monotonic sequences, one ascending, the other descending". This means that a bitonic sequence is two sorted sub sequences of half the size of the full sequence, one of the sequences is sorted in the reverse order.

In the outermost step of the bitonic sorting, the algorithm is modified to produce a fully sorted sequence instead by keeping the same sorting for both subsequence. This produces a normally sorted sequence (as in iteration 4, right part of Figure 5.1a), instead, of a bitonic sequence (as in iteration 3, right part of Figure 5.1a).

5.5.1 Previous bitonic sort implementations

Bitonic sort has been included as an example from all major GPU vendors. AMD and Intel includes an OpenCL example in their Service Development Kit (SDK)'s, and NVIDIA includes an example in their CUDA SDK. The Intel bitonic sort implementation includes some optimizations, but it does not look like it is designed to run on a GPU. The AMD example consists of 50 code lines, and contains no optimizations at all. Both these examples contains no code for synchronizing the sorting across groups. Therefore only one work group is used. This massively underutilizes the GPU, and makes the sorting too slow for any real world use.

NVIDIA however has included a quite complicated example in the CUDA SDK. This example uses shared memory to speed up the computation and multiple kernels and kernel calls to synchronize work between blocks. This ensures that the whole of the GPU is utilized. In the next section I describe how I did the porting of this example to OpenCL and modified it further to suit my application.

5.5.2 Porting bitonic sort from the CUDA SDK examples

First all CUDA specific expressions with direct mappings to OpenCL functions were replaced using preprocessing macros, and search and replace. Very few CUDA features which did not have direct counterparts in OpenCL, were used by the kernels. Some global variables were used, and they had to be moved into the argument lists of the kernels. The comparator function was originally in a separate file, and shared with an odd-even sort CUDA implementation. This function was moved in to the same file as the bitonic sort kernel in the OpenCL port, to simplify linking.

The function calling the kernels was completely rewritten, as OpenCL kernel calls are more cumbersome and detailed than CUDA calls. The shared memory size is also disregarded, as the total thread count per block is smaller than the shared memory size per block for most data types. To simplify debugging and testing of the sorting algorithms, a simple script visualizing the sorted data of the example code provided by NVIDIA was created. This script is included in Section A.3.

5.5.3 Optimizing and adapting bitonic sort for particle sorting

Sorting all particles completely using bitonic sort for each frame, resulted in a 30fps to 20fps drop for 2^{21} particles on a Mac Book Pro with an AMD Radeon 6490M. This fame rate drop was unacceptable for real time performance, and would be very difficult to justify by improved rendering time and features. Therefore schemes for partial sorting were examined.

The first scheme for partial sorting introduced was to create a hybrid of odd-even transitional sort and bitonic sort by modifying the kernel which sorts all elements bounded by the size of an OpenCL group. This results in locally sorted elements inside one block in one iteration. In the next iteration the blocks used for sorting is shifted by half the block length and then sorted in a reverse bitonic sequence. The lower elements of the shifted blocks are thus transferred to the block below and the higher to the block above. This scheme is visualized in Figure 5.3. To transfer an element from the lowest block to the highest block requires as many iterations as there are blocks. For 2^{21} elements with a block size of 512 elements this results in 4096 frames, or 136 seconds. Therefore further optimization was required.

5.5.4 Improving local correctness

To reduce the time used for one element to travel from one end of the array to the other, a step moving data longer distances is introduced. However this step does not produce perfectly sorted sequences, so it is used quite seldom compared to the sorting steps in Section 5.5.3. Using this step each 5th frame can be a good compromise between global and local movement to keep the lists as sorted as possible.

To move data longer distances, each block is divided up to several pieces. Each piece is the length of one coalesced memory access (at the moment defined to 128 as bytes). When there are

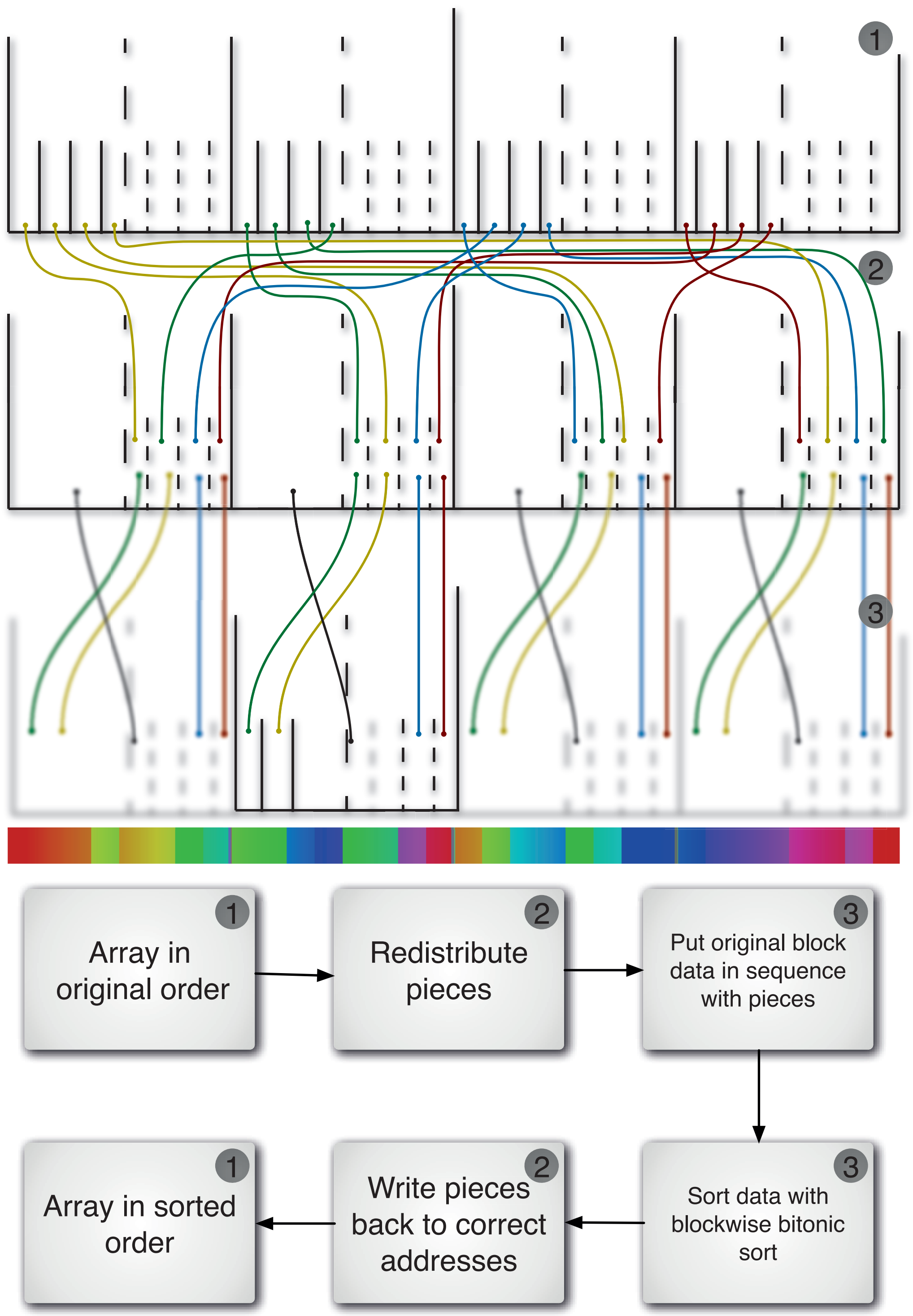

1
2
3
1
Array in original order
2
Redistribute pieces
3
Put original block data in sequence with pieces
3
Sort data with blockwise bitonic sort
2
Write pieces back to correct addresses
1
Array in sorted order

Figure 5.3: Comparison network for improving local correctness, at the bottom: Real network produced by visualization program (position is input, output is color coded on a HSV-scale)

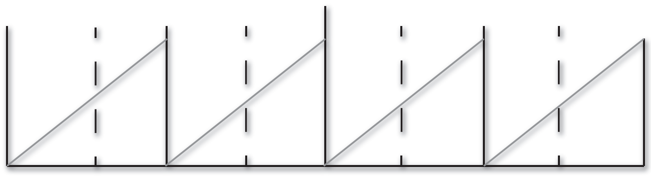
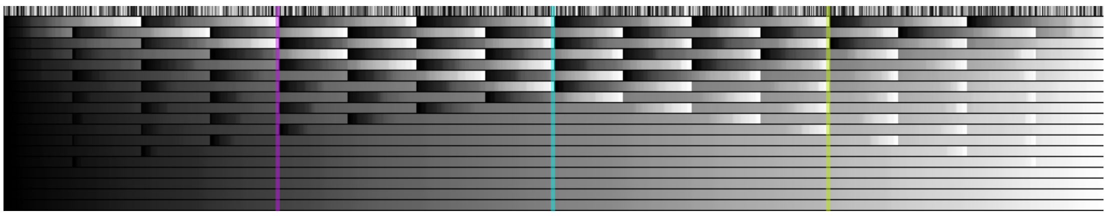
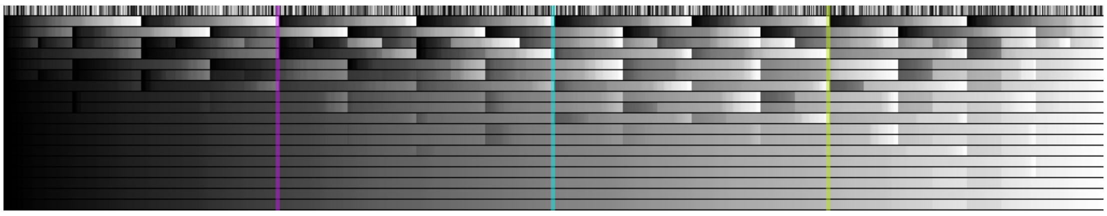


Figure 5.4: Redistribution of blocks to improve local correctness. The diagonal lines represent the index each piece at that point will be compared with.


(a) Only local sorting, with alternating offset


(b) alternating offset, then global improvement (each third line is normal, offseted, global correction sort)

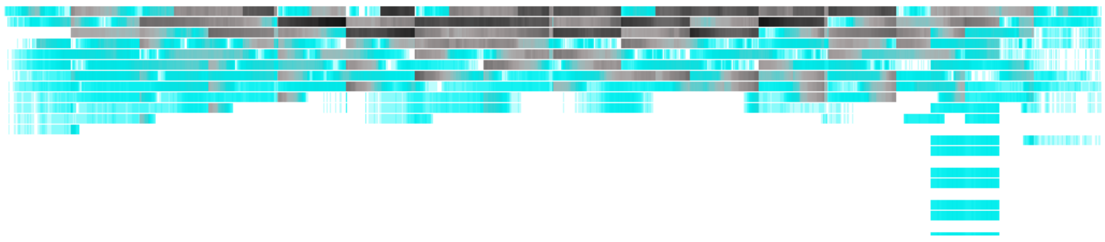

(c) Difference between Figure 5.5a and Figure 5.5b inverted. White means no difference, cyan is enhanced to make small differences visible

Figure 5.5: Bitonic sort: alternating local sort vs alternating local and global sort. The coloured lines are at the same indices as the vertical lines of Figure 5.4

more blocks than pieces, the blocks are shared over pieces belonging to multiple blocks, as in Figure 5.4.

After the pieces are assigned to each block, they are traversed to locate where the block can be inserted to be in order with the pieces. Then the values in the array at the location of all the blocks are read in to shared memory. The blocks are then sorted in shared memory, before the contents pieces are distributed back to their respective position. This process is outlined in Figure 5.3.

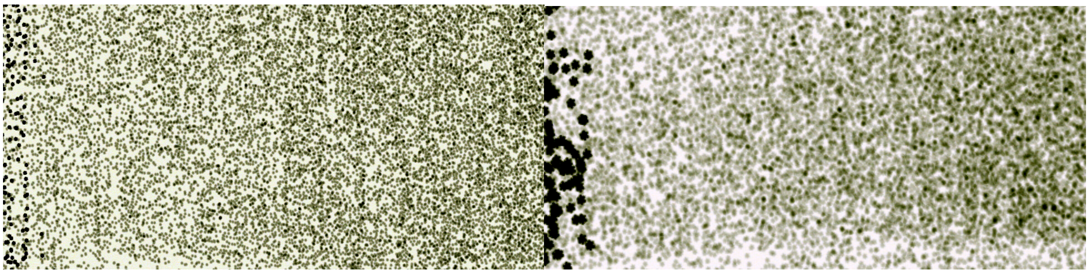
Figure 5.5 shows that both sorting versions uses about the same time to reach a stable completed solution. However the globally improved version displays fewer big differences in the later stages, especially in the lowest and highest values.

All elements in Figure 5.5c which are cyan are small differences which will cause minimal problems for a use case dependent on nearly sorted values. The oscillating elements in the stable solution on the right of Figure 5.5b, are most likely due to a bug in the implementation of the algorithm. However this bug is local to a single block, and can be corrected by running a local sort after the global sort, before the result is used.

5.6 Integration with the snow simulator

After the sorting algorithm was verified graphically, as in Figure 5.5, it was integrated into the move particles function of the snow module in the simulator. A separate class for controlling the sorting algorithms was created. This class was controlled and initialized by the snow system class.

For each frame, one of the sorting functions was called to keep the particles sorted. This caused the particles to change array index. Some parameters related to circular particle movement are randomly generated at the start of the simulation. These parameters are stored in much smaller arrays than the number of particles, and accessed based on the particle index (using modulo to make it fit the smaller arrays). When the array was sorted, the indices of the elements were changed. This caused some artefacts for the particles in the simulator visualized as lines, when showing the trace of the particles. Figure 5.6 is made by capturing a video of the simulator, adding an echo effect to enhance the trace of the particles, and adjusting the colours of a frame of the video.


(a) Rotation of snow particles using array index

(b) Rotation using stored index

Figure 5.6: Artefacts caused by changing the index of particles during simulation

The current state of the circular movement is stored as the forth element of the velocity vector of the particles. This is an angle, which is represented as a value from 0 to 2π. A float variable can store significantly more data. Therefore the offset in the movement tables (integers from 0-31) was combined with the angle and stored in the vector. Thus no extra memory was required to store the offset.

The result of the integration with the snow simulator is presented in Section 6.6.2.

5.7 Future improvements to the sorting algorithm

In a future version the speed of convergence to a nearly correct solution may be improved even further, by altering the distribution of blocks to be make sure that the lower blocks are compared with the upper blocks in every configuration. The snow simulator can also be improved by implementing more features, which will benefit from a sorted list of particles. The comparison function of particles may also be improved, by partitioning the floating point coordinates of the particles in to small bins to simplify the comparison.

Chapter 6

Tests and results

In this chapter the performance and visual results of the simulator will be examined. Different hardware and OpenCL implementations will be tested to see how they affect the performance of the simulator. At the end a brief overview of the visual results and rendering performance of the simulator will be given.

6.1 Methodology, hardware and implementations

Porting the snow simulator to OpenCL made it possible to run it on a number of different platforms. In this chapter the most accessible platforms are tested. The specifications of the platforms tested, are listed in Table 6.1 and 6.2. The table is split into separate tables for each category of systems tested. Because some of the platforms have very different performance, and comparing all platforms at once will result in confusing graphs, the comparison is split in to two parts: CPU versus CPU and GPU versus GPU. In the first part one GPU platform and two CPU platforms running on the same system are compared. The configurations used in this part is listed as 1-4 in Table 6.1. In the second part another system is used to compare different GPU platforms from different vendors. In this part configuration 5-8 from Table 6.2 is used.

Cfg	OS	CPU	GPU	Impl.	Vers
1.	Linux	C2 Quad Q9300@2.5	GeForce GTX 560 Ti	Nvidia	1.1
2.	Linux	C2 Quad Q9300@2.5	GeForce GTX 560 Ti	CUDA	4.2.1
3.	Linux	C2 Quad Q9300@2.5	GeForce GTX 560 Ti	Nvidia, Intel	1.1, 1.1
4.	Linux	C2 Quad Q9300@2.5	GeForce GTX 560 Ti	Nvidia, AMD	1.1, 1.2

Table 6.1: Specifications of the hardware and software used to test the simulator on CPU

Three different aspects of the simulator is tested: Wind simulation, snow simulation and full simulation (wind & snow). The size of the simulation is increased to observe how the performance of the OpenCL implementations vary with different memory requirements and computational complexity.

6.2 Visualization of the results and statistical method

All graphs resulting from the tests, show the simulation size on the x axis and the performance of the simulator on the y axis. The simulation size is sampled and visualized on a logarithmic scale as the number of snow particles or the dimensions of the wind field of the simulator. The sampling is done on a logarithmic scale to fulfil requirements from some of the algorithms and GPGPU APIs which require, or are optimized for problem sizes which is a power of two. The performance loss is often inverse algorithmic which gives a linear graph which is simpler to compare to the other results.

The performance of the simulator along the y axis is, measured in Frames Per Second (FPS). FPS is widely used as a way to describe the performance of real time systems. It has also been used in earlier reports on the snow simulator. Therefore it is easier to compare the results of this thesis and the old thesis when using the same measurement units.

All tests work by running the simulator and measuring the achieved performance. External factors running on the computer may affect the performance differently for different problem sizes and variable combinations. Therefore some of the tests were run two times, especially if the results didn't follow a smooth function. For most of the combinations the noise was negligible. In these cases only one run is included in the graphs. However in some cases the noise was significant. In this case 4-5 runs were performed. Then all the results were averaged and the standard derivation was computed and included as error bars in the graphs.

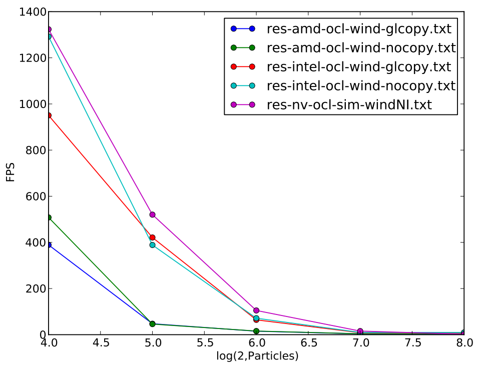
6.3 CPU versus GPU wind simulation

This chapter looks at how the CPU handles executing a wind simulation kernel which is optimized for a certain GPU. When no debug rendering is enabled the wind simulation has very little interaction with the snow simulation and rendering. Therefore the wind simulation was a good candidate for testing CPU implementations of OpenCL. Three OpenCL implementations for CPUs were considered. The Intel SDK for OpenCL Applications [Int12a], the AMD accelerated parallel processing SDK and the OpenCL implementation included in Mac OS X 10.7.4.

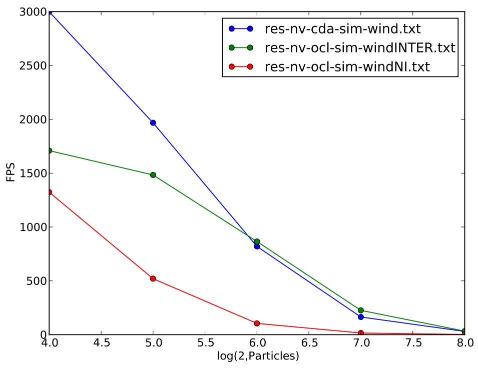
The implementation of OpenCL for CPU's in Mac OS X proved to be too unstable for testing. Code that worked on all the other implementations failed on Mac OS X when ran on a CPU, and when debugging the code testing if a variable was not a number worked once, but not twice in a row. Therefore testing the simulator on CPU on Mac OS X was abandoned.

Originally the simulator uses a standardized extension of OpenCL to transfer the result of the simulations for rendering to OpenGL. The support for this extension is limited for platforms using the CPU as the processor for OpenCL kernels. Therefore the transferring code was rewritten to copy the results of the simulation using the OpenGL API. The transfer of simulation results to the GPU is limited by the bandwidth of the GPU bus. This bottleneck is regarded as one of the largest problems in current GPU design [Ceb04]. Therefore this will probably be one of the biggest bottlenecks of an hybrid CPU - GPU implementation. To see the impact of this bottleneck, all the tests in this section is performed both with and without this transfer.

Figure 6.1a shows how different implementations of wind simulation performs on systems 1, 3 and 4 of Table 6.1. All the implementations do not share the results of the simulation by using the OpenGL-OpenCL interoperability extension. The green and cyan runs does not copy the



(a) Wind simulation without opencl/opengl integration



(b) Wind simulation with opencl/opengl integration

Figure 6.1: Wind simulation only

result at all. Comparing these runs with the blue and red runs gives insight into how big the performance hit of the copying is.

The performance hit of the copying seems smaller than the difference between the OpenCL-implementations. The implementation with the worst performance, is the AMD implementation. The most likely explanation is that the AMD implementation is ran on a CPU produced by Intel, and therefore not tuned for the CPU. The Intel implementation however is nearly on par with the Nvidia GPU-implementation when the memory is copied from the GPU to the CPU and back. It is even faster than the GPU of the MacBook Pro, which averages on about 55 FPS (plotted in Figure 6.4).

6.4 Full simulation

In this section a full simulation, which includes both wind and snow particle simulation is examined. Figure 6.2 outlines a full simulation for CPU, GPU and CUDA. The characteristics of this simulation is very similar to the wind only simulation. The snow simulation (which is illustrated in Figure 6.5 for reference) does not affect the performance of the simulation in any significant way.

6.5 Simulation on different GPUs

All GPU's in this section were tested on the same system which is detailed in Table 6.2. This isolates the implementations and GPUs from external influences from CPUs, system software, motherboards etc. The Tesla-(system 6. of Table 6.2) and Quadro (system 5.) GPU was tested with an NVIDIA driver. Then the cards were removed, the AMD card (system 7.) was inserted and the AMD GPU driver and the AMD APP SDK (OpenCL driver) was installed.

The cards produced by NVIDIA are designed primarily for GPGPU work. These cards are designed for maximum compute performance and to support double precision floating point

Cfg	OS	CPU	GPU	Impl.	Vers
5.	Linux	C2 Quad Q9550@2.83	Quadro FX 5800	Nvidia	1.0
6.	Linux	C2 Quad Q9550@2.83	Tesla C1060	Nvidia	1.0
7.	Linux	C2 Quad Q9550@2.83	ATI Radeon 5870	AMD	1.2
8.	Mac OS X	C i7-2635QM@2.0Ghz	ATI Radeon 6490M	Apple	1.1

Table 6.2: Specifications of the hardware and software used to test the simulator

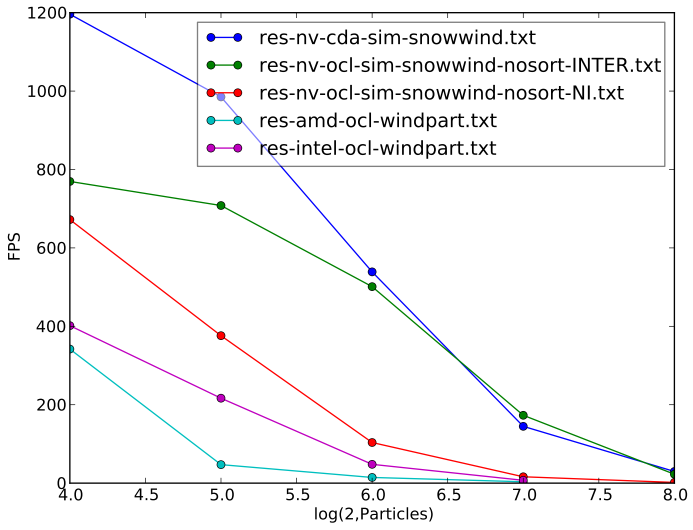


Figure 6.2: Wind & snow simulation on GPU-CPU

variables. The Tesla card is a graphics card without a display. This card is dedicated to GPGPU work. However all results have to be transfered back to the host CPU for further processing or to be displayed.

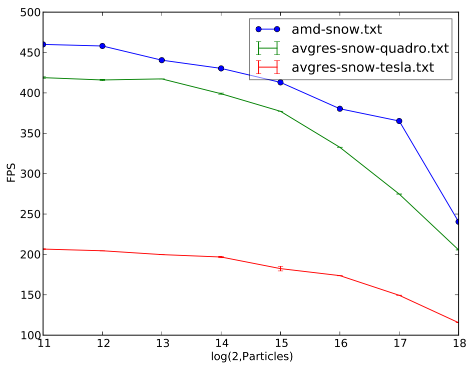
The results of the simulation is plotted in Figure 6.3.

6.5.1 The NVIDIA Tesla card

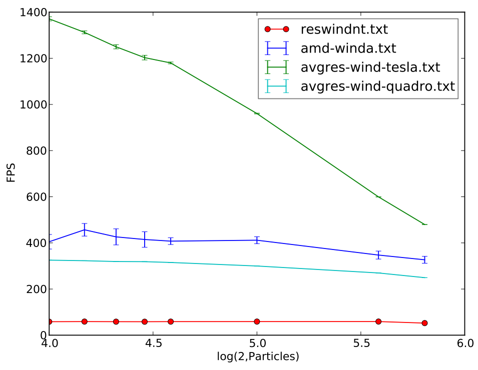
In the snow simulation all elements of the snow simulator were disabled except the snow simulation module. The snow simulation uses the OpenCL / OpenGL for much of it's memory. This is most likely the reason for the bad performance of the NVIDIA Tesla card.

In the wind simulation a separate pass of the Tesla card was performed, and included as the green line. In this run no OpenCL / OpenGL integration was performed. This causes a massive speed up compared to the other cards.

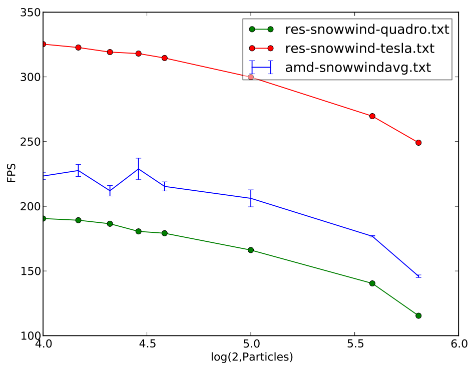
When both the snow and wind simulation was running the Tesla card performed best. In this simulation the computational requirements was much higher compared to the communication requirements. This suits the Tesla card better.



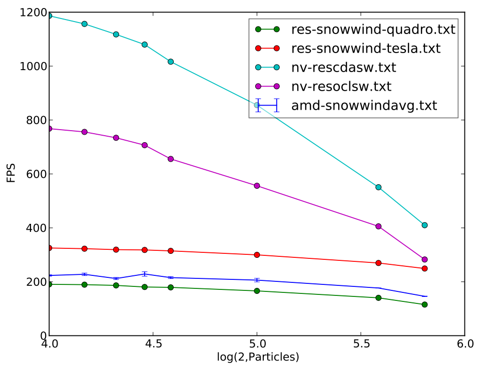
(a) Snow simulation on GPUs



(b) Wind simulation on GPUs



(c) Snow & Wind simulation on GPUs



(d) Snow & Wind simulation on GPUs for all generations

Figure 6.3: Simulation on different GPUs

6.5.2 The GPUS with display output

The Quadro card from NVIDIA is designed for 3D desktop work, like Computer Aided Design (CAD) and modelling 3d objects. Marketwise therefore, it is in the middle ground between a consumer gaming card, and a GPGPU card like Tesla. The performance characteristics of the Quadro card and the AMD card were very similar. All the tests in Figure 6.3 shows that the The AMD card performed about 50 FPS better than the Quadro card. These cards do both support graphics output, and were released nearly at the same date.

6.5.3 Common trends

For most of the runs the frame rate of the simulation is nearly constant until a certain point before it decreases rapidly. However the frame rate of the Tesla GPU without OpenGL interface and the modern GPU of configuration 1 and 2 decreases linearity from the start. This may be explained by improved memory management, which reduces the fixed cost of memory transfer.

In this case the variable performance cost related to the size of the simulation dominates the fixed performance costs even for small problems. Therefore all increases in problem size causes degraded performance. The performance is however higher than the performance of the other GPUs at all times.

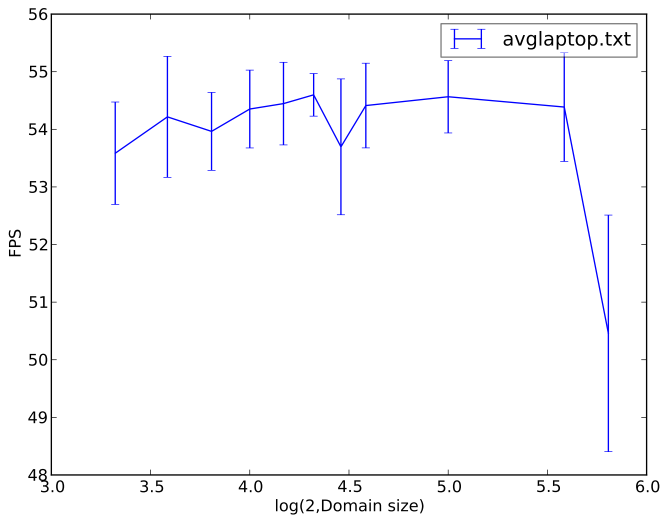


Figure 6.4: Wind simulation on system 8 of Table 6.2 (MacBook Pro)

The results of the simulation for GPUs are very dependent on the type and generation of the GPUs used to run the simulation. All the GPUs tested in Figure 6.3 shows performance in the same ballpark. The more recent GPUs displayed as the top results in Figure 6.3d show double performance compared to the old desktop and GPGPU GPUs. This is expected and can be viewed as a conformation of Moor's Law which suggests that the performance of computer devices doubles each 2nd year.

However the laptop in system 8 of Table 6.2 shows results in Figure 6.4 which are in the magnitude of 55 FPS. This laptop is quite new, and can therefore be compared with the systems 1-4 in Table 6.2. The fact that the results for the laptop is much lower than the results for the desktop shows that different GPU profiles have a huge impact on performance. This indicates that the performance of GPUs for mobile phones may be quite low. It is however difficult to predict this performance because the architecture of the mobile GPUs is quite different from desktop GPUs. However the fact that the architecture is even more focused on locating frequently used memory close to the compute units, makes it probable that memory will become a bottleneck. Similar bottlenecks are visible in Figure 6.3d as the difference between green line which does not transfer the result of the wind simulation and the red line which does transfer the result. In the case of mobile GPUs it is likely that the bottleneck will be bigger, and therefore it is likely that the difference will be more pronounced.

6.6 Rendering performance

The snow simulator gives a graphic output of the results of the simulation. The graphical presentation consists of several elements.

- Terrain
- Snow particles
- Wind debug info: Velocity vectors, pressure

In this section the performance of generating the visual outputs are examined. Then some visual results are presented and discussed.

6.6.1 Terrain rendering performance

When looking for ways to optimize the rendering of the simulator the terrain was identified as the most interesting part to optimize. Rendering terrain has a huge effect on the performance of the simulator. Especially on the MacBook in system 8 of Table 6.2.

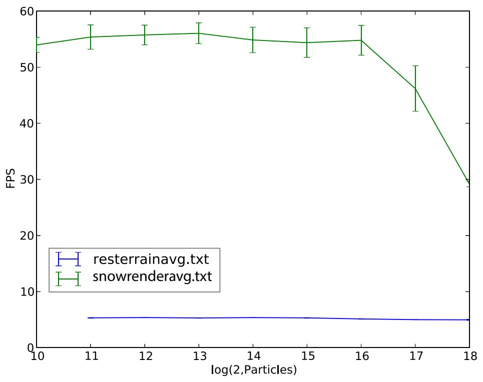
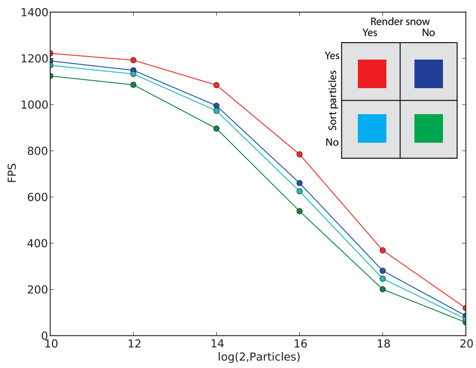


Figure 6.5: Snow simulation with and without terrain rendering on system 8

Figure 6.5 shows this by comparing running a snow simulation with and without rendering terrain. The green line shows snow simulation without rendering terrain. The blue line shows simulation and terrain rendering. The FPS achieved for the simulator without rendering the terrain is 10 times the FPS when rendering it. A fellow student of mine has rewritten the terrain rendering in a thesis which is written in parallel to mine. Therefore I did not focus on optimizing the terrain, and the terrain rendering was disabled in all other tests of the simulator to make sure it did not affect the results of the tests.

6.6.2 Snow rendering performance

Rendering snow is done in OpenGL. Therefore the rendering code is similar for both GPGPU implementations. The original snow particle rendering code rendered some particles with larger



(a) sort vs no sort with and without rendering

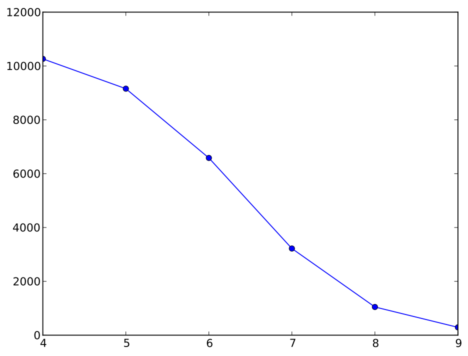

(b) Draw snow

Figure 6.6: snow particle rendering

size for better visual performance. This was turned off during the tests for this thesis to make sure the performance of the rendering was equal in all test cases. All the results for Figure 6.6 are computed on hardware configuration 1 of Table 6.1. Figure 6.6b shows the performance of the OpenGL-only rendering when no simulations are running.

In Figure 6.6a 4 cases are examined. The fact that the distance between the cases with rendering (red and blue) and without rendering (cyan & green) is the same shows that sorting does not improve rendering time.

6.7 Visual results

The goal of porting the snow simulator to OpenCL and adapt it to mobile platforms was to reproduce the visual experience of the simulator as it is when using CUDA on a desktop. This has been done to a certain extent. The simulation of snow in OpenCL works nearly in the same way as the CUDA simulation. The only difference is the initialization, or repositioning, of snow particles at a random position after they have hit the ground. This is done by casting and pointer manipulation in CUDA. OpenCL is a more cross platform API therefore the initialization is done in only floats. This initialization is not properly clamped to the domain of the simulation. This is however managed by the out of bounds code in the snow movement kernel, which moves all particles inside the domain. This is illustrated in Figure 6.7

Several artefacts regarding the snow distribution appears when wind forces are applied. The snow forms curtains where snow particles are concentrated, as shown in Figure 6.8, and areas where nearly no snow particles exists. This may be caused by the distribution function not distributing the particles uniformly if they start to fall at the same position on the ground. The reposition function was modified to try to mitigate the problem, but no working algorithm was found.

The snow simulator contains some debug visualization modes to observe the behaviour of the wind field. The pressure and wind velocity vectors can be studied visually using debug modes

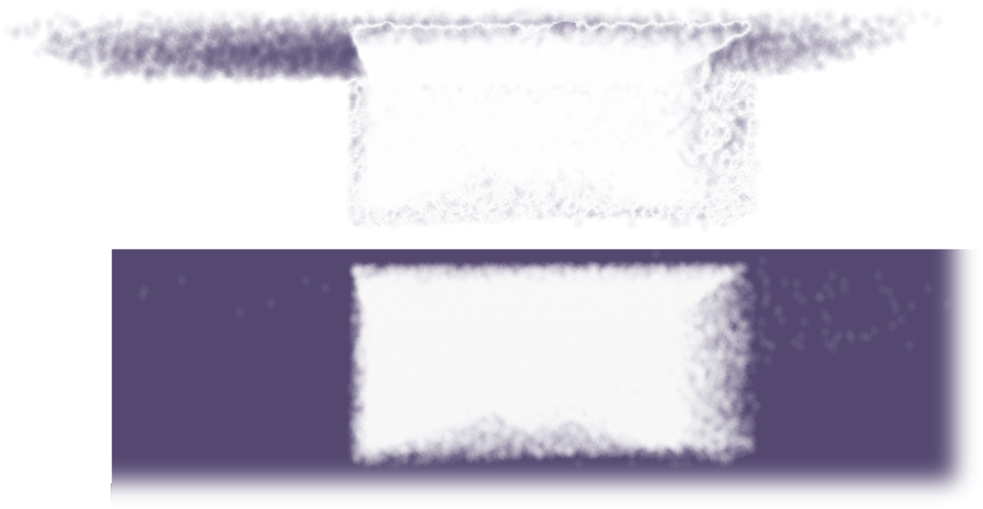


Figure 6.7: Snow outside the domain; upper part: Difference between early frame and after 1 sec. Lower part: After 0.5 sec

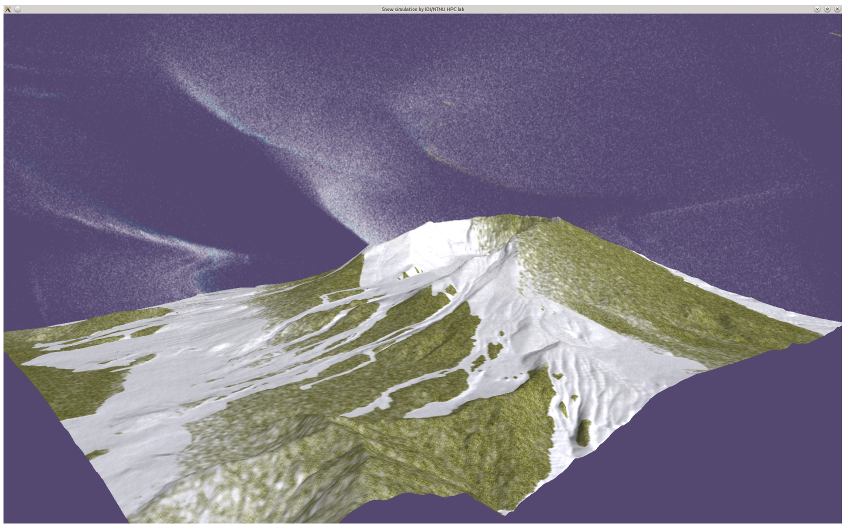


Figure 6.8: Snow forming curtains when wind are enabled because of deficiencies in the reinitialization function

integrated in the simulator. Pressure is visualized by evenly spaced squares as shown in Figure 6.9. The colour of the squares vary with the magnitude of the pressure. Yellow squares means low pressure, while red squares means high pressure.

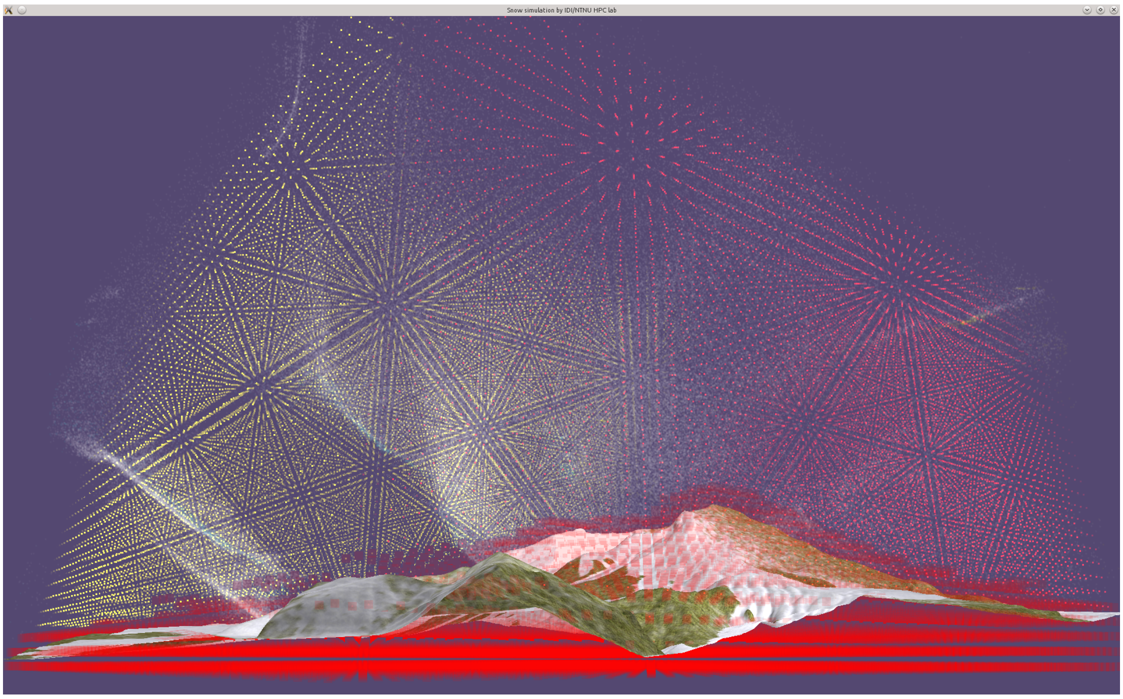


Figure 6.9: Pressure distribution

The wind velocity and direction is visualized by small lines which originate from the same points as the pressure visualization points. This is shown in Figure 6.10. The lines are drawn along the direction of the wind at their origin. The length and colour of the lines are dependent on the velocity of the wind.

The wind of the snow simulator is very stable. In the start of the simulation the difference of pressure causes wind to flow. Soon however the pressure is transferred to a stable pattern depending on the obstacles on the ground. This causes the pressure and wind to settle, especially when simulating with slow frame rates. This may be caused by some scaling factors which have been changed between the OpenCL port and the original CUDA version. Some investigation of this has been done, but more experimentation is required to find the optimal parameters. The main concern of this thesis is however to investigate the performance and characteristics of OpenCL implementations. Therefore making the simulation physically correct has not been an area of focus as long as the simulation is usable for visual inspection and to produce benchmarks.

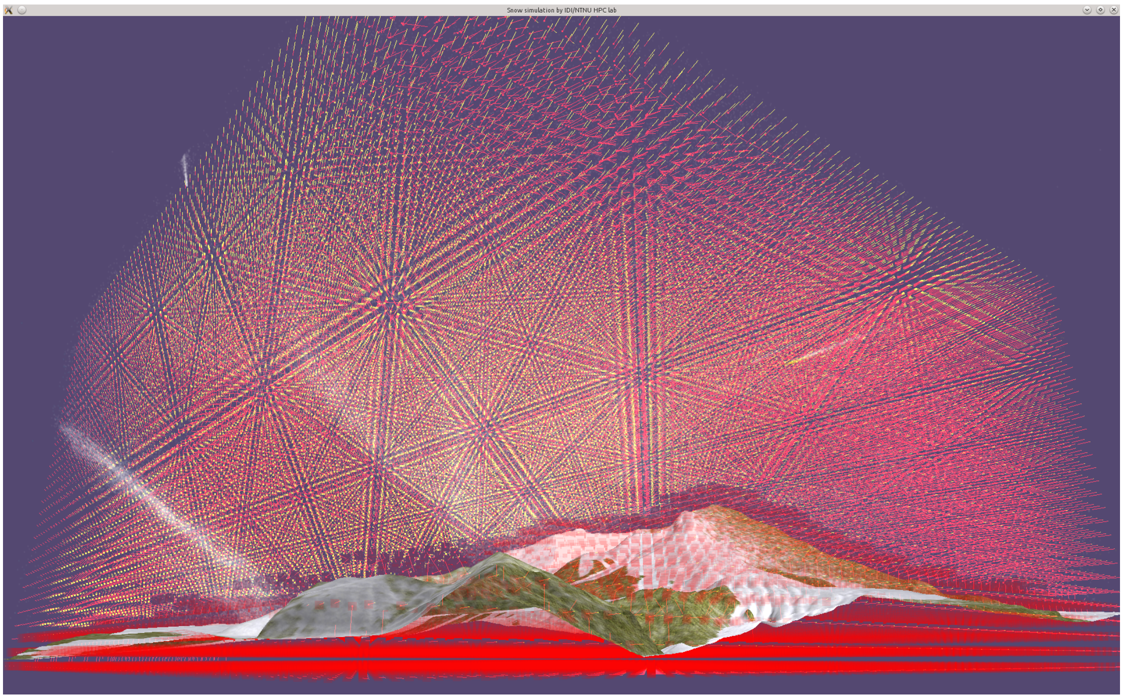


Figure 6.10: Pressure distribution and wind velocities

Chapter 7

Conclusion & Future work

The original goal of this thesis was to investigate modern GPUs for mobile computing by porting a snow simulator to a mobile platform. This included bringing the OpenCL port of the snow simulator done 2 years ago, up to date. Because the existing port could not be found, and the mobile hardware which the simulator was going to run on did not arrive in time, the focus of the thesis was altered. The new goal of the thesis was to prepare the snow simulator for the porting, in order to make it as simple to port as possible when the hardware finally arrives.

7.1 Conclusions

To improve performance of snow particle movement [Eid09], and to enable some more rendering techniques sorting the particles was implemented. However no performance improvement was noticed, most likely due to the fact that sorting the particles degrade performance more than cache locality improves it. For rendering no improvement was made as long as the snow particles were rendered identically.

Porting the snow simulator to OpenCL enabled testing on different devices. The simulator was tested on a CPU with two OpenCL implementations to investigate how CPUs performs compared to GPUs, and to examine how different CPU OpenCL implementations perform on the same hardware. Tests on an Intel CPU indicated that the Intel implementation was faster than the AMD implementation, and on par with a mobile GPU.

The simulator was also tested on multiple GPUs and against the CUDA implementation. In these tests memory management turned out to be the most important performance factor. The CUDA implementation was faster for small problem sizes, and on par with the OpenCL on larger problem sizes. The reason for this is most likely that the CUDA code has been tuned for GPUs using techniques which can not be translated directly to OpenCL.

The CUDA snow simulator code from earlier projects was optimized for speed. Therefore some of the code was quite difficult to understand. Very few comments in the code existed, but all algorithms were described in earlier thesis. However, some implementation details crucial to understanding the code was left out. These details have been figured out and documented in this thesis.

This thesis is intended to serve as a solid base for finishing the port of the snow simulator to a mobile device. The simulator is ported to the OpenCL API which will be available on mobile phones, and verified to work with multiple OpenCL implementations. Preparations for future work on the port are done by documenting and storing the code in a way which simplifies further work to finish the port.

7.2 Future work

The most obvious task for the future in this thesis is to finish the port of the simulator to a mobile device when a device with a GPU supporting OpenCL becomes available.

Another future project could be to look at the ability to execute kernels concurrently by using events. This is a feature introduced to GPGPU by OpenCL, but recently it has also been supported by CUDA. Events can be used to create a dependency network for the OpenCL kernels and memory access functions. Thus the OpenCL implementation can execute the kernels in parallel. Removing serial kernel execution can improve the performance of the simulator, especially if different parts of the simulation can be executed in parallel on multiple devices. For example the wind simulation, snow simulation and rendering can be performed in parallel on three different GPUs.

References

[Aks09] Eirik Ola Aksnes, *Simulation of fluid flow through porous rocks on modern GPUs*, Master's thesis, IDI, Norwegian University of Technology and Science, July 2009.

[AMD] AMD, *A Brief History of General Purpose (GPGPU) Computing*, `http://www.amd.com/us/products/technologies/stream-technology/opencl/pages/gpgpu-history.aspx` accessed May 2012.

[AMS08] T. Akenine-Moller and J. Strom, *Graphics processing units for handhelds*, Proceedings of the IEEE **96** (2008), no. 5, 779–789.

[App10] Apple, *Press release: Apple Launches iPad*, `http://www.apple.com/pr/library/2010/01/27Apple-Launches-iPad.html`, Jan 2010.

[ARM11] ARM, *Mali optimization guide*, `http://infocenter.arm.com/help/topic/com.arm.doc.dui0555a/DUI0555A_mali_optimization_guide.pdf`, 2011.

[Bab11] Kjetil Babington, *Real-Time Ray Tracing for the HPC-lab Snow Simuator*, December 2011.

[Bat68] K. E. Batcher, *Sorting networks and their applications*, Proceedings of the April 30–May 2, 1968, spring joint computer conference (New York, NY, USA), AFIPS '68 (Spring), ACM, 1968, pp. 307–314.

[Boa03] OpenGL Architecture Review Board, *Arb_shading_language_100*, `http://www.opengl.org/registry/specs/ARB/shading_language_100.txt`, 2003.

[BOM11] V.F. Bauset, J.M. Orduna, and P. Morillo, *Performance characterization on mobile phones for collaborative augmented reality (car) applications*, Distributed Simulation and Real Time Applications (DS-RT), 2011 IEEE/ACM 15th International Symposium on, Sept 2011, pp. 52 –53.

[Ceb04] C. Cebenoyan, *Graphics pipeline performance*, GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics, R. Fernando, Ed., Pearson Higher Education (2004).

[Che10] Joel Chelliah, *The NTNU HPC Snow Simulator on the Fermi GPU*, December 2010.

[CT08] D. Cederman and P. Tsigas, *A practical quicksort algorithm for graphics processors*, Algorithms-ESA 2008 (2008), 246–258.

[CvDPH98] Steve Carson, Andries van Dam, Dick Puk, and Lofton R. Henderson, *The history of computer graphics standards development*, SIGGRAPH Comput. Graph. **32** (1998), no. 1, 34–38.

[DSSS05] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier, *High-performance computing: clusters, constellations, MPPs, and future directions*, Computing in Science Engineering **7** (2005), no. 2, 51 – 59.

[Eid09] Robin Eidissen, *Utilizing gpus for real-time visualization of snow*, `http://www.idi.ntnu.no/~elster/master-studs/robine/robin-eidissen-master-ntnu.pdf`, February 2009.

[Els] Anne C. Elster, *IDI HPC-lab web page*, `http://research.idi.ntnu.no/hpc-lab` accessed May 2012.

[Fal06] Falanax (now ARM), *Competitive advantages of the Mali graphics Architecture*, `http://www.design-reuse.com/articles/9591/competitive-advantages-of-the-mali-graphics-architecture.html`, 2006.

[Gje09] Alexander Gjermundsen, *LBM vs SOR solvers on GPUs for real-time snow simulations*, Master's thesis, IDI, Norwegian University of Technology and Science, 2009.

[Gro04] Khronos Group, *OpenGL ES Specification*, `http://www.khronos.org/registry/gles/specs/1.0/opengles_spec_1_0.pdf`, 2004.

[IDI] IDI, *Digital Arkivering og Innlevering av Masteroppgaver*, `http://daim.idi.ntnu.no/` accessed May 2012.

[Ima12] Imagination techologies, *PowerVR series 6 announcement*, `http://www.imgtec.com/news/Release/index.asp`, Jan 2012.

[Int12a] Intel, *Intel SDK for OpenCL Applications 2012*, `http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/`, 2012.

[Int12b] Intel, *Intel SDK for OpenCL* Applications 2012 Release Notes*, `http://software.intel.com/en-us/articles/opencl-release-notes/`, 2012.

[JS09] R. Jarrett and P. Su, *Building tablet pc applications*, O'Reilly Media, Incorporated, 2009.

[KBR04] John Kessenich, Dave Baldwin, and Randi Rost, *The open shading language*, `http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.10.59.pdf`, 2004.

[Kro10] Øystein Eklund Krog, *GPU-based Real-Time Snow Avalanche Simulations*, June 2010.

[KSW04] P. Kipfer, M. Segal, and R. Westermann, *Uberflow: a gpu-based particle engine*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, ACM, 2004, pp. 115–122.

[Lie11] Hallgeir Lien, *Procedural Generation of Roads for use in the Snow Simulator*, December 2011.

[Mar08] Katie Marsal, *Apple proposes OpenCL as high-speed computing standard*, `http://www.appleinsider.com/articles/08/06/17/apple_proposes_opencl_as_high_speed_computing_standard.html`, Jun 2008.

[Mes] Mesa, *Mesa 3D Graphics Library*, `http://www.mesa3d.org` accessed May 2012.

[MG10] Duane G. Merrill and Andrew S. Grimshaw, *Revisiting sorting for gpgpu stream architectures*, Proceedings of the 19th international conference on Parallel architectures and compilation techniques (New York, NY, USA), PACT '10, ACM, 2010, pp. 545–546.

[Mor11] Timothy Prickett Morgan, *ARM Holdings eager for PC and server expansion*, `http://www.theregister.co.uk/2011/02/01/arm_holdings_q4_2010_numbers`, Feb 2011.

[NVI01] NVIDIA, *Press release: Nvidia introduces geforce3–breaks new ground in the quest for real-time cinematic graphics*, `http://www.nvidia.com/object/IO_20010530_6131.html`, Feb 2001.

[Pow09] PowerVR, *PowerVR MBX Technology Overview*, `http://www.imgtec.com/factsheets/SDK/PowerVR%20Technology%20Overview.1.0.2e.External.pdf`, May 2009.

[PSHL10] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, *Fast in-place sorting with cuda based on bitonic sort*, Parallel Processing and Applied Mathematics (2010), 403–410.

[Rov11] Jan Rovde, *Fluid simulations using SPH on GPUs*, December 2011.

[Sal06] Ingar Saltvik, *Parallel Methods for Real-Time Visualization of Snow*, Master's thesis, IDI, Norwegian University of Technology and Science, June 2006.

[Sha05] Dinesh C. Sharma, *CNet: Nokia debuts linux based web device*, `http://news.cnet.com/Nokia-debuts-Linux-based-Web-device/2100-1041_3-5720066.html`, June 2005.

[SKP10] BVN Silpa, G. Krishnaiah, and P.R. Panda, *Rank based dynamic voltage and frequency scaling fortiled graphics processors*, Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, ACM, 2010, pp. 3–12.

[Ste10] Jarle Erdal Steinsland, *Porting the OpenCL snow simulator to opencl for mobile applications*, December 2010.

[Val12] Theo Valich, *Nvidia tegra 4 to get GPGPU*, `http://vr-zone.com/articles/nvidia-tegra-4-to-get-gpgpu-i.e.-gpu-computational-capabilities-kepler-inside-/15361.html`, Mar 2012.

[Ver12] Luc Verhaegen, *The Lima Driver: Liberating the ARM Mali GPU*, `http://people.freedesktop.org/~libv/FOSDEM2012_lima.pdf`, May 2012.

[WMV] WMVare, *Gallum3D GPU driver architecture - open source project page*, `http://wiki.freedesktop.org/wiki/Software/gallium` accessed May 2012.

Appendices

Appendix A

Source Code

A.1 Simulation GPU code

A.1.1 WindSystem.cl

```
//#define CL_USE_TEXTURES
#define REF(p,a,b,c) (p)[((b)*dim.z+(c))*dim.x+(a)]
#ifdef CL_USE_TEXTURES
#pragma OPENCL EXTENSION cl_khr_3d_image_writes : enable
#define WINDVEL_TYPE __read_only image3d_t
#define WINDVEL_TYPE_W __write_only image3d_t
#define WIND_SAMPLES(wind, x, y, z) sample_trilinear(wind, (float4){(float)x, (float)y, (float)z, 0}, dim)
#define WIND_SAMPLE(wind, pos) sample_trilinear(wind, (float4){(float)pos.x,(float)pos.y,(float)pos.z, 0}, dim)
#define WIND_WRITE(wind, pos, value) write_imagef(wind, pos, value)
#define WIND_WRITES(wind, x, y, z, value) {int4 www_pos = (int4){(int)(x), (int)(y), (int)(z), 0}; \
                                        write_imagef(wind, www_pos, value);}
#else
#define WINDVEL_TYPE __global __read_only float4*
#define WINDVEL_TYPE_W __global float4*
#define WIND_SAMPLES(wind, x, y, z) REF(wind, (int)x, (int)y, (int)z)
#define WIND_SAMPLE(wind, pos) REF(wind, (int)pos.x, (int)pos.y, (int)pos.z)
#define WIND_WRITE(wind, pos, value) WIND_SAMPLE(wind,pos) = (value)
#define WIND_WRITES(wind, x, y, z, value) WIND_SAMPLES(wind,x,y,z) = (value);
#endif
#define WIND_SAMPLE_OFFSET(wind, pos, a, b, c) WIND_SAMPLES(wind, pos.x+a, pos.y+b, pos.z+c)
#define INIT() \
    int y = get_global_id(0) / (dim.x*dim.z); \
    int z = (get_global_id(0) - (y * (dim.x*dim.z))) / dim.x; \
    int x = get_global_id(0) - (y * (dim.x*dim.z)) - (z*dim.x);

#define VOX_SELF (1<<0)
#define VOX_LEFT (1<<1)
//..
#define thread_x ((unsigned int) get_local_id(0))
#define block_x ((unsigned int) get_group_id(0))

float4 sample_trilinear(__read_only image3d_t image, float4 pos, int4 dim){
    const sampler_t  sampler = CLK_ADDRESS_CLAMP|
        CLK_FILTER_LINEAR;
    return read_imagef(image, sampler, pos);
}
#else
float4 sample_trilinear(__global __read_only float4* array, float4 pos, int4 dim){
#define REFTLF4(p,a,b,c) REF(p, (int)pos.x+a,(int)pos.y+b,(int)pos.z+c)
#define POSADJ(pos,val) ((pos)==0?1.0f-(val):(val))
    pos.x = max(pos.x, 0.0f); // for y & z too
    pos.x = min(pos.x, dim.x-0.99f); // for y & z too
    float4 fpos = (float4){pos.x-(float)((int)pos.x), pos.y-(float)((int)pos.y), pos.z-(float)((int)pos.z), 0};
    //REFTLF4(array, k,i,j)
    //Unrolled loop:
    float4 ival, jval, kval;
    kval = REFTLF4(array, 0,0,0) * POSADJ(0, fpos.x);
    kval+= REFTLF4(array, 1,0,0) * POSADJ(1, fpos.x);
    jval = kval * POSADJ(0, fpos.z);
    kval = REFTLF4(array, 0,0,1) * POSADJ(0, fpos.x);
    kval+= REFTLF4(array, 1,0,1) * POSADJ(1, fpos.x);
    jval+= kval * POSADJ(1, fpos.z);
    ival = jval * POSADJ(0, fpos.y);
    kval = REFTLF4(array, 0,1,0) * POSADJ(0, fpos.x);
    kval+= REFTLF4(array, 1,1,0) * POSADJ(1, fpos.x);
    jval = kval * POSADJ(0, fpos.z);
    kval = REFTLF4(array, 0,1,1) * POSADJ(0, fpos.x);
    kval+= REFTLF4(array, 1,1,1) * POSADJ(1, fpos.x);
    jval+= kval * POSADJ(1, fpos.z);
    ival+= jval * POSADJ(1, fpos.y);
    return ival;
}
#endif
__kernel void wind_advect(WINDVEL_TYPE wind_vel,
                          WINDVEL_TYPE_W wind_vel_write,
                          __write_only __global float *pressure,
                          __read_only __global int* obstacle,
                          __read_only int4 dim,
                          __read_only float delta_time,
                          __read_only float4 boundary){
    INIT();
    float4 v;
    int mask = REF(obstacle, x, y, z);
    v = WIND_SAMPLES(wind_vel, x, y, z);

    if(x > 0 && x < dim.x-1 && y > 0 && y < dim.y-1 && z > 0 && z < dim.z-1) {
        v.x = (float)x - delta_time * v.x - 0.5f;
        v.y = (float)y - delta_time * v.y - 0.5f;
        v.z = (float)z - delta_time * v.z - 0.5f;
        v = sample_trilinear(wind_vel, v, dim);//_sample(v); //FIXME: Do some filtering here?

        if((mask & (VOX_SELF | VOX_LEFT | VOX_RIGHT))) v.x = 0;
        if((mask & (VOX_SELF | VOX_ABOVE | VOX_BELOW))) v.y = 0;
        if((mask & (VOX_SELF | VOX_UP | VOX_DOWN))) v.z = 0;

        if(y == dim.y-2){
            WIND_WRITES(wind_vel_write, x, y+1, z, v)
        }

    } else {
        v = boundary;

    }
    WIND_WRITES(wind_vel_write, x, y, z, v)
    REF(pressure, x, y, z) = 0.0f;
}

//Z (block),X (thread) - y loop
__kernel void build_solution2(
                          __local float4 *shared,
                          __read_only WINDVEL_TYPE wind_vel,
                          __global float *solution,
                          __read_only __global int* obstacle,
                          __read_only int4 dim,
                          __read_only float factor){
     #define YADVANCE dim.x*dim.z
    int active = (thread_x-1) <= (dim.x+1) && thread_x < (unsigned int) dim.x-1;
    //make relevant pointers for this thread
    __read_only __global int *loc_obs = &obstacle[(1+block_x)*dim.x+thread_x];
    __global float *loc_sol = &solution[(1+block_x)*dim.x+thread_x];
    int4 wvpos = (int4){thread_x,0,1+block_x,0};
    float last = WIND_SAMPLE(wind_vel, wvpos).x;
    wvpos.y+=1;
    //Along the whole y-border
    for(unsigned int y = 1; y < dim.y-1; ++y) {
        float res = -last;
        last = WIND_SAMPLE(wind_vel, wvpos).y;
        //Syncronize resolution
        barrier(CLK_LOCAL_MEM_FENCE);
        if(active) {
            res += WIND_SAMPLE_OFFSET(wind_vel, wvpos, 1,0,0).x;
            res -= WIND_SAMPLE_OFFSET(wind_vel, wvpos,  -1,0,0).x;
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        res += WIND_SAMPLE_OFFSET(wind_vel, wvpos, 0,0,1).z;
        res -= WIND_SAMPLE_OFFSET(wind_vel, wvpos, 0,0,-1).z;
        //Next z 2d-plane - Advance windvel
        wvpos.y++;
        res += WIND_SAMPLE(wind_vel, wvpos).y;
        res *= factor;
        //Advance Obstacle
        loc_obs = &loc_obs[YADVANCE];//y++
        //Advance Pressure
        loc_sol = &loc_sol[YADVANCE];//y++
        //Take obstacle in to account
        res *= (float)((*loc_obs & VOX_SELF) == 0);
        if(active) {
            *loc_sol = res;
        }
    }
}

__kernel void solve_poisson2(__local float *block,
                             __local float *prev,
                             __local int *masks,
                             __global float *pressure,
                             __read_only __global float *solution,
                             __read_only __global int* obstacle,
                             __read_only __global float* poisson_tab,
                             __read_only int4 dim,
                             __read_only float w,
                             __read_only int which
                            ) {
    //divide shared into different memory areas
    float curr;
    int mask;
    int x = get_local_id(0);
    int z = get_group_id(0)+1;

    prev[x] = REF(pressure, x, dim.y-1, z);
    block[x] = REF(pressure, x, dim.y-2, z);

    //#pragma unroll
    for(int y = dim.y-2; y > 1; y--) {
        barrier(CLK_LOCAL_MEM_FENCE);

        mask = REF(obstacle, x, y, z) & 127;

        if((~mask & 126) && (mask & VOX_SELF) == 0) {
            float res;
            //Does this depend on some fancy overflow?
            if((unsigned)(x-1) <= (unsigned)(dim.x+1)) {
                res = 0;
                res += block[x-1] * (float)((mask & VOX_LEFT) == 0);
                res += block[x+1] * (float)((mask & VOX_RIGHT) == 0);

                res += REF(pressure, x, y, z-1) * (float)((mask & VOX_UP) == 0);
                res += REF(pressure, x, y, z+1) * (float)((mask & VOX_DOWN) == 0);

                curr = block[x];
            }
            barrier(CLK_LOCAL_MEM_FENCE);
            block[x] = REF(pressure, x, y-1, z);
            if((unsigned)(x-1) <= (unsigned)(dim.x+1)) {
                res += prev[x] * (float)((mask & VOX_BELOW) == 0);
                res += block[x] * (float)((mask & VOX_ABOVE) == 0);
                res -= REF(solution, x, y, z);
                res *= poisson_tab[mask>>1];
                res *= w;
                res += curr * (1.0f - w);
                prev[x] = curr;
                REF(pressure, x, y, z) = res;
            }
        }
        else {
            REF(pressure, x, y, z) = 0.0f;
        }
    }
```

```
}
__kernel void set_boundary2(__global float *pressure,
                             __read_only __global int* obstacle,
                             __read_only int4 dim,
                             __read_only int which
                            ) {
    INIT();
    int mask = REF(obstacle, x, y, z) & 127;
    if(x > 0 && x < dim.x-1 && y > 0 && y < dim.y-1 && z > 0 && z < dim.z-1) {
        if((mask & VOX_SELF) && mask != 127) {
            if(!(mask & VOX_LEFT))
                REF(pressure, x, y, z) = REF(pressure, x-1, y, z);
            else if(!(mask & VOX_RIGHT))
                REF(pressure, x, y, z) = REF(pressure, x+1, y, z);
            else if(!(mask & VOX_UP) //up, down, above, below ...
        }
        if(x == 1 && which & VOX_LEFT)
            REF(pressure, 0, y, z) = REF(pressure, 1, y, z);
        if(x == dim.x-2 && which & VOX_RIGHT) //up, down, above, below ...
    }
}

__kernel void wind_project2(__local float *shared,
                            WINDVEL_TYPE wind_vel,
                            WINDVEL_TYPE_W wind_vel_write,
                            __read_only __global float *pressure,
                            __read_only int4 dim,
                            __read_only float factor
                            ) {

    //Block = our part of shared memory
    local float *block = &shared[get_local_id(0)];//does the have to be local or private?

    int active = (unsigned)(thread_x-1) <= (unsigned)(dim.x+1) && thread_x < dim.x-1;
    //make relevant pointers for this thread
    int4 wvpos = (int4){thread_x,0,1+block_x,0};
    __read_only __global float *loc_press = &pressure[(1+block_x)*dim.x+thread_x];

    //init loop variables
    float last = loc_press[0];
    loc_press = &loc_press[YADVANCE];//y++?
    block[0] = *loc_press;
    //Along the whole y-border
    for(unsigned int y = 1; y < dim.y-1; ++y) {
        wvpos.y++;
        float4 v = WIND_SAMPLE(wind_vel, wvpos);
        v.z -= factor * loc_press[dim.x];
        v.z += factor * loc_press[-dim.x];

        barrier(CLK_LOCAL_MEM_FENCE);
        if(active) {
            v.x -= factor * block[1];
            v.x += factor * block[-1];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        v.y += factor * last;
        last = block[0];
        loc_press = &loc_press[YADVANCE];//y++
        block[0] = *loc_press;
        v.y -= factor * block[0];
        if(active) {
           WIND_WRITE(wind_vel_write, wvpos, v);
        }
    }
}
```

A.1.2 SnowSystem.cl

```
//#define CL_USE_TEXTURES
#ifdef CL_USE_TEXTURES
#define WINDVEL_TYPE __read_only image3d_t
#else
#define WINDVEL_TYPE __global __read_only float4*
#endif
```

```
void incrementHeight(int4 terrain_dim, int x, int y, float amount, __global float4 *grid) {
  if(x < 0) x = 0; // & y
  if(x >= terrain_dim.x) x = terrain_dim.x-1; // & y
  int ix = terrain_dim.x * y + x;
  if(grid[ix].w < 1.0f)
    grid[ix].w += amount;
}

int2 calcGridPos(float4 p, int4 terrain_dim, int4 scene_dim) {
    int2 gridPos;
    // for both .x and .y:
    gridPos.x = floor((float)terrain_dim.x*p.x/scene_dim.x);
    if(gridPos.x < 0) gridPos.x = 0;
    if(gridPos.x >= terrain_dim.x) gridPos.x = terrain_dim.x-1;

    return gridPos;
}

#define SG_STRIDE 18

__constant int sg_off[4] = { -1, -SG_STRIDE, 1, SG_STRIDE };

__kernel void smooth_ground(__global float4 *grid,
                            __local float4 *shared,
                            int4 terrain_dim,
                            int4 gridDim,
                            int4 scene_dim) {
  const int tx = get_local_id(0);
  const int ty = get_local_id(1);

  __local float4 *block = &shared[(ty+1) * SG_STRIDE +tx+1];
    grid = &grid[get_global_id(1)*terrain_dim.y + get_global_id(0)];
  block[0] = grid[0];

  if(tx == 0) {
    if(get_group_id(0) == 0)
      block[-1] = block[0];
    else
      block[-1] = grid[-1];
  }
  if(tx == get_local_size(0)-1) {
    if(get_group_id(0) == get_num_groups(0)-1)
          block[1] = block[0];
    else
      block[1] = grid[1];
  }
  if(ty == 0) {
    if(get_group_id(1) == 0)
      block[-SG_STRIDE] = block[0];
    else
      block[-SG_STRIDE] = grid[-terrain_dim.y];
  }
  if(ty == get_local_size(1)-1) {
    if(get_group_id(1) == get_num_groups(1)-1)
      block[SG_STRIDE] = block[0];
    else
      block[SG_STRIDE] = grid[terrain_dim.y];
  }

    barrier(CLK_LOCAL_MEM_FENCE);

  float mod = 0.0f;
  const float a = block[0].y + block[0].w;
#pragma unroll
  for(int i = 0; i < 4; ++i) {
    int k = sg_off[i];
    float b = block[k].y + block[k].w;
    float diff = b - a;

    if(fabs(diff) > 0.05f) {
      // I am highest
      if(diff < 0) {
        if(block[0].w > 0.1f) {
          float snow = block[0].w;
          if(-diff < snow)
            snow = -diff;
          mod -= 0.05f * snow;
        }
      }
      else {
        if(block[k].w > 0.1f) {
          float snow = block[k].w;
          if(diff < snow)
            snow = diff;
          mod += 0.05f * snow;
        }
      }
    }
  }

    barrier(CLK_LOCAL_MEM_FENCE);
  block[0].w += mod;
  grid[0] = block[0];
}

void reposition(float4 *pos, int4 scene_dim) {
//Shuffle position bits around to get a new randomized position at top
    float temp = (*pos).x;
    int3 ipos = (int3){(int)((*pos).x*100.0f), (int)((*pos).y*100.0f), (int)((*pos).z*100.0f)};
    (*pos).x = (ipos.x%750 - ipos.z%500 + ipos.y % 20) / 1.0f;
    (*pos).z = (ipos.z%300 + ipos.x%300 + ipos.y % 20) / 1.0f;
  (*pos).y = (float)scene_dim.y - 2.0f;
}

__kernel void part_update(__global float4 *part_pos,
                          __global float4 *part_vel,
                          __global float4 *grid,
                          WINDVEL_TYPE wind_vel,
                          __global float *omegas,
                          __global float *radiuses,
                          __global __read_only float *convert,
                          int4 wind_dim,
                          int4 scene_dim,
                          int4 terrain_dim,
                          float delta_time,
                          float snow_growth,
                          float gravity) {
  int tid = get_global_id(0);
  float4 pos = part_pos[tid];
    float4 vel = part_vel[tid];
    float4 v_drag = sample_trilinear(wind_vel, (float4){pos.x*convert[0] + 1.0f, pos.y*convert[1] + 1.0f, pos.z*↩
         convert[2] + 1.0f, 0.0f}, wind_dim);

  v_drag.x -= vel.x;
  v_drag.y -= vel.y;
  v_drag.z -= vel.z;
  float temp = v_drag.x*v_drag.x + v_drag.y*v_drag.y + v_drag.z*v_drag.z;
  float v_fluid_abs = native_sqrt(temp);
    temp = v_fluid_abs;
  v_drag.x *= temp*gravity*vel.w;
  v_drag.y *= temp*gravity*vel.w;
  v_drag.z *= temp*gravity*vel.w;

    float ax = v_drag.x;
    float ay = -gravity + v_drag.y;
    float az = v_drag.z;

    //Extract rot_index and theta from pos.w
    int rot_idx = (pos.w/(2*M_PI));
    float rot_val = pos.w-((rot_idx)*(2*M_PI));
    rot_idx--;

  float rot_omega = omegas[rot_idx & 31];
    rot_val += rot_omega*delta_time;

    //and put it pback
    rot_val-=(rot_val>2*M_PI)*2*M_PI;//wrap rot_val to make it inside allocated space
    pos.w = rot_val+((rot_idx+1)*2*M_PI);//Store back in pos.w

  rot_omega *= radiuses[rot_idx & 31];
  float v_abs = fmax((float)native_sqrt(vel.x*vel.x + vel.y*vel.y + vel.z*vel.z), 1.0f);
  float v_circ_x = -sin(rot_val) * rot_omega;
  float v_circ_z = cos(rot_val) * rot_omega;

  pos.x += (vel.x + v_circ_x)*delta_time + 0.5f*ax*delta_time*delta_time;
  pos.y += vel.y*delta_time + 0.5f*ay*delta_time*delta_time;
  pos.z += (vel.z + v_circ_z)*delta_time + 0.5f*az*delta_time*delta_time;

    vel.x += ax*delta_time;
  vel.y += ay*delta_time;
  vel.z += az*delta_time;
  part_vel[tid] = vel;

  if(pos.y < -1.0f) pos.y = scene_dim.y-2.0f;//+= scene_dim.y;
  else if(pos.y > scene_dim.y) {
    reposition(&pos, scene_dim);
        float4 wind = sample_trilinear(wind_vel, (float4){pos.x*convert[0] + 1.0f, pos.y*convert[1] + 1.0f, pos.z*↩
            convert[2] + 1.0f, 0.0f}, wind_dim);
    part_vel[tid] =  (float4){wind.x, wind.y, wind.z, vel.w};
  }

    int moved = 0;
    if(pos.x < 0.0f) {
        pos.x += scene_dim.x;
        moved = 1;
    }
    else if(pos.x > scene_dim.x) {
        pos.x -= scene_dim.x;
        moved = 1;
    }
    if(pos.z < 0.0f) {
        pos.z += scene_dim.z;
        moved = 1;
    }
    else if(pos.z > scene_dim.z) {
        pos.z -= scene_dim.z;
        moved = 1;
    }
  int2 ip = calcGridPos(pos, terrain_dim, scene_dim);

  float4 hv = grid[terrain_dim.x * ip.y + ip.x];
  float h = hv.y + hv.w;
    snow_growth = 0.064f;
  if(pos.y < h) {
    if(moved) {
      reposition(&pos, scene_dim);
            float4 wind = sample_trilinear(wind_vel, (float4){pos.x*convert[0] + 1.0f, pos.y*convert[1] + 1.0f, ↩
                pos.z*convert[2] + 1.0f, 0.0f}, wind_dim);
      part_vel[tid] =  (float4){wind.x, wind.y, wind.z, vel.w};
    }
    else if (ip.x != terrain_dim.x-1 && ip.y != terrain_dim.y-1) {
      incrementHeight(terrain_dim, ip.x, ip.y, snow_growth*0.15f, grid);
      //... lots for more increment height function calls, which are removed to shorten the code in the appendix
      reposition(&pos, scene_dim);
            float4 wind = sample_trilinear(wind_vel, (float4){pos.x*convert[0] + 1.0f, pos.y*convert[1] + 1.0f, ↩
                pos.z*convert[2] + 1.0f, 0.0f}, wind_dim);
      part_vel[tid] =  (float4){wind.x, wind.y, wind.z, vel.w};
    }
  }
  part_pos[get_global_id(0)] = pos;
}
```

A.2 Sorting

A.2.1 BitonicSort.cl

In this section only significantly modified functions are included to try to keep the appendix in a managable size.

```
/*
 * Bitonic sort, based on NVIDIA CUDA bitonic sort example which is
 * Copyright 1993-2010 NVIDIA Corporation, but derivative can be used as long as
 * "This software contains sorce code provided by NVIDIA Corporation" is in the documentation.-
 *
 * Changes copyright 2012 Frederik M.J. Vestre under BSD the license with attribution.
 *
 */
//Based on http://www.iti.fh—flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm

#define blockIdx ((unsigned int) get_group_id(0))
#define blockDim ((unsigned int) get_num_groups(0))
#define threadIdx ((unsigned int) get_local_id(0))
#define threadDim ((unsigned int) get_local_size(0))

//Remove, define as dependent 2*threadDim (*var_size)
#define SHARED_SIZE_LIMIT shared_size_limit
#define SORT_TYPE float4
#define KV_SORT
#define VAL_TYPE float4
#define MAKEKEY(key) (((((*key).z*1000)+(*key).y)*10.0)+(*key).x)

inline void Comparator(
    __local SORT_TYPE *keyA,
    __local SORT_TYPE *keyB,
#ifdef KV_SORT
    __local VAL_TYPE *valA,
    __local VAL_TYPE *valB,
#endif
    int dir
){
    SORT_TYPE tk;
#ifdef KV_SORT
    VAL_TYPE tv;
#endif
    if( (MAKEKEY(keyA) > MAKEKEY(keyB)) == dir ){
        tk = *keyA; *keyA = *keyB; *keyB = tk;
#ifdef KV_SORT
        tv = *valA; *valA = *valB; *valB = tv;
#endif
    }
}
//Because of picky amd :(
inline void ComparatorLoc(
     SORT_TYPE *keyA,
     SORT_TYPE *keyB,
#ifdef KV_SORT
     VAL_TYPE *valA,
     VAL_TYPE *valB,
#endif
    int dir
){
    SORT_TYPE tk;
#ifdef KV_SORT
    VAL_TYPE tv;
#endif
    if( (MAKEKEY(keyA) > MAKEKEY(keyB)) == dir ){
        tk = *keyA; *keyA = *keyB; *keyB = tk;
#ifdef KV_SORT
        tv = *valA; *valA = *valB; *valB = tv;
#endif
    }
}
__kernel void bitonicSortQndLocal(
    __local SORT_TYPE *s_key,
    __global SORT_TYPE *d_DstKey,
    __global SORT_TYPE *d_SrcKey,
#ifdef KV_SORT
    __local  VAL_TYPE *s_val,
    __global VAL_TYPE *d_DstVal,
    __global VAL_TYPE *d_SrcVal,
#endif
    __read_only uint arrayLength,
    uint dir,
    uint offseted
){

#define global_offset ((offseted!=0)*threadDim)
#define LOC_START_ELM_IDX global_offset+(blockIdx * 2 * threadDim) + threadIdx
//use start of memory if at the end of block, and global offset
    int offsetLast = (blockIdx==(blockDim—1)&&(offseted!=0));
    //offset last is equal within a block, which makes test based on it cheap

    int end_elm_idx = ((offsetLast==0) * (global_offset+(blockIdx * 2 * threadDim) + threadIdx+ (threadDim)))
```

```
                    + (offsetLast * (threadIdx));
#define LOC_END_ELM_IDX end_elm_idx
    s_key[threadIdx +                       0] = d_SrcKey[LOC_START_ELM_IDX];
    s_key[threadIdx + threadDim] = d_SrcKey[LOC_END_ELM_IDX];

#ifdef KV_SORT
    s_val[threadIdx +                       0] = d_SrcVal[LOC_START_ELM_IDX];
    s_val[threadIdx + threadDim] = d_SrcVal[LOC_END_ELM_IDX];
#endif

    for(uint size = 2; size < threadDim*2; size <<= 1){
        //Bitonic merge
        uint ddd = dir ^ ( (threadIdx & (size / 2)) != 0 );
        for(uint stride = size / 2; stride > 0; stride >>= 1){
            barrier(CLK_LOCAL_MEM_FENCE);
            uint pos = 2 * threadIdx - (threadIdx & (stride - 1));
            Comparator(
                &s_key[pos +      0],
                &s_key[pos + stride],
#ifdef KV_SORT
                &s_val[pos +      0],
                &s_val[pos + stride],
#endif
                ddd
            );
        }
    }
    if(offsetLast)
        {
            for(uint stride = threadDim/2; stride > 0; stride >>= 1){
                barrier(CLK_LOCAL_MEM_FENCE);
                uint pos = 2 * threadIdx - ((threadIdx) & (stride - 1));
                if((pos+stride)>=threadDim)
                    pos = 0;
                Comparator(
                    &s_key[pos +      0],
                    &s_key[pos + stride],
    #ifdef KV_SORT
                    &s_val[pos +      0],
                    &s_val[pos + stride],
    #endif
                    (!dir)
                );
#if 1
                Comparator(
                    &s_key[pos + (threadDim) +      0],
                    &s_key[pos + (threadDim) + stride],
    #ifdef KV_SORT
                    &s_val[pos + (threadDim) +      0],
                    &s_val[pos + (threadDim) + stride],
    #endif
                    (!dir)
                    );
#endif
            }
        }
    else
        //ddd == dir for the last bitonic merge step
        {
            for(uint stride = threadDim; stride > 0; stride >>= 1){
                barrier(CLK_LOCAL_MEM_FENCE);
                uint pos = 2 * threadIdx - (threadIdx & (stride - 1));
                Comparator(
                    &s_key[pos +      0],
                    &s_key[pos + stride],
    #ifdef KV_SORT
                    &s_val[pos +      0],
                    &s_val[pos + stride],
    #endif
                    (!dir)
                );
            }

    }
    barrier(CLK_LOCAL_MEM_FENCE);
    d_DstKey[LOC_START_ELM_IDX] = s_key[threadIdx];
    d_DstKey[LOC_END_ELM_IDX] = s_key[threadIdx + (threadDim)];
```

```
#ifdef KV_SORT
    d_DstVal[LOC_START_ELM_IDX] = s_val[threadIdx];
    d_DstVal[LOC_END_ELM_IDX] = s_val[threadIdx + (threadDim)];
#endif
}
//Continous memory access with in bytes; FIXME:can this be read dynamicly and passed to the kernel?
#define MEM_ACCESS_WIDTH_BYTES 128
__kernel void bitonicSortQndGlobal(
    __local SORT_TYPE *s_key,
    __global SORT_TYPE *d_DstKey,
    __global SORT_TYPE *d_SrcKey,
#ifdef KV_SORT
    __local  VAL_TYPE *s_val,
    __global VAL_TYPE *d_DstVal,
    __global VAL_TYPE *d_SrcVal,
#endif
    __read_only uint arrayLength,
    __read_only uint dir,
    __read_only uint sort_buffer_len,
    __local uint *buf_offsets
){
#define LOCAL_WIDTH (threadDim*2)
#define NUM_BLOCKS (arrayLength/LOCAL_WIDTH)
#define USED_BUFFER_LEN (sort_buffer_len)
#define BUF_PER_BLOCK (threadDim/(USED_BUFFER_LEN))
#define BBM_ADJ 1
#define BLOCKBUF_MISMATCH ((NUM_BLOCKS/(BBM_ADJ*BUF_PER_BLOCK)))

#define RIGHTOFFSET(idx) ((((idx)/USED_BUFFER_LEN)%(NUM_BLOCKS/BLOCKBUF_MISMATCH))*(LOCAL_WIDTH))
#define BLOCKS_IN_MISMATCH (NUM_BLOCKS/BLOCKBUF_MISMATCH)
#define SOFF_IDX_FLIP(idx) (idx%BLOCKBUF_MISMATCH)

#define SHARINGOFFSET(idx) (SOFF_IDX_FLIP(idx)*(arrayLength/BLOCKBUF_MISMATCH))
#define BLKOFFSET(idx) (((idx/BLOCKBUF_MISMATCH)%BUF_PER_BLOCK)*USED_BUFFER_LEN)

    __local uint localblock_offset_idx;
    uint bs_tmp;
    if(threadIdx%USED_BUFFER_LEN==(USED_BUFFER_LEN/2)){
        buf_offsets[threadIdx/USED_BUFFER_LEN] = (SHARINGOFFSET(blockIdx)+BLKOFFSET(blockIdx)+RIGHTOFFSET(threadIdx));
    }
#if 1
#if 1
    barrier(CLK_LOCAL_MEM_FENCE);
    if(threadIdx==0){//FIXME: very bad utillisation , use binary search instead
        localblock_offset_idx=0x4000000;
        for(int i=0;i<BUF_PER_BLOCK;i++){
            if(buf_offsets[i]>(blockIdx*LOCAL_WIDTH)+2){
                localblock_offset_idx=i;
                break;
            }
        }
        if(localblock_offset_idx==0x4000000)
            localblock_offset_idx=BUF_PER_BLOCK;
        else if(localblock_offset_idx>0)
            localblock_offset_idx-=1;

    }
#else
    if(threadIdx==0){//FIXME: very bad utillisation , use binary search instead
        localblock_offset_idx=BUF_PER_BLOCK/2;
    }
#endif
    barrier(CLK_LOCAL_MEM_FENCE);
#define LOCALPART_OFFSET    (localblock_offset_idx*USED_BUFFER_LEN)
#define CUR_BUF_OFFSET(pos) (((pos)<LOCALPART_OFFSET)?(buf_offsets[(pos)/USED_BUFFER_LEN]+threadDim+((pos)%USED_BUFFER_LEN)): \
                            (((pos)-LOCALPART_OFFSET<threadDim)? \
                            ((blockIdx*LOCAL_WIDTH)+(pos)-LOCALPART_OFFSET): \
                            (buf_offsets[((pos)-threadDim)/USED_BUFFER_LEN]+threadDim+((pos-threadDim)%USED_BUFFER_LEN))))
#define GLOB_START_ELM_IDX CUR_BUF_OFFSET((threadIdx))
#define GLOB_END_ELM_IDX CUR_BUF_OFFSET((threadIdx+threadDim))
#else
#define CUR_BUF_OFFSET(pos) (buf_offsets[(pos)/USED_BUFFER_LEN])
#define GLOB_START_ELM_IDX blockIdx*LOCAL_WIDTH+threadIdx
#define GLOB_END_ELM_IDX (CUR_BUF_OFFSET(threadIdx))+threadDim+(threadIdx%USED_BUFFER_LEN)
```

```
#endif

#define MAKE_DIR(pos) dir


    s_key[threadIdx +                       0] = d_SrcKey[GLOB_START_ELM_IDX];
    s_key[threadIdx + threadDim]               = d_SrcKey[GLOB_END_ELM_IDX];
#ifdef KV_SORT
    s_val[threadIdx +                       0] = d_SrcVal[GLOB_START_ELM_IDX];
    s_val[threadIdx + threadDim] = d_SrcVal[GLOB_END_ELM_IDX];
#endif

    for(uint size = 2; size < threadDim*2; size <<= 1){
        //Bitonic merge
        uint ddd = dir ^ ( (threadIdx & (size / 2)) != 0 );
        for(uint stride = size / 2; stride > 0; stride >>= 1){
            barrier(CLK_LOCAL_MEM_FENCE);
            uint pos = 2 * threadIdx - (threadIdx & (stride - 1));
            Comparator(
                &s_key[pos +      0],
                &s_key[pos + stride],
#ifdef KV_SORT
                &s_val[pos +      0],
                &s_val[pos + stride],
#endif
                MAKE_DIR(pos)
            );
        }
    }
    //ddd == dir for the last bitonic merge step
    {
        for(uint stride = threadDim; stride > 0; stride >>= 1){
            barrier(CLK_LOCAL_MEM_FENCE);
            uint pos = 2 * threadIdx - (threadIdx & (stride - 1));
            Comparator(
                &s_key[pos +      0],
                &s_key[pos + stride],
#ifdef KV_SORT
                &s_val[pos +      0],
                &s_val[pos + stride],
#endif
                !MAKE_DIR(pos)
            );
        }
    }
    barrier(CLK_LOCAL_MEM_FENCE);
#if 1
    d_DstKey[GLOB_START_ELM_IDX] = s_key[threadIdx];
    d_DstKey[GLOB_END_ELM_IDX] = s_key[threadIdx + (threadDim)];
#else
    d_DstKey[blockIdx*2*threadDim+threadIdx] = GLOB_START_ELM_IDX;// ((GLOB_END_ELM_IDX)%1000)*8;//↩
         GLOB_START_ELM_IDX;
    d_DstKey[blockIdx*2*threadDim+threadDim+threadIdx] = GLOB_END_ELM_IDX;
#endif
#ifdef KV_SORT
    d_DstVal[GLOB_START_ELM_IDX] = s_val[threadIdx];
    d_DstVal[GLOB_END_ELM_IDX] = s_val[threadIdx + (threadDim)];
#endif
}
```

A.2.2 BitonicSortHostInterface.cpp

```
CLSorting::CLSorting(GPGPU* device){
    // ...
    bsSharedQndLoc = clCreateKernel(program, "bitonicSortQndLocal", &err);
    bsSharedQndGlob = clCreateKernel(program, "bitonicSortQndGlobal", &err);
}

uint CLSorting::factorRadix2(uint *log2L, uint L){
    if(!L){
        *log2L = 0;
        return 0;
    }else{
        for(*log2L = 0; (L & 1) == 0; L >>= 1, *log2L++);
        return L;
    }
}
uint CLSorting::bitonicSortAdv(
    cl_mem d_DstKey,
    cl_mem d_SrcKey,
#ifdef KV_SORT
    cl_mem d_DstVal,
    cl_mem d_SrcVal,
#endif
    uint batchSize,
    uint arrayLength,
    uint dir,
    int doSHMSort,
    int sort_num,
    int sort_den
){
    int err;
    if(arrayLength < 2) //Nothing to sort
       return 0;
    cl_ulong shmSize=512;//min of: opencl work group shm size and work items in work group
    int ai;
    //Only power-of-two array lengths are supported by this implementation
    uint log2L;
    uint factorizationRemainder = factorRadix2(&log2L, arrayLength);
    if( factorizationRemainder != 1){
        printf("Warning, sort not power of 2");
        return 0;
    }
    dir = (dir != 0);
    size_t  blockCount = batchSize*arrayLength / shmSize;
    size_t threadCount = shmSize/2;
    blockCount *= threadCount; // cuda dimensions -> opencl dimensions
    if(false && arrayLength <= shmSize){
        if( (batchSize * arrayLength) % shmSize != 0 ){
            printf("Array length not divisable by shm size");
            return 0;
        }
        ai = 0;
        clSetKernelArg(bsShared, ai++, sizeof(cl_uint), &shmSize);//SHM
        clSetKernelArg(bsShared, ai++, shmSize*sizeof(SORT_TYPE), NULL);//SHM
        clSetKernelArg(bsShared, ai++, sizeof(cl_mem), &d_DstKey);
        clSetKernelArg(bsShared, ai++, sizeof(cl_mem), &d_SrcKey);
#ifdef KV_SORT
        clSetKernelArg(bsShared, ai++, shmSize*sizeof(VAL_TYPE), NULL);//SHM
        clSetKernelArg(bsShared, ai++, sizeof(cl_mem), &d_DstVal);
        clSetKernelArg(bsShared, ai++, sizeof(cl_mem), &d_SrcVal);
#endif
        clSetKernelArg(bsShared, ai++, sizeof(cl_uint), &arrayLength);
        clSetKernelArg(bsShared, ai++, sizeof(cl_uint), &dir);
        size_t mThread = shmSize/2;
        size_t mBlock= arrayLength*mThread;
        err = clEnqueueNDRangeKernel(device->queue, bsShared, 1, NULL, &blockCount, &threadCount, 0, NULL, NULL);
        CL_CHECK_ERROR(err, "Error while calling bss kernel");
        err = clFinish(device->queue);
        CL_CHECK_ERROR(err, "Error while running bss kernel");
    }else{
        ai = 0;
        clSetKernelArg(bsShared1, ai++, sizeof(cl_ulong), &shmSize);//SHM
        clSetKernelArg(bsShared1, ai++, sizeof(SORT_TYPE)*shmSize, NULL);//SHM
        clSetKernelArg(bsShared1, ai++, sizeof(cl_mem), &d_DstKey);
        clSetKernelArg(bsShared1, ai++, sizeof(cl_mem), &d_SrcKey);
#ifdef KV_SORT
        clSetKernelArg(bsShared1, ai++, shmSize*sizeof(VAL_TYPE), NULL);//SHM
        clSetKernelArg(bsShared1, ai++, sizeof(cl_mem), &d_DstVal);
        clSetKernelArg(bsShared1, ai++, sizeof(cl_mem), &d_SrcVal);
#endif
        if(doSHMSort)
            err = clEnqueueNDRangeKernel(device->queue, bsShared1, 1, NULL, &blockCount, &threadCount, 0, NULL, ↩
                 NULL);
        CL_CHECK_ERROR(err, "Error while calling bss1 kernel");
        err = clFinish(device->queue);
        for(uint size = 2 * shmSize; size <= arrayLength; size <<= 1){
            for(unsigned stride = (size * sort_num)/sort_den; stride > 0; stride >>= 1)
                if(stride >= shmSize){
                    ai = 0;
                    clSetKernelArg(bsMergeGlob, ai++, sizeof(cl_mem), &d_DstKey);
                    clSetKernelArg(bsMergeGlob, ai++, sizeof(cl_mem), &d_DstKey);
#ifdef KV_SORT
                    clSetKernelArg(bsMergeGlob, ai++, sizeof(cl_mem), &d_DstVal);
                    clSetKernelArg(bsMergeGlob, ai++, sizeof(cl_mem), &d_DstVal);
#endif
                    clSetKernelArg(bsMergeGlob, ai++, sizeof(cl_uint), &arrayLength);
                    clSetKernelArg(bsMergeGlob, ai++, sizeof(cl_uint), &size);
                    clSetKernelArg(bsMergeGlob, ai++, sizeof(cl_uint), &stride);
                    clSetKernelArg(bsMergeGlob, ai++, sizeof(cl_uint), &dir);
                    size_t mBlock= (batchSize*arrayLength)/(shmSize/2) ;
                    size_t mThread = (shmSize/4);
                    mBlock*=mThread;

                    err = clEnqueueNDRangeKernel(device->queue, bsMergeGlob, 1, NULL, &mBlock, &mThread, 0, NULL, NULL);
                    CL_CHECK_ERROR(err, "Error while calling bmg kernel");
                    err = clFinish(device->queue);
                }else{
                    ai = 0;
                    clSetKernelArg(bsMergeShared, ai++, sizeof(cl_ulong), &shmSize);//SHM
                    clSetKernelArg(bsMergeShared, ai++, sizeof(SORT_TYPE)*shmSize, NULL);//SHM
                    clSetKernelArg(bsMergeShared, ai++, sizeof(cl_mem), &d_DstKey);
                    clSetKernelArg(bsMergeShared, ai++, sizeof(cl_mem), &d_DstKey);
#ifdef KV_SORT
                    clSetKernelArg(bsMergeShared, ai++, sizeof(VAL_TYPE)*shmSize, NULL);//SHM
                    clSetKernelArg(bsMergeShared, ai++, sizeof(cl_mem), &d_DstVal);
                    clSetKernelArg(bsMergeShared, ai++, sizeof(cl_mem), &d_DstVal);
#endif
                    clSetKernelArg(bsMergeShared, ai++, sizeof(cl_uint), &arrayLength);
                    clSetKernelArg(bsMergeShared, ai++, sizeof(cl_uint), &size);
                    clSetKernelArg(bsMergeShared, ai++, sizeof(cl_uint), &dir);
                    err = clEnqueueNDRangeKernel(device->queue, bsMergeShared, 1, NULL, &blockCount, &threadCount, 0, NULL, NULL);
                    CL_CHECK_ERROR(err, "Error while calling bms kernel");
                    break;
                }
        }
    }
    return threadCount;
}
```

A.3 Drawtmp.py - for visualizing the sorted data

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sys
from functools import partial
from itertools import permutations
from PySide import QtCore, QtGui
from PySide.QtCore import QPointF, QRectF, QLineF
from PySide.QtGui import QTransform, QColor, QVector2D
import json
import atexit

def main():
    iter_height = 10
    app = QtCore.QCoreApplication(sys.argv)
    iterations = None
    print "Reading and parsing"
    with open("qnddata.txt", "rb") as data:
        iterations = [[float(element.strip()) for element in iteration.strip(" ").split(" ")] for iteration in data.read()[0:-1].split("\n")]
    print "Parsed"
    numiter = len(iterations)
#    numiter = 3
    iterstep = 1
    img = QtGui.QImage( len(iterations[0]), (numiter/iterstep)*iter_height, QtGui.QImage.Format_RGB32 )
    painter = QtGui.QPainter(img)
    color = QtGui.QColor()
    MXP=11
    for i in xrange(0,numiter,iterstep):
        iteration = iterations[i]
        print "Painting", i
        for j,element in enumerate(iteration):
            if 0.0<element and element<1.0:
                color.setHsvF(0.0,0.0,element);#greyscale
#                color.setHsvF(element,1.0,1.0);#color
            elif element<0:
                color.setHsvF(0.1,0.5,0.5)
            elif element>2**MXP:
                color.setHsvF(0.5,0.1,0.9)
            else:
                 color.setHsvF(element/(2**MXP),1.0,element/(2.0**MXP))
            painter.fillRect(j,i*iter_height,1,iter_height-1,color)
    color.setHsvF(0.5,0.9,0.9,0.5)
    painter.fillRect((len(iterations[0])/2)-2, 0, 4, (numiter/iterstep)*iter_height,color)
    color.setHsvF(0.8,0.9,0.9,0.5)
    painter.fillRect((len(iterations[0])/4)-2, 0, 4, (numiter/iterstep)*iter_height,color)
    color.setHsvF(0.2,0.9,0.9,0.5)
    painter.fillRect((3*len(iterations[0])/4)-2, 0, 4, (numiter/iterstep)*iter_height,color)
    painter.end()
    print "Painted, now saving"
    img.save("qndsortviz.png");
    print "Saved"



if __name__ == "__main__":
    main()
```

A.4 Testing

A.4.1 Script for generating test graph data

```
#!/usr/bin/env python
from jinja2 import Template
from jinja2 import Environment, FileSystemLoader
from copy import copy
import shutil, os, subprocess, time, fcntl, signal, re, pickle
import datetime
codedir = "/Users/freqmod/pgmz/snowgit/"
builddir = codedir+"bld/"
env = Environment(loader=FileSystemLoader(os.path.dirname(os.path.realpath(__file__))))
renderTmp = env.get_template("renderopts.h")
configTmp = env.get_template("config.tmpl")
cfg = {'stereo':0,
       'LBM': 'false',
       'road': "../release/mntroad.rd",
       'numpart': 65536,
       'renderSnow': '0',
       'simSnow': '0',
       'simWind': '1',
       'windX': 62,
       'windY': 14,
       'windZ': 62}
opts = {'drawWind': False,
        'drawObs': False,
        'drawPress': False,
        'drawTerr': False,
        'drawRoad': False}

def dotest(opts, cfg):
    global renderTmp, configTmp;
    renderoptsCnt =  renderTmp.render(opts)
    configCnt =  configTmp.render(cfg)
    #generate renderopts.h and recompile the simulator
    shutil.copy(builddir+"snow", "./snow")
    snowout = open("/tmp/comm.txt","wb")
    snow = subprocess.Popen(["./snow"], stdout=snowout, stderr=subprocess.STDOUT)
    print "Call"
    time.sleep(10)
    snow.send_signal(signal.SIGINT)
    time.sleep(3)
    snowout.close()
    with open("/tmp/comm.txt",'rb') as snowout:
        snowoutput  = snowout.read()
    avgfps = re.search(r'run was: ([0-9\.]+)',snowoutput).group(1)
    with open("results.txt",'rb') as writtenresfle:
        writtenres = writtenresfle.read()
    print "AVGFPS:", avgfps
    print "Terminated"
    snow.terminate()
    return (avgfps, writtenres)
winds = (16,18,20,22,24,32,48,56)#,64,92,128) #(2**10,2**12,2**14,2**16,2**18,2**20)
parts = (2**11,2**12,2**13,2**14,2**15,2**16,2**17,2**18)
results = []
#for part, wind in zip(parts,winds):
#for  part in parts:
for  wind in winds:
#    cfg['numpart'] = part
    cfg['windX'] = cfg['windZ'] = wind
    cfg['windY'] = cfg['windZ']/2
    print opts,cfg
    result = dotest(opts, cfg)
    results.append((copy(opts),copy(cfg),result))
print results
with open("results %s.txt" % (datetime.datetime.now()),'wb') as fh:
    pickle.dump(results, fh)
```