Porting the HPC-Lab Snow Simulator to OpenCL

Imre Kerr

December 2014

SPECIALIZATION PROJECT

Department of Computer and Information Science

Norwegian University of Science and Technology

Supervisor: Dr. Anne C. Elster

Problem Description

OpenCL is a recently defined and widespread open standard that will make the massive parallelism power now available in GPUs, but also newer CPUs (like the Cell B.E.) , more easily accessible to programmers.

This project involves porting an existing OpenCL snow simulation application to Open CL and comparing the implementation and benchmarks to those implemented in CUDA.

Trondheim, 2015-01-04

Supervisor: Dr. Anne C. Elster

Acknowledgment

I would like to thank Dr. Anne C. Elster for heading the HPC-Lab and letting me work there. The lab provides a unique opportunity for students to work together and learn from each other, all in a great work environment.

I would also like to thank Jørgen Kvalsvik for providing valuable expertise on build systems, in particular CMake.

I.K.

Summary and Conclusions

Here you give a summary of your your work and your results. This is like a management summary and should be written in a clear and easy language, without many difficult terms and without abbreviations. Everything you present here must be treated in more detail in the main report. You should not give any references to the report in the summary – just explain what you have done and what you have found out. The Summary and Conclusions should be no more than two pages.

You may assume that you have got three minutes to present to the Rector of NTNU what you have done and what you have found out as part of your thesis. (He is an intelligent person, but does not know much about your field of expertise.)

Contents

Chapter 1

Introduction

1.1 Background

1.1.1 Graphics Hardware

Graphics processing units (GPUs) are special-purpose processors designed with graphics in mind. Computer graphics require large amounts of calculations, typically the same calculations done over and over. This type of workload is very different from the mostly serial workloads typically done by CPUs. Hence, large performance benefits can be gotten by using specialized hardware with a large number of execution units in a single-instruction/multiple-data (SIMD) setup.

1.1.2 General-Purpose GPU Computing

As it happens, there is another field where SIMD hardware is widely used: scientific computing. A typical physics simulation (e.g. a weather report) can involve solving systems of equations with thousands of unknowns. Historically, vector processors were used for this purpose. However, these processors were little used outside of scientific computing, leading to high cost due to the nature of semiconductor manufacturing. Designing a processor, making photolithography masks and buying fabrication equipment are all extremely expensive. Once these are done, though, processors can be made at next to no cost, leading to economies of scale.

GPUs, on the other hand, are sold in vast quantities to the consumer market, due in no small part to computer games. If one could use graphics hardware to compute other things than graphics, one could potentially do scientific computing at much lower cost.

Experiments in general-purpose GPU (GPGPU for short) computing began with Larsen and McAllister (2001), which presents a routine for matrix multiplication using graphics hardware. This was enabled by graphics hardware that supported programmable shaders and floating-point arithmetic, which had previously been lacking. Galoppo et al. (2005) represented one of the first implementations of a common scientific programming primitive which ran faster than the CPU implementation.

However, GPGPU still had one fatal flaw. The only APIs available for interfacing with GPUs were graphics APIs, in particular OpenGL and DirectX. Programming for GPUs therefore meant casting everything in terms of graphics primitives, which was tedious, error-prone and inaccessible.

1.1.3 CUDA

Released in 2006, Nvidia's CUDA sought to make GPU programming easier by letting programmers write programs as they were used to, namely in terms of the task at hand rather than graphics primitives. CUDA is currently widely used, not only in scientific computing, but also for image and video processing, cryptography, physics simulation in games and more.

The CUDA programming model splits code into host-side code, which runs on the CPU, and device-side code (also called kernels) which runs on the GPU. When calling a kernel, the number of threads to be created is included as an argument. Copying data between the host and device, as well as allocating memory on the device, is done explicitly. This allows the programmer to take advantage of the memory hierarchy on the device, placing often-used data in fast memory.

A much more thorough treatment of the CUDA programming model is given in the CUDA C Programming guide (Nvidia, 2014).

1.1.4 OpenCL

While CUDA has enjoyed great popularity, it suffers from vendor lock-in due to the fact that it only supports Nvidia GPUs. In an effort to create an open standard for GPU computing (or more precisely, heterogeneous computing), OpenCL was created. Originally developed by Apple, it is currently maintained by the Khronos Group.

While CUDA and OpenCL have many similarities, there are a few key differences to note:

- CUDA is a *GPU programming* API, while OpenCL is a *heterogeneous programming* API. OpenCL implementations exist not only for GPUs, but also CPUs, APUs, and more exotic hardware such as Adapteva's Epiphany processor.

- To maintain compatibility across platforms, OpenCL device-side code must be compiled at runtime. This incurs a penalty in startup time compared to CUDA.

Finally, although OpenCL code is portable, code which has good performance on one platform is not necessarily performant on others. This issue of code portability versus performance portability, as well as the relative performance of OpenCL versus CUDA in general, is treated by Weber et al. (2011).

1.2 Goals

The main goal for this project can be stated simply as "Port the HPC-Lab Snow Simulator to OpenCL". However, there are a few secondary goals that should be considered as well.

Firstly, this is not the first time the snow simulator has been ported to OpenCL. The previous port was obsolete pretty much out of the gate, which is why the most current version has always been CUDA only. I can see two possible reasons for this. One is simply that working with two APIs is more work than working with one, and people are likely to just take the path of least resistance. To mitigate this, an OpenCL port should have the API specific code isolated from the rest of the code, separated by a clean abstraction layer. This will ensure that making changes to the host-side code will be just as easy in the cross-platform version as in the CUDA only version. One should also take steps to make the OpenCL and CUDA code both as readable and as similar to each other as possible. This will minimize the work required to make changes to the device-side code, and hopefully encourage people to keep both versions of this code up to date.

Additionally, other students are also doing projects involving the snow simulator this semester. If their work and mine are not kept in sync with frequent merging, we might end up with separate versions, and future projects will need to choose between these when choosing what code base to work on.

Finally, the snow simulator is a very computationally intensive program. Producing correct code should of course be priority number one, but one should not neglect performance. Therefore I will analyze the performance of the simulator using whatever tools I can, and explore options for using OpenCL-specific tricks to improve this performance.

1.3 Motivation

Rather than being a purely academic exercise, there are a few reasons why an OpenCL port of the Snow Simulator was desired. Of course, supporting more platforms is “better”, but we had a few things in mind apart from this.

First of all, it paves the way for exploring GPGPU on mobile platforms. Even though Nvidia’s Tegra chips got CUDA support with the Tegra K1, only a few devices with this processor are currently available. An OpenCL version will enable us to use mobile graphics hardware such as e.g. ARM’s Mali GPU.

Also, when this project was started, there were plans for the HPC Lab to build a display wall. Such a display is composed of several smaller screens, allowing for large screen size and resolution at a moderate price. We found that the best choice for a large multi-monitor setup like this would be AMD’s Eyefinity. Running the Snow Simulator on AMD hardware would require an OpenCL port. This project may still happen at some point, but it is less of a priority since the Lab recently purchased an 85-inch 4k monitor, which fills the same role.

1.4 Structure of the Report

The rest of the report is organized as follows. Chapter 2 briefly describes the previous work on the Snow Simulator. Chapter 3 describes the actual porting work (briefly, as this is quite boring and mechanical), before going into the testing that was done, as well as the results of these tests. Chapter 3 discusses the results, suggests future work based on these.

Chapter 2

Previous work

What follows is a brief treatment of previous work that has been done on the Snow Simulator. Only the aspects that are relevant to my project are included. For a fuller treatment, I refer to any of a number of recent masters theses about the subject, e.g. Boge (2014).

History

The Snow Simulator started out as a Master's project by Saltvik (2006). It was CPU only, and could simulate 40 000 snow particles in real time. Later, it was ported to CUDA by Eidissen (2009), which (along with hardware advances, of course) brought the particle count up to 2 000 000.

After this, the simulator has been the subject of several master's theses, which have all improved or expanded on it in some way. Examples include visualization improvements (Nordahl, 2013), avalanche prediction (Boge, 2014), and porting to OpenACC (Mikalsen, 2013) and OpenCL (Vestre, 2012).

Architecture

Initially, the user is greeted with a screen that allows setting various options such as number of particles, wind field size, terrain map file, whether to simulate wind and so forth. When these settings have been entered, the simulation can be started by the user.

The simulator starts by initializing the data structures needed for the simulation. A heightmap is generated based on the terrain map file, the wind field is initialized (no wind initially), and randomized positions for snow particles are generated. After initialization, the simulator enters the main loop, which runs through the various simulation steps on the GPU, before rendering everything to the screen. The main simulation steps are:

1. Update obstacle field
2. Simulate wind
3. Move snow particles
4. Update snow layers

The GPU specific code is kept quite separate from the rest of the simulator. The reason for this is that most of the simulator is written in C++, but CUDA programs must be compiled with nvcc, which only accepts C. Therefore, the CUDA code is compiled into a library, libParticle.a, which is then statically linked to the rest of the simulator. All communication with this library is done using a relatively small set of public methods.

Internally, the particle system library is divided into three subsystems: Snow, Wind and Terrain. Data passes quite freely between these three, mostly using global state.

Existing OpenCL port

As mentioned, there already exists an OpenCL port of the Snow Simulator, written by Vestre (2012). However, this port has become out of date due to not being updated along with the CUDA version. Hence, this project is an attempt to bring it up to date. When planning the project, two approaches were considered:

- Base my work on the previous port, merging in all the improvements that have happened over the past two years.
- Base my work on the most recent version. This effectively means redoing the porting work, but will entail less work due to the possibility of copying code in areas where the simulator is unchanged.

I decided on the second approach, not only because it meant less work, but also because the work would be localized to only one part of the simulator, rather than spread out across the whole codebase.

Chapter 3

Methodology and Testing

3.1 Methodology

3.1.1 Build System

To make the build process smoother on machines with different hardware, an automated build system was needed. A rudimentary CMake script was already present, but this needed some extension. The key problem was identifying which (if any) of CUDA and OpenCL were present, and building the appropriate version(s). The resulting builds two binaries on systems with both OpenCL and CUDA, and will copy all necessary kernel, shader and data files for out-of-tree builds. Installation functionality was not implemented.

Extending the build system to other compute APIs (e.g. PETSc, currently being done by Martin Stølen), should be a simple task. One can simply find the relevant library, and add it to the list of versions.

3.1.2 Library Porting

When starting the project, I set a goal of not changing any of the platform-independent code unless I absolutely had to. Since the particle system had a clearly defined interface and reasonably loose coupling to the rest of the simulator, this seemed like an achievable goal. Thus, the porting work roughly followed these steps:

1. Write header files specifying the interface to the particle system.

2. Write method stubs for all the public methods of the library. At this point the simulator compiled and ran, but of course didn't do anything apart from showing snowflakes hanging motionless in the air.

3. Write the OpenCL boilerplate of finding platforms and devices, and creating a context and queue. Doing this correctly and portably is nontrivial, as shown by e.g. Fastkor (2012).

4. Implement the minimum amout of functionality to see that OpenCL was actually working. This was done by writing a kernel that moved snow particles downward at a constant rate. A lot of work was required to get to this point, and the process was a whole lot smoother after this.

5. Port the rest of the snow and wind systems. These systems were relatively unchanged since the 2012 port, so I was able to copy most of the code for these.

6. Port the terrain system. At this point, I was familiar with the porting process and with OpenCL, which saved me a good amount of stumbling. Also, the CUDA code was written in a single semester by a single person, so it was very clean and well-structured compared to other parts of the simulator.

3.2 Testing

3.2.1 Test Setup

Testing was done on my PC at home, which has an Nvidia GPU. This allowed me to compare the CUDA and OpenCL versions when testing. The full specifications of this machine are given below:

- CPU: Intel Core 2 Quad Q9550
- Graphics card: MSI Gaming GTX 970
- Motherboard: Asus P5Q Pro
- RAM: 4GB

- OS: Ubuntu 14.04

3.2.2 Test Methodology

Functional tests were done by visual inspection, in particular comparing results to those from the CUDA version. This is not a perfect method, for two reasons.

Firstly, visual inspection may not be sufficient to detect small differences, and is only good for uncovering "obvious" errors. More thorough testing would require a deeper knowledge of the mathematics and physics involved in the simulation, as well as writing code for automated testing. Neither of these is something I would have had time to do.

Secondly, it makes the implicit assumption that the CUDA simulator is correct. While improving or uncovering errors in the CUDA simulator would be nice and useful, it is outside the scope of this project.

For performance tests, a bit more tooling was needed. Simple framerates are already measured by the simulator. In order to measure relevant performance (computation rather than rendering) and decrease variance, I sampled the framerate when looking out at the skybox, with no snow or terrain visible. However, I also wanted to use an OpenCL profiler to see exactly how long each function took, to better highlight possibilities for optimization. Three profilers were considered:

1. Nvidia visual profiler (nvvp) - This tool could apparently profile OpenCL applications at one point (at least according to old forum posts), but the support was never great, and now appears to be nonexistent.

2. AMD CodeXL - Apparently an excellent profiler, allowing for in-depth analysis of kernel execution including GPU debugging, various profiling metrics such as GPU counter data, application trace, kernel occupancy and hotspots analysis, and static kernel analysis. However, this would require AMD hardware for testing, which I did not have available. (I was out of town, so using the HPC-Lab's machines was sadly not an option.)

3. Light Temporal Performance Viewer (LTPV) - This is an open source tool, mainly written by Simon Denel in 2013 during an internship at Thales. It is able to collect data while run-

ning the simulator, but only about kernel calls, not the actual kernel execution. Profiling kernel execution would require platform-specific code, so this is understandable.

Performance tests were done by varying the number of snow particles and the windfield size, and measuring framerates and profiling data using LTPV.

3.2.3 Functional tests

1. **Snow, terrain and skybox renders properly**

   Result: PASS

2. **Snow moves towards the ground when there is no wind**

   Result: PARTIAL — Seems to run faster than the CUDA version, despite the framerate being similar.

3. **New snow spawns at top**

   Result: PASS

4. **Snow piles up**

   Result: PASS

5. **No snow below ground level**

   Result: PASS

6. **When ground smoothing enabled, ground is smoothed**

   Result: FAIL — See fig. 3.1.

7. **Snow gets blown around by wind**

   Result: PASS

8. **Changing wind direction/speed causes a corresponding change in snow movement**

   Result: PASS

9. **Wind debug displays render correctly**

   Result: PASS

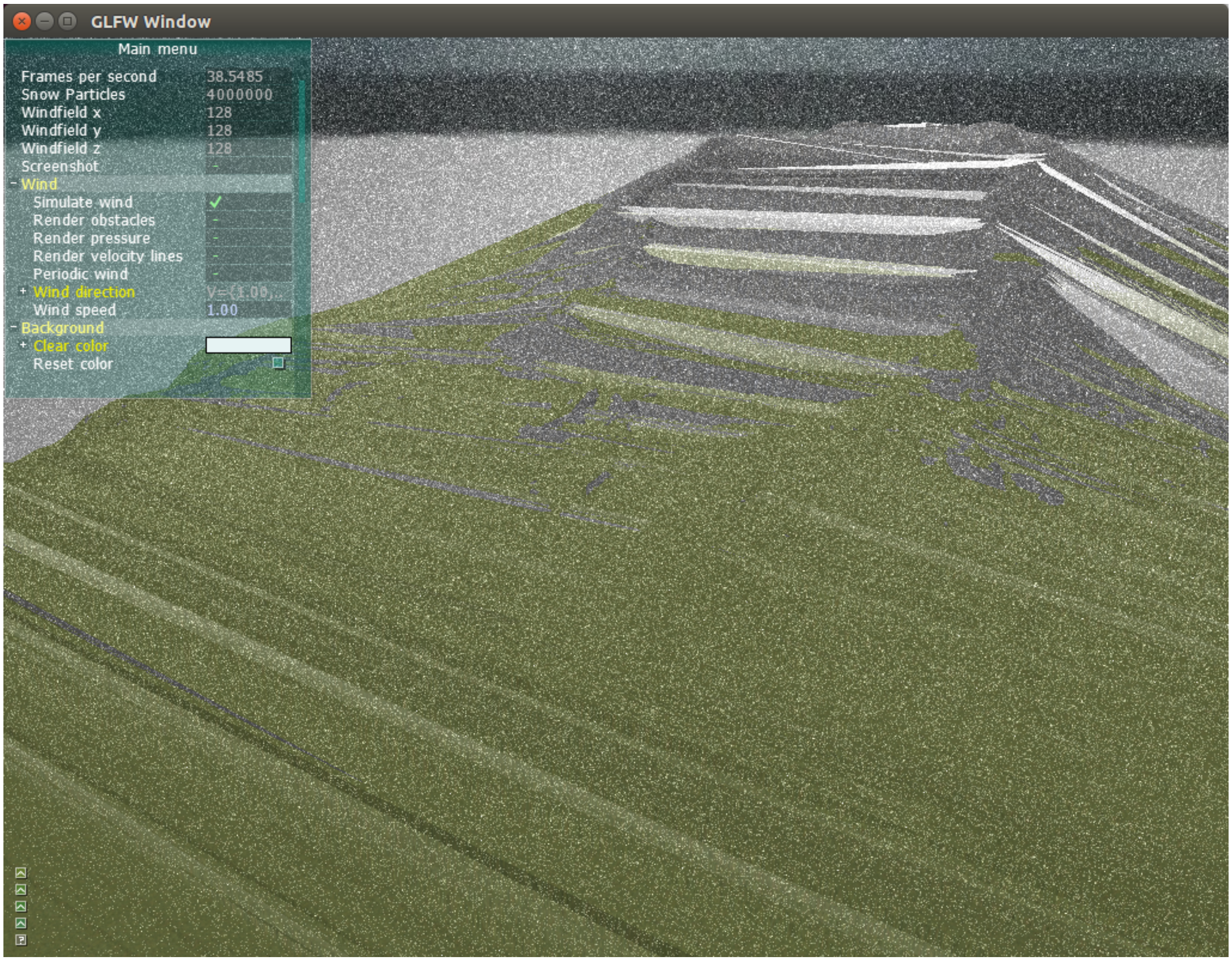

Figure 3.1: Simulator with smoothing function enabled, showing erroneous results.

10. **Wind debug displays look similar to CUDA version**

    Result: PASS

11. **Avalanche prediction looks similar to CUDA version**

    Result: PASS

3.2.4 Performance tests

Table 3.1 contains the framerates of the CUDA and OpenCL versions for various windfield sizes and particle counts. Profiling data is in the form of a bunch of screen shots, which can be seen in appendix B.

Windfield \ Snow particles	1 Million	4 Million	16 Million
32x32x32	OpenCL: 298 CUDA: 336	OpenCL: 113 CUDA: 126	OpenCL: 34.0 CUDA: 45.5
128x128x128	OpenCL: 99 CUDA: 93	OpenCL: 51.1 CUDA: 52.8	OpenCL: 18.1 CUDA: 19.7
256x256x256	OpenCL: 22 CUDA: 18	OpenCL: 17.7 CUDA: 16.3	OpenCL: 10.7 CUDA: 10.4

Table 3.1: Framerates for various problem sizes

3.3 Discussion

3.3.1 Functional tests

The only serious issue uncovered by functional testing was the erroneous results with ground smoothing enabled. The smoothing function works by comparing the height of a grid cell with the neighbors, and increasing or decreasing the height if the difference is, respectively, below or above a certain threshold. I can see two possible explanations for the observed behavior:

- The smoothing function is comparing the height not to that of the cell's neighbors, but to some more distant cell. This would indicate some sort of indexing error.
- Race conditions. The change in height needs to happen after all threads have read the current height.

3.3.2 Performance tests

Chapter 4

Summary and Recommendations for Further Work

In this final chapter you should sum up what you have done and which results you have got. You should also discuss your findings, and give recommendations for further work.

4.1 Summary and Conclusions

Here, you present a brief summary of your work and list the main results you have got. You should give comments to each of the objectives in Chapter 1 and state whether or not you have met the objective. If you have not met the objective, you should explain why (e.g., data not available, too difficult).

This section is similar to the Summary and Conclusions in the beginning of your report, but more detailed—referring to the the various sections in the report.

4.2 Discussion

Here, you may discuss your findings, their strengths and limitations.

4.3 Recommendations for Further Work

You should give recommendations to possible extensions to your work. The recommendations should be as specific as possible, preferably with an objective and an indication of a possible approach.

The recommendations may be classified as:

- Short-term
- Medium-term
- Long-term

Appendix A

Acronyms

FTA Fault tree analysis

MTTF Mean time to failure

RAMS Reliability, availability, maintainability, and safety

Appendix B

Profiling Data

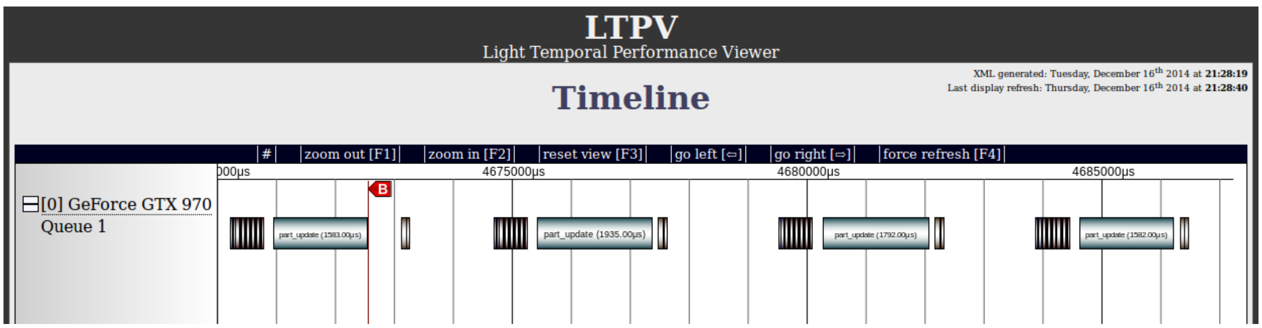

Figure B.1: Profiling data for 1 million particles, 32x32x32 windfield

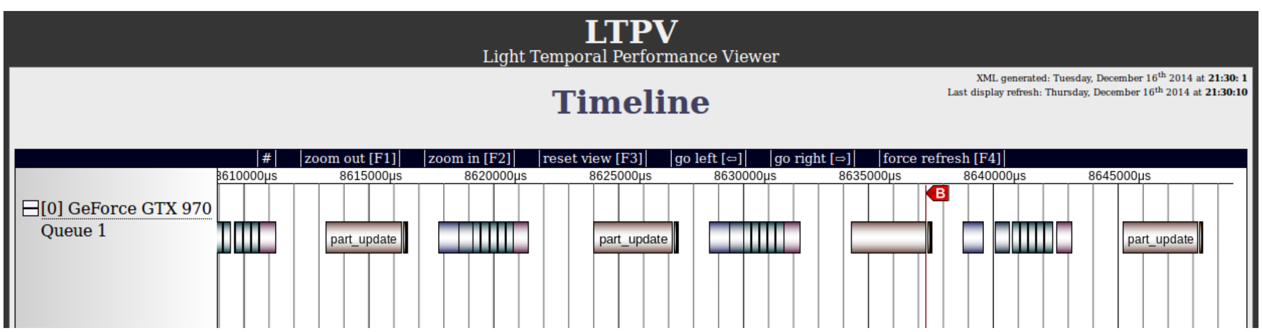

Figure B.2: Profiling data for 1 million particles, 32x32x32 windfield

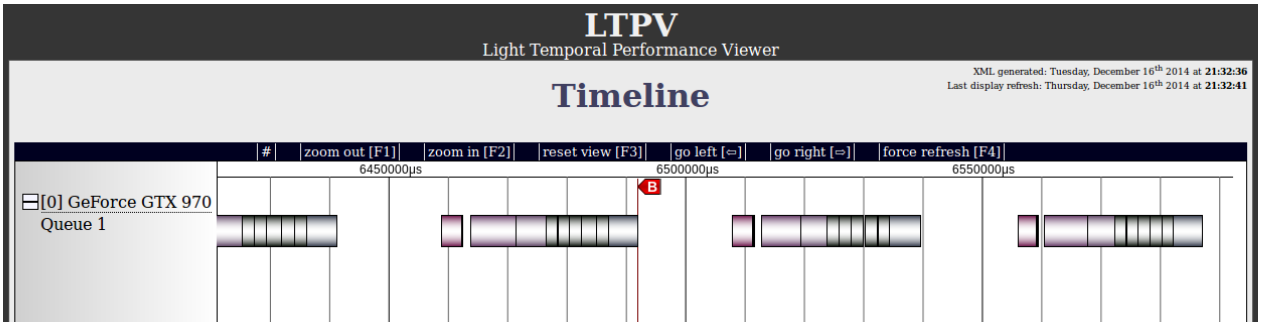

Figure B.3: Profiling data for 1 million particles, 32x32x32 windfield

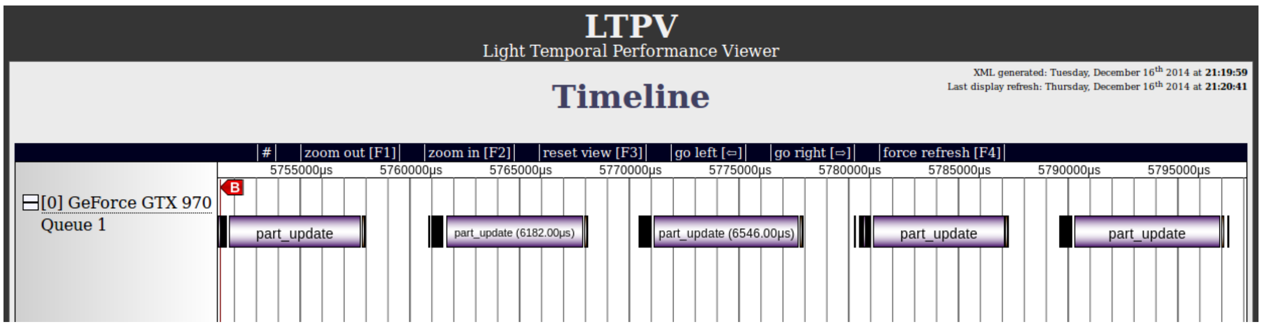

Figure B.4: Profiling data for 4 million particles, 32x32x32 windfield

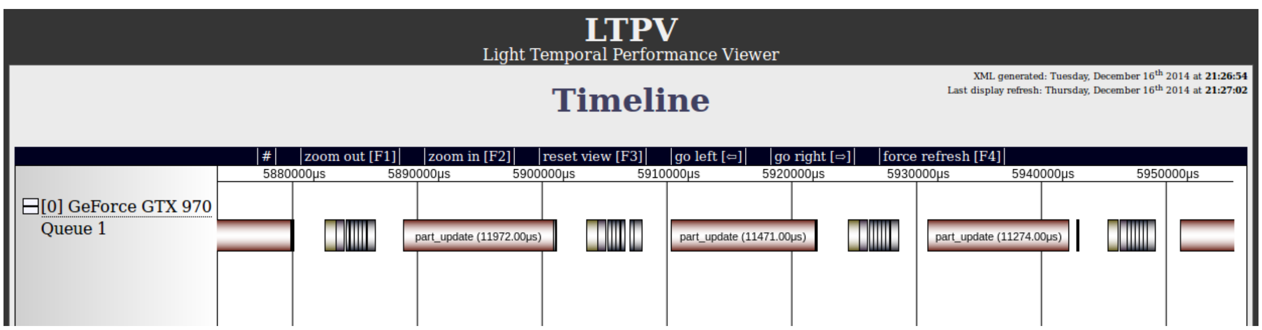

Figure B.5: Profiling data for 4 million particles, 128x128x128 windfield

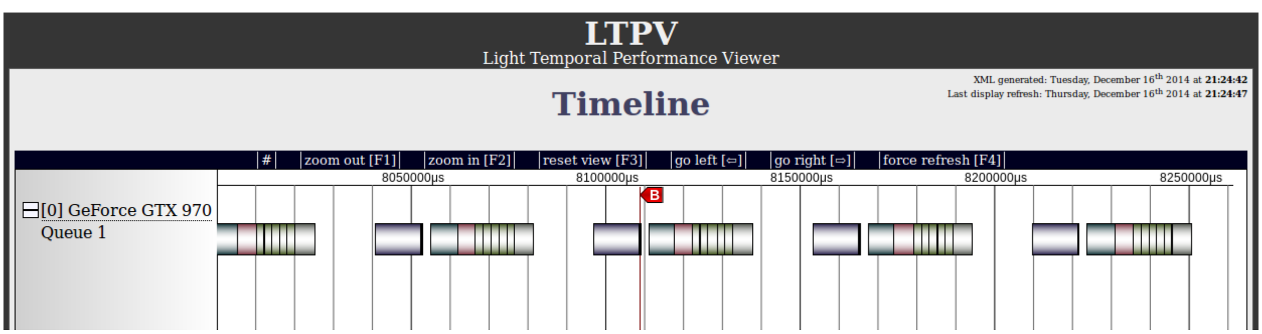

Figure B.6: Profiling data for 4 million particles, 256x256x256 windfield

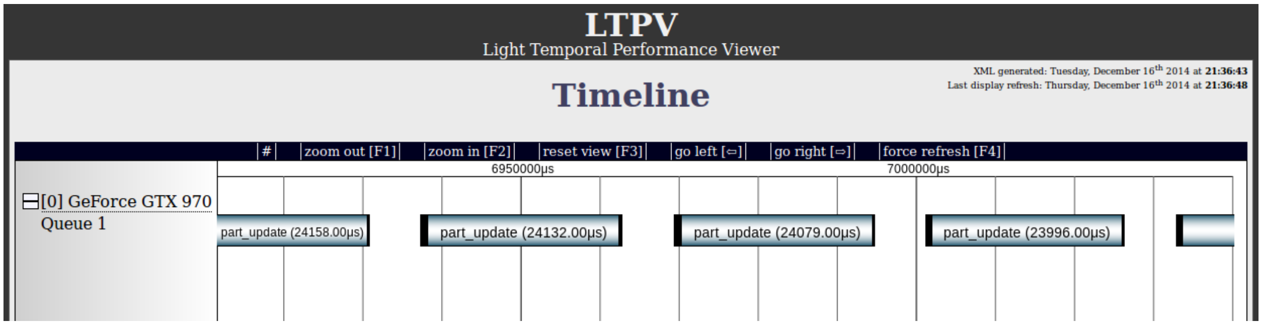

Figure B.7: Profiling data for 16 million particles, 32x32x32 windfield

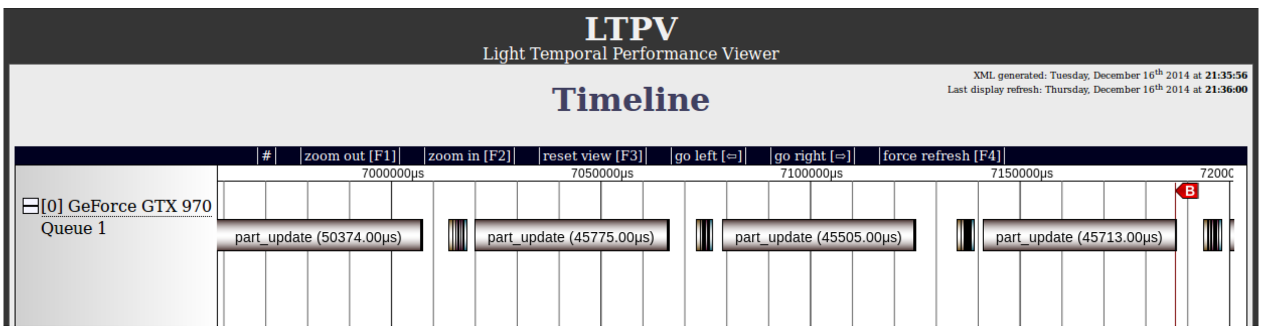

Figure B.8: Profiling data for 16 million particles, 128x128x128 windfield

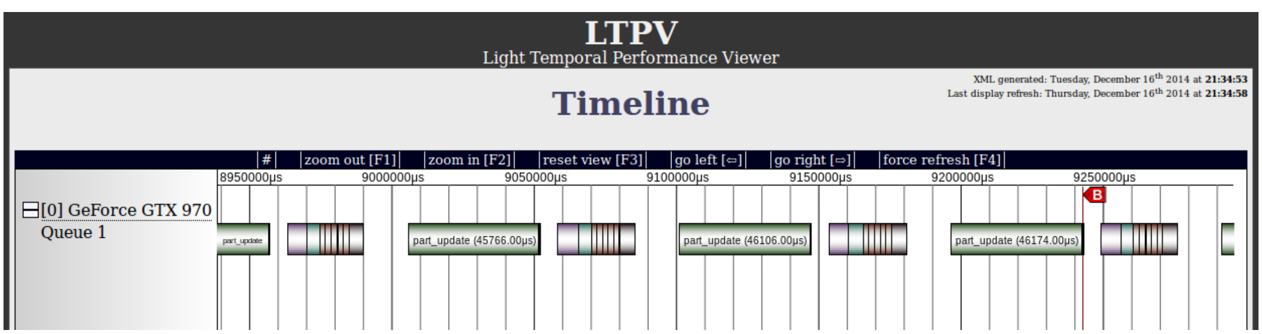

Figure B.9: Profiling data for 16 million particles, 256x256x256 windfield

Bibliography

Boge, O. L. (2014). Avalanche simulations using fracture mechanics on the gpu. Master's thesis, Norwegian University of Science and Technology.

Eidissen, R. (2009). Utilizing gpus for real-time visualization of snow. Master's thesis, Norwegian University of Science and Technology.

Fastkor (2012). Example: Opencl boilerplate. *https://fastkor.wordpress.com/2012/07/22/example-opencl-boilerplate/*.

Galoppo, N., Govindaraju, N. K., Henson, M., and Manocha, D. (2005). Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 3. IEEE Computer Society.

Larsen, E. S. and McAllister, D. (2001). Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 55–55. ACM.

Mikalsen, M. A. (2013). Openacc-based snow simulation. Master's thesis, Norwegian University of Science and Technology.

Nordahl, A. (2013). Enhancing the hpc-lab snow simulator with more realistic terrains and other interactive features. Master's thesis, Norwegian University of Science and Technology.

Nvidia (2014). Cuda c programming guide. *http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html*.

Saltvik, I. (2006). Parallel methods for real-time visualization of snow. Master's thesis, Norwegian University of Science and Technology.

Vestre, F. M. J. (2012). Enhancing and porting the hpc-lab snow simulator to opencl on mobile platforms. Master's thesis, Norwegian University of Science and Technology.

Weber, R., Gothandaraman, A., Hinde, R. J., and Peterson, G. D. (2011). Comparing hardware accelerators in scientific applications: A case study. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):58–68.