View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NORA - Norwegian Open Research Archives

UNIVERSITY OF OSLO
Department of Informatics

Solving systems of hyperbolic PDEs using multiple GPUs

Master thesis

Martin Lilleeng Sætra
martinsa@ifi.uio.no

13th July 2007

Abstract

This thesis spans several research areas, where the main topics being parallel programming based on message-passing, general-purpose computation on graphics processing units (GPGPU), numerical simulations, and domain decomposition. The graphics processing unit (GPU) on modern graphics adapters is an inexpensive source of wast parallel computing power. To harvest this power, general purpose graphics programming is used. The main agenda of the thesis is to make a case for GPU clusters. Numerical simulations of hyperbolic conservation laws using explicit temporal difference methods (finite-difference methods (FDM), finite-volume methods (FVM) and modern high-resolution methods) are used as test-cases. The GPU cluster is proven to be usable, efficient and sufficiently accurate on the chosen test-cases. A white paper where the GPU cluster is used to perform PLU-factorizations of matrices is also included as an appendix.

Contents

Abstract **iii**

Contents **v**

List of Figures **ix**

List of Tables **xi**

Listings **xiii**

1 Introduction **1**
- 1.1 Research questions – Making a case for GPU clusters 1
- 1.2 Methods 2
- 1.3 Numerical simulation of PDEs 3
  - 1.3.1 Partial Differential Equations 3
  - 1.3.2 Hyperbolic PDEs 4
  - 1.3.3 Grids and schemes 7

2 Introduction to parallel programming **11**
- 2.1 The rapid evolution of computing power 11
- 2.2 Motivation: Large-scale complex simulations 12
- 2.3 Classifications of parallel systems and computers 14
  - 2.3.1 Flynn's taxonomy 14
  - 2.3.2 Shared memory and distributed memory computers . . 14
- 2.4 Cluster 15
  - 2.4.1 Beowulf 16
  - 2.4.2 Windows compute cluster server 17
- 2.5 Parallel applications 17
  - 2.5.1 Message passing interface 18
  - 2.5.2 Parallel I/O 20
  - 2.5.3 Profiling and optimizing 21

3 Introduction to GPUs and GPGPU programming 23
3.1 Introduction . . . 23
3.2 OpenGL and the graphics pipeline . . . 24
3.3 Shader programs . . . 26
3.3.1 The GPU . . . 28
3.4 Graphics programming . . . 29
3.4.1 Game programming . . . 29
3.4.2 Visualization . . . 29
3.4.3 Movie industry . . . 30
3.5 GPU vs CPU . . . 30
3.6 GPGPU programming . . . 31
3.6.1 Parallel problems . . . 32
3.6.2 The programming model . . . 32
3.6.3 Low-level languages . . . 33
3.6.4 High-level languages . . . 33
3.6.5 Tools . . . 34
3.6.6 Problems . . . 35
3.7 Conclusions . . . 36
3.7.1 The future . . . 36
3.7.2 Further reading and resources . . . 37

4 Parallel programming using the GPU 39
4.1 Models . . . 39
4.2 Ghost Cell Expansion . . . 41
4.3 Parallel algorithm – Communication time . . . 43
4.4 Upload and readback of textures . . . 44
4.5 Profiling . . . 45

5 Using the GPU to solve systems of hyperbolic PDEs 47
5.1 Hardware . . . 47
5.2 Application . . . 49
5.2.1 Domain decomposition . . . 50
5.2.2 Domain recomposition . . . 50
5.2.3 Shaders . . . 50
5.3 The linear test-case . . . 52
5.4 Finite-volume schemes . . . 55
5.5 High-resolution schemes . . . 58

6 Results 61
6.1 Visualization and animation of simulation data . . . 61
6.1.1 Storing simulation data . . . 61
6.1.2 Visualizing on the GPU . . . 62
6.2 Benchmarking and considerations . . . 65
6.2.1 Setup . . . 65

6.2.2 Benchmarks 67

7 Conclusions 73

7.1 Further research and extensions 74

8 Acknowledgements 77

A The Top 500 project 79

A.1 Introduction 80

A.2 Background 81

A.3 Algorithm 82

A.3.1 Global algorithm 83

A.3.2 Local algorithm 84

A.4 Results 88

A.4.1 Benchmark 88

A.4.2 Analysis 91

A.5 Conclusions and further research 91

A.6 Acknowledgements 92

B Shader programs 93

C PyShallows 97

Bibliography 99

List of Figures

1.1 A 2-dimensional Cartesian grid with equally sized cells. 7
1.2 Two-dimensional Cartesian domain decomposition. 8
1.3 Distributed grid 8

2.1 Distributed memory model. 15
2.2 Shared memory model. 16
2.3 Sequential I/O from one process. 20
2.4 Parallel I/O to multiple files. 21
2.5 Parallel I/O to a single file. 21

3.1 Graphics adapter from XFX 24
3.2 Texturing and rasterizing in the OpenGL graphics pipeline. . 26
3.3 OpenGL graphics pipeline 27
3.4 FLOPS performance of CPUs compared to GPUs 31

4.1 Overview of a GPU cluster. 40
4.2 Ghost cell expansion 42
4.3 Ghost cell data transfer. 43
4.4 Jumpshot profiling tool. 46

5.1 Domain recomposition. 51
5.2 Computational molecule for the linear wave equation. 53
5.3 Boundary conditions. 55

6.1 Screenshots: The linear wave equation 1. 63
6.2 Screenshots: The linear wave equation 2. 63
6.3 Screenshots: The shallow-water equations 1. 64
6.4 Screenshots: The shallow-water equations 2. 64
6.5 Screenshots: The Euler equations 1. 65
6.6 Screenshots: The Euler equations 2. 66
6.7 Plot of overall execution times. 69

6.8 Plot of GCE level impact. 70

A.1 Parallel PLU factorization 83
A.2 Data send patterns for parallel PLU 85
A.3 Data representation for parallel PLU 86
A.4 Setup of nodes for parallel PLU 88

C.1 Screenshot of PyShallows in action 1. 98
C.2 Screenshot of PyShallows in action 2. 98

List of Tables

5.1 Price table for a gigabit node. 48
5.2 Price table for gigabit network components. 48

6.1 Overall execution times. 68
6.2 Impact of GCE on total execution time of the LWE. 69
6.3 Shader benchmarks. 71
6.4 GPU-CPU communication benchmarks. 71
6.5 Inter-node communication benchmarks. 71

A.1 Variation of the subsize parameter. 89
A.2 Variation of the number of processes. 90
A.3 The time spent transmitting data. 90
A.4 Single node computing. 91

Listings

3.1 Vertex shader program 27
3.2 Fragment shader program 28
5.1 A Shallows example 52
5.2 The linear wave equation solver. 54
A.1 Example on a deadlock in an MPI-2 program 82
A.2 Setting up row- and column-communicators 87
B.1 The Lax-Friedrich scheme in GLSL. 94
B.2 The Lax-Wendroff scheme in GLSL. 95

Chapter 1

Introduction

"The Road goes ever on and on
Down from the door where it began."
— Bilbo Baggins (J. R. R. Tolkien)

This master thesis contains several topics of interest, where the main areas are parallel programming using message-passing, general-purpose computation on graphics processing units (GPGPU), domain decomposition, and numerical methods for solving partial differential equations (PDEs). All these topics are more or less well-known and there are a large number of publications in these fields. The main contribution of this thesis is the combination of GPGPU and parallel programming with MPI-2. Chapters 2 and 3 contain background material on the topics listed earlier with some specifics for this thesis. In Chapters 4 and 5 the work performed in this thesis is discussed. The last two chapters are dedicated to results and conclusions. All development and testing was performed at SINTEF ICT, Oslo. Since the field of GPGPU is in such rapid evolution, this thesis should only be considered a snapshot of the available technology at the time of writing. This thesis also features a lot of background material, which is natural since it contains several areas of interest that all need to be introduced.

1.1 Research questions – Making a case for GPU clusters

Throughout my thesis I strive to give complete and precise answers to the following research questions. The answers will be summed up in Chapter 7.

Are GPU clusters a viable computational resource when considering parallel problems?

What are the pros and cons of GPU clusters?

Are the obtained results accurate enough for practical use?

The first research question is narrowed down to the test-cases I use; explicit discretization of an initial-boundary-value PDE problem. More specifically the test-cases are systems of hyperbolic conservation laws solved over a finite two-dimensional Cartesian grid using high-resolution schemes.

1.2 Methods

The main agenda of this master thesis is to test the concept of a GPU cluster and measure its efficiency. A class of equations called nonlinear hyperbolic conservation laws will be used to evaluate this concept. This does not mean that all the results and conclusions in this thesis can be extended to characterize GPU clusters in general, but they will reveal a great deal about the parallel capabilities of a GPU cluster. To understand the background material in the following two chapters it is necessary to have some understanding of the type of problems that will be considered. A detailed description of the problems is therefore presented in Section 1.3 in Chapter 1 and in Chapter 5.

The PDEs will be discretized explicitly and solved over a two-dimensional Cartesian grid. This grid will be split up into sub-domains that are distributed to the different nodes in the cluster and solved locally on the graphics adapter using schemes that will be introduced later. Since the grid is split up and the different parts will be on separate nodes, we will have a communication need in the boundary between sub-domains. Each node will perform computations using a discretized scheme on the following form:

$$u_{i,j}^{n+1} = F(u_{i-I,j}^{n}, ..., u_{i+M,j}^{n}, u_{i,j-N}^{n}, ..., u_{i,j+O}^{n}), \tag{1.1}$$

where $u_{i,j}^{n+1}$ is the solution at time-step $n+1$ at spatial coordinates (i,j) and I, M, N, and O will depend on the scheme. If we use a scheme that has $I = M = N = O = 1$, this means that all information will be displaced with *one* grid cell per time-step.

The main reasons for choosing this particular class of equations and schemes to evaluate the GPU cluster are:

1. They are easy to solve numerically in parallel.
2. The numerical methods for solving these equations are extensively used and tested.
3. The complex schemes we use to numerically solve the equations have a high arithmetic intensity in computing F, and graphics adapters

are perfect for handling this. Simpler schemes do not have as high arithmetic intensity, so we will expect a lower speedup on these.

In addition to using hyperbolic PDEs as test-cases, I have also included a white paper in Appendix A, where the GPU cluster is used to perform PLU-factorization of a matrix. PLU-factorization is used to solve linear systems of equations, e.g., elliptic PDE problems. Elliptic equations give boundary-value problems that require the solution at all points to be simultaneously determined based on the boundary conditions all around the domain. This require a large system of linear equations to be solved for the values of u at each grid-point. PLU-factorization is one way to do this. This white paper was a joint work with two other master students as SINTEF ICT, Oslo.

1.3 Numerical simulation of PDEs

This section presents the problems being solved in this thesis, and some of the theory behind them. The development of the different numerical schemes used to solve these equations is presented in Chapter 5, with focus on the schemes used in this thesis. Some of the physics behind the equations will also be explained. An understanding of the problems presented here will help to make the reasons for my choices in the rest of the thesis clearer.

1.3.1 Partial Differential Equations

A PDE is an equation involving an unknown function of several independent variables and the partial derivatives of the function with respect to these variables. These equations differ from ordinary differential equations (ODEs), which only contain *one* independent variable. PDEs or systems of PDEs are often used to describe different physical phenomena. PDEs model processes that are distributed in space, or in both space and time. PDEs can describe many physical processes in different dimensions. Usually it is simulations in three dimensions that are of greatest interest, since this is most easily transferred to the real world. PDEs can be used to describe for example heat transfer, sound, fluid flows, elasticity and gasses. This makes them a powerful tool in many scientific areas, like engineering, chemistry, physics, biology and mathematics.

PDEs are divided into classes depending on their properties. Equations of the same class exhibit similar general features. We will review three classifications here. Not all PDEs fall into these categories, but many of those arising in practice do. For a linear second-order differential equation in two independent variables of the form:

$$au_{xx} + bu_{yx} + cu_{yy} + du_x + eu_y + fu = 0, \tag{1.2}$$

the type of PDE is categorised depending on the sign of the discriminant:

$$b^2 - 4ac \begin{cases} < 0 \Rightarrow \text{elliptic}, \\ = 0 \Rightarrow \text{parabolic}, \\ > 0 \Rightarrow \text{hyperbolic}. \end{cases}$$

The names are analogies with the conic sections. The *order* of the equation is determined from the highest-order differential, such that $u_{xx}+u_{yy} = g$ (the 2D Poisson problem) is a second-order differential equation. While elliptic equations, such as the Poisson problem, typically are temporal independent, parabolic and hyperbolic equations usually are time-dependent. The canonical examples are the 1D heat equation $u_t = \beta u_{xx}$ (where $\beta > 0$) for a parabolic problem, and the 1D wave equation $u_{tt} = c^2 u_{xx}$ for a hyperbolic problem. The different classes of equations describe different types of phenomena and they require different techniques for their solution, both analytically and numerically. Hyperbolic PDEs is the type of PDEs explored in this master thesis. More specifically, we will numerically solve hyperbolic conservation laws. We will solve them in two spatial dimensions, but the equations and our methods can be extended to more space dimensions.

1.3.2 Hyperbolic PDEs

The simplest example of a hyperbolic PDE is the constant-coefficient *advection equation*, also called the *transport equation*. In 1D, this equation is written as:

$$u_t + a u_x = 0,$$

where u is the conserved scalar quantity stating the advection velocity. The solution to this equation is $u(x,t) = u(x-at, 0)$. This means that any profile in u simply advects with the flow at velocity a.

We will use the linear wave equation, the shallow-water equations and the Euler equations as test-cases in this thesis. The shallow-water equations and the Euler equations are examples of a special type of hyperbolic PDEs derived from conservation laws. The reasons for using this particular class are that all disturbances have a finite speed of propagation, modern *high-resolution* schemes are based on temporal discretizations, and they are easy to solve in parallel. The equations will be solved, all in two spatial dimensions using a simple *finite-difference* scheme for the linear wave equation, and both classical and two high-resolution finite-volume [LeV02] schemes for the shallow-water equations and the Euler equations. The schemes are based on explicit temporal discretization. This means that there is no coupling between unknowns across different grid cells, which makes them ideal test-cases for parallel computing, e.g., on a cluster. Another important point is that high-resolution schemes yield very high arithmetic intensities, which is exactly what the GPU is best at. The schemes used for the linear wave equation and the shallow-water equations will be derived, but the schemes for the Euler equation will only be outlined and described.

Conservation laws Conservation laws are a special type of equations where a set of quantities are conserved. The mathematical model of a conservation law often appears as linear or non-linear hyperbolic partial differential equations in *divergence form*. Seen in one spatial dimension, the equations can be written:

$$Q_t + F(Q)_x = 0,$$

where $Q \in \mathbb{R}^m$ is the set of conserved quantities and F is a flux function.

Definition 1.1:
A conservation law states that the rate of change of a quantity or set of quantities within a given domain Ω equals the flux over the boundaries $\partial\Omega$.

Definition 1.1 gives us the conservation law in integral form:

$$\frac{d}{dt}\iint\limits_{\Omega} Q dx dy + \int\limits_{\partial\Omega} (F, G) \cdot (n_x, n_y) ds = 0. \tag{1.3}$$

Here we assume that $\Omega \in \mathbb{R}^2$ with outer normals (n_x, n_y), and that the flux has two components, $F(Q(x, y, t))$ and $G(Q(x, y, t))$. Using the divergence theorem we can derive the conservation law in differential form:

$$\partial_t Q + \partial_x F(Q, x, y, t) + \partial_y G(Q, x, y, t) = 0. \tag{1.4}$$

Definition 1.2:
When the Jacobi matrix $\partial_Q(F, G) \cdot n$ has only real eigenvalues, and a complete set of eigenvectors for all unit vectors n, the system is said to be hyperbolic.

Knowing the differential form (1.4) of the conservation law, Definition 1.2 can be used to show that a system is hyperbolic.

Hyperbolic systems of conservation laws exhibit very singular behaviour, and admit various kinds of discontinuous and non-linear waves. These can be shocks, rarefactions, phase boundaries, fluid and material interfaces, etc. Resolving these propagating discontinuities is a difficult task. In addition to this; non-linear hyperbolic equations are hard to solve both analytically and numerically. They can also form discontinuous solutions from smooth initial data. We will now have a look at the three equations considered in this thesis.

The linear wave equation

The linear wave equation is a prototype example of an hyperbolic PDE. A derivation of the equation will not be given here. The interested reader can find a derivation in the article “Wave equation” [Wik07d].

The linear wave equation in its simplest form reads:

$$u_{tt} = cu_{xx} + cu_{yy}, \tag{1.5}$$

where u is a scalar function and c is a constant equal to the propagation speed of the wave.

The linear wave equation is not a conservation law and is not derived from the physical principal of conservation of quantities. However, it can be written as a system of first-order conservation laws:

$$u_t - v_x - v_y = 0, \quad (1.6)$$

$$v_t - u_x - u_y = 0, \quad (1.7)$$

where u and v are scalar functions.

This equation is mostly used in this thesis on the ground that it is very simple, and therefore very usable for explaining and demonstrating general principals used on all three test-cases.

The shallow-water equations

The shallow-water equations constitute a more complex model to simulate wave propagation than the linear wave equation. These non-linear equations describe flow below a horizontal pressure surface in a fluid. Without bottom topography (bathymetri), we get the following homogeneous system:

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix},$$

where u is the velocity in the x-direction, v is the velocity in the y-direction, h is the height of the horizontal pressure surface, and g is the gravitation.

It is important to notice that the wave length of the modelled problem needs to be much larger than the water depth in order for the shallow-water equations to be valid.

The Euler equations

The Euler equations describe the dynamics of an ideal gas. It is based on conservation of mass, momentum and energy. The equations, on matrix form, in two spatial dimensions can be seen below:

$$\begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E \end{bmatrix}_t + \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ u(E+p) \end{bmatrix}_x + \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ v(E+p) \end{bmatrix}_y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

where ρ is the density, u and v are velocity in x- and y-directions, respectively, p is pressure, and E is total kinetic energy given by:

$$E = \frac{1}{2}\rho(u^2 + v^2) + p/(\gamma - 1),$$

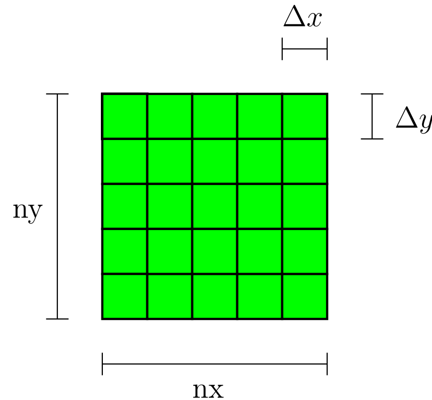

Figure 1.1: A 2-dimensional Cartesian grid with equally sized cells.

where γ is the gas constant, e.g., 1.4 for air. $\gamma = 1.4$ is used in all computations in this thesis.

1.3.3 Grids and schemes

The problems considered in this thesis have been solved on a two-dimensional Cartesian grid, as illustrated in Figure 1.1. Each quantity in the equations has values in each grid cell.

The global domain is decomposed and distributed to all the nodes in the GPU cluster, as Figure 1.2 shows. Each grid cell on each node will be unique, with exception of the *ghost cells* which contain copies of the nearest grid cells on the adjacent sub-domains. When computing the grid cells close to the boundary of the domain, the scheme may require values from outside of the domain. There are two ways of handling this; either we use a modified scheme close to the boundary, or we add ghost cells to the domain boundary. If we e.g., have a scheme on the form (1.1) where $I = M = N = O = 1$, we will need one layer of ghost cells around the domain. The use of ghost cells in this thesis will be introduced in Section 4.2 in Chapter 4, and are not displayed in the figures presented here. The organization of the distributed grid can be seen in Figure 1.3.

The schemes used to solve these three equations have been chosen due to their inherent parallel nature, and the fact that they are very arithmetic intensive. This, as explained earlier, makes them good candidates for testing a GPU cluster. For the linear wave equation, a standard second-order discretization based on the Taylor series is used. This scheme is not as complex, as therefore not as arithmetic intense as the Lax-Friedrich and Lax-Wendroff schemes used for the shallow-water equation, or the central-upwind scheme used for the Euler equations. Derivation and further discussion of the schemes are found in Chapter 5.

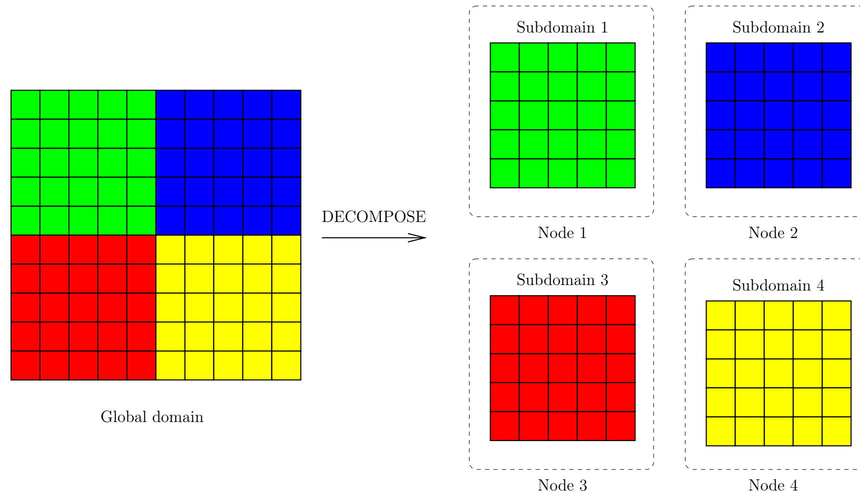

Figure 1.2: Two-dimensional Cartesian domain decomposition.

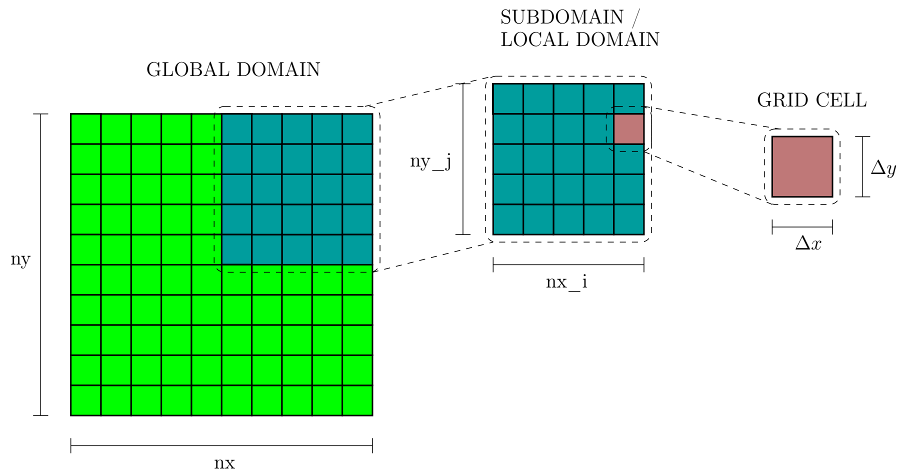

Figure 1.3: The relationship between the global grid and the sub-domain grids.

Further reading To gain a deeper understanding of the mathematical analysis and numerical methods for hyperbolic conservation laws, the interested reader can find a large number of papers and articles on the Conservation Laws Preprint Server [HOHL]. Images of these phenomena can be seen in Van Dyke's Album of Fluid Motion [Van82].

Chapter 2

Introduction to parallel programming

> "*Parallel programming may do something to revive the pioneering spirit in programming, which seems to be degenerating into a rather dull and routine occupation.*"
>
> — S. Gill, Computer Journal, 1958

This chapter will present a generic and traditional approach to parallel programming. The first sections in this chapter sheds some light on the history and evolution of parallel programming, and the last sections describe the Message Passing Interface (MPI) [MPI], and how and why it is used in this master thesis. We will also try to place MPI in the bigger picture that is parallel programming. There exist several implementations of MPI, and I will mention some of them in the last section. This chapter will also give a better understanding of the GPGPU way of programming, in the sense that parallel computing is introduced.

2.1 The rapid evolution of computing power

Prior to the development of computers there were few who thought we would ever be capable of performing several hundred arithmetic computations in a matter of seconds, or even less. When the computer era emerged, however, this soon became a reality. This was a revolution for the engineers and scientist who were able to utilize this new computing power. One was suddenly able to compute numerical solutions to problems that before had to be solved analytically, or in many cases could not be solved at all. When it was made possible to perform hundreds of operations per second, people wanted even more, as they always tend to do. They wanted to be able to per-

form thousands of arithmetic computations per second, then millions, then billions, and this is where we are today (with the exception of the worlds fastest supercomputers, which are already able to perform trillions of computations per second). The current goal is quadrillions of operations per second (10^{15}). A common measure for arithmetic intensity is floating-point operations per second (FLOPS). Billions of operations per second is referred to as gigaflops, trillions as teraflops, and quadrillions as petaflops. This may seem like all the computing power we will ever need, but as we will see, this is not the case. These numbers have not yet been connected to any specific architecture, this will be introduced later.

2.2 Motivation: Large-scale complex simulations

The following example will demonstrate why we need as much as trillions of arithmetic operations per second and more. The example is a modified version of the example presented in the book "Parallel programming with MPI" [Pac97]. Suppose we want to predict the weather over Norway every hour for the next two days. We use a grid that covers all land area, and goes from sea level and 20 km up into the atmosphere. The grid is cubical and we have a grid spacing of 100 meters in all dimensions. This gives us $3.8 \cdot 10^5 \text{km}^2 \times 20\text{km} \times 10^3 \text{cubes per km}^3 = 7.6 \cdot 10^9 \text{cubes}$. If we assume it takes 100 calculations to determine the weather in one cube and that we use 1 hour time-steps, then we will need to perform $7.6 \cdot 10^9 \times 100 \times 48 = 3.65 \cdot 10^{13}$ calculations to predict the weather in every cube for the next 48 hours. If we further expand this with longer forecasts or larger area, it is clear that 10^9 arithmetic computations per second is not nearly enough. If this prediction takes several hours or even several days, it is clearly of little use, as the results would be outdated. Other large-scale simulations may not be under such time pressure. On the other hand, we want the simulations performed as fast as possible.

So what is the solution to this problem? We could solve it by sticking to the well-known sequential model of the von Neumann computer, and simply extend this further. But we are starting to see a decrease in the growth of processor speed. The growth in processor speed has been doubled every 24th month for the past decades, but this is not the case anymore. There are several reasons why, and without going in depth on this subject, I will mention some of them here.

Not only is the evolution and development of CPUs slowing down, but the way we are designing and building CPUs today will probably eventually have to come to a halt. The following example, based on an example from "Parallel programming with MPI" [Pac97], demonstrates this. It is important to note that this is a constructed example to prove the insufficiency of the traditional single-CPU computer when it comes to big and complex

computational problems. One would of course never attempt to run such a large-scale simulation on a single CPU today. Suppose we want to build a computer capable of doing one trillion computations each second. If we want to add two vectors of length one trillion, we would successively fetch one and one entry from memory, add the values and write the answer back to memory. This gives $3 \cdot 10^{12}$ copies between registers and memory each second. If we assume that the data travel with the speed of light ($3 \cdot 10^8$ m/s), and that r is the distance from the CPU to the memory, then r must satisfy:

$$3 \cdot 10^{12} r\text{m} = 3 \cdot 10^8 \text{m/s} \cdot 1\text{s},$$

or $r = 10^{-4}$m. The computer must, of course, contain at least three trillion words of memory to store the three vectors x, y, and the result z. Memory is typically organized in a rectangular grid. If we use a square grid with side length l and place the CPU in the center of the grid, then the average distance from a memory location to the CPU will be $l/2$. This leads to the equation $l/2 = r = 10^{-4}$ or $l = 2 \cdot 10^{-4}$m. We let our memory words form a square grid, a typical row of memory words will then contain $\sqrt{3 \cdot 10^{12}} = \sqrt{3} \cdot 10^6$ words. This means we have to fit a single word of memory into a square with a side length measuring:

$$\frac{2 \cdot 10^{-4}\text{m}}{\sqrt{3} \cdot 10^6} \approx 10^{-10}\text{m}$$

Since this is the size of a small atom, it poses a big problem. This leaves us with a choice; Either we must find a way to represent 32 bit, or preferably 64 bit, with a single atom, or we will have to use a different architecture all together. This is where parallel programming provides us with a solution. The traditional solution is to use high-performance computing (HPC), either in the form of large clusters of computers, or as supercomputers with up to 130 000 processors (IBM's BlueGene/L) [UUN] working in parallel. In this master thesis we will pursue a somewhat different solution, by utilizing the raw computing power of modern graphics hardware combined with a computer cluster.

Another problem that becomes more and more apparent due to the single-minded focus on performance (FLOPS), is the power problem. The power usage is increasing faster than the computational power. IBM's BlueGene/L is currently ranked as number one in the Top 500 list of supercomputers[1]. This supercomputer needs a stunning 2.5 MW to run [SHF06], and this is actually the most "green" computer on the Top 500 list [UUN]. The Japanese Earth Simulator, for example, uses up to 7 MW for power and cooling [SHF06]. A similar problem applies to both conventional single- or dual CPU computers and supercomputers of all sizes.

[1] As of 2007-05-01.

2.3 Classifications of parallel systems and computers

There are many platforms, interfaces, architectures and terms in parallel computing, and this section will hopefully give a basic introduction to the most important ones.

2.3.1 Flynn's taxonomy

According to Flynn [Fly66], parallel computers and computer systems can be classified using a simplified scheme. This scheme has come to be known as Flynn's taxonomy. The four schemes are Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD), Single Instruction Single Data (SISD), and Multiple Instruction Single Data (MISD). These terms are also used in the GPU section to classify GPUs. These schemes were later extended to Single Program Multiple Data (SPMD) and Multiple Program Multiple Data (MPMD). The SIMD scheme executes the same instruction with multiple data, while the MIMD scheme executes multiple instructions on multiple data. The SPMD scheme runs the same program on multiple processors, where each processor works on a different set of data. Typically you start out with a large dataset which you divide into subsets and distribute to each processor. The processors then run the program on the portion of the dataset they were given, and the complete result can finally be collected from the memory of each processor after execution. The MPMD scheme runs different programs on each processor, but it is parallel in the sense that each program handles a task towards a common goal. It is also possible to have hybrid schemes where several schemes are combined, where the processors are grouped, some SPMD and some MPMD.

The application written for this thesis uses the SPMD scheme, where all processors execute the same program in parallel, but uses different datasets. This specific application has data values on a Cartesian 2-dimensional grid, which is decomposed into smaller Cartesian sub-grids and distributed to all the available nodes. Each node then executes the same program on its local sub-grid and communicates with the neighbouring nodes. To recompose the global domain after the simulation is complete, it is necessary to collect data from all nodes. After this is done, the data can be analyzed and visualized.

2.3.2 Shared memory and distributed memory computers

Flynn's taxonomy is used to classify parallel software and parallel schemes. Now we need a way to classify the parallel hardware. Loosely-coupled multiprocessor systems, also known as clusters, have different ways of distributing the resources and different ways of accessing them. The alternative to loosely-coupled systems is tightly-coupled systems, where the CPUs are connected on a bus-level. The different classes of parallel systems refer to where

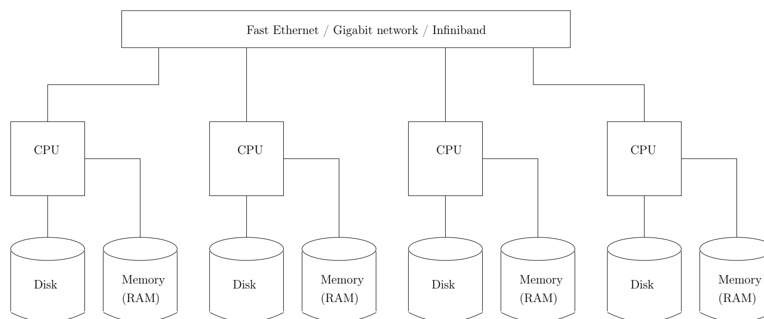

Figure 2.1: Distributed memory model.

the memory is located, and how it is utilized by each node in the cluster. The first model is the distributed memory model, where each node has access to its own memory. The nodes are then typically connected through a fast Ethernet, a gigabit, a Myrinet, or a Infiniband [IBT] network. Since the network often is a major bottleneck in this type of systems, the challenge is to minimize communication between processors. The type of system used in the application for this thesis uses a loosely-coupled distributed memory system, also known as a computer cluster. This type of distributed memory model can be seen in Figure 2.1.

Shared memory systems, on the other hand, typically contain one large block of memory (RAM) that all processors have access to, as seen in Figure 2.2. In addition, each processor has its own on-chip cache memory. Such a system is relatively easy to program, since every processor has access to the same memory (RAM). There are, however, some complications with this model. The bandwidth between CPUs and memory becomes a bottleneck, and it may not scale very well. It is also necessary to update the cache on every CPU when the cache is changed on one of them, such that the processors do not work on incoherent cache data.

Distributed shared memory (DSM) is a variation of the shared memory model, in which each node of a cluster has access to a large shared memory in addition to each node's limited non-shared private memory. In this model each node can use the local memory for local operations to prevent unnecessary traffic on the memory bus, while still having access to a shared global memory.

2.4 Cluster

A computer cluster is a group of loosely coupled computers that work closely together, so that in many respects they can be viewed as though they are a single computer. There exist several types of clusters, including HA-clusters

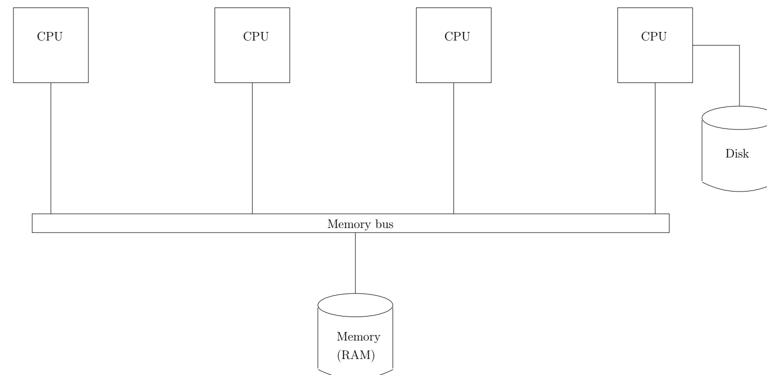

Figure 2.2: Shared memory model.

(High availability clusters) and load-balancing clusters. HA-clusters have redundant nodes, so that if one node should fail, another will take over. Load-balancing clusters use front ends that distribute the workload to the back-end of servers or nodes. This type of cluster is primarily implemented for improved general performance.

A high-performance cluster is the type of cluster that has been used in this thesis. It provides increased performance by splitting a computational task across many different nodes in the cluster. It is most commonly used in scientific computing. The programs that run on high-performance clusters are usually designed specifically for this purpose, and all nodes (processes) should be working as independently of each other as possible. Although each process should work independently, it will often be necessary for one or more processes to send messages between each other. MPI makes this possible. There are several implementations of high-performance cluster services. The most commonly used are Beowulf clusters for UNIX/Linux, Microsoft's Windows Compute Cluster Server, and different implementations of MPI for all kinds of operating systems.

2.4.1 Beowulf

Beowulf is a way of clustering Linux machines. It is not a single piece of software you can buy as a complete package, but rather an idea of how ordinary Linux desktops can be used to achieve the same performance as a million-dollar supercomputer.

The Beowulf project was conceived in early 1994 by Donald Becker and Thomas Sterling. They were working at the Center of Excellence in Space Data and Information Sciences (CESDIS) under the sponsorship of the HP-CC/NASA Earth and Space Sciences (ESS) project. CESDIS is a division of the non-profit University Space Research Association (USRA). CESDIS

is located at the Goddard Space Flight Center in Greenbelt Maryland and supported in part by the ESS project. The initial prototype was a cluster consisting of 16 nodes with DX4 processors connected with bonded Ethernet[2]. When the first Beowulf cluster was built, it was to address problems associated with the large data sets that are often involved in ESS applications. As more and more companies and institutions are starting to see the potential in computer-based simulations, Beowulf clusters have become an inexpensive way of providing the massive (parallel) computational power needed.

There are several pieces of software many people have found useful for building Beowulf clusters [Beo], and together they form a powerful platform for parallel programming. They include MPICH/MPICH2, LAM, PVM, the Linux kernel, the channel-bonding patch to the Linux kernel (which lets you bond multiple Ethernet interfaces into a faster virtual Ethernet interface), the global pid space patch for the Linux kernel (which lets you see all the processes on your Beowulf with ps, and eliminate them), and DIPC (which lets you use `sysv` shared memory, semaphores and message queues transparently across a cluster).

Almost all the components mentioned in the previous section are also available to Windows and other platforms in some form. The big advantage with Beowulf, however, is that it is free. The downside of choosing a Beowulf cluster is that you are not guaranteed support on the software, but you will probably find answers to all your questions online anyhow.

2.4.2 Windows compute cluster server

The closest Windows equivalent to Linux/UNIX Beowulf is Microsoft's Windows Compute Cluster Server[3] [Micb]. This is also a collection of tools and technologies that together form a HPC cluster. The Windows Compute Cluster Server is of course made up of Windows software, and uses MPI-2 for message passing. Companies and institutions that already have a Windows platform might be better off choosing the Windows alternative over a Beowulf cluster.

2.5 Parallel applications

Now that the hardware and some ideas are introduced, we will take a look at parallel programming from a software and application viewpoint. There are two main actors in this area, Parallel Virtual Machine (PVM) and the Message Passing Interface (MPI and MPI-2). Before 1996, PVM was the

[2] Several physical Ethernet interfaces used as one logical/virtual interface.

[3] Windows Compute Cluster Server is new with Windows 2003. Earlier, Microsoft Cluster Server (MSCS) was used instead.

de-facto standard of parallel applications. In 1996, a paper called "PVM and MPI: a Comparison of Features" [GKP96] was published, and in the later years MPI and MPI-2 have taken over a large portion of the parallel software industry. MPI however, is not a virtual parallel machine, but rather a standard for passing messages between processes and groups of processes. The topic of message-passing in parallel applications will be more thoroughly treated in Section 2.5.1.

There are many things that must be considered when developing and using parallel algorithms. Most importantly; it is of no use to rewrite a good sequential algorithm to a poor parallel algorithm. Some algorithms are inherently parallel in nature, but in some cases the algorithm cannot be adopted to the parallel way of thinking. If you, after careful consideration, come to the conclusion that your algorithm and application will benefit from parallel execution, the next issue is to choose a development platform and technology. Since we will be using a GPU cluster in this master thesis, this is the platform we will discuss further.

Another issue that needs consideration when designing parallel programs for clusters is the number of available nodes and processors. Sometimes it is not even necessary to pay attention to the underlaying hardware, it all depends on the algorithm you want to implement. I will illustrate this with an example. If you are writing a program that calculates the decimals of pi using e.g., the circle algorithm, you want as many available nodes and processors as possible. The program can usually even find out how many available processors there are, and distribute the problem itself. In your next program you want to solve some equation with quantities spread over a Cartesian grid (like in this thesis). This time the logic of the application will have a closer relationship with the number of nodes and processors available. For instance, if your grid is of size 100×100, then it is likely that you cannot use more than 10000 processors simultaneously. And furthermore, it would be very ineffective to have only one grid point per process. This is because of the communication cost of sending messages between processes. The number of processes versus the number of nodes must also be considered if possible. Each node can have multiple processors, and many HPC clusters are designed this way. This is often pictured as a $P \times Q$ grid, where P is the number of processors and Q the number of nodes.

2.5.1 Message passing interface

MPI is a standard interface for message-passing. This standard is developed by the MPI Forum (MPIF). The MPIF consists of over 40 different organizations with a common interest in developing a standard for message-passing programs. They state that: *"The goal of the Message Passing Interface, simply stated, is to develop a widely used standard for writing message-passing programs"* [MPI]. Version 1.0 was released in June 1994. In Novem-

ber 2003 the MPI-2 standard was published; it contained specifications for version 1.2 of MPI, and MPI-2. The MPI-2 specification describes additions to the MPI standard. These include process creation and management with a dynamic process model, one-sided communication, extended collective operations, external interfaces, parallel I/O, and additional language bindings. Dynamic process model basically means that all processes do not need to be created at startup, which was the case with MPI 1.2. From now on, MPI will always refer to the MPI-2 standard, if not stated otherwise.

Implementations of MPI There are many implementations of the MPI standard. The most used ones, to name a few, are MPICH2 [Argc], Microsoft MPI [Mica], LAM MPI [BDV94], OpenMPI [GFB+04], DeinoMPI [Arga], and pyMPI [Law]. For this master thesis, MPICH2 has been used, which is an implementation made by the Mathematics and Computer Science Division of Argonne National Laboratory. This software package consists of two main components, the MPI launcher and the MPI process manager.

The MPI process manager, also called SMPD, is the primary process manager for MPICH2 on Windows. It is also used when running on a combination of both Windows and UNIX/Linux machines, and it is written in C. On UNIX/Linux, MPD is the default process manager, and it is SMPD's equivalent. MPD is written in Python. Together with the MPI process launcher this is what primarily makes up the MPICH2 software package. The MPI launcher is called `mpiexec`, and this is the program that is executed when running MPI-jobs. This program can take a large number of arguments to control how and where the MPI job is executed.

Programming with MPI Since MPI is a widely used cross-platform standard, there are many freely available resources and utilities. Some of them are mentioned in Section 2.5.3. The parallel way of programming takes some getting used to, and there are numerous issues to keep track of. I will mention some of them here, and some will be mentioned in the description of my application.

When programming with MPI, you use C/C++ or Python, and MPI methods and constants. MPI is as a message-passing interface, which means it is used to pass messages between different processes. These processes could be on a single computer, or distributed among any number of computers. MPI implements a large number of functions, but usually only a relatively small subset is needed. There are two main classes of methods, called blocking and non-blocking calls. Blocking calls make the process stop and wait for the call to complete. Non-blocking calls on the other hand, do not wait for the call to return, but continues executing the program. This means that you cannot be certain whether the MPI-call was successful or not when using non-blocking functions. The advantage of using non-blocking calls over

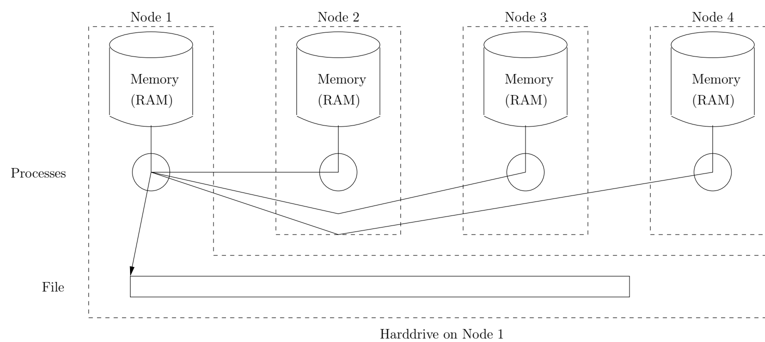

Figure 2.3: Sequential I/O from one process.

blocking calls, however, is that you avoid idle processes, since the process continues executing the program immediately.

2.5.2 Parallel I/O

There are several ways to handle file reading and writing with MPI [GLT99]. Which way is most effective depends on the setup of the parallel system (one disk, multiple disks, etc.) and the application itself. The easiest method is to let one process do all the file handling, and let all the other processes send their data to this process. The file will then be written to sequentially, but you will have no parallelism. The method is shown in Figure 2.3.

There are several advantages with this approach:

- This method will work on all types of parallel systems.
- It is possible to use I/O libraries without parallel support since only one process executes I/O operations.
- The resulting single file is easy to handle.

The next method is non-MPI parallel I/O to multiple files on multiple nodes as shown in Figure 2.4. The big advantage here is that you have no communication between processes. This approach, however, has multiple disadvantages as well:

- The resulting multiple files must be joined together to get the complete results.
- To work with the resulting files, it may be necessary to start another MPI-program with exactly the same number of processes, in order to access the files spread across the nodes. Alternatively one could transfer all the files to a single node and access them there, but even then one would probably choose to use MPI.

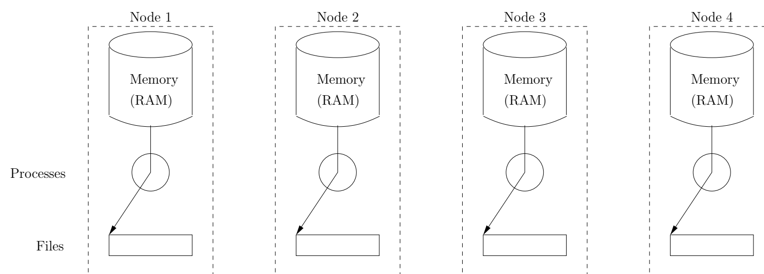

Figure 2.4: Parallel I/O to multiple files.

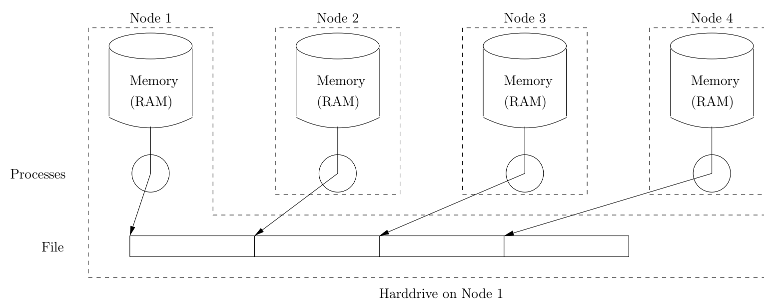

Figure 2.5: Parallel I/O to a single file.

- It is difficult to handle multiple files when moving, copying and/or sending them.

This method can also use MPI parallel I/O, with the same advantages and disadvantages.

The last method is parallel MPI I/O to a single file. There are many ways to do this, since MPI has a broad collection of functions for file handling. One way of doing it is to use the `MPI_File_set_view` function to give each process a different view in the file. In this way each process will write to different parts of the file using the `MPI_File_write` function as shown in Figure 2.5.

There can also be sequential I/O operations performed on each node in addition to the parallel I/O, e.g., for debugging and log messages.

2.5.3 Profiling and optimizing

When dealing with sequential programs one needs only examine the algorithm to predict the behaviour of the program. This is not the case with parallel programs. It is notoriously difficult to predict the behaviour of parallel programs, precisely because they are executed in parallel. In addition to the normal bugs, parallel programs are subject to performance issues, where

you will get the expected results, but more slowly than anticipated. To find and correct this type of issues, a profiling tool is used.

There are many tools and utilities for profiling parallel programs. The most well-known are Upshot [Argd], AIMS [NAS], ParaGraph [HF], Traceview [MHJ91], XPVM [Oak], XMPI [Cen] and Pablo [oI]. This is a detailed description of some of them:

Gist A proprietary tool that was developed to study log files produced on BBN[4] parallel computers.

Upshot A tool developed by Argonne National Laboratory, inspired by gist. Upshot provides a graphical view of the log file, where all process states can be viewed simultaneously in parallel time lines. In that way problems can be identified by looking at the time lines and adjusting the program so that no processes are forced to wait for other processes. Upshot was originally developed for the X windowing system with the Athena widget set system, but was expanded and partially rewritten several times later.

Jumpshot In order to address the portability and maintainability issues faced by Upshot, it has been ported to a Java version called Jumpshot [Argb], which is in use today.

AIMS Automated Instrumentation and Monitoring System (AIMS) is a software toolkit developed by NASA. It is a software suite with tools for measuring and analyzing the performance of FORTRAN and C/C++ message-passing programs that uses the NX [Pie94], PVM, or MPI communication libraries. AIMS can be used to illustrate algorithm behavior, to help analyze program execution, and to highlight problem areas. In other words, AIMS works much like Jumpshot.

[4] BBN Technologies (originally Bolt Beranek and Newman) is a high-technology company that provides research and development services.

Chapter 3

Introduction to GPUs and GPGPU programming

"*To err is human - and to blame it on a computer is even more so.*"
— Robert Orben

This chapter presents the use and the intended use of graphics hardware in the past and today. The use of GPUs for general purpose programming and some tools and programming languages are also presented. The chapter will cover the subjects necessary to understand the use of GPU in this master thesis, with relevant examples. Other uses of the GPU which are not directly relevant for this thesis are also included for completeness.

3.1 Introduction

The most widely used computational resource today is the CPU. These processors have grown in capacity according to Moore's law in the past 50 years. Moore's law states that the CPU doubles in capacity every 24 months. More accurately it says that the complexity of integrated circuits, with respect to minimum component cost, doubles every 24 months [Moo00]. Moore has, however, stated that his law may soon become obsolete. While the evolution of CPUs is slowing down, the need for computational resources is increasing at a high rate. We may turn to supercomputers or clusters for these resources, or we may turn to new alternative solutions.

One alternative that became viable ground in 1999 is the use of commodity off-the-shelf (COTS) graphics adapters [OLG+07], like the one seen in Figure 3.1, for general-purpose computing. This was an effect of the introduction of programmable graphics adapters, with the GPU as the programmable processing unit. In 1999 the current generation of GPUs already

Figure 3.1: The newest generation graphics adapter from XFX with NVIDIA technology [XFX].

surpassed the performance of contemporary CPUs, and the trend seems to continue. One of the main reasons for this is that while the CPU uses much of its chip for logic and cache, the transistors in the GPU's chip are almost dedicated to performing floating-point operations (FLOPs). It is important to notice that the GPU only outperforms the CPU with regards to parallel tasks. The CPU is still more effective on inherently serial tasks, e.g., text processing. It may seem strange that the GPU is not more utilized when it offers such massive computational power, but as we will see, this power comes with a cost. The cost is a relatively hard programming interface, poorly portable code, and only single-precision representation of floating-point numbers. However, this is a small price to pay when one considers that while Moore's law for CPUs describe a yearly growth rate of $\sim$1.4, the yearly growth rate for GPUs have been 1.7 (pixels/second) to 2.3 (vertices/second) [OLG+07].

There are many suppliers of graphics adapters, but the two biggest actors today are ATI and NVIDIA. A graphics adapter consists of the GPU, memory, a chipset, and a BIOS, today often packaged as a PCI-Express adapter. The chipset controls the graphics adapter and the BIOS provides software access to the hardware. You will find the same setup on a motherboard, with a chipset, memory (RAM), a BIOS, and a processor (CPU).

3.2 OpenGL and the graphics pipeline

To access and run programs on the GPU, we can use the Open Graphics Language (OpenGL) [SWND05] graphics API. OpenGL is traditionally used to *render* graphics, and in this thesis it will be our access point to the graphics adapter and the GPU. Rendering is the process of generating an image from

a model. OpenGL utilizes a *graphics pipeline* to perform rendering, and all graphics data passes through this pipeline. Before the graphics pipeline is outlined, there are some terms that need to be introduced and explained. I will briefly explain the terms, and try to connect them to their most likely "counterpart" in traditional CPU programming where this is possible.

State machine OpenGL is a state machine. This means that when a state is set, it will be used until changed or unset. The current color is one such state, so that when the color is set it will be applied to all consequent drawing, until it is changed again. Many of the state variables in OpenGL can be set and unset using the `glEnable(SOME_STATE)` and `glDisable(SOME_STATE)` functions. This functionality is possible due to the data-driven nature of OpenGL.

Vertices and fragments OpenGL draws geometric shapes like points, lines, polygons etc. These graphics *primitives* are defined by special points (e.g., the corners for polygons) called *vertices* (*vertex* in singular form). For each defined vertex, there is also possible to supply additional information, like the color of the vertex, the normal, the material, etc. After the primitives have gone through the *rasterization* part of the graphics pipeline, they are split up into *fragments*, which will eventually be what we see as *pixels* on the screen.

Textures Instead of defining the color of a vertex, you may connect a *texel* in a *texture* to the vertex. A texture can be thought of as a vector or a matrix in the graphics adapter's memory, usually containing an image. Texels are point values from the texture that are connected to the vertices. When the graphics primitives are *rasterized* later in the pipeline, all points in each primitive will get values from the attached texture, and the texture will appear as "glued onto" the primitive.

The relationship between vertices, texels, pixels and fragment is illustrated in Figure 3.2. This figure also demonstrates how the texturing and rasterizing process works.

The graphics pipeline The GPU is traditionally used for rendering graphics scenes, and it is part of a rendering pipeline as the one in Figure 3.3. In this figure we see the OpenGL graphics pipeline [Ros06] with the vertex and fragment processors clearly marked as 1 and 2, respectively. Most graphics APIs use a pipeline very similar to this. The vertex processor takes vertices and topology information as input, and then calculates lighting and color. Next is the primitive assembly stage, where the vertices are assembled into geometric primitives like triangles and quads. The primitives then get clipped and projected according to the modelview and projection

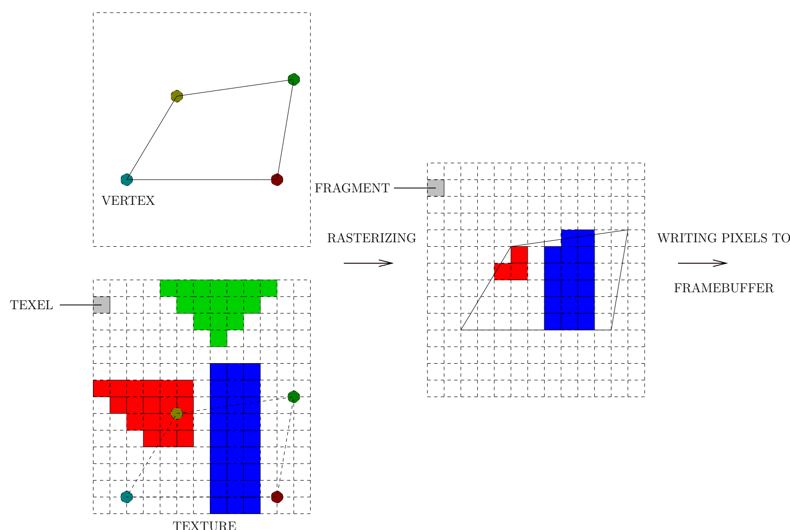

Figure 3.2: Texturing and rasterizing in the OpenGL graphics pipeline.

matrices and the current viewport settings. Then the rasterizer interpolates per-vertex values over the primitives and breaks them down to fragments with color values attached. The fragments are in turn sent to the fragment processor. The fragment processor executes per-fragment operations, like texturing. Last, fixed-functionality operations are executed on each fragment before it is written to the *framebuffer*, which is what we see on the screen during traditional graphics rendering. The computations performed on the vertex and the fragment processors in this thesis are described in more detail later.

OpenGL's graphics pipeline and the GPU are data-driven, in contrast to the instruction-driven CPU. This means that the GPU has a predetermined set of operations that will be performed on any specified input data. The CPU is not fed with data in this way, but executes a program and accesses memory as instructed by that program.

3.3 Shader programs

Programs that are executed on the GPU are called *shaders*. The vertex processor executes *vertex shaders*, and the fragment processor executes *fragment shaders*. This section contains a simple, but complete, example of how to program the graphics pipeline of your graphics adapter. In addition to these shader programs, it is necessary to have an OpenGL program that initializes the execution, uploads and compiles the shader programs, and defines input and output. The OpenGL program is executed on the CPU, an example can

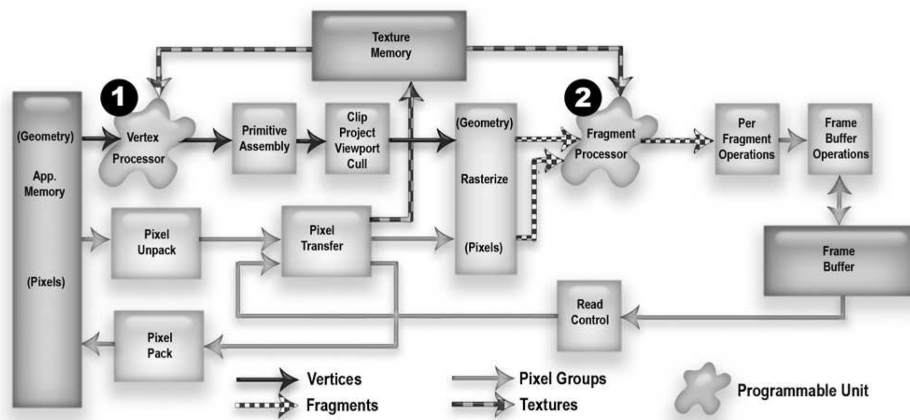

Figure 3.3: OpenGL graphics pipeline [Ros06]. Figure courtesy of R. Rost.

```
varying vec4 texcoord;

void main(void) {
    texcoord = gl_MultiTexCoord0;
    gl_Position = ftransform();
}
```

Listing 3.1: Vertex shader program

be seen in Listing 5.1 in Chapter 5. The shader programs are executed on multiple data simultaneously; this will be explained in detail later.

Vertex shader In the vertex shader you have access to the vertices specified in your main OpenGL application (the CPU application). These vertices are available through the built-in *attribute* input `gl_Vertex`. Attribute variables are per-vertex input to the vertex shader from the OpenGL program, and there are more of them than just the vertex attribute. Another variable named `gl_MultiTexCoord0` contains the appurtenant texture coordinates.

The vertex shader in Listing 3.1 does nothing more than one of the operations from the usual fixed-function pipeline computation; the function `ftransform` multiplies all vertices with the current modelview matrix (`gl_ModelViewProjectionMatrix`) to compute the homogeneous coordinates of each vertex in clip space. The results are stored in the special output variable `gl_Position`. The vertex shader *must* perform this computation. Another thing worth noticing is the initialization of the *varying* variable `texcoord`. Varying variables are interpolated and sent as read-only input to the fragment processor and can then be accessed by the fragment shader program. The attribute `gl_MultiTexCoord0` contains the texture coordin-

```
varying vec4 texcoord;
uniform sampler2D texCurrent;

void main(void) {
    vec4 tex = texture2D(texCurrent, texcoord.xy);
    gl_FragColor = tex;
}
```

Listing 3.2: Fragment shader program

ates associated with the vertex coordinates and is transfered to the fragment processor via `texcoord` in this shader.

Fragment shader The fragment shader has access to texture memory, varying variables passed from the vertex shader, and some built-in special variables.

The fragment shader in Listing 3.2 simply sets all fragments to the value of the input texture. The texture value is fetched with the `texture2D` function. `texCurrent` is a *uniform* that is passed from the OpenGL application that specifies which texture to read from, and `texcoord` specifies which texel to read. Uniforms are read-only variables that can be passed to both vertex and fragment shaders. `gl_FragColor` is a special output variable that sets the color of each fragment that later is written to the framebuffer.

In addition to the already mentioned input and output variables, it is also possible to specify constants in both vertex and fragment shaders. Constants are declared using the `const` keyword.

3.3.1 The GPU

GPUs are primarily used, and intended, for rendering and are equipped with parallel processing units. This is because graphics operations can be executed in parallel. The GPU logically consists of two types of processing units, the vertex processors and the fragment processors. The vertex processor computes the per-vertex operations and can be programmed. Vertex transformation, normal transformation and normalization, texture coordinate generation, texture coordinate transformation, lighting, and color material application are the operations performed in the fixed pipeline by the vertex processor. When a vertex shader is in use on the vertex processor, it can replace all these operations, if wanted. It is not possible to let these operations be performed by the fixed-functionality pipeline as long as a vertex shader is active. However, it is not required that your vertex shader performs all these operations. Vertex shaders can also perform many other per-vertex operations if you want.

The operations usually performed by the fragment processor are: Oper-

ations on interpolated values, texture access, texture application, fog effects, and color sum. While the vertex processor handles vertices as input, the fragment processor works on a stream of fragments. Fragments are pixels that have not yet been written to the framebuffer, but they may also contain more information than just the final pixel color, e.g., depth.

The intended use of the vertex and the fragment processor units is in rendering, but as we are going to see, they can also be used in more general purpose programming.

3.4 Graphics programming

The purpose of graphics programming is usually to present graphics on the screen in some form. There are many uses of computer graphics, in the range from entertainment to visualization of scientific data. Some of the main uses are covered in this section. This section is dedicated to non-GPGPU use of graphics hardware.

3.4.1 Game programming

The largest, single user of computer graphics today is the computer gaming industry. This industry drives the development of new graphics technology, both software and hardware. DirectX [Mic07] and OpenGL [SWND05] are the two most used graphics APIs. DirectX is developed by Microsoft. OpenGL was founded by SGI, and is currently being maintained by the OpenGL Architecture Review Board (ARB) [The07a]. ARB consists of several of the biggest actors in computer graphics and development, for instance 3DLabs, ATI, and Apple. In the fall of 2006, the OpenGL ARB became a part of the Khronos Group [The07b].

The graphics in the newest computer games is very computationally demanding, even though many techniques to lower the cost of certain graphics operations exist. Some of these techniques include specialized hardware on the graphics cards, and effective software algorithms. The demands from modern computer games have made the GPUs today's most powerful computational hardware relative to price [OLG+07].

3.4.2 Visualization

Visualization is the process of taking raw scientific data, e.g., values collected with sonar or MR, and processing them for visual presentation. The goal is to convey as much of the information from the raw data as possible, in an effective way. Since the short-term picture memory in the human brain is very brief, it is of great advantage if the visualization can be made interactive. This obviously requires heavy computations to be carried out fast, especially since the datasets often are big and also can be 3-dimensional.

Writing effective shader programs for the GPU is one way of coping with these demands.

3.4.3 Movie industry

The movie industry is a heavy consumer of computer graphics, for special effects in movies. Some movies contain more computer graphics than actual film, and can have all-virtual characters. The people responsible for creating characters and special effects using computer graphics are referred to as artists. They are often not that familiar with the technical aspects of computer graphics; instead they rely on programs like 3D Studio Max [Auta] and Maya [Autb] to render their drawings and films. 3D Studio Max and similar programs often generate shaders for use on the GPU. Pixar's RenderMan [Ups89] was one of the first offline renderers that utilized shader programs.

Most of the computer graphics in movies, however, is computed offline. Until recently it has been computed by the CPU and not the graphics hardware. The attitude towards using graphics hardware in rendering is changing, and as the movie industry starts to demand more powerful graphics hardware, this will further push the development.

3.5 GPU vs CPU

The GPU is architecturally quite different from the CPU. Today's GPUs have multiple pipelines (128 scalar processors on NVIDIA GeForce 8800 [NVIb]) and a huge memory bandwidth (103,7 GB/s on NVIDIA GeForce 8800 [NVIb]). GPUs are one of the most, if not *the* most, computationally effective processors currently available in the mass market, with over 384 million transistors built on a 90-nanometer fabrication process [OLG$^+$07]. This makes the GPU an extremely powerful parallel processing unit. The memory is also designed differently than on CPUs. The GPU allocates memory in 2D, while the CPU allocates memory in only one dimension.

Figure 3.4 shows a comparison of the floating-point performance of the GPU and the CPU from the period 2000-2008. As this figure shows, the GPUs have outperformed the CPU increasingly over these four years, and this development is likely to continue. Although GPUs outperform CPUs in many computational experiments like the one shown in Figure 3.4, the CPU will still in most cases beat the GPU when the program is of a certain size and complexity that demands a large instruction set from the processor to be executed efficiently. This is because the CPU has a relative large cache memory and a much broader set of instructions than the GPU. This is, however, not necessarily a bad thing since it means that the GPU's transistors are almost exclusively used for floating point operations. That is, GPUs will outperform CPUs in number of floating-point operations per second

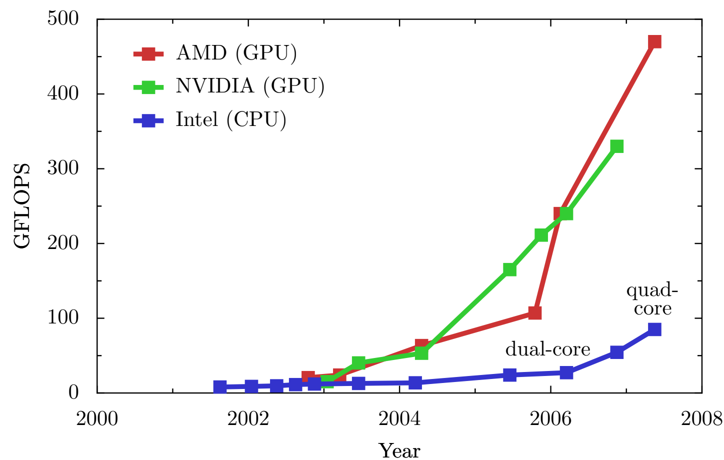

Figure 3.4: The programmable floating-point performance of GPUs (measuring one multiply-add instruction as 2 floating-point operations) compared to CPUs [OLG+07]. Figure courtesy of J. D. Owens.

(FLOPS), but the CPU will in most cases handle advanced operations like branching better than the GPU. Therefore, one should avoid constructions like if-tests, etc., when writing shaders for the GPU. The key to high efficiency in a GPU cluster comes from the use of CPU *and* the GPU, such that each processor type performs the subtasks for which it is most efficient.

3.6 GPGPU programming

Writing programs, or shaders, for GPUs is not a straight-forward process. In traditional CPU-programming you have a variety of different programming languages available to you, both high-level like Java and low-level like assembly. CPU-programmers also have a magnitude of highly developed compilers, profilers and debugging tools available. For GPU-programmers this is not the case. Until recently, GPU-programmers had to use assembly or (relatively) low-level shading languages made for graphics programming to implement their general purpose programs. Today there are several new languages available, including high-level languages made for GPGPU programming. Many of these tools and languages are described in Sections 3.6.4 and 3.6.5. Even with these new programming languages and tools, you are usually forced to think unconventionally when writing shaders for the GPU.

Graphics hardware vendors also make it difficult to write compilers and other tools for GPUs since the exact specifications of the hardware often are well-kept business secrets.

3.6.1 Parallel problems

Parallel problems are problems that can be divided into pieces which are possible to solve simultaneously and independently of each other, as introduced in Chapter 2. Parallel programs can use several processors in parallel to solve a single problem. A subset of this programming model is known as stream programming, and the GPU can be explained according to this model: The vertex- and the fragment shaders are known as *kernels* in stream programming terms. The vertex shader takes a *stream* of vertices as input, and the fragment shader a *stream* of fragments. The kernels process the data in the stream in an arbitrary order. There are several processors on the graphics adapter, and streams are processed in parallel.

3.6.2 The programming model

When a program is executed on the CPU, your code is executed one instruction at a time in a sequential order. The GPU on the other hand, executes in a parallel fashion.

In a scene there are almost always many more fragments than vertices, and because of this the fragment processor is more powerful than the vertex processor. Based on this, most GPGPU-programs use the fragment processor as the main computational engine. The newest generation of graphics adapters feature *unified shaders* [Bly06], where each processor is capable of executing both vertex- and fragment-shader programs. In this model, the graphics adapter will decide how many processors that will process vertices, and how many that will process fragments. However, fragment-shaders will still be preferred, because they are more suitable for GPGPU-programming. A typical GPGPU program is structured as follows [OLG+05]:

1. The application is segmented into independent parallel sections. Each of these sections can be considered a kernel, and is implemented as a fragment program, or possible as a vertex program. The input and output are data arrays stored as textures in the GPU memory. These data arrays can be considered as the streams. The kernels are then executed on each stream in parallel.

2. To invoke a kernel, the range (output to the framebuffer) of the computation must be specified by passing vertices to the GPU. Typically this is a quadrilateral oriented parallel to the image plane, sized to cover a rectangular region of pixels matching the desired size of the output array.

3. The rasterizer then generates a fragment for every pixel location in the input quad.

4. Each of the generated fragments is then processed by the active fragment program. The fragment program can read from arbitrary global memory locations (texture reads), but can only write to locations corresponding to the fragment in the framebuffer as determined by the rasterizer.

   The domain (input texture) for the computation is specified for each input texture (stream) by specifying texture coordinates at each of the input vertices, which are then interpolated at each generated fragment. Texture coordinates can be specified independently for each input texture, and can also be computed on the fly in the fragment program, allowing arbitrary memory addressing.

5. The output from the fragment program is a value or a vector of values per fragment. This may be your final output, or they may be stored in an offscreen *auxiliary buffer*, and used in additional computations. This is referred to as *multipass* computation. Multipassing allows for applications of arbitrary complexity.

3.6.3 Low-level languages

GPU programming started out with assembly-level programs. The introduction of programmable graphics hardware and an assembly language for the vertex- and the fragment processor made it possible to program for general purposes on GPUs. The assembly language was used to specify programs to run on each vertex or each fragment. Later, several new languages and tools have arrived. Some of them are described in the following section.

3.6.4 High-level languages

The OpenGL Shading Language (GLSL) [Ros06] is the most used language for programming shader programs on modern GPUs. The choice of language, depends on the type of application, the degree of control you need over GPU-operations, and which languages the programmer is accustomed to.

NVIDIA has developed a language for GPU-programming named C for graphics, or Cg [MGAK03]. Their goal was to make a high performance language similar to C that was easy to program, portable, and with complete support for hardware functionality. They have achieved the last point by introducing profiles. One profile fits one graphics processor, and ensures that you do not use any functionality not supported by the hardware by giving you access to only a subset of Cg. It should be mentioned, however, that Cg is not developed for GPGPU, but for graphics shader development.

High Level Shading Language (HLSL) [Mic05a] is Microsoft's shading language for their DirectX API, and a counterpart to GLSL. It has a syntax very similar to Cg. Since OpenGL is used in this master thesis, GLSL will be the shading language of choice.

In the past few years several new languages have been developed. I will briefly describe some of them here. Brook [BFH+04b] is a system for general-purpose computation on GPUs. It consists of a compiler and a runtime system. Brook is built on the stream-processing principals, defining a program as a kernel and the dataset as a stream. Brook is also the code-base for PeakStream [Pea].

Glift [LSK+06] is a collection of data structures for use in GPGPU-programming. The great advantage with using Glift is that you can separate data structures from logic, something that has been and still is a problem for GPGPU-programmers. When the actual logics of the shader program is separated from the data structure, it makes the program far much easier to modify and develop further. Each structure in Glift has a Cg and a C++-implementation, and hence does not introduce yet another language or tool.

Ashli [PB03] is a shading language interface which makes it easier for artists to access shader functionality more directly, and not just through shaders generated by different 3D-programs. Ashli takes code written or generated in RenderMan Shading Language, Maya Shading Network or 3DStudio Max Standard Materials as input and outputs DirectX 9.0 Vertex and Pixel Shader version 2.0 and OpenGL ARB_{vertex,fragment}_programs. It also outputs shader *formals* describing parameters.

Shallows [SIN05] is a GPGPU interface developed at SINTEF. It allows easy access to GPU-functionality through a C++-library, like writing to offscreen buffers and compiling and running shader programs.

3.6.5 Tools

There are a few development tools and debuggers for the GPU, and in this section I will mention some of the most useful ones.

gDEBugger [Gra05] and GLIntercept [Tre05] are tools for debugging OpenGL programs. Both are able to capture and log the OpenGL state from a program. With gDEBugger it is possible to set breakpoints and watch OpenGL state variables at runtime, but debugging of shaders is not supported. GLIntercept provides runtime shader editing, but debugging of shaders is not supported.

The Microsoft Shader Debugger [Mic05c] is integrated into Visual Studio IDE [Mic05b] and provides watches for runtime variables and breakpoints for shaders. The catch is that the shaders must be run in software emulation, and not on the actual hardware. In contrast, the Apple OpenGL Shader Builder [App05] can run shaders on hardware, but is designed for writing shaders for rendering and not for GPGPU. As a result, the shaders are not run in the context of the application, but in a separate environment designed to help shader writing. This prohibits debugging of advanced GPGPU-programs since you do not have the degree of control needed, e.g., over OpenGL-GLSL interaction.

The Image Debugger [Bax05] is a tool that provides support for shader visualization in the form of a `printf`-like function over a region of memory. The region of memory gets mapped to a display window and is visualized as an image. This is often a very useful form of debugging as you can see the results from a computation directly. The Image Debugger does not have any special support for shader programs, so the programmer must map the output from a shader to an output buffer for visualization.

The Shadesmith Fragment Program Debugger [PS03] provides `printf`-style debugging, and also basic shader debugging functionality like breakpoints and stepping. Shadesmith decomposes a fragment program into multiple programs, one for each instruction, and then adds an output instruction to each of these smaller programs. Shadesmith automates the `printf`-debugging by running the appropriate shader for the register that one wants to track, and then draws the output to an image window.

PyShallows is a small and simple tool written in Python and C++/OpenGL. It is an GUI-application with two text fields where the vertex shader program goes in one field and the fragment shader program in the other. It then visualizes the results of the shaders on some chosen OpenGL object, or prints out the compilation errors if the shaders fail to compile. It is a project of mine, and is currently under development. It is presented in Appendix C.

3.6.6 Problems

Since the graphics hardware is built for a specific use, it puts some restraints on GPGPU-programmers. For instance, there is currently no support for scatter operations i.e., assignments of type $A[i] = x$ for an arbitrary i; some graphics adapters have a limit on the number of instructions you can have in your shader program (not the newer cards); and reading and writing to the same texture memory simultaneously only works under very restricted conditions, since it is an undefined operation. Limited resources on the graphics adapter such as too little texture memory can also be a problem when running large simulations and other programs requiring extended use of memory. This can be solved by virtualization of hardware resources. Virtualization implies that the computation executed in several passes, also known as multipass. This implies global management of application data and hardware, since the program will have to generate different shader programs for each pass. Without this virtualization, a complex program using more than the available resources will abort at runtime.

Branching is supported by current GPUs. However, since current vertex processors and fragment processors are SIMD, the use of branching will slow down your program significantly. This is because a branch will force many processors to execute both sides of the branch, if both branches are taken by different fragments. GPUs are currently missing support for integers and bit operands, which include operations such as bit-shifts and bitwise logical

operations like AND, OR, XOR and NOT. This makes the GPU ill-suited for some computationally intense tasks such as cryptography, that involve heavy integer-calculations. GPUs now have 32-bit single-precision, but are missing 64-bit double-precision. NVIDIA have disclosed, however, that they plan to have support for double-precision in late 2007.

3.7 Conclusions

The demand for powerful computational engines are ever-increasing, especially by the need for large-scale simulations. Traditionally the CPU and supercomputers have been used for this, but now there exists an inexpensive alternative. Today, GPUs actually give you most computational power for the dollar [OLG+07]. Many simulations are of a parallel nature and thus well-suited for computation on GPUs.

Writing shader programs for the GPU, however, is not a trivial process. It involves dividing your problem in such a way that it can be solved in parallel, figuring out how to handle dataset input and output, and finally writing the shader or shaders in a language that can compiled for the GPU.

The biggest obstacle in GPGPU-programming is the unusual programming model as described before. But one cannot ignore the potential gain in utilizing the GPU for computation. The next generation of hardware will probably have increased generality and will certainly have even greater performance.

3.7.1 The future

Graphic adapters and GPUs are being developed at a high rate and new adapters are released sometimes as often as 3-4 times a year. The graphics hardware of today is already fast, but is also accelerating. This could mean that GPUs will become an important computational resource in near future, as it is already a fast-growing field of interest.

In Shader Model 4.0, a new stage has been added to the rendering pipeline, the geometry stage [Bly06]. This stage allows you to write shaders that manipulate assembled primitives. One of the most anticipated feature are probably the arrival of double-precision floating-point accuracy, but this will not be available for some time yet. In addition to further development on existing hardware and software, we also have some new additions:

- NVIDIA has released a new technology and SDK called Compute Unified Device Architecture (CUDA) [NVIa]. CUDA is made specifically for developing GPGPU-applications.

- IBM has released a new chip; the Cell Broadband Engine (Cell BE) [IBM], with 64-bit Power Architecture technology. The Cell BE is directed toward distributed processing. The chip consists of one or more Power

Processor Elements (PPEs) and multiple high-performance SIMD Synergistic Processor Elements (SPEs).

- AMD is also working on stream processing. They have founded The Torrenza Initiative [ATIb], which will be working towards finding a common "ecosystem" for hardware providers. Torrenza was conceived to enable a tight coupling of accelerator coprocessors and other various option cards to AMD64 technology-based systems.
- NVIDIA has released a new platform for high-performance computing named Tesla [NVIc]. Tesla is a dedicated, high-performance GPU computing solution.

3.7.2 Further reading and resources

`http://www.gpgpu.org` is a fine site for news, articles and papers in the field of GPGPU programming.

`http://www.opengl.org` is home to the OpenGL graphics API. Here you will find everything related to OpenGL.

`http://www.microsoft.com/directx` is home to Microsoft's DirectX graphics API.

`http://www.nvidia.com` and `http://www.ati.com` are the two largest manufacturers of graphics adapters today.

`A Survey of General-Purpose Computation on Graphics Hardware`, the State of the Art report on GPGPU from Eurographics 2005 [OLG+05].

`A Survey of General-Purpose Computation on Graphics Hardware`, the State of the Art report on GPGPU from Computer Graphics Forum 2007 [OLG+07].

Chapter 4

Parallel programming using the GPU

““The time has come,” the Walrus said,
“to talk of many things.””
— L. Carrol

Chapters 2 and 3 introduced two concepts; the parallel programming paradigm and general-purpose computation on GPUs (GPGPU). In this chapter we will try to connect these two, and use both GPGPU and parallel programming together in the same application. In reality, this gives us two layers of parallelism; the first one in the MPI application, and then a second layer is added when each node uses the GPU for further execution of the application. The main topics of this chapter will be how to best combine these two layers to achieve the most efficient and clean application, and how to best test and profile such an application. This is a complex and new research-area, with little or no existing material to build on. There is a great deal of published material on parallel programming, and also quite a lot on GPGPU programming, but almost nothing on combining these two. However, Fan et al. at Stony Brook University have written “GPU Cluster for High Performance Computing” [FQKYS04], where they use a GPU cluster to do flow simulation using the lattice Boltzmann model (LBM). Figure 4.1 shows the GPU cluster used for this thesis, and gives an overview of what a GPU cluster consists of from a hardware perspective.

4.1 Models

The top layer of the parallel GPGPU-programming model is standard parallel programming, using for example MPI, and the second layer is the graphics

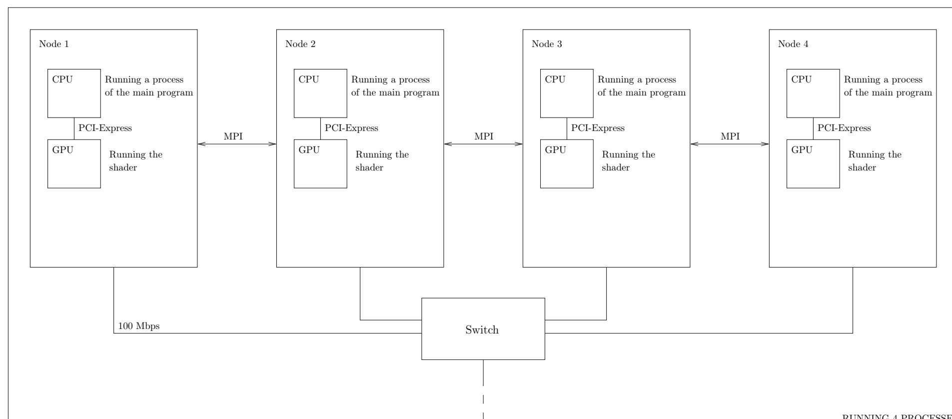

Figure 4.1: Overview of a GPU cluster.

programming model of the GPU. A very likely scenario is that MPI is used for domain decomposition and distribution, the GPUs solve some problem over the sub-domains, and MPI is then used to collect data or to recompose the original domain again. It is important to minimize communication between processes, as this is a costly task. This can be achieved by running multiple time-steps on each node before sending border values to neighbours, as detailed in Section 4.2.

Another consideration is how you handle the simulation data from each node. Do you collect data from each node after each time-step, or for each 20th time-step, or maybe not before the simulation is completed? Storage of simulation data is also important. When simulating over a very large grid with perhaps ten quantities to save for each time step, large amounts of data are generated. When and how these data are saved becomes important. They can be kept in memory until the end of the simulation, or they can be written to file on the fly, either on the local node or on a master node. How this is handled depends mostly on the available hardware. A cluster with little RAM needs to write data to disk relatively often to avoid running out of memory, and a cluster with a slow network is better off storing data on each node's harddrive instead of using the slow network connection to send all data to a master node. One needs to consider what is practical against what is effective. If simulation data are saved on each node during the execution of the simulation, it is necessary to have another application that makes use of the data. This application can collect all the data and write them to a single file, or visualize them directly. This all depends on how the data are intended used. Maybe it is only the final state that is interesting. In that case, it would be a simple task to collect the last time-step from each node and put them together to form the global domain again.

A more or less generic model of parallel applications that solves some

time-dependent equation or equations over some domain utilizing GPUs on each node, can be as follows:

1. Initialize MPI.
2. Set up an OpenGL context on each node.
3. Decompose the problem and distribute description of local domains to each node (e.g., spatial and temporal grid sizes, number of time steps, domain dimensions etc.).
4. Upload and compile shader program(s) to the GPU, set output render texture(s) and framebuffer(s), and set uniforms if there are any.
5. Make and upload textures containing initial conditions.
6. Run shader(s) to solve the local problem on each node.
7. Read back results from render texture(s) as needed.
8. Communicate with other processes and nodes as needed.
9. Save simulation data to file or memory as needed throughout the execution.
10. Collect data to main node, close all open files, free allocated memory, and clean up.
11. Finalize MPI.

This is also a slightly abstracted flow-chart of the application written for this thesis.

4.2 Ghost Cell Expansion

Ghost Cell Expansion [DH01] (GCE) is a method used to minimize communication in a parallel system solving PDEs. The idea is to add an extra number of ghost cells e in addition to the ghost cells L needed by the PDE scheme. This allows your PDE solver to perform e time steps on each sub-domain before it is necessary to exchange data with neighbouring subdomains. It is important to notice that the reason why we can use GCE is the fact that the equations used in this thesis have a finite speed of propagation, and the information is displaced by a known number of spatial grid cells per time-step. Recall equation (1.1) from Chapter 1, and assume that $I = O = M = N = 1$, i.e.:

$$u_{i,j}^{n+1} = F(u_{i-1,j}^n, ..., u_{i+1,j}^n, u_{i,j-1}^n, ..., u_{i,j+1}^n)$$

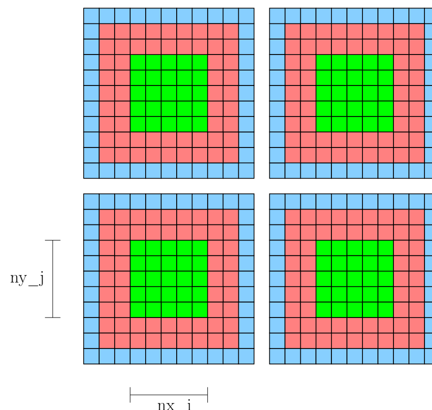

Figure 4.2: Four sub-domains (green) with expanded ghost cells (red) and regular PDE scheme ghost cells (blue).

Such a scheme requires only one ghost cell. One example of such a scheme is the standards finite-difference method for the heat equation $u_t = u_{xx} + u_{yy}$.

In Figure 4.2 the computational scheme requires one ghost cell, and there are added two additional ghost cells, so we have that $e = 2$ and $L = 1$. This allows for three time steps to be performed before transferring data between sub-domains. If the original sub-domain has size N, the total size of the domain in 1D then becomes $N + 2 \cdot (L + e)$. In this thesis, GCE will have an additional positive effect: By making each process able to perform several time-steps without inter-node communication, we also reduces readback from the GPU, since this only occurs in connection with exchange of ghost cells.

Figure 4.3(a) and 4.3(b) shows how data are transfered between neighbouring sub-domains in the y- and x-direction, respectively. This is performed every $\text{mod}(n, e + L)$ time steps, where n is the current time-step. The sub-domains that have one or two global boundaries are treated in the same way as the internal sub-domains, with the obvious exception that the global boundaries are left untouched. When using MPI's `MPI_Cart_shift` to identify neighbouring sub-domains, no extra programming is required to achieve this. One important thing to notice here is that the corner areas of each grid is implicitly transfered correctly. This is a result of always doing the vertical exchange before the horizontal exchange. Doing the horizontal exchange before the vertical exchange will also work, as long as its done consistently throughout the simulation. This method was first presented by Ding and Ye [DH01], and it saves four explicit ghost cell exchanges in a two-dimensional Cartesian grid. Furthermore, if the domain were to be extended to three spatial dimensions, it would require as much as 26 different ghost-cell transfers between neighbouring sub-domains. This makes the savings due to implicit corner transfers even more attractive.

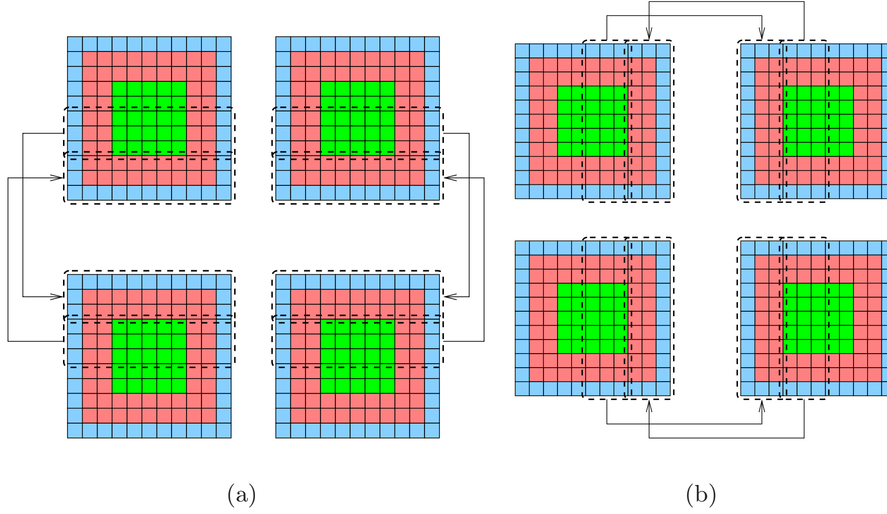

Figure 4.3: Exchange of ghostcell data between neighbouring sub-domains located on different nodes: (a) Ghost cell exchange in vertical directions. (b) Ghost cell exchange in horizontal directions.

Rojas and Hoemmen [RH04] have measured communication savings of utilizing GCE for domain decompositions of finite-difference grids. They concluded that there are considerable savings in one and two spatial dimensions, although it was hard to extrapolate a clear trend from the results, which varied from platform to platform. The specific savings obtained in this thesis can be reviewed in Chapter 6.

4.3 Parallel algorithm – Communication time

It is vital to minimize communication, especially if your cluster is connected through a slow network, e.g., Ethernet. We will examine some communication models related to the application written for this thesis.

The total message volume for a two-dimensional decomposition for each update of the ghost cells per sub-domain is:

$$v_{\text{old}} = (nx + 2L) \cdot (ny + 2L) - nx \cdot ny = 2L \cdot (nx + ny + 2L),$$

while the message volume for each time-step using the GCE method is:

$$v_{\text{new}} = (2L + 2e) \cdot (nx + ny + 2L + 2e)/(e + L)$$

We divide by $e + L$ since ghost cells only are updated every $e + L$ time-

step. If we compare these volumes, we get:

$$\frac{v_{\text{new}}}{v_{\text{old}}} = \frac{(2L+2e)\cdot(nx+ny+2L+2e)}{2L\cdot(nx+ny+2L)\cdot(e+L)} \simeq \frac{L+e}{L(e+L)}$$

This is true because in most cases we will have $nx + ny \gg L + e$. The total communication volume *per* exchange will of course increase when using this method.

In addition to the message volume, we also need to investigate the communication time for updating the ghost cells in each time-step, t_{comm}. A simple, generic communication time model may look like this:

$$t_{\text{comm}}(n) = T + \frac{v}{B},$$

where t_{comm} is the total time used for communication, v is the message volume, and T is the startup overhead of setting up the communication (opening a socket etc.), i.e., the *latency*. B is the bandwidth of the network. Without GCE this gives us the following communication time for our problem:

$$t_{\text{old}} = \frac{2L\cdot(nx+ny+2L)}{B} + 4T.$$

With the new method, we remember that ghost cell updates only occurs every $e + 1$ time-step, and that gives us this model:

$$t_{\text{new}} = \left(\frac{(2L+2e)\cdot(nx+ny+2L+2e)}{B} + 4T\right)/(e+L).$$

Suppose we have a global domain of size 1024×1024, a scheme where $L = 1$, and that we are using 50 ghost cells. If we ignore the latency this gives us the following (scaled) numbers $t_{\text{old}} = 41$ and $t_{\text{new}} = 22$ on a fast Ethernet (100 Mbit) network. When we at the same time realize that the network latency cost of t_{new} is 51 times smaller than that of t_{old}, we would expect significant savings. The speedups recorded by using GCE for this thesis can be seen in Section 6.2 in Chapter 6.

4.4 Upload and readback of textures

One of the possible bottlenecks issued in this thesis is the readback from the GPU. Since the traditional use of the GPU, namely for gaming, does not require fast readback, it has not been a priority for the card manufacturers. In fact, most computer games do not use readback at all, and this has resulted in a very slow readback on GPUs [Göd06]. Uploading of textures on the other hand, is much faster, since this is done in most computer games. This means that not only is it important to minimize communication between nodes, but also between the GPU and the CPU, and then

especially readback. One possible extension that could hide some of the cost of uploading and reading back textures and network communication, is asynchronous upload and readback. This is possible since a large portion of the computational domain can be evolved without using the boundary and ghost cell values, allowing for uploading and readback while computing a reduced computational domain, and then computing the remaining grid cells after the ghost cell exchange has been completed.

4.5 Profiling

The type of application written in this thesis is somewhat difficult to debug and profile. The combination of parallel execution and multiple programming languages is the reasons for this. For debugging C++ code, the Visual Studio 8 debugger has been used. The parallel parts of the code have also been debugged in Visual Studio by running the program as a single process. Trial and error has also been necessary to debug communication errors between multiple processes. The shader programs have been debugged by printing out compile errors using the Shallows [SIN05] library, and manually by trial and error. Profiling of the parallel execution has been done with the Jumpshot application described in Chapter 2. A screen shot of Jumpshot in action can be seen in Figure 4.4. Each bar in the figure represents the timeline of one process. By examining the states of the different processes throughout the execution one can identify possible idle periods in a process or a group of processes. This makes it much easier to find bottlenecks in the parallel algorithm.

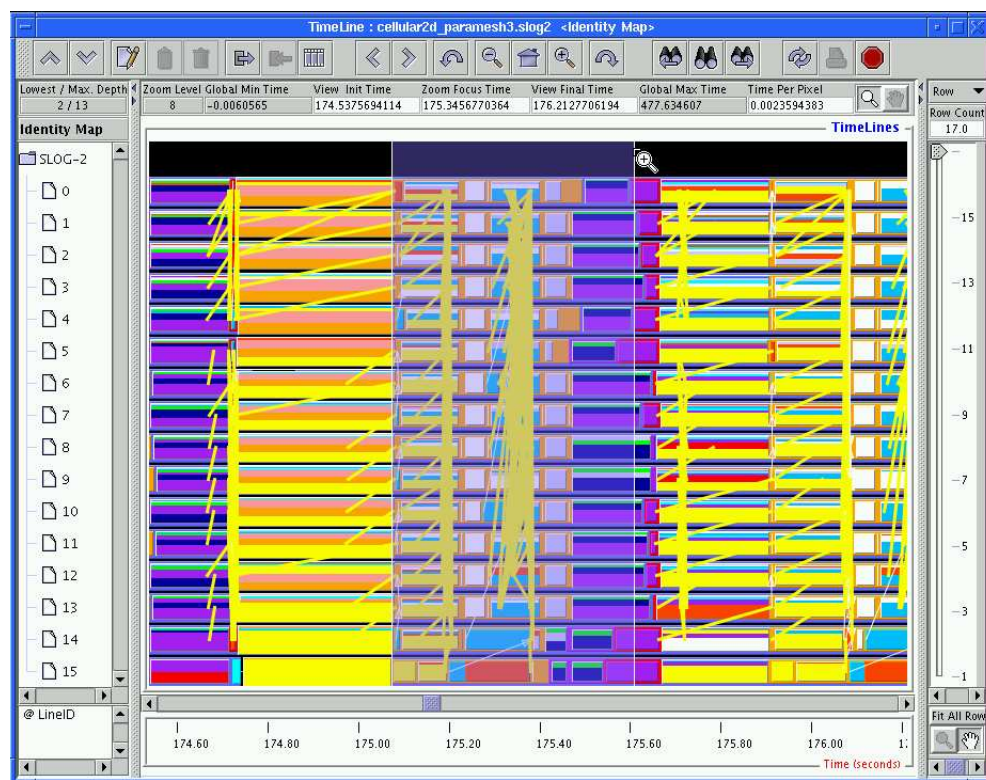

Figure 4.4: Jumpshot profiling tool.

Chapter 5

Using the GPU to solve systems of hyperbolic PDEs

"*It is impossible to exaggerate the extent to which modern applied mathematics has been shaped and fueled by the general availability of fast computers with large memories. Their impact on mathematics, both applied and pure, is comparable to the role of the telescopes in astronomy and microscopes in biology.*"

— Peter Lax[a], Siam Rev. Vol. 31 No. 4

[a]Peter Lax was awarded the Abel Prize in 2005.

This chapter describes how the equations presented in Section 1.3 in Chapter 1 are simulated on a GPU cluster. The chapter is divided into two parts: The first part contains general information about how to build a GPU cluster. The last part discusses the software and the algorithms used to solve the three hyperbolic PDEs.

5.1 Hardware

We previously discussed that there are many types of parallel systems. Some classifications are based on hardware, some on network, and others on software. While some parallel technologies are closely related to the underlying hardware, the MPI package does not have this relation. A cluster that will run MPI-applications can be assembled using ordinary desktop computers, and connected through a regular Ethernet network. The hardware on each node will, of course, have a direct influence on how the cluster performs as a whole. There are, however, no special specifications needed to allow execution of parallel programs. Each node needs the standard components; a CPU, a motherboard, memory, a hard drive, and a network adapter. You

Table 5.1: Price table for a gigabit node with and without graphics adapter.

Component	Description	Price
CPU	Intel Pentium 4 641 3.2GHz Socket LGA775, 2MB, BOXED w/fan	750 NOK
Memory	Crucial DDR2 PC4200 2048MB CL4 Kit w/two matched DDR2 PC4200 1024MB CL4	1095 NOK
Motherboard	ECS P965T-A, P965, Socket-775, SATAII, GbLAN, DDR2, ATX, 2x PCI-Ex16	729 NOK
Hard drive	Samsung SpinPoint T166 500GB SATA2 16MB 7200RPM	1150 NOK
Network adapter	Built-in gigabit LAN on motherboard	–
Graphics adapter	XFX GeForce 7900GS 525M 256MB GDDR3, XT-X, PCI-Express, 525/1550 Mhz, 2x DVI	1699 NOK
Sum Sum w/o gfx adapter	– –	5423 NOK 3724 NOK

Table 5.2: Price table for gigabit network components for 16-nodes cluster.

Component	Description	Price
Switch GB	SMC EZ Switch GS16 16P gigabit	1895 NOK
Cables	Patch cable UTP CAT 5E Grey 10 m RJ-45/RJ-45, AWG24	109 NOK $\times$ 16 = 1744 NOK
Sum	for a 16-nodes cluster	3639 NOK

may also need some sort of removable storage drive, like a CD-ROM or USB-pen for installation purposes. Table 5.1 shows a typical node, with prices collected from a Norwegian web shop [Kom]. In large quantities, each node would probably cost even less than this. This is *not* the configuration of the GPU-nodes used for benchmarking in this thesis, but only an example chosen with regards to price and efficiency to prove the attractive FLOPS/-dollar ratio of a GPU cluster. In addition to the nodes, the other essential part of the cluster is the network that binds the nodes together. Some prices and specifications are shown in Table 5.2. Altogether, this gives a total price of 90407 NOK for a 16-node gigabit cluster. Furthermore, we will examine what components are important for good performance of our application, and the reasons why.

The most important factor is the network, since this is almost always

the limiting factor, and because fast communication between nodes are vital. The second most important components for performance are the CPU and the graphics adapter, since these parts will be performing the computations. RAM and harddrive are important for storing simulation data. Different methods for storing simulation data were discussed in Section 2.5.2 in Chapter 2, and different hardware configurations should be carefully chosen with respect to the implementation. For instance, if data are to be stored on one single node, this node will need a very large harddrive and probably also a large amount of RAM. If the data are to be saved on each node, every node will need a relatively large harddrive.

One major advantage with GPU clusters is that they are very inexpensive compared to traditional clusters with regards to FLOPS per dollar. The node in Table 5.1 has a theoretical capacity of 24.6 GFLOPS [Gee05], but with the added FLOPS capacity of the graphics adapter (144 GLOPS [Com]) we get over 500% increase in efficiency spending only 1699 NOK extra (which is less than 50% increase in price). Harvesting these extra FLOPS to the full extent is, however, not a straightforward task, as this thesis demonstrates. It should also be noted that these numbers are theoretical, and as we will see there are many other factors that will have an impact on the performance of the GPU cluster. However, they are very encouraging for further research into possible uses of the GPU. The costs per FLOP for the CPU- and the GPU-version of the cluster presented here are 160.5 NOK and 33.5 NOK, respectively.

5.2 Application

In this section we will examine the application that has been written for this master thesis. It is written in C++ using MPICH2 for message passing between processes and nodes. GLSL is used in the shader programs. The Shallows library [SIN05] is also used for setting up a GPGPU-programming environment. In addition to standard C++, the STL library and the Boost C++ libraries [Boo] have also been used. All code is compiled with the Microsoft VS 8 C++ compiler, `cl.exe`. The other software and languages used, have already been described in Chapters 2 and 3. This includes MPI-2, OpenGL, and GLSL.

Boost Boost is a large C++ library that has a broad variety of uses. In this application, shared pointers is the only class used. Shared pointers feature atomic reference counting and automatic deletion of the object when no more `shared_ptr`s are pointing to it.

5.2.1 Domain decomposition

The application in this thesis allows for an arbitrary decomposition of a random rectangular grid in two spatial dimensions. To achieve this, the `MPI_Cart`-class of MPI-functions is used:

1. `MPI_Dims_create` calculates the dimensions of the process grid based on the number of processes available, and what spatial dimension we want the grid to have.

2. `MPI_Cart_create` creates a new communicator with the nodes distributed according to the dimensions calculated in Step 1.

3. `MPI_Cart_map` makes each process aware of its own *rank* in the new communicator.

4. `MPI_Cart_coords` fetches the coordinates of the process in the new communicator, based on the position of the process in the two-dimensional grid of processes.

In this thesis the initial conditions of each sub-domain are set using simple if-tests on the process rank, after the logical distribution of the global domain. If the simulation is based on some measured initial condition data, one must also transfer those data to the appropriate sub-domains after the global domain has been decomposed. Figure 1.2 in Chapter 1 shows an example of a two-dimensional domain decomposition. The sub-domains do not need to be squares, but they will always be rectangular when using the algorithm described above.

5.2.2 Domain recomposition

In the application written for this thesis the sub-domains are not recomposed into the original domain before the simulation results are to be visualized. The visualization application reads each sub-domain from file and places it correctly in the global domain, and in the correct time-step. The visualization application is covered in Section 6.1.2 in Chapter 6. Figure 5.1 illustrates the domain recomposition process.

5.2.3 Shaders

The background material on shader programs can be found in Chapter 3, and this section will cover implementation specific details. A description of the shaders used to solve the linear wave equation is included in Section 5.3. The shaders for the shallow-water equations and the Euler equations will not be described in detail; however, the two shaders used to simulate the shallow-water equations are included in Appendix B. All shaders used for the Euler equations are written at SINTEF by Hagen et al. [HHHL07].

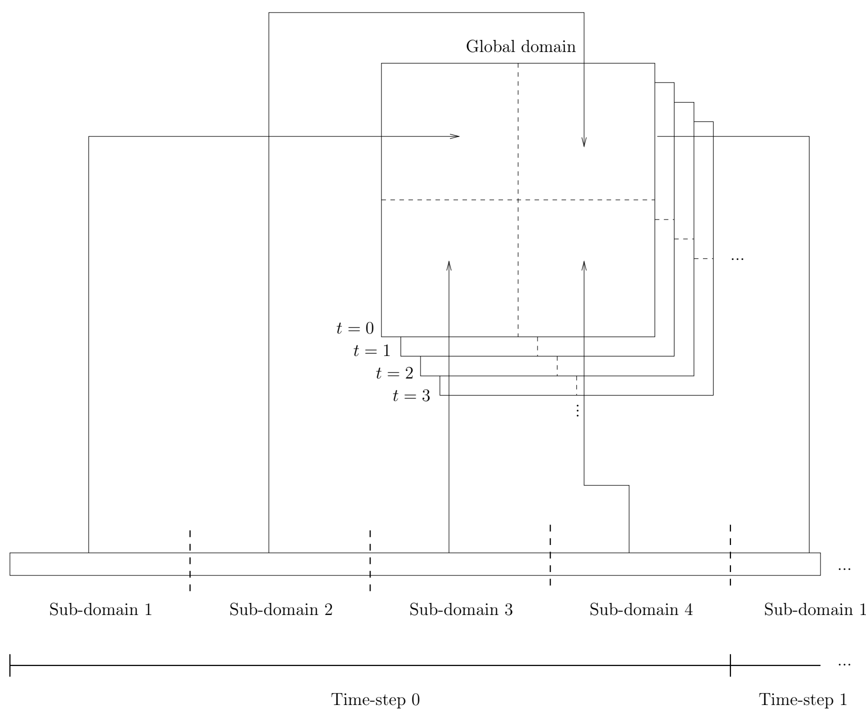

Figure 5.1: Domain recomposition from saved simulation data. The simulation data are saved in four files, one file for each of the four channels, R, G, B and A. Each file, and each channel, contains one quantity. All files are recomposed like this figure illustrates, before the visualization process can start. The time-steps of the simulation is organized in ascending order in each file, and within each time-step we have all the sub-domains of the global domain, also organized in ascending order. This recomposition is similar for each file.

```
// declare variables
boost::shared_ptr<shallows::OffScreenBuffer> fb;
boost::shared_ptr<shallows::GLProgram> shader;
boost::shared_ptr<shallows::RenderTexture2D> rt;

// initialize variables
int nx=10, ny=10;
shallows::init_shallows();
fb.reset(new OffScreenBuffer(nx, ny));
rt=fb->createRenderTexture2D();
shader.reset( new GLProgram );

// load shader from file and set input/output
shader->useNormalizedTexCoords();
shader->readFile("C:/shaders/example.shader");
shader->setFrameBuffer(fb);
shader->setParam2f("dXY", 1.0/nx, 1.0/ny);
shader->setInputTexture("someLabelInShader", someTexture);
shader->setOutputTarget( 0, rt );

// run the shader
shader->run();
```

Listing 5.1: A Shallows example

One important thing to notice is that the three different equations have different number of unknowns; the linear wave equation has one, the shallow-water equations three, and the Euler equations four. Since the GPU operates on vectors of length four, one would expect a much higher speedup for the Euler equations, than for the linear wave equation. However, it is possible to pack several unknowns of the same variable into one fragment, and this is discussed in Section 7.1 in Chapter 7.

Shallows has been used in my application as a layer on top of OpenGL. Some places it was easier to use OpenGL and GLSL directly, to get the degree of control I wanted, but in most cases Shallows proved to be a fast and easy way of performing GPGPU tasks. It was primarily used for easy texture- and shader handling throughout my application.

To compile, load, and run a shader with Shallows, it is sufficient to execute the code in Listing 5.1. For this example to work, it is necessary to include the Shallows classes used, and the Boost library. The example code will read its input from `someTexture` and write the results to `rt`.

5.3 The linear test-case

We will now present and examine the test-cases considered, starting with the linear wave equation. In two spatial dimensions the linear wave equation

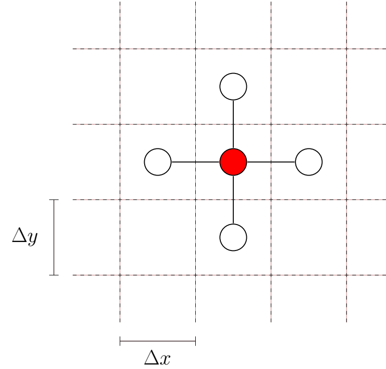

Figure 5.2: The computational molecule for the linear wave equation in 2D.

can be discretized as:

$$u_{i,j}^{n+1} = \Delta t^2 \left(\frac{u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n}{\Delta x^2} + \frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{\Delta y^2} \right) + 2u_{i,j}^n - u_{i,j}^{n-1}, \quad (5.1)$$

where $u_{i,j}^{n+1} \approx u(i\Delta x, j\Delta y, n\Delta t)$.

Equation (5.1) is derived from a higher-order (second-order) approximation of the derivatives, either by using the Taylor series (5.2) directly, or by combining a forward (5.3) and a backward (5.4) approximation.

$$f(x \pm h) \approx f(x) \pm hf'(x) + \frac{h^2}{2} f''(x) \pm \frac{h^3}{6} f^{(3)}(x) + \mathcal{O}(h^4) \quad (5.2)$$

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad (5.3)$$

$$f'(x) \approx \frac{f(x) - f(x-h)}{h} \quad (5.4)$$

The computational molecule for the linear wave equation in two spatial dimensions is illustrated in Figure 5.2. The white-colored values are taken from the current time step, $u_{i,j}^n$, while the red value is from both the current and the previous time step, $u_{i,j}^{n-1}$.

Listing 5.2 shows the full GLSL source code of the shader used to solve the linear wave equation. Line 1 and 21 shows how to tell Shallows which is the vertex shader and which is the fragment shader. `texcoord` holds the texture coordinates, which is interpolated over the rendering area. `texCurrent` is the texture containing the current time-step and `texLast` holds the previous time-step. `dXY` is a vector of size two which holds the spatial resolution

```
 1 [Fragment shader]

   varying vec4 texcoord;

 5 uniform sampler2D texCurrent;
   uniform sampler2D texLast;
   uniform vec2 dXY;

   void main(void) {
10   vec4 texC = texture2D(texCurrent, texcoord.xy);
     vec4 texE = texture2D(texCurrent, texcoord.xy + vec2(dXY.x, 0.0));
     vec4 texW = texture2D(texCurrent, texcoord.xy - vec2(dXY.x, 0.0));
     vec4 texN = texture2D(texCurrent, texcoord.xy + vec2(0.0, dXY.y));
     vec4 texS = texture2D(texCurrent, texcoord.xy - vec2(0.0, dXY.y));
15   vec4 texL = texture2D(texLast, texcoord.xy);

     gl_FragColor = (0.5 * ((tex0-2.0*tex+tex1 + tex2-2.0*tex+tex3))
             + 2.0*tex - texL);
   }
20
   [Vertex shader]

   varying vec4 texcoord;

25 void main(void) {
     texcoord = gl_MultiTexCoord0;
     gl_Position = ftransform();
   }
```

Listing 5.2: The linear wave equation solver.

in each dimension. Lines 10-15 extracts the vectors holding the actual values from the textures, using the texture coordinates according to the computational molecule (Figure 5.2) of the scheme. The values are then used in the computational scheme and written to `gl_FragColor` as vectors of size four again, which is rendered to the chosen output texture. Remember that in the case of the linear wave equation we only use one value in each four-vector. It does not matter which channel we use (R, G, B or A), as long as it is done consistently, all channels are computed anyway. Input textures and parameters, and output rendertexture are set like in Listing 5.1.

Boundary-conditions For the linear wave equation reflective boundaries are used:

$$\nabla u \cdot n = 0 \tag{5.5}$$

They are implemented in a very simple manner, by drawing a frame of ghost cells around the computational domain and shifting the texture coordinates one pixel in. This is illustrated in Figure 5.3.

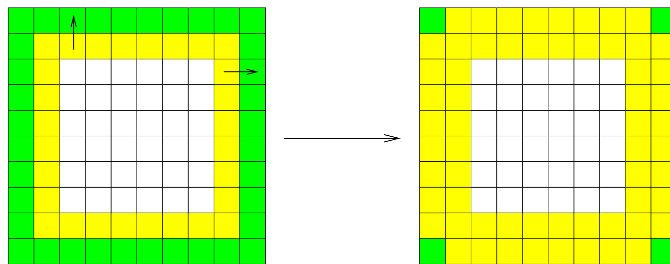

Figure 5.3: Boundary conditions for the linear wave equation in 2D. The yellow frame is copied one pixel outwards.

The linear wave equation is stable under the following CFL-condition:

$$\frac{\Delta t}{h} < \frac{1}{2}, \tag{5.6}$$

where $h = \Delta x = \Delta y$. This means that the length of each time-step is limited by the spatial resolution.

5.4 Finite-volume schemes

Classical high-order schemes tend to generate unwanted oscillations that results in errors near discontinuities. Low-order schemes have another problem, they introduce numerical diffusion, and the solutions become inaccurate. Two such classical schemes are the diffusive first-order Lax-Friedrich and the oscillatory second-order Lax-Wendroff scheme. One way of controlling the oscillations introduced by the Lax-Wendroff scheme is to combine it with the Lax-Friedrich scheme in a *composite* scheme. Such a scheme is used to solve the shallow-water equations. This is the simplest possible high-resolution scheme, and consists of e.g., three steps Lax-Wendroff, followed by one Lax-Friedrich step. Since the second-order Lax-Wendroff scheme introduces spurious oscillation, we use one first-order Lax-Friedrich step to smoothen the solution. This is possible since numerical dissipation occurs in the Lax-Friedrich scheme.

In the following derivations the paper "How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational engine" [HHHL07] is used as a guide. It should be noted that this is a somewhat unusual derivation of these schemes, that in the mentioned article was motivated by the need to derive similar high-resolution schemes used for the Euler equations. Both Lax-Friedrich and Lax-Wendroff use finite-volumes as basic building blocks:

$$Q_i(t) = \frac{1}{|\Omega_i|} \iint_{\Omega_i} Q(x, y, t) dx dy. \tag{5.7}$$

This is a approximation to the cell-average of Q within the cell Ω_i. What we want to find now is a equation for the evolution of these cell-averages over time. If we use the conservation law on integral form, (1.3), with the grid cell definition $\Omega_i = [x_{i-\frac{1}{2}}...x_{i+\frac{1}{2}}, y_{j-\frac{1}{2}}...y_{j+\frac{1}{2}}]$, we get:

$$\frac{d}{dt}\int_{x_{i-1/2}}^{x_{i+1/2}}\int_{y_{j-1/2}}^{y_{j+1/2}} Q_{i,j}(x,y,t)dxdy = \int_{y_{j-1/2}}^{y_{j+1/2}} (F(Q(x_{i-1/2},y,t)) - F(Q(x_{i+1/2},y,t)))dy + \int_{x_{i-1/2}}^{x_{i+1/2}} (G(Q(x,y_{j-1/2},t)) - G(Q(x,y_{j+1/2},t)))dx. \quad (5.8)$$

We then obtain the following semi-discrete ODE from (5.8):

$$\frac{d}{dt}Q_{i,j}(t) = -\frac{1}{\Delta x}\left(F_{i+1/2,j}(t) - F_{i-1/2,j}(t)\right) - \frac{1}{\Delta y}\left(G_{i,j+1/2}(t) - G_{i,j-1/2}(t)\right), \quad (5.9)$$

where the numerical approximation to the fluxes are given by

$$F_{i\pm1/2,j} \approx \frac{1}{\Delta y}\int_{y_{j-1/2}}^{y_{j+1/2}} F(Q(x_{i\pm1/2},y,t))dy, \quad (5.10)$$

$$G_{i,j\pm1/2} \approx \frac{1}{\Delta x}\int_{x_{i-1/2}}^{x_{i+1/2}} G(Q(x,y_{j\pm1/2},t))dx. \quad (5.11)$$

To derive a numerical scheme from this, we need to discretize (5.9) and evaluate the fluxes (5.10) and (5.11) along the cell boundaries. We use the midpoint rule to approximate the integrals in (5.10) and (5.11). The midpoints of the edges are estimated by averaging the one-sided values in each direction:

$$F(Q(x_{i+1/2,j},y_j,n\Delta t)) = \frac{1}{2}\left(F(Q^n_{i,j}) + F(Q^n_{i+1,j})\right), \quad (5.12)$$

etc. A similar approximation is used in the y-direction.

Using the forward Euler method on (5.9), and the approximated fluxes,

we get the following scheme:

$$Q_{i,j}^{n+1} = Q_{i,j}^{n} - \frac{1}{2}\frac{\Delta t}{\Delta x}\left(F(Q_{i+1,j}^{n}) - F(Q_{i-1,j}^{n})\right) - \frac{1}{2}\frac{\Delta t}{\Delta y}\left(G(Q_{i,j+1}^{n}) - G(Q_{i,j-1}^{n})\right). \quad (5.13)$$

This scheme, however, is notoriously unstable. By adding artificial diffusion $\beta(Q_{xx} + Q_{yy})$ (where β is the diffusion coefficient) and discretising it by a central difference, we obtain a more stable scheme; the first-order Lax-Friedrichs scheme:

$$Q_{i,j}^{n+1} = \frac{1}{4}\left(Q_{i+1,j}^{n} + Q_{i-1,j}^{n} + Q_{i,j+1}^{n} + Q_{i,j-1}^{n}\right) - \frac{1}{2}\frac{\Delta t}{\Delta x}\left(F(Q_{i+1,j}^{n}) - F(Q_{i-1,j}^{n})\right) - \frac{1}{2}\frac{\Delta t}{\Delta y}\left(G(Q_{i,j+1}^{n}) - G(Q_{i,j-1}^{n})\right), \quad (5.14)$$

where $Q_{i,j}^{n} = Q(x, y, n\Delta t) = Q_{i,j}(n\Delta t)$ is piecewise constant inside each grid cell.

If we use the midpoint rule for integrating (5.9) instead of the forward Euler method, we get the second-order Lax-Wendroff scheme:

$$Q_{i,j}^{n+1} = Q_{i,j}^{n} - \frac{\Delta t}{\Delta x}\left(F(Q_{i+1/2,j}^{n+1/2}) - F(Q_{i-1/2,j}^{n+1/2})\right) - \frac{\Delta t}{\Delta y}\left(G(Q_{i,j+1/2}^{n+1/2}) - G(Q_{i,j-1/2}^{n+1/2})\right), \quad (5.15)$$

where we need to solve a one-dimensional conservation law along the grid-cell boundaries:

$$Q_{i+1/2,j}^{n+1/2} = \frac{1}{2}\left(Q_{i+1,j}^{n} + Q_{i,j}^{n}\right) - \frac{1}{2}\frac{1}{\Delta y}\left(F_{i+1,j}^{n} - F_{i,j}^{n}\right), \quad (5.16)$$

$$Q_{i,j+1/2}^{n+1/2} = \frac{1}{2}\left(Q_{i,j+1}^{n} + Q_{i,j}^{n}\right) - \frac{1}{2}\frac{1}{\Delta x}\left(G_{i,j+1}^{n} - G_{i,j}^{n}\right). \quad (5.17)$$

These schemes can also be derived by using finite-differences, with a different interpretation of the unknown quantities. One of the strengths of the finite-volume interpretation over finite-difference, is that it is easily extendible to more complex grids. A more thorough derivation of these schemes can be found in [HHHL07].

Boundary-conditions The shallow-water equations are also equipped with reflective boundaries, implemented in the same manner as for the linear wave equation. However, in addition to copying the water height we also need to reverse the sign of the velocity component in the direction parallel to the normal of the boundary.

CFL-condition Both Lax-Friedrich and Lax-Wendroff are stable provided that:

$$\frac{\Delta t}{h} \max_{ik} |\lambda_k^F(Q_i)| \leq \frac{1}{2}, \tag{5.18}$$

where $h = \Delta x = \Delta y$ and λ_k^F is the eigenvalues of the Jacobian matrix of the flux function F.

5.5 High-resolution schemes

In 1983, Ami Harten [Har97] introduced a new class of schemes called high-resolution schemes. These schemes were introduced to overcome the problems of oscillations and other errors introduced when using traditional high-order schemes. To solve the Euler equations, a more complex high-resolution scheme than the simple composite scheme for the shallow-water equations is used. I will not give a derivation of these schemes, but they are also based on finite-volumes, using cell-averages as approximations. This gives us the evolution equation (5.9) derived in the last section. But instead of using the midpoint method to approximate the fluxes, we use a Gaussian quadrature. To evaluate the flux across the interfaces, we use the central-upwind flux [KNP01]. For integrating the time-dependent ODE for the cell-averages, we use a second-order Total variation diminishing (TVD) [Shu88] Runge-Kutta method:

$$Q_{i,j}^{(1)} = Q_{i,j}^n + \Delta t R_{i,j}(Q^n), \tag{5.19}$$

$$Q_{i,j}^{n+1} = \frac{1}{2} Q_{i,j}^n + \frac{1}{2}\left(Q_{i,j}^{(1)} + \Delta t R_{i,j}(Q^{(1)})\right), \tag{5.20}$$

where $R_{i,j}$ is the right-hand side of (5.9).

The process of computing $R_{i,j}(Q)$ involves many steps and has a very high arithmetic intensity, which makes this scheme ideal as a test-problem for the GPU; arithmetic intensity being the strength of the GPU. Without going through all details, this is the process in short:

1. Reconstruct a piecewise linear Q.
2. Compute point-values.
3. Compute eigenvalues.
4. Compute fluxes.

A thorough derivation of these schemes can be found in "How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational engine" [HHHL07] and "Solving the Euler equations on graphics processing units" [HLN06].

Boundary-conditions For the Euler equations, an outflow Dirichlet boundary condition is used. This simply means that all quantities are simply advected out of the computational domain when reaching the boundary. Another way of looking at it is that the boundary absorbs the conserved quantities. This is achieved by copying values from within the computational domain onto the boundary, in the same manner as for the shallow-water equations, but this time *without* reversing the sign of the velocity components.

CFL-condition The disturbances may travel at most one half grid cell per time-step:

$$\max(a^+, -a^-)\Delta t \leq \frac{\Delta x}{2}, \tag{5.21}$$

where

$$a^+ = \max(0, \lambda^+(Q_L), \lambda^+(Q_R)), a^- = \min(0, \lambda^-(Q_L), \lambda^-(Q_R)), \tag{5.22}$$

and similarly for the y-direction. Q_L and Q_R are the left- and right-sided point value at each integration point and $\lambda^{\pm}(Q)$ are the slow and fast eigenvalues of dF/dQ.

Chapter 6

Results

> “*If something’s hard to do,*
> *then it’s not worth doing.*”
> — Homer J. Simpson

This chapter describes how the simulation data has been stored and visualized, with screenshots of the visualizations. The benchmarks that were used are then presented together with the results. The conclusions based on these results, and the master thesis as a whole, are presented in Chapter 7.

6.1 Visualization and animation of simulation data

To visualize the simulation data from each of the three equations, a separate application has been written. This application reads the file(s) outputted from the simulation application and visualizes them using OpenGL and GLSL.

6.1.1 Storing simulation data

The simulation data are collected on one node at the end of each time-step. All data are then stored in one huge array on the collecting node, and then written to file(s) all at once when the simulation is finished. This limits disk access during execution, but it also eats away memory which could influence the execution time of the simulations. This also poses another problem; what happens if there is generated more simulation data than can be stored in memory on one single node? One solution here is, of course, to have a master node with much more memory than the other nodes, the other solution is to change the way data is stored. The different methods of storing data to disk were presented in Section 2.5.2 in Chapter 2. The reason behind the choice of this somewhat naïve storing strategy is that it

is simple, and I do not need to store the simulation data for other purposes than making screenshots and animations. If the application were to be run as production code and save simulations on a daily basis, we would have to employ one of the more complex and efficient storing strategies.

Another issue that is important to notice, is how the use of GCE limits the opportunity to store simulation data. If we have a GCE-level of 50, this means that the GPU can perform 50 time-steps without any data being read back to the CPU. By extension, this means that we are only able to store each 50th time-step. This is not the only drawback associated with storing; it also forces us to read back the complete computational domain, and not just the ghost cells. The conclusion is, the fewer time-steps we need to store, the faster the simulation will run.

Data The only values stored when simulating the linear wave equation and the shallow-water equations are the water height. It is stored at each spatial grid point for each time-step. This is sufficient information to visualize the propagating wave. To properly visualize the Euler equations we need to store more than one variable. Actually, we could just store the pressure and visualize that, but a more advanced visualization is used here. In this case we end up with four files, one file for each variable. These files are equivalent to the R, G, B and A-channels collected for each time-step from the texture the variables are stored in.

6.1.2 Visualizing on the GPU

There are two different shader programs used to visualize the data, both written by Hagen et al. The first one is based on the reflection law from physics, and uses the water height to determine how the light from some light source is reflected off the water surface over some background. This makes the background appear to be the bottom of a body of water, with the simulated wave on top. Figures 6.1 and 6.2 show screenshots from the visualizations of the linear wave equation, and Figure 6.3 and 6.4 shows screenshots from the visualization of the shallow-water equations. The initial conditions for both cases are a small collection of elevated values somewhere in the computational domain. This simulates a drop of water onto a still-water surface.

The other type of visualization, used for the Euler equations, are Schlieren gradients. Schlieren imaging [Wik07c] is really a photo-optical technique for visualizing the flow of fluids of different densities. This technique is imitated here by depicting $(1 - |\nabla\rho|/\max|\nabla\rho|)^p$ for $p = 15$ as a grey-map. Schlieren imaging is frequently used for studying shock-phenomena. Screenshots from this visualization can be seen in Figures 6.5 and 6.6. The air is initially at rest with a circle of low density ($\rho = 0.1$) that is hit by a shock in the air

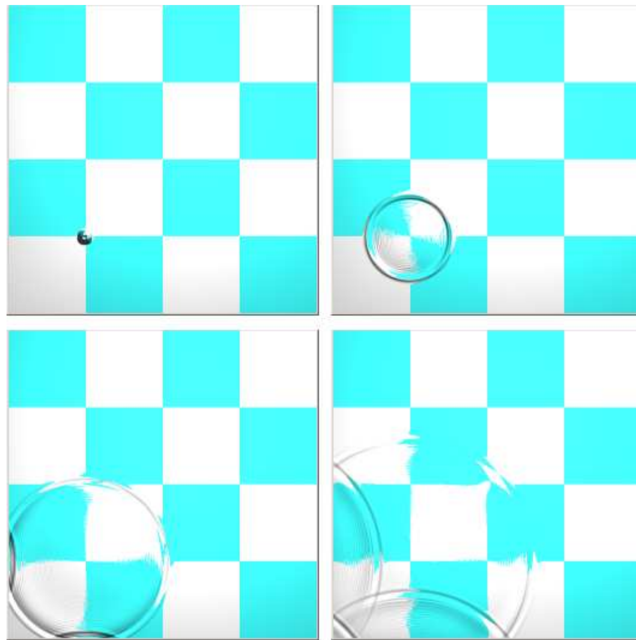

Figure 6.1: Screenshots from the visualization of the linear wave equation, using only one sub-domain. (Time-steps $t = 10, 100, 200, 300$)

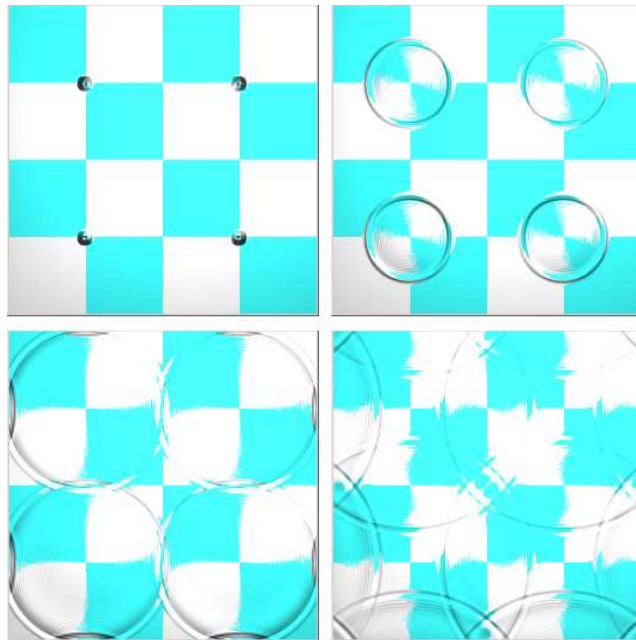

Figure 6.2: Screenshots from the visualization of the linear wave equation, using four sub-domains. (Time-steps $t = 10, 100, 200, 300$)

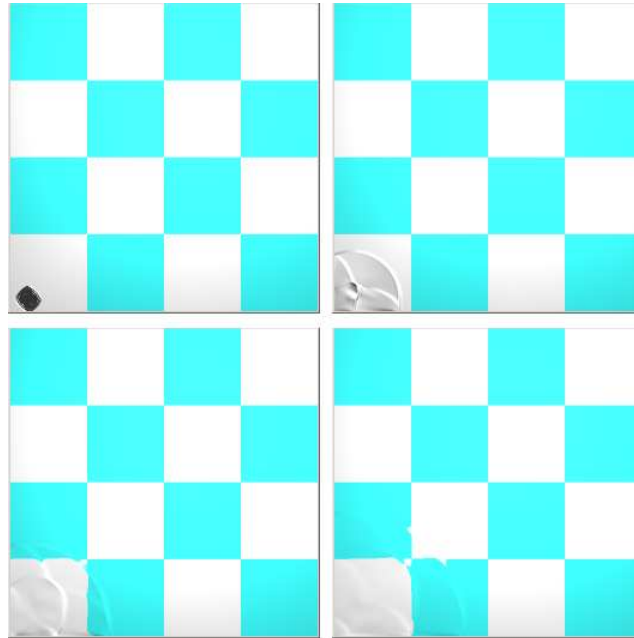

Figure 6.3: Screenshots from the visualization of the shallow-water equations, using only one sub-domain. (Time-steps $t = 10, 100, 200, 300$)

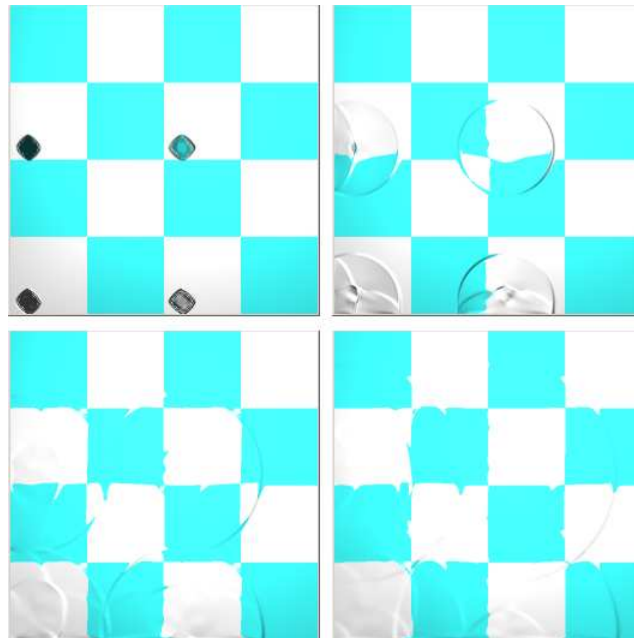

Figure 6.4: Screenshots from the visualization of the shallow-water equations, using four sub-domains. (Time-steps $t = 10, 100, 200, 300$)

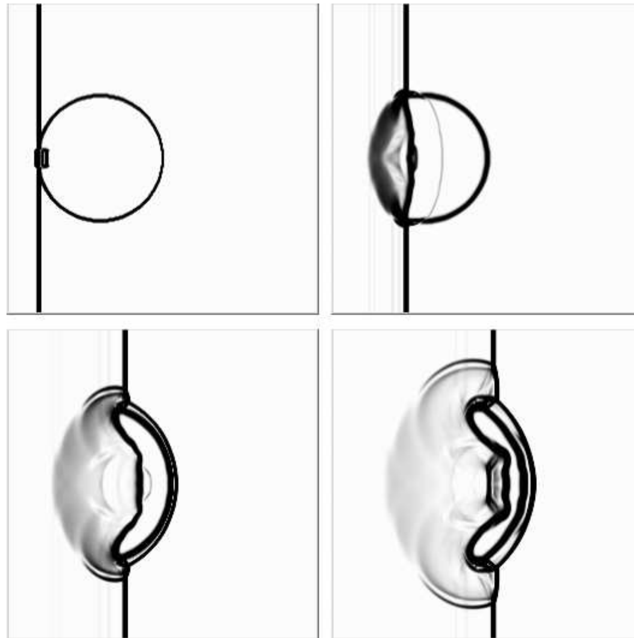

Figure 6.5: Screenshots from the visualization of the Euler equations, using only one sub-domain. (Time-steps $t = 10, 400, 800, 1200$)

propagating from left to right in the x-direction. The pressure p behind the shock is 10.

6.2 Benchmarking and considerations

Hagen et al. showed in "How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational engine" [HHHL07] that hyperbolic conservation laws have a significant speedup on a single GPU compared to a single-CPU reference implementation. This does not, however, guarantee that a cluster of GPU-nodes will be effective. There are a number of factors that influence the efficiency of this cluster, where the two most important are inter-node communication over the network and communication between the CPU and the GPU. The rest of this section will describe the node setup, the different benchmarks, and the results obtained from these.

6.2.1 Setup

This section covers the benchmarking setup. The configuration described here was used in all testing and for all visualization screenshots in this chapter. It is the minimal configuration needed to test the application in two spatial dimensions. The main reason for this is that no more nodes were

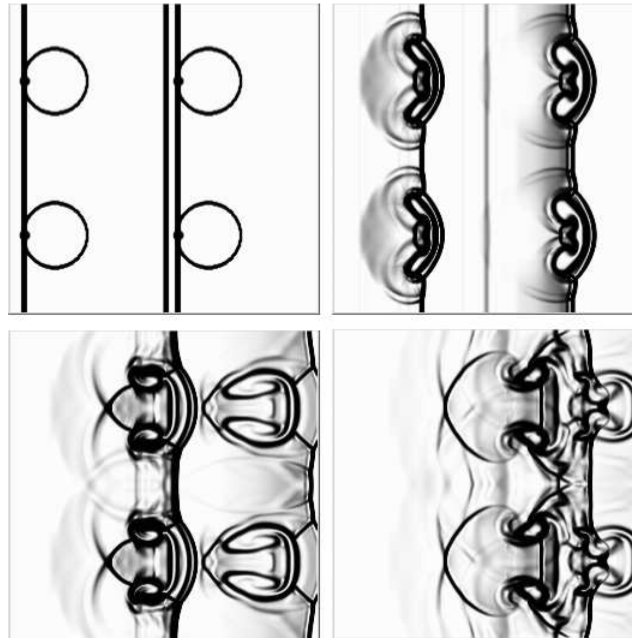

Figure 6.6: Screenshots from the visualization of the Euler equations, using four sub-domains. (Time-steps $t = 20, 800, 1600, 2400$)

available. However, this setup will still demonstrate some of the strengths and weaknesses of a GPU cluster.

Physical setup

The physical setup of the GPU cluster can be seen in Figure 4.1 in Chapter 4. Each node in the cluster had the following configuration:

CPU: Intel Pentium 4 with 3GHz (with HT technology)

System memory (RAM): 2GB

Graphics adapter (GPU): NVIDIA GeForce 7800GT with 256MB memory on a PCI-Express bus

Network adapter: Broadcom NetXtreme Gigabit Ethernet

The nodes were connected through a Fast Ethernet switch.

The graphics adapter was tested using GPUBench [BFH04a], and the results of the readback test can be seen here:

```
Fixed   Hostmem   GL_RGBA       Mpix/sec: 193.71  MB/sec: 738.95
Fixed   Hostmem   GL_ABGR_EXT   Mpix/sec: 201.50  MB/sec: 768.66
Fixed   Hostmem   GL_BGRA       Mpix/sec: 220.11  MB/sec: 839.66
```

This tells us that the network should prove to be the biggest bottleneck, because of its low bandwidth compared to the PCI-Express bus.

Application setup

All benchmarks were run with the configuration below, if not stated otherwise. Variable values in the benchmark tables supersede these configurations.

Number of time-steps: 1000

Number of nodes: 4

Number of processes: 4

Number of expanded ghostcells: Tweaked to obtain the best possible result

Total number of ghostcells: Dependent on scheme.

The three PDE test-cases will be abbreviated as LWE, SW and EE, which refers to the linear wave equation, the shallow-water equations, and the Euler equations, respectively. N is the global grid size.

Parameters As can be clearly seen in the two last paragraphs, we have a large number of parameters that can be adjusted. It has been a challenge to construct informative benchmarks, and the focus has been on showing trends that can be used in further research on GPU clusters. In addition to the variables already presented, it is also possible to have several GPUs per CPU-node, as discussed in Section 7.1 in Chapter 7.

6.2.2 Benchmarks

Benchmarking of the application written for this thesis is a fairly complex issue, since it utilizes both the CPU and the GPU, and in the same time is distributed on multiple nodes. Since one of the main concerns is to detect possible bottlenecks in the application, it was necessary to benchmark several parts of the application. All benchmarks have been run without storing simulation data. The following areas were benchmarked:

- Overall execution time and performance
- Impact of GCE
- GPU shaders
- GPU-uploading and -readback
- Inter-node communication

Table 6.1: Overall execution times.

-		Equation		
Nodes	N	LWE	SW	EE
1	256	1.46425	1.69143	11.35120
	512	2.87217	3.02078	24.43520
	1024	7.91547	8.76265	84.83230
	2048	27.7595	27.8843	336.01203
4	256	4.91324	4.60230	19.2864
	512	6.86207	5.29687	25.2825
	1024	13.2893	8.05684	47.3395
	2048	26.6323	18.5888	207.3422

MPI's `MPI_Wtime()` was used in all five benchmarks.

The reasons behind the choice of exactly these benchmarks are that they are naturally segmented in the application, and because they are likely to reveal where the bottleneck of the cluster can be found.

Overall execution This benchmark simply measures the execution time of the application as a whole, only excluding the writing of simulation data to disk. The results can be seen in Table 6.1 and Figure 6.7. We can see that all the equations eventually perform better on the GPU cluster than on a single GPU, when increasing the global domain size. We also see that the equations that require the most arithmetic intensity have more to gain from the GPU cluster than the ones with lower arithmetic intensity.

Impact of GCE This benchmark measures the impact different levels of GCE have on total execution time. These execution times are for the linear wave equation, however, the two other equations have similar trends. Results are found in Table 6.2 and Figure 6.8.

Determining on optimal GCE level has proven to be somewhat difficult. The optimal GCE level depends on static and dynamic network parameters, the computational molecule of the scheme, and the size of the local sub-domain. Nevertheless, GCE has proven to reduce the cost of network communication and the total execution time significantly. Partly by eliminating much of the overhead associated with network latency, and partly by allowing several time-steps to be performed on the GPU before a readback to the CPU is necessary.

GPU shaders The results from the execution of the shader programs can be viewed in Table 6.3. This shows that the execution time grows with

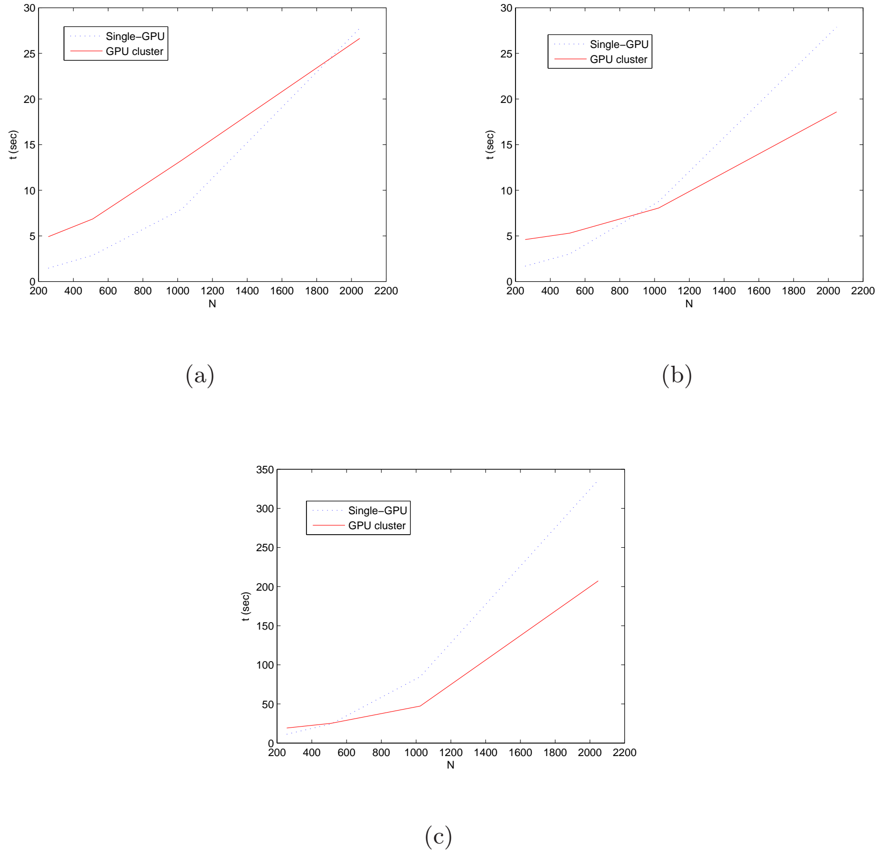

Figure 6.7: Plot of overall execution times. N is global domain size. (a) The linear wave equation. (b) The shallow-water equations. (c) The Euler equations.

Table 6.2: Impact of GCE on total execution time of the LWE.

-	N			
GCE level	256	512	1024	2048
1	8.56322	11.4503	21.6686	52.2775
2	6.33932	8.61161	15.6966	33.9080
4	5.30109	7.33385	13.4487	31.9056
8	4.91378	6.83736	12.8738	26.1673
16	3.98509	6.48395	12.2361	25.7829
32	5.95926	8.13920	12.5027	27.5297
64	6.11298	8.16748	14.8376	28.4251
128	7.43962	9.69644	16.0252	-

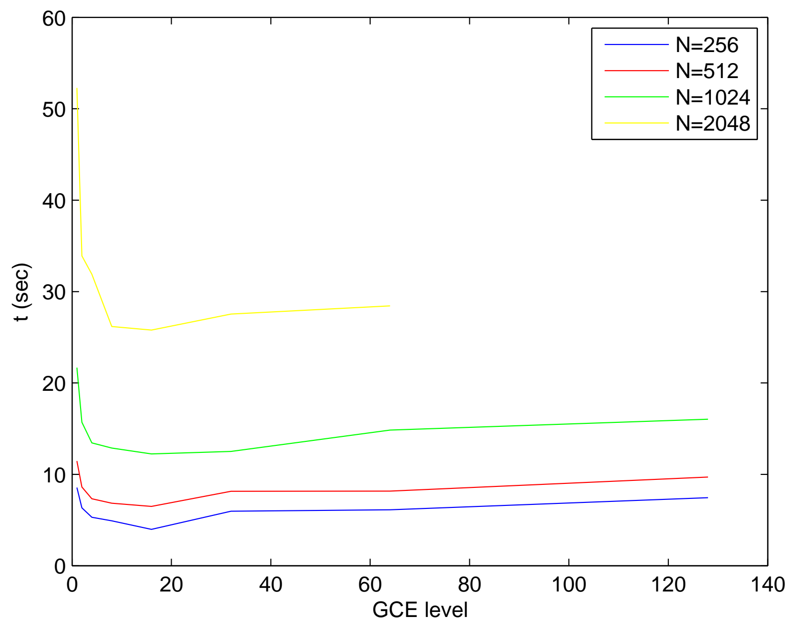

Figure 6.8: Plot of GCE-level impact on overall execution time for the linear wave equation. N is global domain size.

the size of the computational domain and the arithmetic intensity of the computational scheme, as expected.

GPU-uploading and -readback Before a ghost-cell exchange can occur, the values to be exchanged need to be read from the GPU. After the exchange, the new values need to be uploaded to the GPU. The level of ghost cell expansion together with the size of the domain determine how much time this takes. This benchmark measures the time used on these operations. Readback is slower than upload due to reasons discussed in Section 4.4 in Chapter 4. The results can be seen in Table 6.4.

Inter-node communication This benchmark measures the time used on inter-node communication, meaning the exchanging of ghost cells between nodes. The initial broadcasting of sub-domain size and other one-time broadcasts are ignored, since these contributions are minimal and constant. They are, however, included in the overall execution benchmark. The results can be seen in Table 6.5, and they clearly shows that the network is the biggest bottleneck, just as expected.

Table 6.3: Shader benchmarks.

Equation	N	Time
LWE	256	0.57063
	512	1.79228
	1024	2.08224
	2048	23.1243
SWE	256	1.04674
	512	2.28828
	1024	6.75697
	2048	25.0086
EE	256	10.4211
	512	22.0435
	1024	80.0092
	2048	325.8762

Table 6.4: GPU-CPU communication benchmarks for the LWE.

N	Upload	Readback
256	0.06397	0.15794
512	0.10979	0.36200
1024	0.22813	1.03398
2048	0.47758	3.34950

Table 6.5: Inter-node communication benchmarks for the LWE.

N	Time
256	4.18286
512	5.74454
1024	10.9581
2048	22.9631

Chapter 7

Conclusions

"*Do not say a little in many words but a great deal in a few.*"
— Pythagoras

This thesis has been about fusing traditional parallel programming with the power of GPGPU-programming. This chapter sums up the experiences and results from this experiment. I will give answers to the research questions posed in Chapter 1, extract the important points from the text, and discuss possible extensions beyond this thesis.

This thesis is supported on a great deal of research fanned over several scientific areas. To use a GPU cluster it is therefore necessary to have a broad background, and knowledge of both traditional parallel programming, GPGPU-programming, networking, and, of course, detailed knowledge of the computations you wish to perform. For inherently parallel problems, however, the potential gain in efficiency could very well be worth the cost. Even without a speedup in comparison with a single-GPU implementation, a GPU cluster would still be attractive. This is because it allows you to solve almost arbitrary sized problems (limited by the number of nodes) much faster than a standard CPU-cluster is able to. This is supported in my results by the fact that most of the communication overhead is in the network traffic, not in the GPU-CPU communication, and the network communication would also be necessary on a CPU-cluster. When also considering that a single-GPU implementation significantly outperforms a single-CPU implementation, the rest follows.

Accomplishments The main accomplishment of this master thesis is the demonstration of a working GPU cluster. This demanded the use of message-passing, GPGPU-programming, ghost-cell expansion, and domain decomposition. I also had to come up with a reasonable algorithm for storing the simulation data, and then recompose them before visualizing them. In

addition, the linear wave equation, the shallow-water equations and the Euler equations all have been simulated in two spatial dimensions as test-cases for the GPU cluster. Much of the GPGPU work is based on earlier code and work performed at SINTEF ICT, Oslo.

Efficiency As can be clearly seen from the results, the GPU cluster needs to work on domains of some size before it becomes effective in comparison with a single-GPU run of the same global domain size. The graphics adapters in the test-cluster had only 256MB, which limited the texture size to $\sim 2048 \times 2048$. However, already on this size, the GPU cluster outperformed the single-GPU solution. In fact, on the Euler equations, the GPU cluster outperformed a single-GPU node on a 1024×1024 global grid. More video memory and larger textures would further improve the efficiency of the GPU cluster. The GCE level has also proven to be critical for the performance of the GPU cluster, and needs to be adjusted according to grid size and the nature of the computations. The GPU cluster has also proven to be more effective on problems with a high arithmetic intensity than problems with less arithmetic intensity, as expected.

Accuracy Though the floating-point accuracy only extends to single-precision, this is sufficient for the problems considered in this thesis. Double-precision floating point representation will also soon be available, making accuracy an obsolete problem. The arrival of double-precision will open the GPGPU-scene for problems that require a higher degree of accuracy than the test-cases presented in this thesis.

7.1 Further research and extensions

There are many aspects in this thesis that deserve further work, and several alternative configurations of the GPU cluster that may improve efficiency. Testing and benchmarking with gigabit and infiniband networks would have been the next natural thing to do, since the network has proven to be the biggest bottleneck in the cluster.

The linear wave equation and the shallow-water equation do not use the full per-fragment vector. It would increase efficiency to use a packing scheme on the unknowns, such that all the GPU pipelines are utilized to the maximum, e.g., one could pack four grid cells into each fragment when simulating the linear wave equation.

Another possible extension is to have several GPUs per physical node, which would eliminate some of the communication need and thereby lessen execution time. This could be done by using NVIDIA's Scalable Link Interface (SLI) [NVId, Wik07b] or ATI's CrossFire [ATIa, Wik07a]. By clustering interfacing sub-domains on one SLI-node, the cost of ghost cell transfer

would be significantly reduced compared to using MPI for ghost cell exchanges between the same sub-domains, if located on different nodes. SLI offers data transfer directly through the PCI-Express bus without the costly readback to the CPU, i.e. the GPUs could share the workload without going through the CPU. Asynchronous execution of computations, CPU-GPU communications, and network communications like discussed in Section 4.4 in Chapter 4 could also help to hide some of the cost of ghost cell transfers.

Further benchmarking with additional nodes would have been preferable, as this could reveal further details of interest concerning GPU clusters. Unfortunately, I did not find a suitable site with more than four GPU nodes since most freely available clusters have poor graphics adapters at best.

This thesis have in some sense been about "assembling the puzzle". I can only now, at the time of writing a conclusion, claim to have a good overview of all the components used in the thesis, and a good code platform for performing computations on a GPU cluster. This thesis has hopefully taken you to the starting line of GPU cluster computations, and given some impressions of its usability. And surely, the many new exciting languages and technologies (PeakStream, CUDA, new generations of graphics adapters etc.) will make the use of the GPU and its like, increasingly important in the future.

Chapter 8

Acknowledgements

"*Misquotations are the only quotations that are never misquoted.*"
— Hesketh Pearson

This thesis could not have been written without the help and support from many people and institutions, and I would like to thank some of them here. First, I want to thank my supervisors Knut-Andreas Lie and Trond Runar Hagen for their support, ideas and valuable help. This thesis would have had significantly more errors, both grammatically and theory-wise, if it had not been for Knut-Andreas' red pen and keen eyes. I would also like to thank my fellow master students André R. Brodtkorb, Trygve Fladby, Thomas Lunde, Hanne Moen and Lars Moastuen at SINTEF ICT, and Guo Wei Ma at Simula Research Laboratory. SINTEF ICT deserves a big thank you for the great offices we had at our disposal, not many master students get a corner office in the top floor with a view. The Department of Informatics provided us with new computers, and excellent wide-screen monitors, which we were all very grateful for. Everybody needs some distraction from the studies from time to time, and I would like to thank Cybernetisk Selskab for providing this to me. Finally, I would like to thank my family for believing in me and always supporting me in all my choices, and everyone else that in some way have contributed to this thesis.

Appendix A

The Top 500 project

The parallel PLU project is a cooperation between André R. Brodtkorb, Trygve Fladby and myself. The original idea was to run the application at The Gathering (TG) [KAN], the worlds largest computer party. Time constraints, and other issues (e.g., security), prevented us from reaching our original goal. We have completed the project, despite that we were unable to run the application at TG, and reported our findings in the following white-paper.

My contribution in this white-paper is mainly the parts that concerns communication and message-passing (MPI-2). The PLU algorithms are written by Fladby and Brodtkorb, and my role in these algorithms is limited to advising and discussions.

PLU Factorization on a Cluster of GPUs Using Fast Ethernet

André Rigland Brodtkorb, Martin Lilleeng Sætra and Trygve Fladby

13th July 2007

Abstract In this white paper, we present a novel approach to solve linear systems of equations on a cluster using the PLU factorization. We use the graphics processing unit (GPU) as the main computational engine at each node, and a block-cyclic data distribution to solve the system. The local computation is a new way of solving the PLU factorization on the GPU. It utilizes the full four-way vectorized arithmetic found in most GPUs, and a new pivoting strategy. The global algorithm uses the message passing interface (MPI) for communication between nodes. We show that our algorithm is highly efficient on the local nodes, but bounded by the relatively slow network. A faster network will eliminate this bottleneck, and the speed of the local computations show promising results.

A.1 Introduction

This paper explores the field of general purpose computation on graphics processing units (GPGPU). We specifically target the PLU factorization of a large system of linear equations on a cluster of nodes. Solving large linear systems of equations using dense algorithms is used extensively as a benchmark for clusters and supercomputers. The High Performance LINPACK benchmark (HPL) [PWDC] which computes the PLU factorization, is the standard way of benchmarking and ranking the fastest 500 supercomputers in the world [UUN]. This benchmark, however, has been criticized for neglecting the importance of faster inter-node communication. This is because the HPL benchmark can run the benchmark with different parameters that compensate for slow network communication by letting each node execute extra computations (e.g., look-ahead).

While the HPL benchmark uses the CPU to compute partial results on each node, we utilize the graphics processing unit (GPU) as the main computational engine to solve the same problem. The GPU is a massively parallel processor with vast amounts of processing power [OLG+07]. Current GPUs have a theoretical peak of 400 GFLOPS [Neo07], compared to 90 GFLOPS [Neo07] for current high-end CPUs. When comparing the price[1] per FLOP, the GPU comes out ahead as well with approximately $1.50 per GFLOP, compared to the CPU that costs approximately $18 per GFLOP.

During the last years, we have seen an enormous development in 3D-graphics. The demand for more powerful programmable graphics processing units (GPU) from for example the gaming industry has led to increased flexibility in the processors. The rapid evolution in speed and flexibility has made the GPU interesting for scientific purposes as well. The field of general-purpose computation on GPUs (GPGPU) has emerged as a new and exiting research area [OLG+07]. Even though the GPU is a far more powerful and cost-effective processor than the CPU, there is another price. While the CPU has complex logic for branch prediction, cache management, and instruction pipelining, most of the transistors on the GPU are used for pure floating-point operations. There is another architectural difference as well. The CPU

[1]Prices are from the Norwegian web shop komplett.no 2007-04-23.

is designed to operate on sequential code, such as word processing where each character is entered and processed sequentially. The GPU on the other hand, is designed to simultaneously compute all the pixels that together make up the screen image. In addition, the GPU could traditionally only be accessed via a graphics API, such as OpenGL [SWND05] or DirectX [Mic07]. The architectural differences, and the need to access the GPU through a graphics API require new algorithms and techniques to be employed when the GPU is to be used for general-purpose computing.

A.2 Background

The Top 500 project [UUN] was started in 1993 to provide a reliable basis for tracking and detecting trends in the field of high-performance computing. It is a list of the 500 most powerful supercomputers, which is updated twice per year. The ranking of the supercomputer sites is determined by how well they perform on the LINPACK benchmark. A parallel version of LINPACK named HPL [PWDC] was introduced by Dongarra, for this purpose. HPL is short for High-Performance LINPACK Benchmark for Distributed-Memory Computers. HPL utilizes the Message Passing Interface (MPI) and the Basic Linear Algebra Subprograms (BLAS). The algorithm used by HPL implements a two-dimensional block-cyclic data distribution. In addition a look-ahead strategy and bandwidth reducing swap-broadcast algorithm is used to increase performance. The complete operation count sums up to $\mathcal{O}(\frac{2}{3}n^3) + \mathcal{O}(n^2)$.

LU factorization on the GPU has previously been implemented by Galoppo et al. [GGHM05]. One of their main contributions was index-pair streaming, which uses texture coordinates to make a cache-oblivious algorithm. The index-pair streaming technique sets texture coordinates from the CPU in order for the GPU to pre-fetch data, in contrast to computing them on the fly on the GPU. This data pre-fetch resulted in about 25% speed increase [GGHM05]. They also reported their algorithm as faster than ATLAS, but the benchmark was highly synthetic.

To run our application in parallel on multiple nodes, we have utilized the Message Passing Interface 2.0 (MPI-2) [MPI]. MPI-2 is a C/C++ and Fortran interface for message passing between multiple processes spread over any number of nodes. It can be used in many different setups, e.g., supercomputers, distributed memory clusters, and shared memory clusters. Several implementations of MPI-2 exist, where we have chosen MPICH2 [Argc] for our application. The most important uses of MPI-2 in our application are the automatic generation of a block-cyclic Cartesian grid of processes and broadcast of data to groups of processes.

There are two concepts related to our use of MPI-2 that require some explanation; communicators, and blocking- and non-blocking calls. A *communicator* in MPI is a collection of processes. Many functions in MPI-2 take a communicator as argument and perform the requested operation on all processes in that communicator. A call to the broadcast function in MPI, for example, can look like this: `MPI_Bcast(buf, 10, MPI_FLOAT, 0, MPI_COMM_WORLD)`. This call will broadcast ten elements of the array `buf` to all processes in the `MPI_COMM_WORLD` communicator. The other processes in the communicator must also call the `MPI_Bcast` function to receive these elements. The `MPI_COMM_WORLD` communicator is a special communicator that contains all processes, and it is initialized automatically by MPI. When

```
1 MPI_Init(&argc, &argv);

  if(processId == 0) {
    MPI_Recv(buf, 10, MPI_INT, 1, 101, MPI_COMM_WORLD, &status);
5   MPI_Send(buf, 10, MPI_INT, 0, 100, MPI_COMM_WORLD);
  } else(processId == 1) {
    MPI_Recv(buf, 10, MPI_INT, 0, 100, MPI_COMM_WORLD, &status);
    MPI_Send(buf, 10, MPI_INT, 1, 101, MPI_COMM_WORLD);
  }
10

  MPI_Finalize();
```

Listing A.1: Example on a deadlock in an MPI-2 program

an MPI function is called on all processes within a communicator (or group) it is referred to as a collective operation. `MPI_Bcast` is a collective operation.

A *blocking* call will make the application wait for the call to complete before continuing execution. In this way you will know if the call has finished successfully or aborted due to some error. This also means that the application may get deadlocked, where two or more processes have called competing blocking functions that are circularly dependent on each other [CES71]. For example, if we have two processes that execute the code in Listing A.1, it will result in a deadlock. Both processes are waiting for the other to send data, thus blocking program execution. A non-blocking call on the other hand, will not cause the application to wait for the call to return. In this way it is possible to call a function and continue executing the application before the function returns. Collective operations in MPI-2, however, are always blocking.

A.3 Algorithm

The LU factorization of a matrix A can be written as $LU = A$, where L and U are *lower* and *upper* triangular respectively. Using the Doolittle algorithm, we can construct the upper triangular matrix U using Gaussian elimination. The lower triangular matrix is constructed from the multipliers used to reduce A to an upper triangular form. For our algorithm to be numerically stable, we also permute the rows of A. This is known as partial pivoting, and ensures that the row we are eliminating with creates smaller perturbations of the result than would normally occur. With the permutation of the rows in A, our factorization takes the form $A = P^T LU$, where P is the permutation matrix that permutes rows of A.

Our algorithm has two layers, the global and the local computation. The global algorithm solves the PLU factorization of the matrix spread over all the nodes, shown in Figure A.1(b), whilst the local algorithm is what each node needs to compute for the global algorithm to be correct.

Each node in the computation receives a block-cyclic part of the matrix, as shown in Figure A.1(b). Then, all the processors compute what type of operation they need to compute. Our algorithm splits the computation into four distinct operations: pivot, normalize, eliminate and reduce, as shown in Figure A.1(a). The operation computed on each node depends on the global position of the pivot operation. All processors that hold elements in the

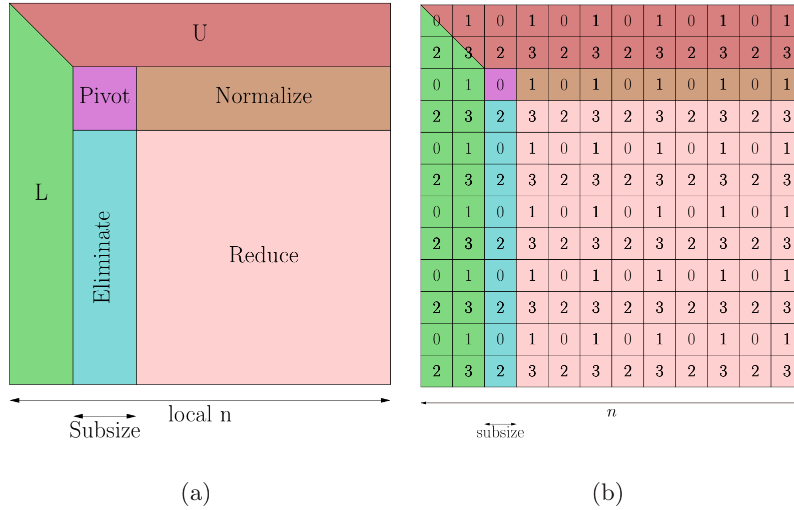

Figure A.1: PLU decomposition on a cluster of nodes: (a) the four different parts of the LU factorization. (b) the block-cyclic distribution of data on four nodes, 0, 1, 2 and 3.

same row as the pivot operation need to compute the normalize operation, and similarly all nodes with elements in the same column as the pivot operation need to compute the eliminate operation. All remaining nodes need to compute the reduction operation. In Figure A.1(b) this means that process 0 is the *pivot*, process 1 executes *normalize*, process 2 *eliminate*, and process 3 *reduce*. The pivot node shifts one down along the diagonal for each global pass.

A.3.1 Global algorithm

Computing the PLU factorization is an almost embarrassingly parallel operation. However, vanilla implementations demand a lot of data to be transferred between nodes, which is a very costly operation. In addition, many nodes would simply idle as we reach the end of the computation.

To reduce the idling, we distribute the matrix A block cyclically in the same fashion as the HPL algorithm [PWDC]. Figure A.1(b) shows this distribution, where all nodes have a part of the matrix to process throughout the whole factorization, except for the very last block. The last block is computed by the last node in an extra pass. For each pass in the global domain, we compute the result of one row of blocks, and one column of blocks. In the following, we refer to these as *block-row* and *block-column* respectively.

To lessen the amount and number of transfers between nodes, we use partial pivoting within in-core memory, thus eliminating the need to transfer rows between processors. It is trivial to create examples where partial pivoting fails, but sufficient accuracy is attainable in practice. This also holds for our pivoting, which pivots in a subset of the regular pivot candidates.

In order to compute one pass in the global domain, we have to execute the four different operations *pivot, normalize, eliminate* and *reduce*. It should be mentioned that this data distribution, and splitting into different operations per node allows for multiple nodes, not only four as shown in this example. In the third pass of this algorithm, we have the following situation (see also Figure A.2):

Pivot: The pivot position (process 0) must compute the PLU factorization of the current active pivot block in its local domain. The block size is subsize $\times$ subsize. In addition, it has to reduce the rest of the local matrix according to the computed L and U. These blocks belong elsewhere in the global domain (see Figure A.1(b)). In each global pass, there is always only one pivot node.

Normalize: The normalize operation (process 1) needs to compute U according to the P and L computed by the *pivot* operation. It will also have to reduce all remaining elements in the local matrix, which again belong elsewhere in the global domain. There are $s - 1$ nodes that compute the normalize operation in each global pass, where s is the width and height of the processor grid.

Eliminate: Eliminate (process 2) calculates the multipliers needed to forward substitute one block by using the computed U's from *pivot*. In addition, it has to reduce the rest of the local matrix, according to the computed U. In each global pass, the number of eliminate nodes is also $s - 1$.

Reduce: The reduce operation simply reduces the local matrix according to the L and U computed in *eliminate* and *normalize* respectively. All remaining processes compute this operation, $s \times s - 2(s - 1) - 1$ nodes.

As stated in the list of operations, the different processes depend on data from other processes. This dependency is not static, but varies with the operation the current node is set to execute. Figure A.2 shows how the data is sent in the already used example. The nodes waiting for data cannot continue before they have received the data. This effectively limits the computational speed to the slowest node. The HPL [PWDC] algorithm uses look-ahead to remedy this somewhat. As this chart shows, there is still quite a lot of idling for the four nodes. The pivot node, for example, computes its result and then waits until all other nodes have completed their computations.

A.3.2 Local algorithm

The local algorithm includes four stages *pivot, eliminate, normalize* and *reduce*, but first we will introduce the matrix representation. The data is row-wise represented in four-wide vectors [Mor03]. This is to utilize as much computational power and bandwidth as possible, since most GPUs can execute one MAD instruction on four-long vectors per clock cycle. The advantage of this packing scheme is that it does not require restructuring of the data in main memory before it is sent to the GPU[2]. Another reason for this choice is that it fits well with the solution we have for pivoting. In addition to storing the matrix, we add an extra column leftmost in the

[2] Assuming its width is divisible by four.

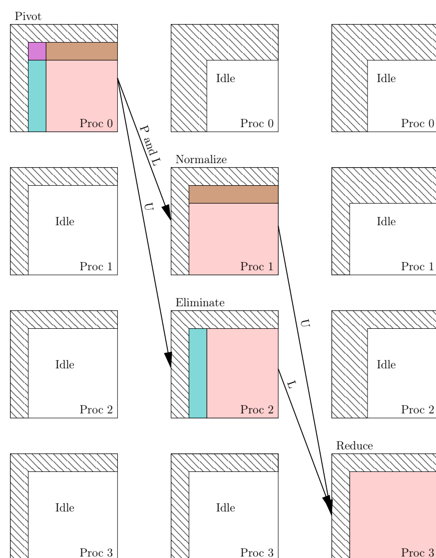

Figure A.2: Data send patterns for PLU decomposition using four nodes in the third global pass (corresponds to the situation in Figure A.1(b)). The shaded areas represent the part of the matrix we already have computed.

matrix, as shown in in Figure A.3(a). This column is used to speed up the calculation of the next pivot element, explained later. Because the result of writing to the same buffer as we read from is explicitly undefined in OpenGL, we have to use an extra texture. The two textures are used as one virtual matrix, but we alternate between reading / writing and writing / reading to the front and back textures, respectively. This technique is referred to as ping-ponging in the field of GPGPU.

Pivot

The pivot procedure computes the PLU factorization of A, but stops when one block-row and one block-column has been computed (see Figure A.1(b)). It can roughly be split into two tasks: multiplier calculation, and reduction, each explained below. To compute a single row and column, we start by permuting the first column simultaneously as we compute the multipliers. Then, we reduce the rest of the matrix, whilst permuting the rows here as well.

To compute one column of multipliers, we read from the correct location in the source texture, and write to the leftmost column in the destination, as shown in Figure A.3(a). The top element is rendered at the position of the pivot element. Because the multiplier for the top row always is one, we do not need to compute it. In addition to computing the multipliers, we also compute the values of the column to the right of the pivot position and store in one of the other color channels (see Figure A.3(b)).

When the computation is complete, we transfer the multipliers and the reduced next column to the CPU using a pixel buffer object (PBO). The PBO uses asynchronous read-back to the CPU, allowing both the CPU and

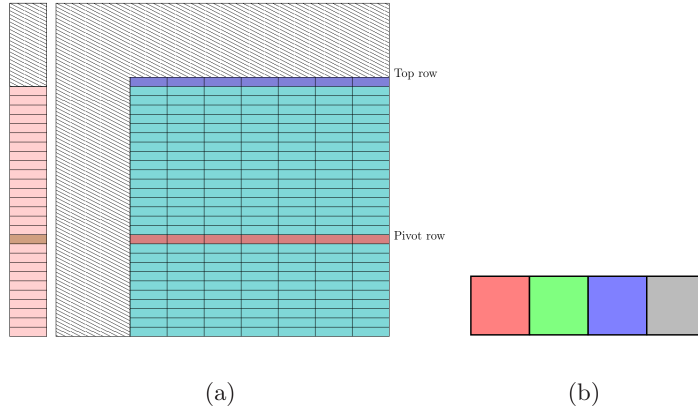

Figure A.3: Data representation on the GPU: (a) Row interchange of the multipliers (leftmost column) and the rest of the matrix (cyan part). (b) The leftmost column of the texture, with both the multiplier, and the reduced next column in the PLU factorization. The multiplier is stored in the red color channel, and the reduced next column is stored in the blue color channel.

the GPU to continue execution. When the whole leftmost column has been transferred to the CPU, the next pivot element is found by the CPU. Simultaneously as the data is copied, and the CPU searches for the pivot element, the GPU subtracts the multiplier times the top row throughout the rest of the matrix. The top and pivot row are also interchanged simultaneously in the same manner as in the first column. In addition, we employ the index pair streaming technique to increase performance [GGHM05]. When the computation is complete, the top row is copied to the CPU, again using a PBO. The algorithm continues until we have computed the whole block-row of U, and block-column of L.

Normalize

The normalize step computed on the local domain executes as follows: The L matrix from this global time-step's pivot node is uploaded to the GPU as a texture. Then, we execute a for-loop that sequentially computes one row of U at a time. First, the current top row and pivot row are swapped, simultaneously as we eliminate using the multipliers in L. Because we are using two buffers, we read back the pivot row simultaneously using PBO's, and store them in main memory. When all rows in the block-row have been computed, U is sent to all nodes in the same column for the reduction operation.

```
1 /* Set up row communicators */
  MPI_Cart_sub(origcom, {0, 1}, &rowcom);

  /* Set up column communicators */
5 MPI_Cart_sub(origcom, {1, 0}, &colcom);
```

Listing A.2: Setting up row- and column-communicators

Eliminate

The elimination procedure calculates multipliers. Normalized rows (U) are sent from the current time-step's pivot node, and the multipliers are calculated using these. The elimination step follows much of the same procedure as the pivot step, but it is a simpler case since there is no complications with row interchanges. This is again because the pivot node only pivots within in-core memory.

Reduce

The reduction step is trivial on the local node. Using a for-loop, we sequentially reduce the whole remaining sub-matrix by looking up one row from U and one column from L, and calculating the reduced A as $A_{i,j} := A_{i,j} - L_{i,k} \cdot U_{k,j}$.

Sending of data

This section describes how data is sent between different nodes. The use of MPI-2 for this inter-node communication will also be explained in detail.

Based on the algorithm discussed in Section A.3.1 we have the following communication scenarios:

1. Sending data to all processes in the same row as active process (to *normalize* and *reduce*).
2. Sending data to all processes in the same column as active process (to *eliminate* and *reduce*).

For broadcasting data to all processes in the same row as the active process, the broadcast function in MPI, `MPI_Bcast`, is used. This function takes a communicator, a pointer to the data, and a count of data elements as arguments. When called, it broadcasts the data to all processes within that communicator. Broadcasting data to the same row as yourself is done by calling `MPI_Bcast` with the row communicator.

To broadcast to columns we use the column communicator instead of the row communicator.

Since the `MPI_Bcast` function is collective, it needs to be called in every process within the current communicator. This implies that each process needs to know a priori from which node it will receive the next broadcast. In our application we have a function dedicated to calculate this. This function bases the calculation on which global pass the process is currently in, and which type it currently is (*pivot, normalize, eliminate* or *reduce*). This method is fairly complicated, but can be briefly explained as follows: The *normalize* nodes will always receive a broadcast from the *pivot* node, which

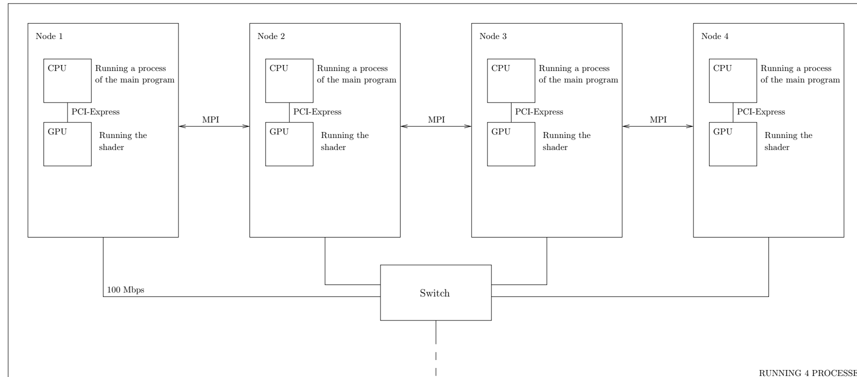

Figure A.4: Overview of physical setup of nodes.

is the diagonal element in its row communicator. *Eliminate* is similar, but will receive from the diagonal element in its column communicator. Finally, *reduce* will receive data from *normalize*, which is the node with the same column index as the current node, and the same row index as the current *pivot* node. *Reduce* also receives data from *eliminate*, which is computed in a similar fashion.

To facilitate the communication needed by our algorithm, row- and column-wise communicators are set up. Listing A.2 shows the code used to create these communicators. In this listing, the array sent as the second parameter sets which dimension we wish to keep in the new communicators. When we create the row communicators we keep the y-dimension intact, and when creating the column communicators we do the opposite and keep the x-dimension. When the code is executed, each process will set up a row communicator called `rowcom` and a column communicator called `colcom` relative to the process' location in the grid.

A.4 Results

The cluster which we benchmarked our application on consists of four one-CPU, one-GPU nodes as shown in Figure A.4. The nodes were all equipped were Intel Pentium 4 processors with Hyper-Threading Technology (HTT) and 2 GB of RAM. All nodes had an NVIDIA GeForce 7800 GT graphics adapter on a PCI-Express 16× slot.

A.4.1 Benchmark

Benchmarking of our algorithms showed that it gives sufficiently accurate results considering that all computation is executed on single precision hardware.

When benchmarking the algorithm, we have varied several variables to identify possible bottlenecks. The variables we have varied are:

1. Number of nodes.
2. Number of processes.

Table A.1: Variation of the subsize parameter, as well as the impact of several nodes. The number of processes is 4, and the times are in seconds.

-		Nodes			
n	Subsize	1	2	3	4
128	8	0,20607	0,14006	0,13756	0,28482
	16	0,23247	0,11918	0,14593	0,28739
	32	0,19208	0,10213	0,11030	0,27609
	64	0,13572	0,09238	0,08232	0,24506
512	32	0,54457	0,25811	0,28454	1,11726
	64	0,49388	0,24161	0,26360	0,78500
	128	0,32307	0,23648	0,24194	0,64518
	256	0,24012	0,20242	0,21688	0,36138
2048	128	3,17257	2,43952	2,95311	3,19620
	256	3,07729	2,43028	2,95513	3,15248
	512	2,88925	2,41467	2,87859	3,05907
	1024	2,59612	2,39955	2,68849	2,93310
4096	256	13,76410	13,03520	14,77890	15,26550
	512	13,70820	13,18710	14,74090	15,78910
	1024	13,59550	13,59640	14,73430	16,26440
	2048	14,62520	14,45370	14,50600	16,66760

3. The size of the block to factorize in each global pass (subsize).
4. The total size of the problem matrix (n).

In addition, we have benchmarked the pivot operation on a single node executed on the full matrix, as well as only network communication. This gives us performance results for our network setup, the local algorithm, as well as the global algorithm, enabling analysis of the limiting factor.

Table A.1 shows the time used to compute the PLU factorization while varying the number of nodes, size of the matrix, and the block size. The maximum achieved performance is 3.5 GFLOPS (for $n = 4096$ on two nodes), and the general trend seems to suggest that using only two nodes is faster than using four. This can somewhat be explained by interprocess communication being faster with two processes per node, than one process per node, as this eliminates a lot of network communication.

Table A.2 shows the time used to compute the PLU factorization while varying the number of processes on four nodes. As the table shows, the speed of the algorithm can be greatly influenced by tuning this parameter. However, the optimal number of processes seems to vary with the size of the matrix. The maximum achieved performance achieved was now increased to 4.2 GFLOPS (16 processes on four nodes). We also timed the network-communication, and measured the percentage of the total time used for network communication. The percentages show that there is a substantial time used to send and receive data alone.

To analyze the impact of the network, we ran the network communication while varying the number of nodes. Table A.3 shows the time of the network

Table A.2: Variation of the number of processes. The number of nodes is four, and the times are in seconds.

Procs	Subsize	Time	Network time %
4	256	97,78950	42
	512	98,19350	36
	1024	99,65560	31
	2048	102,98800	30
16	256	86,30310	37
	512	88,69330	35
	1024	89,53110	35
	2048	95,1656	33
64	128	122,72900	24
	256	124,48000	23
	512	120,79000	23
	1024	124,32000	21

Table A.3: The time spent transmitting data. The number of processes is four and the problem size is 2048, while the number of nodes is varied. This shows the impact of the network communication.

-	Nodes			
Subsize	1	2	3	4
128	0,57228	3,18597	5,70082	6,30781
256	0,59500	3,14385	5,72201	5,73072
512	0,62175	3,13140	5,64543	5,65009
1024	0,69741	3,02938	5,37086	5,38025

Table A.4: The time spent computing using only a single node where subsize = n. The times are in seconds.

n	Time
64	0,0284489
256	0,0491337
1024	0,280545
2048	1,44955
4096	10,051

communication, and the impact of the subsize parameter, as well as the use of multiple nodes. The subsize parameter seems to have little effect on the time, whilst the number of nodes has a massive impact. Using two nodes with four processes is approximately half as expensive as using four nodes.

Finally, we have benchmarked the pivot operation on one node. This is the most computationally heavy operation, and a limiting factor. Table A.4 shows the time spent to compute a full matrix using the pivot operation. The peak performance was measured for the largest matrix, 4096×4096, where the algorithm performed 4.6 GFLOPS. As a comparison, we timed the ATLAS implementation used in MATLAB, which achieved 3.5 GFLOPS on the same problem size.

A.4.2 Analysis

Our global algorithm had a maximum measured performance of 4.2 GFLOPS using four nodes, while our local algorithm showed a promising 4.6 GFLOPS. The network communication could account for at least 20% of the total runtime. However, because of the way the presented algorithm is executed, most of the processes simply idle, waiting for data. This is the largest bottleneck, but there are some solutions.

Using a look-ahead strategy, as used in the HPL [PWDC] algorithm, will increase the workload per node, and decrease the idling. In addition, restructuring the computation into smaller parts, so that pivot, eliminate, normalize and reduce are split into smaller subproblems, will also decrease the time spent idling per node.

We have not been able to show the full potential of this algorithm, because we have only have had four nodes at disposal. Having only four nodes makes almost all the computation execute serially, because we only have one node per operation at each global time-step. This parallelizes the computation of normalize and eliminate only. Using more nodes, will parallelize the reduction step of the algorithm as well, and probably speed up the total computational speed.

A.5 Conclusions and further research

We have presented a new way of computing the PLU factorization of a matrix, by using the GPU on a cluster of nodes. We have shown that the algorithms computed locally are efficient, even outperforming ATLAS.

Our global algorithm, however, is less efficient. We have pointed to a slow network link, a lot of idling of nodes, and the use of only four nodes as the main reasons.

A faster network link will decrease the impact of the network communication in our algorithm. It is also possible to lessen the issue with idling of nodes by using techniques such as look-ahead, or splitting up the computation further.

It is possible to extend our algorithm to include forward and backward substitution, as the HPL algorithm does. The computation of the forward substitution will be virtually free, while the backward substitution will require more global passes. Including the forward and backward substitution in the algorithm will fulfill the complexity demands for the Top500 benchmark [UUN].

A.6 Acknowledgements

We would like to thank J. Hjelmervik for first proposing this project to us, and our supervisors K.-A. Lie and T. R. Hagen for their helpful guiding and notes on our white paper. We would also like to thank G. W. Ma for all his assistance, and our fellow master students at SINTEF ICT for insightful discussions.

Appendix B

Shader programs

Listings B.1 and B.2 shows the two shaders in the composite scheme used to simulate the shallow-water equations. Listing B.1 shows the Lax-Friedrich scheme, and Listing B.2 shows the Lax-Wendroff scheme.

```
1  [Fragment shader]

   varying vec4 texXcoord;
   varying vec4 texYcoord;
5
   uniform sampler2D Q;
   uniform float r;
   uniform float halfG;

10 vec4 f(in vec4 Q) {
     float v = Q.y/Q.x;
     float u = Q.y/Q.x;

     return vec4(Q.y, (Q.y*u + halfG*Q.x*Q.x), Q.z*u, 0.0);
15 }

   vec4 g(in vec4 Q) {
     float v=Q.z/Q.x;

20   return vec4(Q.z, Q.y*v, (Q.z*v + halfG*Q.x*Q.x), 0.0);
   }

   void main(void) {
    // extract the values needed in the scheme from the input texture
25   vec4 QE = texture2D(Q, texXcoord.wx);
     vec4 QW = texture2D(Q, texXcoord.zx);
     vec4 QN = texture2D(Q, texYcoord.xw);
     vec4 QS = texture2D(Q, texYcoord.xz);

30   gl_FragColor = 0.25*(QE+QW+QN+QS)
                  - 0.5*r*(f(QE)-f(QW))
                  - 0.5*r*(g(QN)-g(QS));
   }

35 [Vertex shader]

   varying vec4 texXcoord;
   varying vec4 texYcoord;
   uniform vec2 dXY;
40
   void main(void) {
     texXcoord = gl_MultiTexCoord0.yxxx + vec4(0.0, 0.0, -1.0, 1.0) * dXY.x;
     texYcoord = gl_MultiTexCoord0.xyyy + vec4(0.0, 0.0, -1.0, 1.0) * dXY.y;

45   gl_Position = ftransform();
   }
```

Listing B.1: The Lax-Friedrich scheme in GLSL.

```
[Fragment shader]

varying vec4 texXcoord;
varying vec4 texYcoord;

uniform sampler2D Q;
uniform float r;
uniform float halfG;

vec4 f(in vec4 Q) {
  float u = Q.y/Q.x;

  return vec4(Q.y, (Q.y*u + halfG*Q.x*Q.x), Q.z*u, 0.0);
}

vec4 g(in vec4 Q) {
  float v = Q.z/Q.x;

  return vec4(Q.z, Q.y*v, (Q.z*v + halfG*Q.x*Q.x), 0.0);
}

void main(void) {
  // extract the values needed in the scheme from the input texture
  vec4 QC = texture2D(Q, texXcoord.yx);
  vec4 QE = texture2D(Q, texXcoord.wx);
  vec4 QW = texture2D(Q, texXcoord.zx);
  vec4 QN = texture2D(Q, texYcoord.xw);
  vec4 QS = texture2D(Q, texYcoord.xz);

  // compute cell-averages
  vec4 F  = f(Q);
  vec4 G  = g(Q);

  // use midpoint rule on cell-edges
  vec4 QnhE = 0.5*(QE + QC) - 0.5*r*(f(QE) - F);
  vec4 QnhW = 0.5*(QC + QW) - 0.5*r*(F     - f(QW));
  vec4 QnhN = 0.5*(QN + QC) - 0.5*r*(g(QN) - G);
  vec4 QnhS = 0.5*(QC + QS) - 0.5*r*(G     - g(QS));

  gl_FragColor = Q - r*(f(QnhE) - f(QnhW)) - r*(g(QnhN) - g(QnhS));
}

[Vertex shader]

varying vec4 texXcoord;
varying vec4 texYcoord;

uniform vec2 dXY;

void main(void) {
  texXcoord = gl_MultiTexCoord0.yxxx + vec4(0.0, 0.0, -1.0, 1.0) * dXY.x;
  texYcoord = gl_MultiTexCoord0.xyyy + vec4(0.0, 0.0, -1.0, 1.0) * dXY.y;

  gl_Position = ftransform();
}
```

Listing B.2: The Lax-Wendroff scheme in GLSL.

Appendix C

PyShallows

PyShallows is a utility suitable for debugging simple shader programs. It will try to compile and run the shaders, and then either display an OpenGL object that these shaders are used on, or display the compilation error(s) if one or both shaders should fail to compile.

Figure C.1 shows how PyShallows works when the shaders compiles correctly, and Figure C.2 shows what happens when a shader fails to compile.

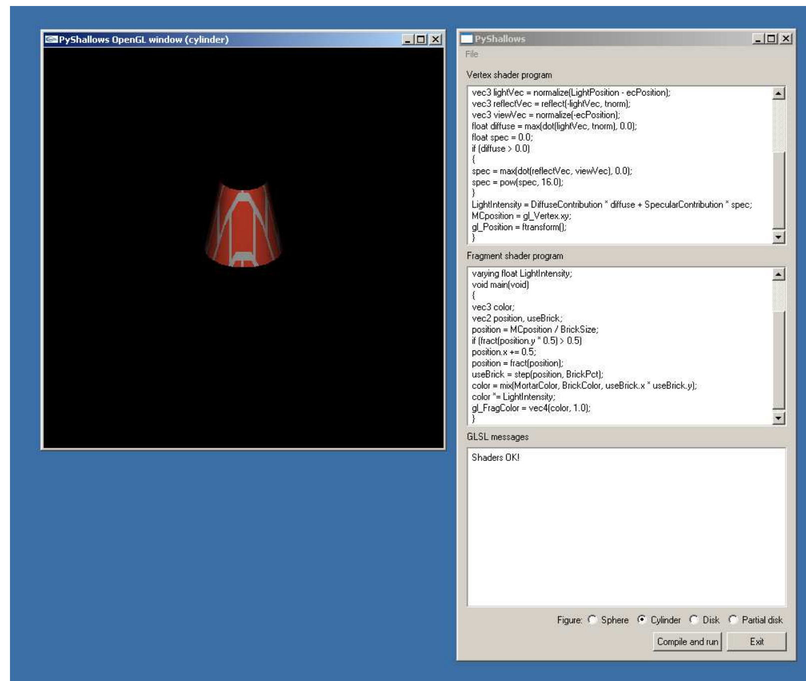

Figure C.1: Screenshot of PyShallows in action 1. The brick-shaders compiles correctly, and are used on the chosen OpenGL object.

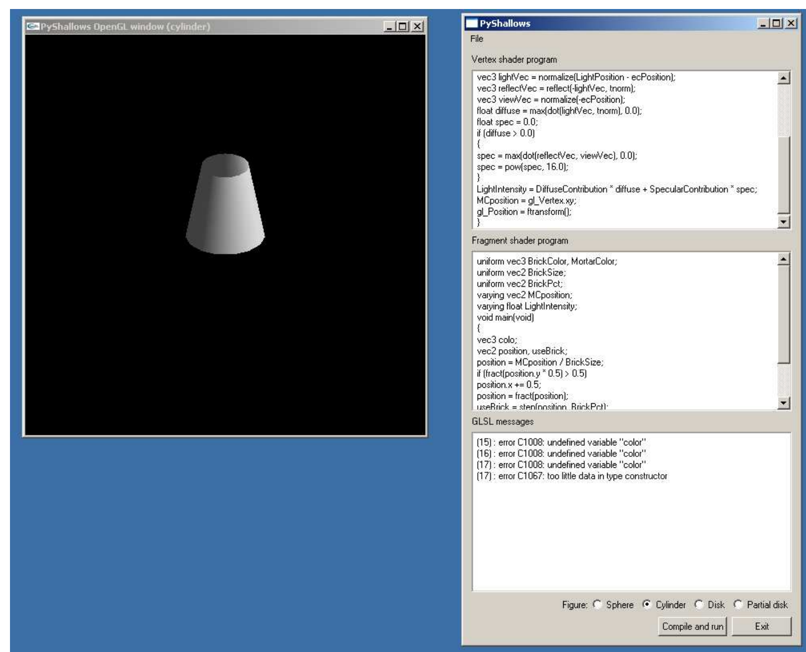

Figure C.2: Screenshot of PyShallows in action 2. The brick-shaders fail to compile, and the compilation error(s) are displayed.

Bibliography

[App05] Apple Computer. Opengl shader builder / profiler. Online; http://developer.apple.com/graphicsimaging/opengl/, 2005. [accessed 2007-04-18].

[Arga] Argonne National Lab. DeinoMPI. Online; http://mpi.deino.net/. [accessed 2007-07-11].

[Argb] Argonne National Laboratory. Jumpshot. Online; http://www-unix.mcs.anl.gov/perfvis/. [accessed 2007-04-18].

[Argc] Argonne National Laboratory. MPICH2. Online; http://www-unix.mcs.anl.gov/mpi/mpich2/. [accessed 2007-04-18].

[Argd] Argonne National Laboratory. Upshot. Online; http://www-fp.mcs.anl.gov/~lusk/upshot/. [accessed 2007-04-18].

[ATIa] ATI. CrossFire. Online; http://ati.amd.com/technology/crossfire/index.html. [accessed 2007-06-21].

[ATIb] ATI. Torrenza. Online; http://enterprise.amd.com/us-en/AMD-Business/Technology-Home/Torrenza.a%spx. [accessed 2007-06-12].

[Auta] Autodesk. 3d studio max. Online; http://www.autodesk.com/3dsmax. [accessed 2007-06-11].

[Autb] Autodesk. Maya. Online; http://www.autodesk.com/maya. [accessed 2007-06-11].

[Bax05] B. Baxter. The image debugger. Online; http://www.billbaxter.com/projects/imdebug, 2005. [accessed 2007-04-18].

[BDV94] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[Beo] Beowulf.org. Beowulf faq. Online; `http://www.beowulf.org/overview/faq.html`. [accessed 2007-04-18].

[BFH04a] I. Buck, K. Fatahalian, and P. Hanrahan. Gpubench: Evaluating gpu performance for numerical and scientific applications. In *ACM Workshop on General-Purpose Computing on Graphics Processors*, pages C–20, August 2004.

[BFH+04b] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23:777–786, August 2004. Special Issue: Proceedings of the 2004 SIGGRAPH Conference.

[Bly06] D. Blythe. The Direct3D 10 system. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 724–734, New York, NY, USA, 2006. ACM Press.

[Boo] Boost.org. Boost c++ libraries. Online; `http://www.boost.org`. [accessed 2007-04-18].

[Cen] Ohio Supercomputer Center. XMPI. Online; `http://www.lam-mpi.org/software/xmpi/`. [accessed 2007-07-11].

[CES71] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[Com] Computer Base. Xfx GeForce 7900 GS 480M extreme. Online; `http://www.computerbase.de/artikel/hardware/grafikkarten/2006/test_xfx_%geforce_7900_gs_480m_extreme/drucken/`. [accessed 2007-07-02].

[DH01] C. Ding and Y. He. A ghost cell expansion method for reducing communications in solving PDE problems. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 50–50, New York, NY, USA, 2001. ACM Press.

[Fly66] M. J. Flynn. Very high-speed computing systems. In *Proceedings of the IEEE*, volume 54, pages 1901–1909, December 1966.

[FQKYS04] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pages 47–47, November 2004.

[Gee05] D. Geer. Taking the graphics processor beyond graphics. In *Computer*, volume 38, pages 14–16, 2005.

[GFB+04] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[GGHM05] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

[GKP96] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. Pvm and mpi: a comparison of features. *Calculateurs Paralleles*, 8, 1996.

[GLT99] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2.* Massachusetts Institute of Technology, 1999.

[Gra05] Graphic Remedy. gdebugger. Online; `http://www.gremedy.com`, 2005. [accessed 2007-04-18].

[Göd06] Dominik Göddeke. Gpgpu::fast transfer tutorial. Online; `http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial3.html`, 2006. [accessed 2007-06-25].

[Har97] A. Harten. High resolution schemes for hyperbolic conservation laws. *J. Comput. Phys.*, 135(2):260–278, 1997.

[HF] M. T. Heath and J. E. Finger. ParaGraph. Online; `http://www.csar.uiuc.edu/software/paragraph/`. [accessed 2007-07-11].

[HHHL07] T. R. Hagen, M. O. Henriksen, J. M. Hjelmervik, and K.-A. Lie. *How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational engine*, pages 211–264. Springer, 2007.

[HLN06] T. R. Hagen, K.-A. Lie, and J. R. Natvig. Solving the euler equations on graphics processing units. In *International Conference on Computational Science*, pages 220–227, 2006.

[HOHL] H. Hanche-Olsen, H. Holden, and K.-A. Lie. Conservation laws preprint server. Online; `http://www.math.ntnu.no/conservation/`. [accessed 2007-06-13].

[IBM] IBM. Cell broadband engine architecture. Online; http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA277%6387257060006E61BA/$file/CBEA_v1.01_3Oct2006.pdf. [accessed 2007-07-12].

[IBT] IBTA. Infinibandta.org. Online; http://www.infinibandta.org. [accessed 2007-04-18].

[KAN] KANDU. The gathering. Online; http://www.gathering.org. [accessed 2007-05-01].

[KNP01] A. Kurganov, S. Noelle, and G. Petrova. Semidiscrete central-upwind schemes for hyperbolic conservation laws and hamilton–jacobi equations. *SIAM J. Sci. Comput.*, 23(3):707–740, 2001.

[Kom] Komplett.no. komplett.no. Online; http://www.komplett.no. [accessed 2007-04-18].

[Law] Lawrence Livermore National Laboratory. pyMPI. Online; http://pympi.sourceforge.net/index.html. [accessed 2007-07-11].

[LeV02] R. J. LeVeque. *Finite Volume Methods for Hyperbolic Problems.* Cambridge Texts in Applied Mathematics, Cambridge University Press, 2002.

[LSK+06] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25:60–99, January 2006.

[MGAK03] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics archive*, 22:896–907, July 2003. Special issue: Proceedings of ACM SIGGRAPH 2003.

[MHJ91] A. D. Malony, D. H. Hammerslag, and D. J. Jablonowski. Traceview: A trace visualization tool. *IEEE Softw.*, 8(5):19–28, 1991.

[Mica] Microsoft. About microsoft MPI. Online; http://msdn2.microsoft.com/en-us/library/bb524831.aspx. [accessed 2007-07-11].

[Micb] Microsoft. Microsoft windows compute cluster server. Online; http://www.microsoft.com/windowsserver2003/ccs/default.mspx. [accessed 2007-04-18].

[Mic05a] Microsoft. Microsoft high-level shading language. Online; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/direct%x9_c/directx/graphics/reference/hlslreference/hlslreference.asp, 2005. [accessed 2007-04-18].

[Mic05b] Microsoft. Microsoft visual studio. Online; http://msdn.microsoft.com/vstudio/, 2005. [accessed 2007-04-18].

[Mic05c] Microsoft. Shader debugger. Online; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/direct%x9_c/directx/graphics/Tools/ShaderDebugger.asp, 2005. [accessed 2007-04-18].

[Mic07] Microsoft. Microsoft DirectX. Online; http://www.microsoft.com/directx, 2007. [accessed 2007-04-25].

[Moo00] G. E. Moore. *Cramming more components onto integrated circuits*, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[Mor03] A. Moravánszky. Dense matrix algebra on the GPU. Online; http://www.shaderx2.com/shaderx.pdf, 2003. [accessed 2006-05-11].

[MPI] MPI Forum. Mpi documents. Online; http://www.mpi-forum.org/docs/docs.html. [accessed 2007-04-18].

[NAS] NASA Advanced Supercomputing Division. AIMS. Online; http://www.nas.nasa.gov/Resources/Software/swdescriptions.html#AIMS. [accessed 2007-04-18].

[Neo07] Neoptica. Programmable graphics – the future of interactive rendering. Online; http://www.neoptica.com/NeopticaWhitepaper.pdf, 2007. [accessed 2007-04-23].

[NVIa] NVIDIA. Cuda. Online; http://developer.nvidia.com/object/cuda.html. [accessed 2007-06-12].

[NVIb] NVIDIA. Geforce 8 series. Online; http://www.nvidia.com/page/geforce8.html. [accessed 2007-07-02].

[NVIc] NVIDIA. Nvidia tesla. Online; http://www.nvidia.com/object/tesla_computing_solutions.html. [accessed 2007-07-02].

[NVId] NVIDIA. SLIZone Home. Online; http://www.slizone.com/page/home.html. [accessed 2007-06-21].

[Oak] Oak Ridge National Laboratory. XPVM. Online; `http://www.netlib.org/utk/icl/xpvm/xpvm.html`. [accessed 2007-07-11].

[oI] University of Illinois. Pablo. Online; `http://wotug.kent.ac.uk/parallel/performance/tools/pablo/`. [accessed 2007-07-11].

[OLG+05] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, August 2005.

[OLG+07] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[Pac97] P. S. Pacheco. *Parallel programming with MPI.* Morgan Kaufmann Publishers, Inc., 1997.

[PB03] A. J. Preetham and A. Bleiweiss. Ashli - advanced shading language interface. In *Eurographics 2003*, 2003.

[Pea] Peakstream Inc. Peakstream. Online; `http://www.peakstreaminc.com/`. [accessed 2007-05-03].

[Pie94] P. Pierce. The nx message passing interface. *Parallel Computing*, 20(4):463–480, 1994.

[PS03] T. J. Purcell and P. Sen. Shadesmith fragment program debugger. Online; `http://graphics.stanford.edu/projects/shadesmith`, 2003. [accessed 2007-04-18].

[PWDC] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. Hpl. `http://www.netlib.org/benchmark/hpl/`. [accessed 2007-04-18].

[RH04] C. Z. Rojas and M. Hoemmen. Communication savings with ghost cell expansion for domain decompositions of finite difference grids. May 2004.

[Ros06] R. J. Rost. *OpenGL® Shading Language, Second Edition.* Addison Wesley Professional, January 2006.

[SHF06] S. Sharma, C.-H. Hsu, and W.-C. Feng. Making a case for a green500 list. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006)/ Workshop on High Performance - Power Aware Computing*, 2006.

[Shu88] C.-W. Shu. Total-variation-dimishing time discretisations. *Sci. Stat. Comput.*, 9:1073–1084, 1988.

[SIN05] SINTEF. Shallows. Online; http://sourceforge.net/projects/shallows, 2005. [accessed 2007-04-18].

[SWND05] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide, Fifth Edition, The Official Guide to Learning OpenGL, Version 2*. Addison-Wesley, 2005.

[The07a] The Khronos Group. OpenGL Architecture Review Board. Online; http://www.opengl.org/about/arb/, 2007. [accessed 2007-06-11].

[The07b] The Khronos Group. The Khronos Group, open standards, royaltee free, dynamic media technologies. Online; http://www.khronos.org/, 2007. [accessed 2007-06-11].

[Tre05] D. Trebilco. GLIntercept. Online; http://glintercept.nutty.org, 2005. [accessed 2007-04-18].

[Ups89] S. Upstill. *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[UUN] University of Mannheim, University of Tennessee, and NERSC/LBNL. Top 500 supercomputer sites. Online; http://www.top500.org/about. [accessed 2007-04-18].

[Van82] M. Van Dyke. Album of fluid motion, 1982. [accessed 2007-06-13].

[Wik07a] Wikipedia. ATI CrossFire — wikipedia, the free encyclopedia, 2007. [Online; accessed 21-June-2007].

[Wik07b] Wikipedia. Scalable Link Interface — wikipedia, the free encyclopedia, 2007. [Online; accessed 21-June-2007].

[Wik07c] Wikipedia. Schlieren photography — wikipedia, the free encyclopedia, 2007. [Online; accessed 10-July-2007].

[Wik07d] Wikipedia. Wave equation — wikipedia, the free encyclopedia, 2007. [Online; accessed 21-June-2007].

[XFX] XFX. GeForce 8800 ultra 768MB DDR3 RoHS dual DVI extreme. Online; http://xfxforce.com/web/product/listConfigurations.jspa?seriesId=730995%&productId=1085635. [accessed 2007-05-23].