Shallow Water Simulations on Graphics Hardware

Martin Lilleeng Sætra
2014

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1524

ISSN 1501-7710

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Akademika Publishing.
The thesis is produced by Akademika Publishing merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

Abstract: Conservation laws describing one or more conserved quantities in time arise in a multitude of different scientific areas. Mathematically, conservation laws are expressed as partial differential equations (PDEs). In this thesis, the shallow water equations are the particular system of interest, and flood simulations the main application area. There are several numerical methods for approximating the solution of hyperbolic PDEs like the shallow water equations, and finite volume methods constitute an important class. Explicit finite volume methods typically rely on stencil computations, making them inherently parallel, and therefore a near perfect match for the many-core graphics processing unit (GPU) found on today's graphics cards. The GPU is one example of the accelerators now used in high performance computing. Accelerators are typically power efficient, and deliver higher computational performance per dollar than traditional CPUs. Through the scientific papers in this thesis, we present efficient hardware-adapted shallow water simulations on the GPU, based on a high-resolution central-upwind scheme. The topics range from best practices for stencil computations on the GPU to adaptive mesh refinement. This work extends to architectures similar to the GPU and to other hyperbolic conservation laws.

Contents

Preface

This thesis is submitted in partial fulfilment of the requirements for the degree of Philosophiae Doctor (Ph.D.) at the Faculty of Mathematics and Natural Sciences, University of Oslo. The work has been performed as part of the Parallel3D project (Research Council of Norway (RCN) grant number 180023/S10, 80% funded by the RCN and 20% funded by Kongsberg Oil & Gas Technologies as an industry partner), partially funded by the Norwegian Meteorological Institute (MET Norway). A three month period as a visiting scholar was partially funded by the National Center for Computational Hydroscience and Engineering (NCCHE).

The thesis consists of two parts. Part I contains three chapters: Chapter 1 gives an introduction to the scientific fields and topics the work in this thesis is based on; Chapter 2 summarizes and comments the scientific papers in Part II; and Chapter 3 offers some concluding remarks and outlooks. Part II contains the published and submitted scientific papers that constitute the main research contribution.

Acknowledgements This thesis is the outcome of four years of work and study, mainly conducted at SINTEF ICT in Oslo. I conducted the research for this thesis as a member of the Heterogeneous Computing research group in the Department of Applied Mathematics, while employed by the Centre of Mathematics for Applications at the University of Oslo. I want to thank both the University of Oslo and SINTEF ICT for providing me with a productive frame for my work. Furthermore, I thank MET Norway for financial support, and NCCHE for an exceptional stay as a visiting scholar in Oxford, Mississippi.

A great many people have helped me along the way, and I want to especially mention my supervisors, Christopher Dyken, Knut-Andreas Lie, and Eigil Samset, and my good friend and room-mate for most of the period, André Brodtkorb. This thesis would not have existed without their ideas, support, and enthusiasm. I also want to thank my co-authors, Mustafa Altinakar, André Brodtkorb, Trond Runar Hagen, and Knut-Andreas Lie, for good collaborations. To every member of the Heterogeneous Computing group; I would like to say thank you for insightful discussions, good collaborations, and excellent foosball games, I really appreciated working with you all. I hope to be able to cooperate with both my co-authors and my colleagues at SINTEF ICT again in the near future. Last, but in no way least, I would like to thank my girlfriend, Mari, who had to endure periods of long hours and stress, for always being there for me, and my family for always supporting me.

January 2014
Martin Lilleeng Sætra

Part I
Introduction

Chapter 1

Background

The aim of this thesis is to develop state-of-the-art algorithms and techniques for flood simulation on modern many-core architectures. To this end, numerical simulations of the shallow water equations, a system of hyperbolic partial differential equations (PDEs), is employed. More specifically, we explore how high-resolution explicit schemes can be efficiently implemented on graphics hardware, based on parallelization of stencil computations. The shallow water equations can be used to describe many naturally occurring phenomena, e.g., tsunamis, dam breaks, and storm surges. Physically correct simulations of such real-world phenomena typically require large domains and are computationally demanding. By utilizing graphics processing units (GPUs) and hardware-adapted algorithms, we are able to efficiently solve the shallow water equations on very large domains.

Modern graphics hardware has been shaped through the last decades to perform one of the most computationally intensive tasks on a standard desktop or laptop computer, namely updating the color of each pixel on the computer screen many times per second. This is an embarrassingly parallel task, which maps very well to stencil computations. The computational power of the GPU has been increasingly harnessed to solve general problems, starting in the early 2000s, and an ecosystem of languages, libraries, and tools for GPU computing has evolved. Today, GPUs are found in nearly every desktop and laptop computer, and they are used in supercomputers to accelerate computations.

The contributions of the thesis include efficient shallow water simulations, investigation of parallel and heterogeneous architectures, and hardware-adapted algorithms for these emerging architectures. We have utilized GPUs, but the algorithms presented herein should be transferable to other similar architectures, e.g., the Intel Xeon Phi[1]. Furthermore, we believe the algorithms to be applicable to other hyperbolic conservation laws and other numerical schemes. This thesis covers several topics, in which computer science and scientific computing are the main scientific fields, supplemented by numerical analysis and hydrology.

The aim of this chapter is to give the reader insight into the scientific fields, topics, and technologies that are used in the six papers that constitute the core of the thesis. Section 1 places the research contribution of this thesis, and its application, into a practical real-world context. Section 2 gives an overview of conservation laws in general, and Section 3 discusses the shallow water equations in particular. Section 4 introduces heterogeneous computing and

[1]The programming model of the Xeon Phi differs from that of the GPU, and hence requires a reimplementation. However, the parallel algorithms and techniques should be directly transferable.

offers an explanation to why we have seen a shift from serial computing on CPUs to parallel computing on both CPUs and novel computing architectures. While GPU computing is a form of heterogeneous computing, Section 5 is dedicated to GPU computing in its whole, as it is an integral part of every scientific paper in this thesis.

1 Simulating Real-World Phenomena

The phenomena we are simulating in this thesis are costing lives, as well as enormous sums in property damage, every year. Figure 1 shows two examples of large disasters; the Indian Ocean tsunami and the Vajont dam overtopping. According to statistics from the UN Office for Disaster Risk Reduction [54], in the period 1980–2008, the combined human casualties from floods and tsunamis were 425,394. The combined economic damage were a staggering 407.38 billion USD. In fact, freshwater floods alone caused 175,000 deaths and affected more than 2.2 billion persons worldwide in a total of 1816 events in the period 1975–2001 [23]. Furthermore, more than a billion people live under the constant threat of these devastating phenomena [1]. In an effort to mitigate this we turn to numerical simulations, in which a mathematical model outputs water height and wave speed for discrete points in space and time. The value of such numerical simulations rests on two key factors: The phenomena must be simulated *accurately enough*, and *fast enough*. Fast enough means that simulation results must be available at such a time that it is possible to act upon them, i.e., initiate emergency plans, inform the public, start evacuations, etc. By utilizing the GPU, our code has in Paper III been proven capable of efficient shallow water simulations, also for many real-world problems, which could be used in the operative phase of an emergency to help the decision making process. The capacity and efficiency of the code has later been further increased through multi-GPU simulations, the sparse domain algorithms[2], and adaptive mesh refinement (AMR), in Papers III, V, and VI, respectively.

Which physical property that is most important to capture will depend on the class of problem. For a dam break it will be important to accurately predict the wave-front arrival time, and the distribution of the water. For a flash flood the peak water height is typically very important. The simulator's ability to accurately capture these values depends on many factors, e.g., the given bottom topography, the initial conditions, the grid resolution, the numerical scheme, and the hardware. It is also important to understand which factor or factors that are limiting accuracy, e.g., if the limiting factor is the numerical scheme, it will not help to use double-precision instead of single-precision floating-point numbers [6]. A strategy used in the development of our shallow water simulator has therefore been to identify the current limiting factor, try to mitigate it, then find the next limiting factor, and continue until the code validates within the required degree of accuracy. Results from verification and validation of our code is given in Paper III.

It is not just when disaster has struck it is essential with accurate simulations; it is also important for planning ahead. This is something humankind has proven to be surprisingly bad at, and for each 100 USD used in post-disaster emergency help, only 1 USD is used for pre-disaster preparedness [1]. While tsunamis and floods are powerful destructive forces, water is also one of the most precious resources on the planet, and hence, river banks, islands, and coastal regions tend to be heavily populated. The pressure to live and work in these flood-prone

[2]Sparse domain methods for only representing and computing "wet" parts of the domain.

(a) The 2004 Indian Ocean tsunami.

(b) The 1959 Vajont Dam overtopping.

Figure 1: (a) The 2004 Indian Ocean tsunami spanned a large portion of the globe, claiming a total of 230,000 lives. (b) The Vajont Dam overtopping in 1959 flushed away several Italian villages in the Piave valley below, claiming 2000 lives in the process. (Picture (a) courtesy of David Rydevik, and picture (b) is in the public domain.)

areas, which typically feature attractive rich soils, abundant water supplies and ease of transport, will only increase as the world's population continues spiraling upward to a projected 10 billion by 2050. People will always want to live near water, making simulation of potential disaster scenarios and creation of emergency plans highly important, and the backlog is large. In the US alone, there are 84,000 registered dams. More than 25,000 of these dams are classified as significant or high hazard dams, meaning that life and property would be in danger if the dams were to fail, yet only half of them have an emergency action plan in place [2]. As the climate becomes more volatile and the population increases further, it will become even more important with readily available and sufficiently accurate data in the future.

Improved computational capacity due to parallel and heterogeneous architectures will, of course, not only benefit shallow water simulations. There are many scientific areas in which numerical simulations will allow for simulating and forecasting potential disasters and saving lives, e.g., earthquakes, volcanoes, diseases (through molecular dynamics simulations, e.g., Folding@home [28]), pollution, toxic spills, astrophysical catastrophes, etc. It is not without reason that Peter Lax compared the role of “fast computers with large memories” in mathematics to the “role of the telescopes in astronomy and microscopes in biology” [29]. The work presented in this thesis is not only applicable to shallow water simulations, but is also valuable to advance the above mentioned efforts.

2 Solving Conservation Laws Numerically

Before going in detail on the shallow water equations, we will give some background on conservation laws in general. A conservation law states that a particular property within an isolated system does not change as the system evolves in time. The shallow water equations are based on two basic conservation laws: The first, conservation of momentum, is an exact conservation law. The second, conservation of mass, is an approximate conservation law, which is true under certain conditions that are met in systems under our consideration, namely non-relativistic speed and no nuclear reactions. Mathematically, we define a conservation law as PDEs or systems of PDEs that in one spatial dimension can be written on the form:

$$Q_t + F(Q)_x = 0, \tag{1}$$

in which

$$Q = \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_m \end{bmatrix}, F(Q) = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix}. \tag{2}$$

Here, Q is the vector of conserved variables, $F(Q)$ is the vector of *fluxes*, and each of its components f_i is a function of the components q_j of Q. Some of these PDEs and systems of PDEs may be solved analytically, usually under given initial conditions and boundary conditions. However, analytical solutions will not be treated in this thesis, except for verification purposes.

Hyperbolic PDEs usually arise in connection with mechanical oscillators, or in convection-driven transport problems, e.g., the linear wave equation. A great many of the equations found in mechanics are hyperbolic, and although water waves and fluid dynamics are the particular problems class in this thesis, hyperbolic PDEs are also used to model, e.g., acoustic waves and electromagnetic waves. An important feature of hyperbolic PDEs is their ability to support discontinuities in the solution, such as shock waves [52], which is a necessity when modeling the types of problems we investigate. Which methods that are best suited for solving or approximating a solution to a particular PDE depends largely on the class of the PDE. Hyperbolic PDEs are generally challenging to solve, particularly in nonlinear cases, and while implicit schemes certainly are applicable, *explicit time stepping schemes* are frequently used[3]. Using explicit methods, the solution at time t^{n+1} is explicitly computed from the previous solution at time t^n. The solution of hyperbolic PDEs can be approximated with explicit methods, enabled by the fact that the solution has a finite speed of propagation, as opposed to elliptic and parabolic PDEs, in which a change in one point in space instantly influences all other points in space. Furthermore, as we will see, this class of schemes yield excellent performance on the type of hardware architecture we are using; the massively parallel GPU.

Explicit schemes for solving hyperbolic PDEs typically rely on stencil computations. By imposing a 1D uniform grid on the computational domain, the solution is discretized in space and given for each *grid point*, $x_0, x_1, ..., x_N$, with an equal distance, Δx, between grid points (see Figure 2). The grid point values from time t^n used to compute one grid point value at time t^{n+1} is known as the stencil. To use as an example we introduce a first-order hyperbolic PDE; the advection equation in 1D:

$$\frac{\partial q}{\partial t} + a\frac{\partial q}{\partial x} = 0, \tag{3}$$

in which a is the velocity. This equation describes the transport of a conserved quantity by movement of a mass of fluid, e.g., ink added at some point in a river, advected downstream.

[3]A benefit of implicit schemes is that they in principle allow for large time steps without becoming numerically unstable. Unfortunately, large time steps lead to inaccurate solutions, and requirements on accuracy may limit the time step size to a range in which explicit schemes are less expensive. While this argument is true for systems with wave speeds in the same order of magnitude, explicit schemes may not be practical if wave speeds are largely different.

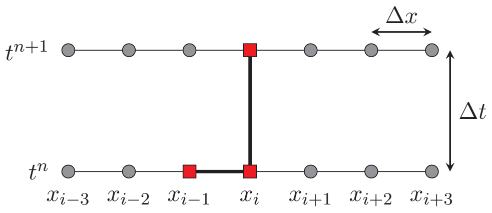

Δx
t^{n+1}
Δt
t^n
x_{i-3} x_{i-2} x_{i-1} x_i x_{i+1} x_{i+2} x_{i+3}

Figure 2: The space-time stencil used for the one-dimensional advection equation when $a > 0$.

There exist several methods for constructing explicit schemes, and one alternative is to use *finite differences* to approximate the derivatives of the equation. A first-order accurate *upwinding scheme* yields the following discretization:

$$\frac{q_i^{n+1} - q_i^n}{\Delta t} + a\frac{q_i^n - q_{i-1}^n}{\Delta x} = 0 \qquad \text{for} \qquad a > 0, \tag{4}$$

$$\frac{q_i^{n+1} - q_i^n}{\Delta t} + a\frac{q_{i+1}^n - q_i^n}{\Delta x} = 0 \qquad \text{for} \qquad a < 0, \tag{5}$$

in which $q_i^n = q(x_i, t^n)$ is the value of the conserved quantity at point x_i at time t^n, Δx is the spatial step size, and Δt is the temporal step size. In a more compact and general form these equations can be written as:

$$q_i^{n+1} = q_i^n - \Delta t \left(a^+ q_x^- + a^- q_x^+\right), \tag{6}$$

in which

$$a^+ = \max(a, 0), \qquad a^- = \min(a, 0), \qquad q_x^- = \frac{q_i^n - q_{i-1}^n}{\Delta x}, \qquad q_x^+ = \frac{q_{i+1}^n - q_i^n}{\Delta x}. \tag{7}$$

The stencil described by (6) is visualized in Figure 2, showing every grid point in space at time t^n involved in computing the value of grid point x_i at time t^{n+1}. (For this particular scheme, the sign of the velocity determines which of the input grid points that are used.) The scheme needs to obey the following Courant–Friedrichs–Lewy (CFL) condition to be stable:

$$\left|\frac{a\Delta t}{\Delta x}\right| \leq 1. \tag{8}$$

Using this CFL condition, the maximum size of the time step can be computed such that the numerical domain of dependence of any point in space and time includes the analytical domain of dependence.[4] By extending the grid to a 2D uniform Cartesian grid, the solution becomes a surface instead of a line. Finite difference methods extend naturally to 2D.

We will now turn to *finite volume schemes* [30], which is the type of schemes used in this thesis. Instead of grid points, we now have *grid cells* which are intervals in 1D and finite areas

[4]Obeying the CFL condition is necessary for stability when solving hyperbolic PDEs, but it does not guarantee a stable solution, e.g., in dry areas of the domain.

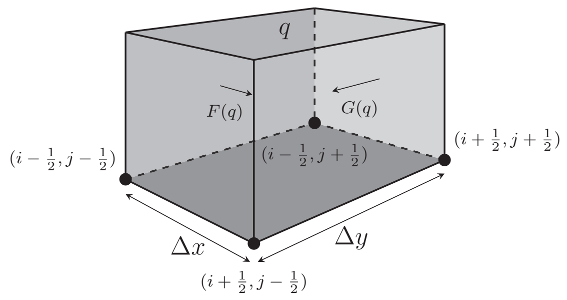

q
$F(q)$
$G(q)$
$(i-\frac{1}{2}, j-\frac{1}{2})$
$(i-\frac{1}{2}, j+\frac{1}{2})$
$(i+\frac{1}{2}, j+\frac{1}{2})$
Δx
Δy
$(i+\frac{1}{2}, j-\frac{1}{2})$

Figure 3: The changes in the conserved quantities for one finite volume from one time step to the next is determined by the net fluxes, sources, and sinks. Only two of the four fluxes into the finite volume are shown.

in 2D (see Figure 3). First, we expand (3) to 2D:

$$\frac{\partial q}{\partial t} + u\frac{\partial q}{\partial x} + v\frac{\partial q}{\partial y} = 0, \tag{9}$$

in which u and v are the velocities in the x and y direction, respectively. Each grid cell has a width, a height, and a set of conserved quantities, which each is represented by the cell average over the area of the cell. We will in the following take the cell average to be the value of q at the center of the cell. By integrating (9), we get:

$$\begin{aligned}\Delta x\Delta y\frac{d\bar{q}_{i,j}}{dt} &+ \int_{y_j-\frac{\Delta y}{2}}^{y_j+\frac{\Delta y}{2}} \Big[u(x_{i+\frac{1}{2}}, y)q(x_{i+\frac{1}{2}}, y) - u(x_{i-\frac{1}{2}}, y)q(x_{i-\frac{1}{2}}, y)\Big]\,dy \\ &+ \int_{x_i-\frac{\Delta x}{2}}^{x_i+\frac{\Delta x}{2}} \Big[v(x, y_{j+\frac{1}{2}})q(x, y_{j+\frac{1}{2}}) - v(x, y_{j-\frac{1}{2}})q(x, y_{j-\frac{1}{2}})\Big]\,dx \\ &= 0,\end{aligned} \tag{10}$$

in which $x_{i\pm\frac{1}{2}} = x_i \pm \frac{\Delta x}{2}$, $y_{j\pm\frac{1}{2}} = y_j \pm \frac{\Delta y}{2}$, and $\bar{q}_{i,j} = (\iint_{i,j} q(x,y)dx\,dy)/(\Delta x\Delta y) = q(x_i, y_j) + O(\Delta x^2, \Delta y^2)$. Finite volume schemes are similar to finite difference schemes, but the derivatives are not approximated by finite differences, instead we compute numerical fluxes, $F(q)$ and $G(q)$, going across the cell interfaces (see Figure 3). If we denote the fluxes by $(F, G) = (uq, vq)$, we can rewrite the equation as:

$$\begin{aligned}\Delta x\Delta y\frac{d\bar{q}_{i,j}}{dt} &+ \int_{y_j-\frac{\Delta y}{2}}^{y_j+\frac{\Delta y}{2}} \Big[F(x_{i+\frac{1}{2}}, y) - F(x_{i-\frac{1}{2}}, y)\Big]\,dy \\ &+ \int_{x_i-\frac{\Delta x}{2}}^{x_i+\frac{\Delta x}{2}} \Big[G(x, y_{j+\frac{1}{2}}) - G(x, y_{j-\frac{1}{2}})\Big]\,dx = 0.\end{aligned} \tag{11}$$

Then we use a simple integration scheme, the mid-point rule, and write:

$$\begin{aligned} \int_{y_j-\frac{\Delta y}{2}}^{y_j+\frac{\Delta y}{2}} F(x_{i+\frac{1}{2}}, y)dy &\approx F(x_{i+\frac{1}{2}}, y_j)\Delta y \\ \int_{x_i-\frac{\Delta x}{2}}^{x_i+\frac{\Delta x}{2}} G(x, y_{j+\frac{1}{2}})dx &\approx G(x_i, y_{j+\frac{1}{2}})\Delta x. \end{aligned} \tag{12}$$

We are now left with an ordinary partial differential equation in time:

$$\frac{d\bar{q}_{i,j}}{dt} = \frac{1}{\Delta x}\left[F(x_{i+\frac{1}{2}}, y_j) - F(x_{i-\frac{1}{2}}, y_j)\right] + \frac{1}{\Delta y}\left[G(x_i, y_{j+\frac{1}{2}}) - G(x_i, y_{j-\frac{1}{2}})\right], \tag{13}$$

in which the right-hand side can be computed explicitly:

$$\begin{aligned} F(x_{i+\frac{1}{2}}, y_j) &= u(x_{i+\frac{1}{2}}, y_j)q(x_{i+\frac{1}{2}}, y_j) \\ G(x_i, y_{j+\frac{1}{2}}) &= v(x_i, y_{j+\frac{1}{2}})q(x_i, y_{j+\frac{1}{2}}). \end{aligned} \tag{14}$$

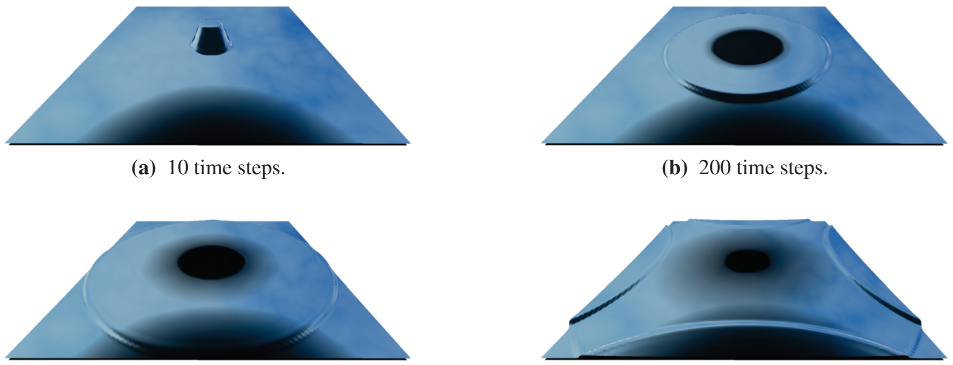
By using a piecewise linear reconstruction of q evaluated at the cell interfaces, we have the last missing piece:

$$\begin{aligned} q^{\pm}(x_{i\pm\frac{1}{2}}, y_j) &= \frac{q(x_i, y_j) + q(x_{i\pm 1}, y_j)}{2} \\ q^{\pm}(x_i, y_{j\pm\frac{1}{2}}) &= \frac{q(x_i, y_j) + q(x_i, y_{j\pm 1})}{2}. \end{aligned} \tag{15}$$

We now have two $q^{\pm}$ values for each integration point at the cell interfaces, and a numerical flux function is used to pick the correct combination of the two.

For this type of schemes we generally know the average of the conserved quantity over each cell, $\bar{q}^n_{i,j}$, at time step n. To compute the fluxes we *reconstruct* these cell averages to a piecewise polynomial function, which is evaluated at the interfaces between cells. Since the reconstruction is performed locally per cell, discontinuities may form at the cell interfaces, which must be handled by a numerical flux function. Next, we *evolve* the equations in time using some generic integration method. The net flux, and any contributions from *sources* and *sinks*, evolved in time, is the basis for new cell averages. Acceleration due to gravity, for example, is a source in the shallow water equations. Finally, we *average* the function over each grid cell again to obtain new cell averages, $\bar{q}^{n+1}_{i,j}$. This is commonly referred to as the REA algorithm [30], and how the three steps are implemented differ between the various numerical schemes. Explicit spatial discretization (see (12)–(15)) results in a compact stencil, in which each cell average $\bar{q}^{n+1}_{i,j}$ can be computed independently of all other cells. This is the inherent parallelism we will be exploiting.

For schemes that rely on stencil computations the boundary cells need special treatment. Depending on the size of the stencil two or more rows and columns along the domain boundary cannot use the same scheme as the internal cells. There are two common methods for dealing with this: A *modified scheme* may be used for the boundary cells, in which no cells outside the domain are used in the modified stencil. The second approach, which we are using in this thesis, is to add *ghost cells* around the boundary, outside the physical domain. With this approach the same scheme may be used on all internal cells, but the added ghost cells must be updated



(a) 10 time steps. **(b)** 200 time steps.

(c) 300 time steps. **(d)** 400 time steps.

Figure 4: The shallow water equations visualized using the simulator code developed throughout the work presented in this thesis. The initial condition is a round cylinder (with radius 6.5 m) of water in the middle of the domain with water elevation of 10 m, while the rest of the domain has a water elevation of 1 m, over a flat bottom topography. The domain size is 100 m^2 and the resolution is 128^2 cells. Euler time integration is used, and wall boundary condition is set for all four boundaries.

between each time step as well. Different types of *boundary conditions* may be implemented, e.g., updating the ghost cells with the same value as their interior neighbors, but with negated speeds in the normal direction, simulates a wall placed at the domain boundary.

The schemes presented in this section are simple compared to the high-resolution schemes used throughout this thesis, nevertheless, it serves as a good introduction to this type of schemes. See also Paper II, in which the classical Lax–Friedrichs finite volume scheme is mapped to the GPU using stencil computations. The interested reader should consult one of the many textbooks on hyperbolic conservation laws and finite volume methods [30, 53, 17, 13, 26, 52]. A large selection of scientific papers on the subject of conservation laws can also be found on the NTNU “Preprints on Conservation Laws” server [14].

Relevant application areas for simulating conservation laws often involves very large domains, e.g., oceans, parts of the atmosphere, and glaciers. The grids required to simulate real-world phenomena on these large scales are typically too computationally demanding to be solved on an average desktop computer. The traditional way of tackling this is by domain decomposition and solving the domain divided on nodes in a cluster or a supercomputer. Decomposing the domain presents its own set of challenges, some of which will be investigated later, particularly in Paper IV. By using a massively parallel architecture and hardware-adapted algorithms, we will herein demonstrate that real-world simulations can be performed on a standard desktop computer by parallelization of stencil computations.

3 The Shallow Water Equations

Now that conservation laws and hyperbolic PDEs have been briefly introduced, we will focus on the shallow water equations in particular, before exploring the hardware employed to solve the shallow water equations in this thesis. The shallow water equations model gravity-induced motion of a fluid, and by assuming negligible vertical density gradients and fluid accelerations, they describe the movement of a surface in two dimensions (see Figure 4). The equations are able to capture many natural occurring phenomena, in which the horizontal wavelengths are much larger than the depth, e.g., tsunamis, inundation scenarios, and some dam break cases. It may seem counterintuitive to think of for example oceans as being shallow, but it is the depth relative to the wavelengths that is the important factor. For example, if we consider the devastating 2004 tsunami in the Indian Ocean; we have depths of two to three kilometers, but wavelengths ranging from dozens to hundreds of kilometers. It is also worth noting that the shallow water equations can describe other fluids than water, and weather forecasting was in fact earlier performed with a modified version of the shallow water equations [42]. The shallow water equations cannot, however, be applied when three-dimensional effects become too large, or the waves become too short or too high. On differential form in two spatial dimensions the equations can be written:

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix}, \qquad (16)$$

in which h is the water depth and hu and hv are the discharges along the abscissa and ordinate, respectively. Furthermore, g is the gravitational constant and B is the bottom topography measured from a given datum.

The mathematician Adhémar Jean Claude Barré de Saint-Venant first formulated the shallow water equations for one-dimensional unsteady open channel flows [44], and the 1D equations are therefore also known as the Saint-Venant system. The equations are derived from the basic principles of conservation of mass and conservation of momentum. For conservation of mass we have:

$$\begin{aligned} \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 & \qquad \Big| \text{ incompressible flow} \\ \nabla \cdot \mathbf{v} = 0, & \qquad \Big| \; \mathbf{v} = (u, v, w) \end{aligned} \qquad (17)$$

in which ρ is density and $\mathbf{v}$ is the velocity vector. Conservation of momentum is described as:

$$\begin{aligned} \frac{\partial}{\partial t}(\rho \mathbf{v}) + \nabla \cdot (\rho \mathbf{v} \otimes \mathbf{v} + p - \eta) = \rho \mathbf{g} & \qquad \Big| \text{ ignore viscosity} \\ \frac{\partial}{\partial t}(\rho \mathbf{v}) + \nabla \cdot (\rho \mathbf{v} \otimes \mathbf{v} + p) = \rho \mathbf{g} & \qquad \Big| \text{ incompressible flow} \\ \frac{\partial}{\partial t}\mathbf{v} + \nabla \cdot (\mathbf{v} \otimes \mathbf{v}) + \frac{1}{\rho}\nabla \cdot p = \mathbf{g} & \qquad \Big| \\ \frac{\partial}{\partial t}\mathbf{v} + \nabla \cdot (\mathbf{v} \otimes \mathbf{v}) = -\frac{1}{\rho}\nabla \cdot p + \mathbf{g}, & \qquad \Big| \; \mathbf{g} = [0, 0, -g]^T \end{aligned} \qquad (18)$$

in which p is pressure, η is viscosity, and g is the gravitational constant. By combining (17) and

(18) we get the following system of four PDEs with four unknowns:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{19}$$

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial x} \tag{20}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial y} \tag{21}$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial z} - g. \tag{22}$$

This system of equations is solvable given appropriate initial conditions and boundary conditions. The problem is now that boundary conditions have to be applied to the surface, which we do not know the position of a priori. Through several simplifications, such as depth averaging (assuming horizontal scales are much larger than vertical scales), neglecting vertical acceleration, and assuming constant density along the z-axis, we end up with the following system of equations:

$$\frac{\partial}{\partial t}h + \frac{\partial}{\partial x}(hu) + \frac{\partial}{\partial y}(hv) = 0 \tag{23}$$

$$\frac{\partial}{\partial t}(hu) + \frac{\partial}{\partial x}(hu^2 + \frac{1}{2}gh^2) + \frac{\partial}{\partial y}(huv) = -gh\frac{\partial B}{\partial x} \tag{24}$$

$$\frac{\partial}{\partial t}(hv) + \frac{\partial}{\partial x}(huv) + \frac{\partial}{\partial y}(hv^2 + \frac{1}{2}gh^2) = -gh\frac{\partial B}{\partial y}, \tag{25}$$

which is equal to (16).

The shallow water equations are a nonlinear system, and the solution may form discontinuities. For a practical example of such a discontinuity, in the form of a single stationary shock wave; simply turn on the kitchen faucet (provided the kitchen sink surface is flat) and observe. There are generally two ways of handling discontinuities; *shock tracking* or *shock-capturing*. Shock tracking involves explicitly tracking discontinuities through the domain, while using a standard finite difference or finite volume scheme in smooth regions. This method adds considerable complexity to the numerical algorithm. Shock-capturing schemes, however, are capable of resolving discontinuities automatically without special handling. The high-resolution numerical finite volume scheme employed in this thesis is a shock-capturing scheme.

There exists a large number of publications investigating the shallow water equations, and even more with shallow water flows as the application area, see Part II and references therein. In the next section we will give background to the change from serial architectures to parallel architectures and frequency scaling to concurrency[5] scaling, before we turn to GPUs and stencil computations on GPUs in particular.

4 Heterogeneous Computing – From Serial to Parallel

Computers have made it possible to do a great number of calculations in almost no time. With the latest supercomputers performing as much as 33.862 petaFLOPS (33.862×10^{15} floating-point operations per second) [55], it may seem like we have all the computing power we will

[5]With regards to the number of cores per chip.

ever need. However, as we explore increasingly complex problems, more coupled problems like climate simulations, and use increasingly larger domains, the need for computational power is ever increasing. After the development of the integrated circuit, the number of on-chip transistors in processors has followed Moore's Law [32], formulated by Intel co-founder Gordon E. Moore in 1965. Moore's Law states that:

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year [...]. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years."

Later, the formulation was slightly altered to say that the number of transistors on integrated circuits doubles approximately every *two* years. The *clock frequency*[6] development of processors has also followed Moore's law closely, making computer performance steadily increase. However, in the early 2000s, the increase in clock frequency started to slow down, before coming to a complete halt around 2005. On most CPUs we have even seen a decrease in clock frequency. This is due to three problems with serial computing, which together constitute what has been dubbed "the brick wall" for serial computing [4].

The first problem is dubbed *the power wall*. Due to the small construction scale and high power consumption of modern processors, they reach very high temperatures when operating at maximum performance. In other words: They have a high *power density* (W/m^2). In fact, todays processors have a power density matching a nuclear reactor core [50]. Since the materials they are constructed of cannot withstand these temperatures over longer periods of time, we must somehow cool the processors. The power density is proportional to the clock frequency cubed [25, p. 88], and as the frequency increases it quickly becomes expensive and difficult to provide sufficient cooling, and it eventually becomes a hopeless endeavor.

The second problem is known as *the memory wall* or *the von Neumann bottleneck*[7]. Data (and instructions) need to be transferred from main memory to the processor (and the Arithmetic Logic Unit) before processing. Hardware manufacturers have developed increasingly larger memory sizes and faster processors with great success, but the memory bus over which data and instructions are transferred has not kept up with the speed, and has become a major bottleneck. The von Neumann architecture refers to the stored-program computer architecture described by John von Neumann in the 1940s. Although computer architecture has evolved since then, the memory bus remains a bottleneck. CPU vendors have taken steps to decrease the effects of the von Neumann bottleneck by adding more levels of on-chip caches, increasing the sizes of these caches, and increasing concurrency. However, as the cache sizes increase, the performance gain is eventually lost due to increased latency [5].

The third problem is named *the Instruction Level Parallelism wall*. Instruction Level Parallelism (ILP) allows the processor to overlap the execution of multiple instructions, or even to change the order in which instructions are executed, as long as the data dependencies are not violated. This type of optimization has also reached its peak, and does no longer contribute to increased processor performance in single-core processors [5].

As a result of the brick wall, hardware vendors were forced to choose a new path. Instead

[6]A measure of cycles per second in units of Hertz. Different instructions may require one or several cycles to complete, depending on their complexity.

[7]John Backus coined this term in his Turing Award lecture in 1977.

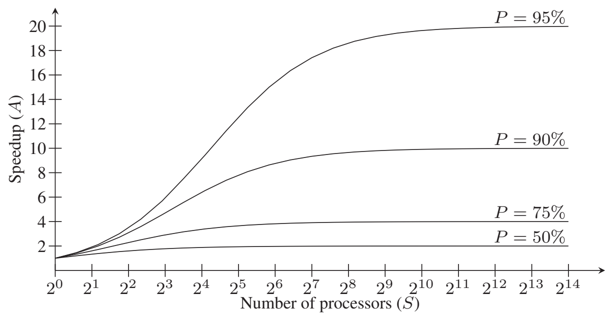


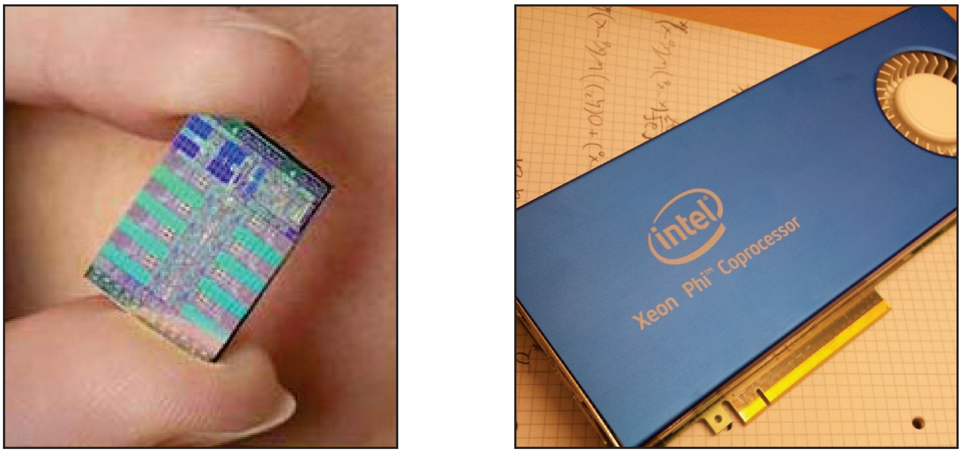
Figure 5: Amdahl's Law. (Adapted from work by André R. Brodtkorb.)

of increasing the clock frequency, they started increasing the vector width and the number of cores on the processors. Vector instructions were introduced with the Pentium III processor in 1999, and Advanced Vector Extensions (AVX, AVX2, and AVX-512) [19] is the newest addition. In 2002, Intel introduced Hyper-Threading Technology (HTT) [31] with the Pentium 4 processor. HTT enables each physical core to mask latencies and stalls, e.g., due to data dependencies, by instantaneously switching between two hardware threads. Furthermore, on current CPUs there are typically two to eight physical cores, each doing the job of a traditional single-core processor. Such multi-core processors are examples of parallel architectures found in commodity-level desktop computers, e.g., Intel's Core and AMD's FX processors. The GPU, with its hundreds to thousands of compute cores, is a many-core processor, operating in a data-parallel fashion, performing the same instructions simultaneously on a large number of input data.

The shift from serial to parallel means that old programs will not automatically scale with new processor generations, as they have previously done. Consequentially, programmers will have to make their programs exploit the new architectures of multi-core and many-core processors. This is a major concern since almost all of today's software is written for single-core processors. At the same time it is an opportunity to significantly speed up all embarrassingly parallel algorithms, and increase efficiency in all other algorithms containing at least some parallel sections of code. Programming for parallelism means recognizing the parallel portions of the algorithm and business logic to be implemented. It is very rarely possible to parallelize a complete program, since there will always be some part of the code that needs to be run serially. Amdahl's Law from 1967 [3] states that:

$$A = \frac{1}{(1-P) + \frac{P}{S}}, \tag{26}$$

in which A is the ideal speedup, P is the portion of parallel code, and S is the number of processors. Figure 5 shows Amdahl's Law for different values of P, where the x axis represent the number of processors S. We see that when the number of processors tend towards infinity, the runtime of a program is not limited by the parallel portions of the program, but rather the serial portion. A heterogeneous platform is one solution to this problem, in which one very fast "traditional" core typically handles the serial portion of the code, and several "lighter" *accelerator cores* that perform the bulk of the arithmetic operations. In other words; different



(a) The die of a Cell BE processor. **(b)** The Intel Xeon Phi coprocessor.

Figure 6: Examples of heterogeneous and parallel architectures. (Cell BE picture courtesy of Érick Luiz Wutke Ribeiro. Intel Xeon Phi picture courtesy of Johan Seland.)

types of compute cores working in symphony, in which each core performs the task it is best suited for. Examples of heterogeneous platforms and architectures include the Cell BE [22] and a compute node utilizing both CPUs and GPUs, Intel Xeon Phis [20], or FPGAs [5].

Computer architectures today are still often classified according to Flynn's taxonomy [12]. Flynn divides systems into four classifications based on the degree of parallelism. Sorted from least parallelism to most parallelism, we have: *Single Instruction, Single Data stream* (SISD) is the model serial computer with no parallelism in neither instructions nor data streams. *Single Instruction, Multiple Data streams* (SIMD) executes a single instruction on multiple streams of data in parallel, i.e., a vector of data. *Multiple Instructions, Single Data stream* (MISD) have multiple instructions operating simultaneously on a single data stream, which makes it a highly uncommon architecture. Last, *Multiple Instructions, Multiple Data streams* (MIMD) means multiple instructions operating simultaneously on multiple data streams in parallel, and most modern supercomputers fall under this classification. Two additional classifications have been added since Flynn proposed his taxonomy: *Single Program, Multiple Data* (SPMD) is multiple processors (not cores) executing in parallel on multiple data, and typically not in the lockstep fashion indicated by SIMD. *Multiple Programs, Multiple Data* (MPMD) is a variation of SPMD with more than one single program being executed simultaneously. Standard desktop computers without multi-core CPUs can generally be classified as a SISD architecture, while multi-core CPUs executing vector instructions can be classified as MIMD. The GPU can be classified as a SIMD architecture.

The Cell BE [22] (see Figure 6a), the processor used in the PlayStation 3 and Blade Servers from IBM, is an interesting heterogeneous architecture. The Cell BE is a joint effort between Sony, Toshiba, and IBM, in which different compute cores resides on the same chip. A standard CPU is connected to eight accelerator cores on a single chip. The CPU will work with conventional operating systems and generally handles serial portions of the program, controls

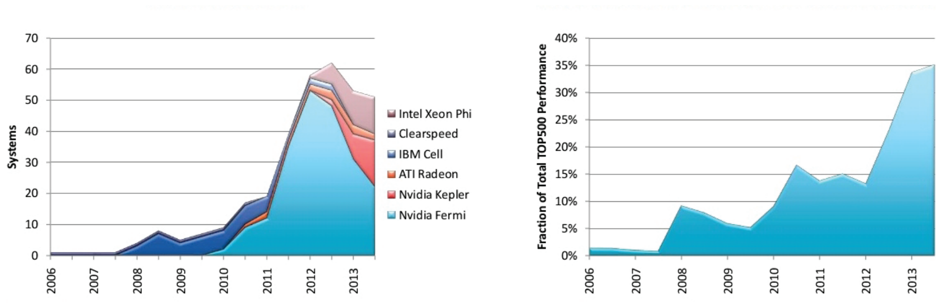
Figure 7: The world's fastest supercomputer: The Tianhe-2 cluster.

the accelerator cores, and performs DMA operations. Most of the computational workload is handled by the accelerator cores, which are designed specifically for fast vectorized floating point operations. The accelerator cores can be chained together, forming a pipeline in which each core performs a different operation on a stream of data, or they can be set to perform the same operations in parallel on a large set of data, i.e., the Cell BE can have both SIMD and MIMD behavior.

The new Xeon Phi [20] coprocessor (see Figure 6b) family from Intel is a massively parallel architecture similar to the GPU. Xeon Phi is based on the Intel MIC[8] [18] architecture, incorporating earlier research on the Larrabee [46] many-core architecture, the Teraflops Research Chip [56] multi-core chip (Polaris), and the Single-chip Cloud Computer [21] multi-core processor. The Xeon Phi coprocessors are x86-compatible and capable of utilizing existing software, including OpenMP [40], OpenCL [24], and specialized versions of Intel's Fortran, C++, and math libraries. The Xeon Phi coprocessors can be classified as a MIMD or MPMD architecture.

Major changes in processor architecture triggered by the brick wall is apparent in both commodity-level hardware and state-of-the-art hardware used in high-performance computing (HPC) for scientific research, financial calculations, advanced control systems, etc. Some examples of HPC used in research include numerical weather prediction models, quantum mechanics, molecular dynamics, reservoir simulations, and advanced signal processing. HPC has traditionally been synonymous with clusters of compute nodes, in which each individual node closely resembles a powerful desktop computer. These clusters are frequently referred to as supercomputers, and can be classified as MIMD or MPMD. In the Tianhe-2 cluster [11] (see Figure 7), each node contains two Intel Ivy Bridge Xeon processors, three Intel Xeon Phi processors, and 88 GB of memory. Tianhe-2 contains a total of 3,120,000 heterogeneous nodes, interconnected by the "TH Express-2", and was the world's fastest supercomputer on the Top 500 November 2013 list [55]. It should be noted that the Top 500 list was introduced over 20 years ago, and that the floating-point performance is assessed by a single benchmark, Linpack, which performs a dense matrix calculation. Some argue that the Top 500 list has become irrel-

[8]Many Integrated Core



(a) Systems utilizing accelerators. **(b)** Accelerator performance share.

Figure 8: Starting in 2008 there has been a significant increase in the use of accelerators in the supercomputers on the Top 500 list. Today, over 35% of the total performance of the Top 500 list comes from accelerators. (Figures courtesy of Top 500 [51].)

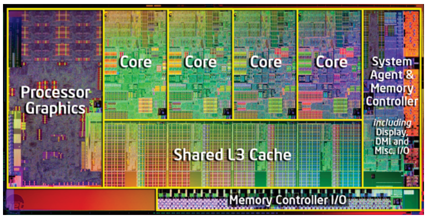
evant, and that it does not reflect the sustained performance and the application performance of the systems. Therefore, the US NCSA[9] chose not to submit the Blue Waters supercomputer to the list [33], which would probably have been ranked among the top systems, with its Nvidia Kepler GPUs, and an achieved one petaFLOPS of sustained performance on a range of scientific applications. Nevertheless, the Top 500 list is still very much active, and it is still the best available measurement and ranking of supercomputers in the world.

Parallel computing has been used in the HPC community for decades, and it is rather on the consumer-level hardware that parallelism is a new concept. What is a more recent addition to HPC, however, is the concept of heterogeneous computing, in which several different types of processors are working in concert on a single problem instance. The typical setup is still multiple nodes with multiple traditional CPUs, but accelerators, e.g., GPUs and Intel Xeon Phis, are now being used to speed up computations. An increasing part of the Top 500 list is now using accelerators (see Figure 8), and even though the first accelerated systems did not appear before 2008, over 10% of the systems now use accelerator technology, and over 35% of the total performance comes from accelerators. In a five-year period this constitutes a large shift, maybe a paradigm shift, in the supercomputer segment.

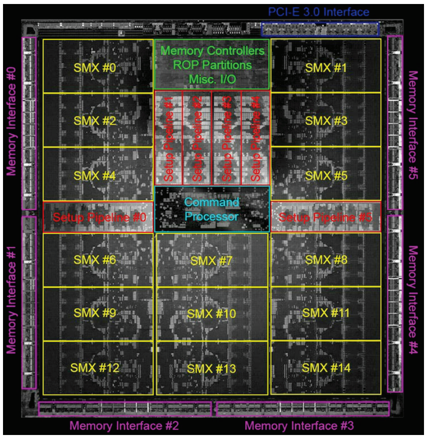
While HPC is a vital part of modern science, and an important tool for modern society, commodity-level hardware has been a significant driver in the development of computing hardware. This is due to the high performance requirements of modern software; demanding operating systems, applications for 3D-modeling, CAD software, and image and video processing tools. Perhaps even more demanding are today's computer games with their often close to photo-realistic graphics. It is the computer games industry which has driven the development of the GPU from being a fixed-pipeline graphics co-processor into a massively parallel processor, capable of delivering over one teraFLOPS of compute performance. The introduction of such powerful commodity-level hardware has made the Nvidia company coin the fitting term "Personal Supercomputer" [35].

Historically, there has also been other parallel and heterogeneous architectures similar to the

[9]National Center for Supercomputing Applications



(a) Intel Sandy Bridge (Core) die.



(b) Nvidia GK-110 (K20) die.

Figure 9: Comparison of the dies of a CPU and a GPU. The GPU chip die, measuring 551 mm^2, dedicates a much larger area to floating-point units than the CPU chip die, measuring 216–294 mm^2, in which a large area is used for cache. The SMX's are the GPU's cores.

ones we see today, which died out, e.g., *vector supercomputers*. The vector machines emerged in the 1970s, most notable the Cray machines, and died out during the 1990s. However, given the massive problems now facing serial computing, and the increasing momentum in terms of both parallel hardware and parallel software (both adapted and new), it seems unlikely that the current trend will die out in the same way. The parallel and heterogeneous hardware of today is also far more ubiquitous then previous comparable hardware, both within HPC and in the commodity-level market.

5 GPU Computing – Massively Data Parallel Computing

Novel architectures for massively data parallel computing typically have broad vector units, explicit memory operations, and many cores. Code developed for one parallel architecture will never be directly transferable to other similar architectures. What we aim to achieve in this thesis, however, is to develop algorithms, techniques, and principles, which are transferable between similar architectures. Our architecture of choice is the GPU, and in the following we will give the motivation behind this choice, introduce the GPU architecture in general, and stencil computations on the GPU in particular.

All aspects of early graphics processing was performed by the CPU, except the generation of the final video output signal. This was possible since it was not yet too demanding. As the graphics became increasingly more computationally demanding, due to the size of the scenes to be rendered and the techniques used to render them, graphics operations started to clog the CPU. One of the first computers featuring a processor intended to offload the CPU of graphics operations was the Commodore Amiga in the 1980s. Although the term GPU was not introduced until the late 1990s by Nvidia [34], this was indeed the first GPU. It supported line draw and area fill, and the "Blitter" (Block Image Transfer) accelerated copying and manipulation of bitmaps. After its introduction, the GPU's performance continued to improve, fueled mainly by the increasingly advanced and demanding real-time graphics in computer games. GPUs have a much higher number of arithmetic units than CPUs have, and operate on wider vectors (four times wider in single precision). This results in a much higher degree of parallelism, and a performance gap of approximately 7–10 times, compared to CPUs. Figure 2 in Paper I shows the gap between CPUs and GPUs for both FLOPS and bandwidth when comparing peak performance. The GPUs have this design because of the task they are designed for; computing the color of each pixel on the screen as fast as possible. Modern computer displays typically support very high resolutions. If we consider a resolution of 1920×1080 (Full HD), in which each pixel consists of four components (red, green, blue, alpha), and the image must be updated at least 30 times per second, this sums up to more than 248 million pixel updates per second. The result is a hardware architecture in which a larger portion of the transistor budget is used on floating-point units, than compared to the CPU (see Figure 9). GPUs are optimized for net throughput rather than single-thread performance and low latency. Utilized correctly for the right type of problems, GPUs will significantly improve the computational capacity of an ordinary desktop computer.

One of the first uses of graphics hardware for general purpose computations were simple matrix multiplication by Larsen and McAllister [27] in 2001. GPUs have since then been used in a multitude of different scientific areas, ranging from simulation of protein folding [28] to simulation of flow in the earth's mantle [7] and the formation of galaxies in the universe [57]. There are thousands of academic papers on GPU computing, big companies are investing in GPU-acceleration (Adobe, Autodesk, Mathworks, Microsoft, Wolfram, etc.), and as we have seen, the GPU is gaining momentum within HPC. In fact, more than half of the accelerated systems on the Top 500 list for November 2013 are utilizing GPUs (see Figure 8a). GPGPU.org and the Nvidia showcase [39] contain many more examples of GPU codes.

In the first few years of GPU computing, it did not exist any programming environment suited for general-purpose programming. The only available libraries and programming languages were intended for graphics programming, mainly OpenGL [48] and DirectX, with their respective *shading languages*. In addition, everything had to be reformulated in terms of graphics. To make code run fast it was necessary to optimize as if you were programming graphics, e.g., using vectors of size four (RGBA), exploiting mathematics tailored for graphics operations, etc. Because of the cumbersome development process, and the rapidly changing hardware and compilers, the first publications on GPU computing were mostly limited to proof-of-concept codes [41]. Programming languages and libraries specifically made for GPU computing started to emerge around 2003, and the most prominent ones are shown in Figure 10. Today, the three main programming languages are Nvidia's CUDA [38], OpenCL [24], and Microsoft's DirectCompute. In addition to the programming languages, OpenACC and the PGI Accelerator can be used to accelerate parts of a Fortran or a C/C++ program by using special compiler directives. Microsoft's C++ AMP works in a similar way. There exist several high-level libraries capable of utilizing GPUs as well, e.g., Thrust [16] and CUDPP [15], containing data structures and parallel algorithms. Thrust resembles a parallel version of the C++ Standard Template Library for CUDA. With all these new languages, libraries, and tools, it is less demanding to get an algorithm running on the GPU, but it remains difficult to optimize it for performance and maximum utilization of the hardware.

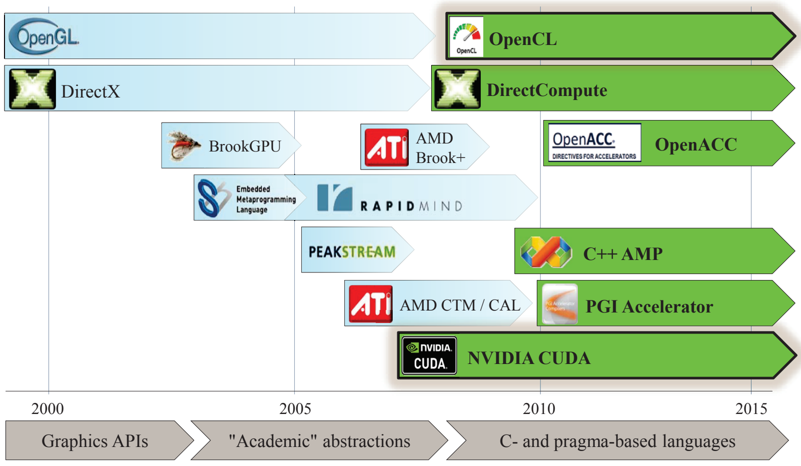
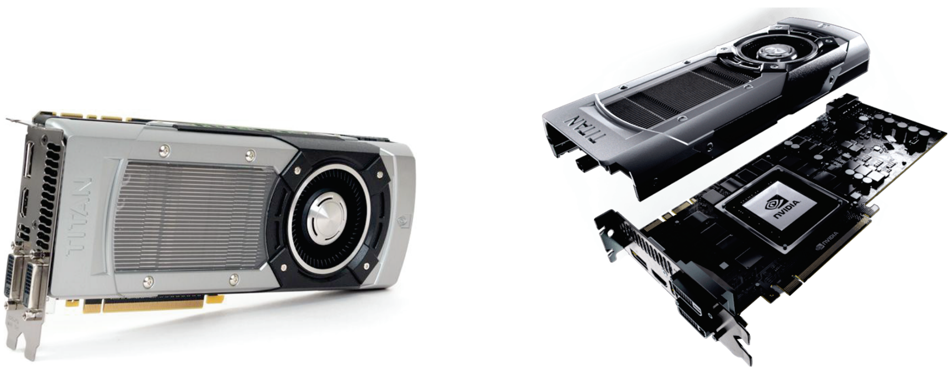


Figure 10: Programming languages and libraries used for GPU computing. The green bars indicate languages and libraries still used today. (Figure courtesy of André R. Brodtkorb.)



(a) GeForce GTX Titan.   **(b)** Without cover and fan.

Figure 11: Nvidia's newest graphics adapter in the GeForce class is the GeForce GTX Titan, based on the Kepler architecture.

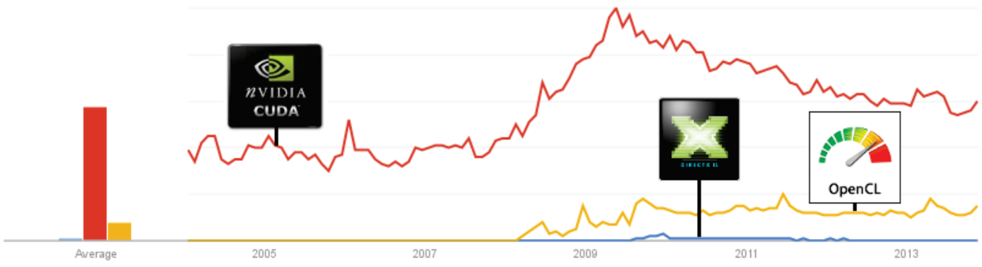


Figure 12: Google search volumes for CUDA, DirectCompute, and OpenCL.

The code in this thesis is written in C++, for Nvidia GPUs (see Figure 11), using CUDA for computations and OpenGL for visualization. CUDA, short for "Compute Unified Device Architecture", was released in 2007 by Nvidia. We chose CUDA because of its maturity, the availability of programming environments and tools, and its large professional environment (see Figure 12). OpenCL code is in theory easier to port between different hardware architectures, nevertheless, manual tuning and optimization is still required in most cases. It should also be noted that the similarities between CUDA and OpenCL are many, and that they both use the same principles, although keywords and conventions may differ. For more details, Paper I investigates GPU architecture, GPU computing, and how to write efficient code by having knowledge of the underlying hardware and using debugging and profiling.

The GPU execution model is very different from the CPU, and more work is left to the programmer. CUDA operates with GPU programs known as *kernels*, executing a given number of *blocks* configured in a 2D or 3D *CUDA grid*, in which each block contains the same number of threads (see Figure 5 in Paper I). In our shallow water simulator, we assign one thread to each cell in the finite volume grid. Since computing the value in one grid cell depends only on a small number of adjacent grid cells forming a compact stencil, this is close to a perfect parallel problem. Furthermore, threads within the same block can communicate using fast on-chip *shared memory*, eliminating many expensive global memory transfers. The shared memory can be viewed as a programmable cache. While CPU cache blocking techniques often involve some educated guesswork to optimize for cache hits, algorithms for the GPU can explicitly move often-used data into shared memory, thus minimizing the effects of slow memory transfers. The CUDA blocks are automatically scheduled to cores on the GPU by the hardware, where each block is executed independently. Expensive global synchronization is required for communication between blocks. For the stencil computations we need to read values from neighboring blocks, leading to some inter-block dependencies at the block boundaries (see Figure 4 in Paper III). The number of values to read from neighboring blocks is dictated by the size of the compact stencil. CUDA block size is typically determined based on the algorithm to implement, the problem domain, and optimization parameters. There are many, often conflicting, optimization parameters to take into account when choosing block size, e.g., total size of shared memory, shared memory size per block, and maximum and minimum threads per block. Some of these parameters are set at compile-time and some at run-time. Other key performance factors for efficient GPU code include minimizing data transfers, keeping data on the GPU[10], and

[10]The GPU global memory bandwidth for the Kepler K20 (> 200 GB/s) is more than 12 times the PCI Express bus bandwidth (up to 16 GB/s).

efficient utilization of shared memory. The CUDA grid is the lowest level of parallelism in the shallow water simulator, and the building block used for multi-GPU simulations and AMR simulations. For further details on CUDA optimization see Paper I, the CUDA C Programming Guide [38], and the CUDA C Best Practices Guide [37].

Hardware-adapted algorithms like the ones described in this thesis give highly efficient problem-specific codes, as opposed to generic "Swiss army-knife" algorithms like linear algebra solvers. Generally, the GPU is excellently suited for algorithms that require high computational performance, e.g., explicit stencil computations with complex schemes and dense matrix computations. Paper II goes more in depth on stencil computations on the GPU, and an example using the classical Lax–Friedrichs finite volume scheme is given. The application of stencil computations is not limited to finite difference and finite volume discretization for solving PDEs. Stencils represent an important computational pattern used in scientific applications within a variety of domains, e.g., computational electromagnetics [49], and image processing for CT and MRI imaging [8, 9]. Furthermore, as stencil computations are embarrassingly parallel, the algorithms presented in this thesis will scale with future GPU hardware and hardware with an architecture similar to the that of the GPU.

Chapter 2

Summary of Papers

This chapter gives an overview of the scientific papers comprising the main part of this thesis, along with comments to each paper. Paper I and II serve as general introductions to GPU programming and implementation of explicit shallow water schemes on GPUs, respectively. In Paper III the implementation of an explicit high-resolution scheme for solving the shallow water equations on the GPU is detailed, and Paper IV extends this to multiple GPUs. Paper V presents algorithms for saving both compute time and memory usage for stencil-based explicit PDE solvers, and the earlier presented shallow water simulator in particular. Last, in Paper VI a full GPU implementation of an adaptive mesh refinement algorithm is given.

Figure 1: The GeForce GTX 480 Fermi-generation GPU from Nvidia.

Paper I: GPU Programming Strategies and Trends in GPU Computing

A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra.

Journal of Parallel and Distributed Computing, ***73****(1) (2012), pp. 4–13*

This paper serves as an introduction to GPU programming and the GPU hardware and execution model. Especially for those new to GPU computing, this paper will be a good starting point. It is based on the Fermi architecture from Nvidia, but also extends to GPUs in general. In the introduction we present a historical overview of GPU computing (or general-purpose computing on GPUs (GPGPU) as it was previously known as) and different architectures and programming languages. We also compare the GPU with the standard CPU, and discuss the motivation behind the development of the GPU as a many-core general compute engine. My contributions to this paper include performing background research and literature surveys, especially related to the sections on development strategies and optimization.

The main part of the paper concerns practical GPU development, using available knowledge and tools. More specifically, we teach the reader how to do profile-driven development and debugging, in which the goal is to build robust and efficient GPU programs. Efficient meaning that the GPU operates on close-to full capacity when it comes to both memory bandwidth utilization and utilization of arithmetic units. At the end of the paper we briefly sum up the latest developments in parallel and heterogeneous computing, and offer our expectations in terms of future developments.

Comments

Since this paper was published, Nvidia has released a new architecture named Kepler [36]. Kepler differs from Fermi in some respects, and has some new features:

- *Dynamic parallelism* enables a GPU kernel to dynamically launch other kernels or itself recursively.
- Multiple CPU cores may launch work on a single GPU simultaneously. *Hyper-Q* increases the total number of work queues between the host CPU and the GPU to 32 simultaneously hardware-managed connections, whereas Fermi operated with one work queue. Hyper-Q allows separate connections from multiple CUDA streams, MPI-processes[1], and threads within a process.
- *GPUDirect* allows copying of data directly between GPUs within a single node, or GPUs in different nodes through the network, without going through CPU and system memory.

Kepler also provides over one teraFLOP of double precision throughput, at up-to three times the performance per watt of Fermi. Most of the content is also applicable to Kepler GPUs, except the section describing the Fermi GPU architecture, since the Kepler GPU architecture differs on some points. Eclipse NSight is now available for Linux users, and debugging is possible with only one GPU on both Linux and Windows, making profile-driven development free of cost.

[1]Message Passing Interface

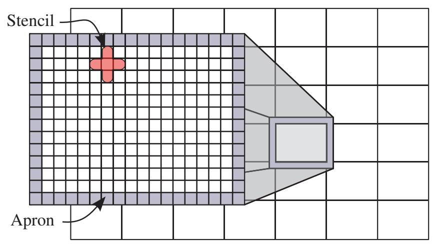


Figure 2: Illustration of the block and grid concepts with one block highlighted. The *apron* (also called local ghost cells) is used to fulfill the requirements of the stencil, so that all blocks may be executed independently and in parallel by the GPU.

Paper II: Explicit Shallow Water Simulations on GPUs: Guidelines and Best Practices

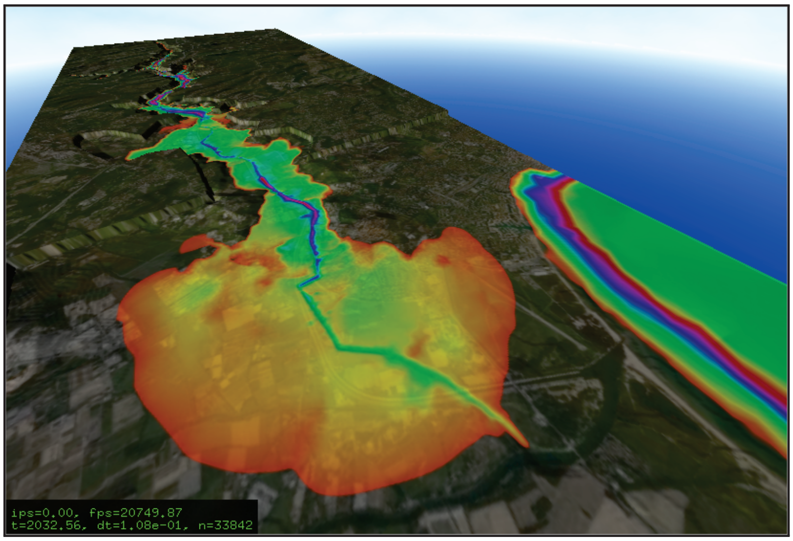
A. R. Brodtkorb and M. L. Sætra. *In proceedings of the XIX International Conference on Computational Methods in Water Resources, 2012*

While the first paper gives an introduction to the GPU and GPU computing, the topic of this paper is the mapping of explicit schemes for the shallow water equations to the GPU. The classical Lax–Friedrichs finite volume scheme is employed as an example, and a detailed step-by-step guide from mathematical model to GPU code is given. In addition to being a general introduction to mapping explicit schemes with a compact stencil to the GPU, simulations on multiple GPUs, sparse domain[2] as an optimization method, and accuracy and performance issues are also discussed. This paper is in many ways a summary of our experiences from working with shallow water simulations on GPUs, and my contributions are based on the research presented in Papers III, IV, and V.

Comments

Together with the first paper, this paper gives an excellent basis on which to read the remaining four papers. The presented guidelines and best practices are based on our experiences from working with explicit shallow water simulations on GPUs over many years.

[2]Sparse domain methods for only representing and computing "wet" parts of the domain.



```
ips=0.00, fps=20749.87
t=2032.56, dt=1.08e-01, n=33842
```

Figure 3: Simulation of the Malpasset dam break in south-eastern France where the initial flood wave caused over 420 casualties. The color indicates water depth.

Paper III: Efficient Shallow Water Simulations on GPUs: Implementation, Visualization, Verification, and Validation

A. R. Brodtkorb, M. L. Sætra, and M. Altinakar. *Computers & Fluids, **55** (2012), pp. 1–12*

This paper gives a complete description of the implementation of the high-resolution Kurganov–Petrova scheme. Novel contributions include optimization strategies, verification and validation, physical friction terms, multiple time integrator schemes, a minimal memory footprint, efficient implementation of multiple boundary conditions, and efficient direct visualization. We further verify and validate our implementation against both analytical and experimental data. The verification is performed by comparing simulation of oscillating motion in a parabolic basin with an analytical solution. In this case, our implementation captures the analytical solution well for the water elevation. However, with an increasing number of simulation steps, there is a growing error in the fluxes along the wet/dry interface, which eventually also affects the water elevation. Our validation is performed against the experimental data from the Malpasset dam break case, a dam break that caused over 420 casualties in south-eastern France in December 1959 (see Figure 3). Our implementation accurately predicts both the wave arrival time and maximum water depth for this real-world case. The research presented in this paper was mainly performed during a three month stay at the National Center for Computational Hydroscience and Engineering in Mississippi, USA. I have contributed to researching different strategies for implementing the numerical scheme, and the verification, validation, and benchmarking of the simulator.

The most promising optimization is the early exit strategy, in which cells that do not change during the next time step are not computed. Cells that do not change are either dry or in a steady-state. Early exit is implemented by exploiting the compact stencil of the scheme, the blocked execution model of the GPU, and a small auxiliary buffer that keeps track of which blocks of cells are “dry”. This strategy speeds up simulations significantly, particularly for simulation domains that has large dry areas, which are typical for, e.g., dam breaks and river inundations, and it is further explored in Paper V.

Comments

By combining a high-resolution numerical scheme and Manning-type friction, the implemented simulator is capable of accurately simulating real-world phenomena, e.g., dam breaks, tsunamis, storm surges, etc. Thus, delivering both realistic physics and high performance. Direct visualization of simulation results is also implemented, and by copying the solution from CUDA memory space to OpenGL memory space without going through the CPU, the visualization constitutes only about a tenth of the run time of the application. Both photo-realistic rendering and rendering of water depth and speed are available. Several promising research directions presented itself during this work. One of which is described in Paper IV; simulation on multiple GPUs. Extending this to heterogeneous computing, in which both GPUs and CPU cores are utilized, may yield more efficient simulations as the number of cores in modern CPUs are steadily increasing. This will, however, require further research into domain decomposition and efficient load-balancing between the different computational resources, i.e., the CPU and the GPU.

Mixed boundary conditions is another possible extension of this work, enabling modeling of, e.g., river inlets and outlets. The current boundary conditions efficiently handles wall boundaries, inflow boundaries, outflow boundaries, and linear functions emulating, e.g., storm surges and tsunamis.

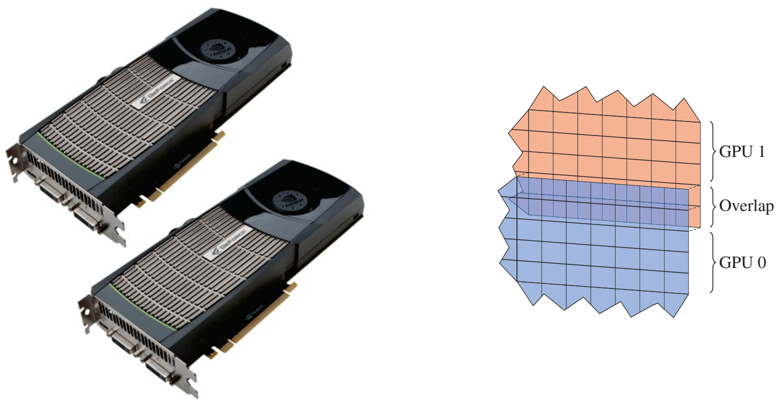


Figure 4: Multi-GPU domain decomposition using overlapping ghost cell regions. The two GPUs exchange the overlapping region after a certain number of time steps.

Paper IV: Shallow Water Simulations on Multiple GPUs

M. L. Sætra and A. R. Brodtkorb. *Lecture Notes in Computer Science,* ***7134*** *(2012), pp. 56–66*

By building on the simulator from Paper III, this paper details a multi-GPU version of the simulator, capable of utilizing several GPUs in one node. The research in this paper is partly based on my master's thesis [43], and I have contributed to researching multi-GPU hardware architectures, strategies for exploiting these, and performance benchmarking and profiling.

The numerical scheme of the simulator remains unchanged. Row domain decomposition is used to enable multi-GPU simulation, and traditional CUDA block decomposition is used within each GPU, giving two layers of parallelism. A *control thread* manages domain decomposition and all synchronization and communication between GPUs, and *worker threads*, one per GPU, run the simulation on its designated *subdomain*, oblivious of all other worker threads. This separation of concerns makes the code more flexible, and facilitates further development of the simulator, e.g., adding other domain decomposition strategies. All subdomains need to communicate with the neighbors (the top and bottom subdomains will only have one neighbor each) it shares boundaries with, i.e., across all boundaries that are not global boundaries. This is handled by copying data from the interior of subdomain 0, to the ghost cell region of subdomain 1, and vice versa (see Figure 4). The row decomposition is favorable as it minimizes the number of memory transfers and allows for transferring continuous chunks of memory. To further decrease the overhead connected with communication between GPUs, *pinned* CPU memory[3] is used. By using *ghost cell expansion* (GCE) the ghost cell region is expanded, i.e., the overlap seen in Figure 4 is also expanded, thus enabling each subdomain to perform more than one time step before exchanging data with its neighbors. The maximum number of time steps per data exchange is determined by both the stencil size and the size of the GCE. In addition to the boundary data, the time step size must also be synchronized globally as every subdomain need to be at the same simulation time when exchanging data with neighbors.

[3]Pinned memory provides higher transfer throughput for large memory transfers.

Benchmarking of the code is performed on three different systems, two generations of server GPU-nodes with four GPUs, and a commodity-level desktop PC with two GPUs. The implementation shows near perfect linear weak and strong scaling, and we were able to simulate on domains consisting of as much as 235 million cells, computing over 1.2 billion cells per second, using four Fermi-generation GPUs. The impacts of GCE and global synchronization of time step sizes are also benchmarked.

Comments

A further research direction is to extend the multi-GPU simulator to allow inter-node communication as well. Enabling the use of multiple nodes adds a third layer of parallelism, and simulation run times can be shortened and domain sizes and grid resolution increased even more. We also expect GCE to have a larger impact when data exchanged between subdomains must be moved over the network. Implementing asynchronous execution of computations and transferring of ghost cells would also increase performance, given the right ratio of ghost cells to internal cells. It should also be noted that using GPUDirect remote DMA, introduced with CUDA 5, would speed up the transfer of ghost cells between GPUs. Adaptive domain decomposition dictated by features in the bottom topography or the solution is another possible extension.

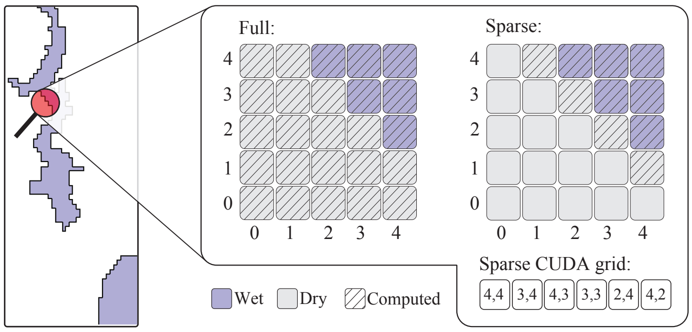


Figure 5: Map of dry blocks during a real-world dam break simulation. By storing whether or not a block contains water, we can create a list of blocks that require computation before each time step, and launch the kernel only on these blocks. Since water from a wet block may flow to a neighboring dry block in the next time step, we must also compute these dry blocks. This approach saves both computation and bandwidth, yielding a significant performance improvement on typical domains.

Paper V: Shallow Water Simulation on GPUs for Sparse Domains

M. L. Sætra. *In: Numerical Mathematics and Advanced Applications 2011, Springer, 2013*

In this paper we revisit the early exit optimization from Paper III, and develop two novel algorithms based on the same premise; that for large parts of the domain the solution will not change from one time step to the next. This time, however, we do not even load any data from "dry" blocks. Dry blocks also include blocks containing water in a steady state. We accomplish this by maintaining a map of the wet blocks, and adding new wet blocks for each time step (see Figure 5). This map is then used to lookup which blocks to read when performing time integration. Since water may propagate across an interface to a neighboring block, we also must include all neighbors of wet blocks in the computations.

Two algorithms are proposed: *Sparse compute* saves compute time, but does not impact memory use, except the minimal extra map for the wet blocks. *Sparse memory*, however, does not save any of the conserved variables for the dry cells, thus significantly reducing memory usage for many simulation scenarios. Instead of saving the conserved variables in a 2D layout consistent with the logical domain layout, each block is saved in a *block-linear* fashion, and new wet blocks are appended after each time step. There is an added computational cost connected with the sparse memory algorithm; an additional mapping is needed to resolve neighbors of wet blocks, since this is no longer given by the memory layout.

Using a circular dam break test case, we compare the efficiency of the different algorithms, including early exit. The results show that the sparse compute algorithm outperforms early exit. When the area of wet cells are below 35%, the sparse memory algorithm also outperforms early exit. Considering that the early exiting also yields a significant performance improvement, these are promising results.

Comments

CUDPP [15], a library of data-parallel algorithm primitives for CUDA, was used for the compaction part of the algorithm. This library has since been optimized, making this part of the algorithm more efficient. There also exist other libraries that can replace CUDPP, e.g., Thrust [16], or a custom compaction algorithm (scan and scatter) can be implemented.

An interesting further research direction for the sparse memory algorithm could be delayed loading of bottom topography, thus allowing for off-core simulations. This means that the domain size no longer would be limited by the amount of available GPU memory, but by the number of wet cells in the solution. Combining sparse domain algorithms with multi-GPU simulations could allow for extremely large domains, but it is not straightforward how this can be accomplished in an efficient manner.

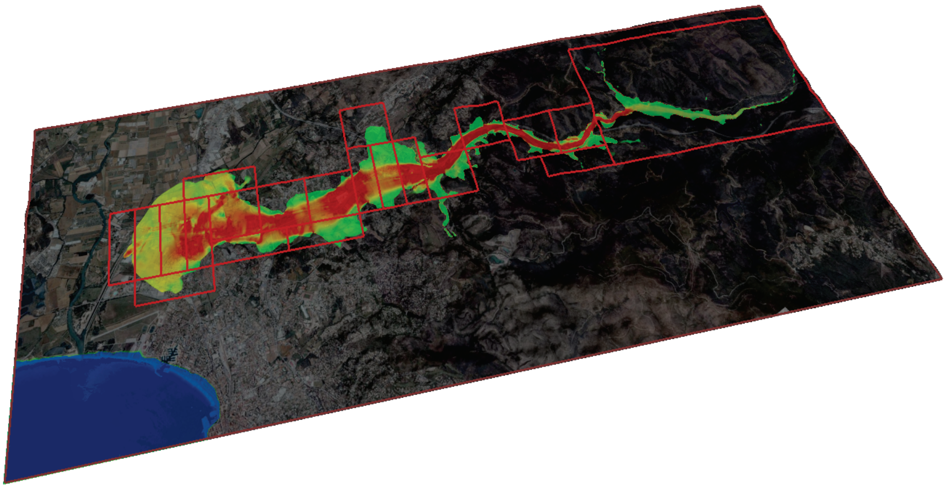


Figure 6: Visualization of water velocity in the 1959 Malpasset dam break case. New grids are added dynamically to cover the flood as it progresses.

Paper VI: Efficient GPU-Implementation of Adaptive Mesh Refinement for the Shallow-Water Equations

M. L. Sætra, A. R. Brodtkorb and K.-A. Lie.

Journal of Scientific Computing, submitted for 2nd review, 2014

The hierarchy of scales is a significant challenge in numerical simulations, particularly when working with real-world cases. For example, a storm surge may start out at sea, hit the continental shelf, inundate the coastline (which may be a harbor with complex geometry), and travel through wetland and up rivers. Accurately capturing the storm surge travelling up a river requires a much higher resolution than out to sea, but to get accurate simulation results the domain needs to cover both areas. It would be too computationally demanding to compute the full domain using the highest necessary resolution, hence we need some way of connecting the scales. Adaptive mesh refinement (AMR) is one possible solution. AMR involves maintaining a hierarchy of *subgrids*, in which each level of new subgrids has a higher resolution than its parent grid, thus enabling higher accuracy in the solution and better resolution of features in the bottom topography.

In the AMR algorithm, a child grid cannot be advanced in time before its parent grid is one time step ahead. By using the solution at the beginning and end of each time step on a parent grid, we perform space-time interpolation to determine ghost cell values for all intermediate time steps on all its child grids. For each single time step on a parent grid, its child grids typically require two time steps, since the child grids have twice the resolution of their parent grid. The time interpolation is performed during time stepping, as we do not know the time-step sizes a priori. For any grid cell that is covered by a grid with higher resolution, the solution is replaced by an average of the values from the cells in the grid with higher resolution covering this cell.

An additional flux correction must be made after each time step to maintain conservation of mass.

Again, as in Paper IV, the shallow water simulator from Paper III is used as the "building block" in the AMR framework code. The result is an AMR algorithm and code featuring a well-balanced high-resolution numerical scheme with conservation of mass. There are two classes of AMR; *cell-based* and *block-based*. This paper describes, to the best of our knowledge, the first block-based AMR code fully implemented on the GPU. I contributed to development of algorithms for AMR on the GPU based on earlier work by Berger and Colella, and implementation of the AMR simulator. Furthermore, I have participated in benchmarking and profiling of the AMR simulator.

Several tests and experiments are performed, and the AMR algorithm is shown to be both efficient and sufficiently accurate compared to global domain refinement. We also verify simulation results against an analytical solution given by the SWASHES library [10], and test the overhead connected with maintaining multiple subgrids. Furthermore, we demonstrate simple shock tracking, give an example of subgrid features resolved by using AMR, and show that the code is capable of handling a real-world case with complex bottom topography and high velocities; the 1959 Malpasset dam break case (see Figure 6).

Comments

We believe the presented AMR algorithm is transferable to other hyperbolic conservation laws, numerical schemes, and other massively parallel architectures similar to the GPU.

The next natural step in the AMR code is to implement, or use an available library, for better subgrid management. Combining new subgrids with existing subgrids will probably reduce the effects of regridding, and certainly increase efficiency, since the total number of grids is decreased. More extensive benchmarking and testing, particularly on real-word cases using many levels of refinement and the newest generation GPU hardware, is also needed. The reduction of time step sizes on a parent to avoid very small time steps on the child grids showed promising results that should be investigated further. It would also be interesting to combine AMR with multi-GPU simulations and the sparse domain algorithms, to enable very large domains and even more efficient simulations.

Chapter 3

Outlook

The previous chapters have given background and introduction to the main topics of this thesis, as well as a short presentation of the scientific results that will be discussed in more detail in Part II. In this chapter we present an outlook on the future of parallel and heterogeneous computing from a scientific computing view, based on the experience and insight gained from this work. Although this thesis investigates a particular problem within scientific computing on a particular architecture, and is naturally colored by this fact, we argue that the findings also extend to other similar contemporary and future architectures.

After frequency scaling met the brick wall, concurrency and parallelism became the new way of increasing FLOPS, in the form of increased vector width and multi-core and many-core architectures. The GPU is a part of this trend. Even though the game industry has always been the main commercial area for GPU vendors, they were able to successfully enter the HPC market. This made GPUs one of the first massively parallel accelerators used. Thus, the GPU's architecture, programming model, and tools and environment have had a long time to develop and mature. The GPU is also the accelerator with the highest theoretical peak performance, the one utilized by the largest share of systems on the Top 500 list [55], and one of the most energy-efficient architectures [45, 36]. Furthermore, it is present in practically every new desktop and laptop computer today. Together, this makes a compelling case for using GPUs when investigating parallel and heterogeneous architectures.

Energy is an expensive and scarce resource which should be efficiently utilized. Supercomputers, and their cooling systems, generally have a very high energy consumption. While the clock frequency was the biggest performance constraint earlier, power has now become the new biggest constraint. This is widely recognized, and the Green 500 list [47], ranking the top 500 most *energy-efficient* supercomputers in the world, emphasizes this. Moreover, energy-efficient architectures will also typically deliver the best FLOPS per dollar ratio, as we also have demonstrated throughout several of the presented scientific papers.

As heterogeneous computing and accelerator technology in HPC becomes more widespread, there is an increasing need for research based on these new architectures and platforms to ensure increased performance and scaling also in the future. It is not just the switch from frequency scaling to concurrency scaling that has affected HPC over the last decades. Coinciding with this switch, several other factors and constraints have also changed: First, FLOPs are increasingly inexpensive, while on-chip data movement becomes a bottleneck. This makes it imperative to minimize data movement, something we have explored throughout this thesis. Second, the memory bottleneck is increasing, as main memory latency is increasing and main memory bandwidth decreasing relative to processor cycle time [25, p. 27]. Writing and reading to main memory should therefore, just as on-chip data movement, be minimized. Third, the uniform latencies within nodes and between nodes are gone, and replaced by heterogeneity in which we need to think very carefully about data locality and the topology of the hardware. Last, growth in parallelism was earlier due to adding new nodes to the cluster, but now comes from parallelism within the chips on each node.

The impact of these changes is significant. There exists a very large number of codes today that need rewriting to be able to scale (or run at all) on the new parallel and heterogeneous architectures. At the same time we see that the need for large-scale simulations are important across a vast range of scientific disciplines and applications, in which shallow water simulations is only one example. It will be important for both users (scientists and programmers) and supercomputing centers to choose “the right” technology to achieve scaling in the future, and to utilize hardware-adapted numerics and algorithms to maximize performance. The technology choice is not trivial, and at the time of writing this thesis there are at least three alternatives: Standard multi-core with complex cores; embedded many-core and custom solutions, e.g., IBM’s BlueGene; and many-core accelerators, e.g., GPUs and Intel Xeon Phi. The development of novel parallel and heterogeneous architectures has been rapid, and even disruptive at times, adding to the difficulty. Nevertheless, these architectures gain momentum as their hardware and software mature, as development is continuously made easier by new languages, libraries, and tools, and by their ubiquity in modern computers. In summary, we foresee a challenging and exciting future for heterogeneous computing, numerical simulation, and hardware-adapted algorithms.

Bibliography

[1] Two billion will be in flood path by 2050, UNU expert warns. *The newsletter of United Nations University*, 32, August 2004.

[2] M. S. Altinakar. Simulation water infrastructural safety DSS-WISE and WGFEM. FEMA region IV capability gaps review meeting, November 2010. [presentation].

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In M. D. Hill, N. P. Jouppi, and G. Sohi, editors, *Readings in computer architecture*, chapter 2, pages 79–81. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[4] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, EECS Department, University of California, Berkeley, December 2006.

[5] A. Brodtkorb, C. Dyken, T. Hagen, J. Hjelmervik, and O. Storaasli. State-of-the-art in heterogeneous computing. *Journal of Scientific Programming*, 18(1):1–33, May 2010.

[6] A. R. Brodtkorb, T. R. Hagen, K.-A. Lie, and J. R. Natvig. Simulation and visualization of the Saint-Venant system using GPUs. *Computing and Visualization in Science*, 13(7):341–353, 2011.

[7] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. Wilcox. Extreme-scale AMR. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE Computer Society, 2010.

[8] J. Cong, M. Huang, and Y. Zou. Accelerating fluid registration algorithm on multi-FPGA platforms. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 50–57, 2011.

[9] J. Cong and Y. Zou. FPGA-based hardware acceleration of lithographic aerial image simulation. *ACM Trans. Reconfigurable Technol. Syst.*, 2(3):17:1–17:29, September 2009.

[10] O. Delestre, C. Lucas, P.-A. Ksinant, F. Darboux, C. Laguerre, T.-N.-T. Vo, F. James, and S. Cordier. SWASHES: a compilation of shallow water analytic solutions for hydraulic and environmental studies. *International Journal for Numerical Methods in Fluids*, 72(3):269–300, 2013.

[11] J. Dongarra. Visit to the National University for Defense Technology Changsha, China. Technical report, Oak Ridge National Laboratory, 2013.

[12] M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[13] E. Godlewski and P. Raviart. *Numerical Approximation of Hyperbolic Systems of Conservation Laws*. Number 118 in Applied Mathematical Sciences. U.S. Government Printing Office, 1996.

[14] H. Hanche-Olsen and H. Holden. Preprints on conservation laws. `http://http://www.math.ntnu.no/conservation`.

[15] M. Harris, S. Sengupta, and J. D. Owens. *GPU Gems 3*, chapter 39. Addison-Wesley Professional, first edition, 2007.

[16] J. Hoberock and N. Bell. Thrust: A parallel template library, 2010. Version 1.7.0.

[17] H. Holden and N. H. Risebro. *Front Tracking for Hyperbolic Conservation Laws*. Number 152 in Applied Mathematical Sciences. Springer Berlin Heidelberg, 2011.

[18] Intel. Intel Many Integrated Core Architecture. `http://www.many-core.group.cam.ac.uk/ukgpucc2/talks/Elgar.pdf`, December 2010. [presentation].

[19] Intel. Intel architecture instruction set extensions programming reference, 2013.

[20] Intel. Intel Xeon Phi coprocessor developer's quick start guide. Technical report, 2013.

[21] Intel Labs. The SCC platform overview. Technical report, Intel Corporation, 2010.

[22] C. R. Johns and D. A. Brokenshire. Introduction to the cell broadband engine architecture. *IBM J. Res. Dev.*, 51(5):503–519, 2007.

[23] S. Jonkman. Global perspectives on loss of human life caused by floods. *Natural Hazards*, 34(2):151–175, 2005.

[24] Khronos Group. OpenCL. `http://www.khronos.org/opencl/`, 2013.

[25] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems. Technical report, DARPA, IPTO, 2008.

[26] D. Kröner. *Numerical schemes for conservation laws*. Advances in numerical mathematics. Wiley, 1997.

[27] E. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *ACM/IEEE conference on Supercomputing*, pages 55–55, New York, NY, USA, 2001. ACM.

[28] S. M. Larson, C. D. Snow, M. Shirts, V. S. P, and V. S. Pande. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology.

[29] P. Lax. The flowering of applied mathematics in America. *SIAM Review*, 31(4):533–541, 1989.

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

[31] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):1–12, 2002.

[32] G. E. Moore. Cramming more components onto integrated circuits. In *Readings in computer architecture*, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[33] NCSA. Top problems with the TOP500. `http://www.ncsa.illinois.edu/News/Stories/TOP500problem`.

[34] Nvidia. GeForce 256. `http://www.nvidia.com/page/geforce256.html`.

[35] Nvidia. Nvidia Tesla Personal Supercomputer. `http://www.nvidia.com/object/personal_supercomputing.html`.

[36] Nvidia. Nvidia GeForce GTX 680. Technical report, Nvidia Corporation, 2012.

[37] Nvidia. Nvidia CUDA C best practices guide version 5.5, 2013.

[38] Nvidia. Nvidia CUDA C programming guide version 5.5, 2013.

[39] Nvidia. CUDA community showcase. `http://www.nvidia.com/object/cuda_showcase_html.html`, 2014.

[40] OpenMP.org. Openmp.org. `http://openmp.org`.

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

[42] L. F. Richardson and S. Chapman. *Weather prediction by numerical process*. Dover publications, New York, NY, USA, 1965.

[43] M. L. Sætra. Solving systems of hyperbolic PDEs using multiple GPUs. Master's thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, August 2007.

[44] A. Saint-Venant. Theorie du mouvement non permanent des eaux, avec application aux crues des rivieres et a l'introduction de marees dans leurs lits. *Comptes-Rendus l'Académie des Sciences*, 73:147–273, 1871.

[45] T. Scogland, B. Subramaniam, and W.-c. Feng. Emerging Trends on the Evolving Green500: Year Three. In *7th Workshop on High-Performance, Power-Aware Computing*, Anchorage, Alaska, USA, May 2011.

[46] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):18:1–18:15, August 2008.

[47] S. Sharma, C.-H. Hsu, and W. chun 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.

[48] D. Shreiner, G. Sellers, J. M. Kessenich, and B. M. Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley, 8th edition edition, 2013.

[49] A. Taflove and S. C. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House, 3 edition, June 2005.

[50] G. Taylor. Energy efficient circuit design and the future of power delivery. Electrical Performance of Electronic Packaging and Systems, October 2009. [presentation].

[51] Top500. Highlights of the 42nd TOP500 list. `http://top500.org/blog/slides-from-the-top500-bof-session-during-sc13-in-denver`, November 2013. [presentation].

[52] E. F. Toro. *Shock-Capturing Methods for Free-Surface Shallow Flows*. John Wiley & Sons, Ltd., 2001.

[53] J. Trangenstein. *Numerical Solution of Hyperbolic Partial Differential Equations*. Cambridge University Press, December 2007.

[54] UN Office for Disaster Risk Reduction. Preventionweb. `http://www.preventionweb.net`.

[55] University of Mannheim, University of Tennessee, and NERSC/LBNL. Top 500 supercomputer sites. `http://www.top500.org/about`.

[56] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w teraflops processor in 65-nm CMOS. *Solid-State Circuits*, 43(1):29–41, January 2008.

[57] P. Wang, T. Abel, and R. Kaehler. Adaptive mesh fluid simulations on GPU. *New Astronomy*, 15(7):581–589, 2010.

Part II
Scientific Papers

Paper I

GPU Programming Strategies and Trends in GPU Computing

A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra

Journal of Parallel and Distributed Computing, **73**(1) (2012), pp. 4–13

Abstract: Over the last decade, there has been a growing interest in the use of graphics processing units (GPUs) for non-graphics applications. From early academic proof-of-concept papers around the year 2000, the use of GPUs has now matured to a point where there are countless industrial applications. Together with the expanding use of GPUs, we have also seen a tremendous development in the programming languages and tools, and getting started programming GPUs has never been easier. However, whilst getting started with GPU programming can be simple, being able to fully utilize GPU hardware is an art that can take months and years to master. The aim of this article is to simplify this process, by giving an overview of current GPU programming strategies, profile driven development, and an outlook to future trends.

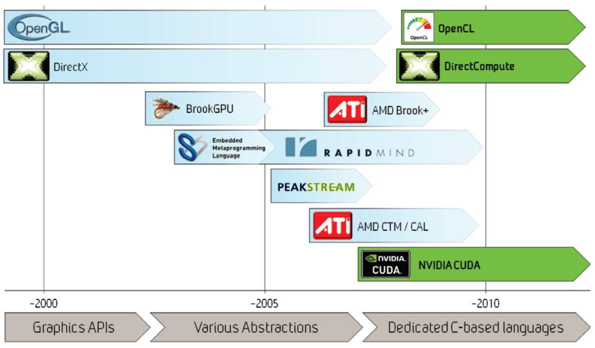


Figure 1: History of programming languages for GPGPU. When GPUs were first being used for non-graphics applications, one had to rewrite the application in terms of operations on graphical primitives using languages such as OpenGL or DirectX. As this was cumbersome and error prone, several academic and third-party languages that abstracted away the graphics appeared. Since 2007, however, vendors started releasing general purpose languages for programming the GPU, such as AMD Close-to-Metal (CTM) and NVIDIA CUDA. Today, the predominant general purpose languages are NVIDIA CUDA, DirectCompute, and OpenCL.

1 Introduction

Graphics processing units (GPUs) have for well over a decade been used for general purpose computation, called GPGPU [8]. When the first GPU programs were written, the GPU was used much like a calculator: it had a set of fixed operations that were exploited to achieve some desired result. As the GPU is designed to output a 2D image from a 3D virtual world, the operations it could perform were fundamentally linked with graphics, and the first GPU programs were expressed as operations on graphical primitives such as triangles. These programs were difficult to develop, debug and optimize, and compiler bugs were frequently encountered. However, the proof-of-concept programs demonstrated that the use of GPUs could give dramatic speedups over CPUs for certain algorithms [20, 4], and research on GPUs soon led to the development of higher-level third-party languages that abstracted away the graphics. These languages, however, were rapidly abandoned when the hardware vendors released dedicated non-graphics languages that enabled the use of GPUs for general purpose computing (see Figure 1).

The key to the success of GPU computing has partly been its massive performance when compared to the CPU: Today, there is a performance gap of roughly seven times between the two when comparing theoretical peak bandwidth and gigaflops performance (see Figure 2). This performance gap has its roots in physical per-core restraints and architectural differences between the two processors. The CPU is in essence a serial *von Neumann* processor, and is highly optimized to execute a series of operations in order. One of the major performance factors of CPUs has traditionally been its steadily increasing frequency. If you double the frequency, you double the performance, and there has been a long withstanding trend of exponential frequency growth. In the 2000s, however, this increase came to an abrupt stop as we hit the *power wall* [3]: Because the power consumption of a CPU is proportional to the frequency cubed [4], the power density was approaching that of a nuclear reactor core [22]. Unable to cool such chips suffi-

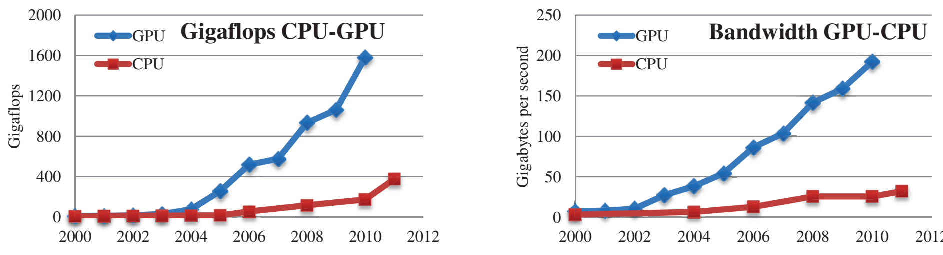


Figure 2: Historical comparison of theoretical peak performance in terms of gigaflops and bandwidth for the fastest available NVIDIA GPUs and Intel CPUs. Today, the performance gap is currently roughly seven times for both metrics.

ciently, the trend of exponential frequency growth stopped at just below 4.0 GHz. Coupled with the *memory wall* and the *ILP wall*, serial computing had reached its zenith in performance [3], and CPUs started increasing performance through multi-core and vector instructions instead.

At the same time as CPUs hit the serial performance ceiling, GPUs were growing exponentially in performance due to *massive* parallelism: As computing the color of a pixel on screen can be performed independently of all other pixels, parallelism is a natural way of increasing performance in GPUs. Parallelism appears to be a sustainable way of increasing performance, and there are many applications that display embarrassingly parallel workloads that are perfectly suited for GPUs. However, increased parallelism will only increase the performance of parallel code sections, meaning that the serial part of the code soon becomes the bottleneck. This is often referred to as Amdahl's law [2]. Thus, most applications benefit from the powerful combination of the massively parallel GPU and the fast multi-core CPU. This combination is known as a heterogeneous computer: the combination of traditional CPU cores and specialized accelerator cores [4].

GPUs are not the only type of accelerator core that has gained interest over the last decade. Other examples include field programmable gate arrays (FPGAs) and the Cell Broadband Engine (Cell BE), which have both been highly successful in many application areas [4]. Today, however, these receive only a fraction of the attention of GPUs (see Figure 3). One of the major reasons for this is that a very large percentage of desktop and laptop computers have a dedicated GPU already, whilst FPGAs and the Cell BE are only found in specially ordered setups. Furthermore, the future of the Cell BE is currently uncertain as the road map for the second version has not been followed through. FPGAs, on the other hand, have a thriving community in the embedded markets, but are unfortunately too hard to program for general purpose computing. The cumbersome and difficult programming process that requires detailed knowledge of low-level hardware has recently improved dramatically with the advent of C-like languages, but the development cycle is still slow due to the time consuming place and route stage.

There are three major GPU vendors for the PC market today, Intel being the largest. However, Intel is only dominant in the integrated and low-performance market. For high-performance and discrete graphics, AMD and NVIDIA are the sole two suppliers. In academic and industrial environments, NVIDIA appears to be the clear predominant supplier, and we thus focus on GPUs from NVIDIA in this article, even though most of the concepts and techniques are also

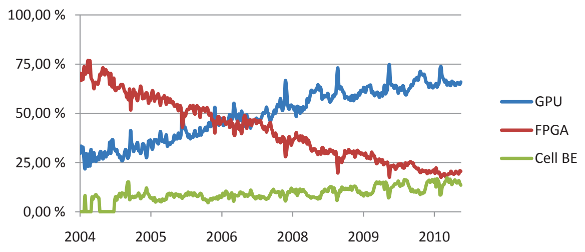


Figure 3: Google search trends for GPU, FPGA and Cell BE. These terms were chosen as the highest ranking keywords for each of the architectures. Whilst both FPGAs and the Cell BE today receive a modest number of searches, the GPU has a large growing interest.

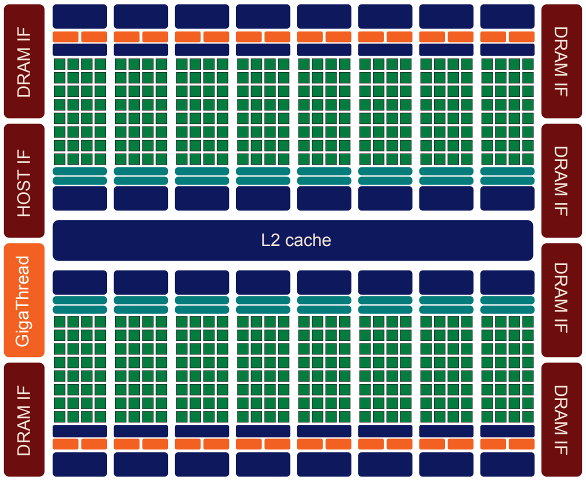
directly transferable to GPUs from AMD. For NVIDIA GPUs there are three languages suitable for general purpose computing as shown in Figure 1. Of these, we focus on NVIDIA CUDA. Even though the three languages are conceptually equivalent and offer more or less the same functionality, CUDA is the most mature technology with the most advanced development tools.

Much of the information in this article can be found in a variety of different sources, including books, documentation, manuals, conference presentations, and on Internet fora. Getting an overview of all this information is an arduous exercise that requires a substantial effort. The aim of this article is therefore to give an overview of state-of-the-art programming techniques and profile driven development, and to serve as a step-by-step guide for optimizing GPU codes. The rest of the article is sectioned as follows: First, we give a short overview of current GPU hardware in Section 2, followed by general programming strategies in Section 3. Then, we give a thorough overview of profile driven development for GPUs in Section 4, and a short overview of available debugging tools in Section 5. Finally, we offer our view on the current and future trends in Section 6, and conclude with a short summary in Section 7.

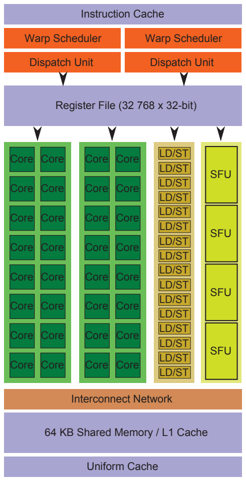
2 Fermi GPU Architecture

The multi-core CPU is composed of a handful of complex cores with large caches. The cores are optimized for single-threaded performance and can handle up-to two hardware threads per core using Hyper-Threading. This means that a lot of transistor space is dedicated to complex instruction level parallelism such as instruction pipelining, branch prediction, speculative execution, and out-of-order execution, leaving only a tiny fraction of the die area for integer and floating point execution units. In contrast, a GPU is composed of hundreds of simpler cores that can handle thousands of concurrent hardware threads. GPUs are designed to maximize floating-point throughput, whereby most transistors within each core are dedicated to computation rather than complex instruction level parallelism and large caches. The following part of this section gives a short overview of a modern GPU architecture.

Todays Fermi-based architecture [17] feature up to 512 accelerator cores called CUDA cores (see Figure 4a). Each CUDA core has a fully pipelined integer arithmetic logic unit (ALU) and a floating point unit (FPU) that executes one integer or floating point instruction per clock cycle. The CUDA cores are organized in 16 streaming multiprocessors, each with 32 CUDA cores (see Figure 4b). Fermi also includes a coherent L2 cache of 768 KB that is shared across all



(a)



(b)

Figure 4: Current Fermi-class GPU hardware. The GPU consisting of up-to 16 Streaming Multiprocessors (also known as SMs) is shown in (a), and (b) shows a single multiprocessor.

16 multiprocessors in the GPU, and the GPU has a 384-bit GDDR5 DRAM memory interface supporting up-to a total of 6 GB of on-board memory. A host interface connects the GPU to the CPU via the PCI express bus. The GigaThread global scheduler distributes thread blocks to multiprocessor thread schedulers (see Figure 4a). This scheduler handles concurrent *kernel*[1] execution and out of order thread block execution.

Each multiprocessor has 16 load/store units, allowing source and destination addresses to be calculated for 16 threads per clock cycle. Special Function Units (SFUs) execute intrinsic instructions such as sine, cosine, square root, and interpolation. Each SFU executes one instruction per thread, per clock. The multiprocessor schedules threads in groups of 32 parallel threads called *warps*. Each multiprocessor features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently. The Fermi dual warp scheduler selects two warps, and issues one instruction from each warp to a group of 16 CUDA cores, 16 load/store units, or four SFUs. The multiprocessor has 64 KB of on-chip memory that can be configured as 48 KB of *shared memory* with 16 KB of L1 cache or as 16 KB of shared memory with 48 KB of L1 cache.

A traditional critique of GPUs has been their lack of IEEE compliant floating point operations and error-correcting code (ECC) memory. However, these shortcomings have been addressed by NVIDIA, and all of their recent GPUs offer fully IEEE-754 compliant single and double precision floating point operations, in addition to ECC memory.

3 GPU Programming Strategies

Programming GPUs is unlike traditional CPU programming, because the hardware is dramatically different. It can often be a relatively simple task to get started with GPU programming

[1]A kernel is a GPU program that typically executes in a data-parallel fashion.

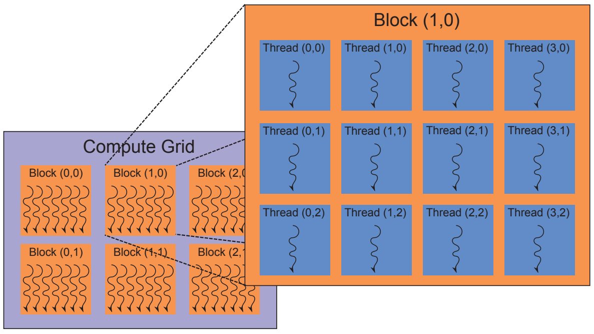


Figure 5: The CUDA concept concept of a grid of blocks. Each block consists of a set of threads that can communicate and cooperate. Each thread uses its block index in combination with its thread index to identify its position in the global grid.

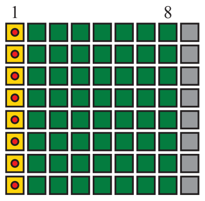
and get speedups over existing CPU codes, but these first attempts at GPU computing are often sub-optimal, and do not utilize the hardware to a satisfactory degree. Achieving a scalable high-performance code that uses hardware resources efficiently is still a difficult task that can take months and years to master.

Guidelines for Latency Hiding and Thread Performance

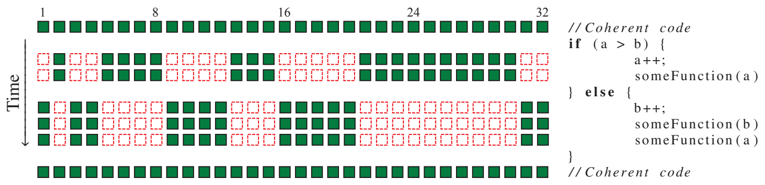
The GPU execution model is based around the concept of launching a kernel on a grid consisting of blocks (see Figure 5). Each block again consists of a set of threads, and threads within the same block can synchronize and cooperate using fast shared memory. This maps to the hardware so that a block runs on a single multiprocessor, and one multiprocessor can execute multiple blocks in a time-sliced fashion. The grid and block dimensions can be one, two, and three dimensional, and determine the number of threads that will be used. Each thread has a unique identifier within its block, and each block has a unique global identifier. These are combined to create a unique global identifier per thread.

The massively threaded architecture of the GPU is used to hide memory latencies. Even though the GPU has a vastly superior memory bandwidth compared to CPUs, it still takes on the order of hundreds of clock cycles to start the fetch of a single element from main GPU memory. This latency is automatically hidden by the GPU through rapid switching between threads. Once a thread stalls on a memory fetch, the GPU instantly switches to the next available thread in a fashion similar to Hyper-Threading [14] on Intel CPUs. This strategy, however, is most efficient when there are enough available threads to completely hide the memory latency, meaning we need a lot of threads. As there is a maximum number of concurrent threads a GPU can support, we can calculate how large a percentage of this figure we are using. This number is referred to as the *occupancy*, and as a rule of thumb it is good to keep a relatively high occupancy. However, a higher occupancy does not necessarily equate higher performance: Once all memory latencies are hidden, a higher occupancy may actually degrade performance as it also affects other performance metrics.

Hardware threads are available on Intel CPUs as Hyper-Threading, but a GPU thread operates quite differently from these CPU threads. One of the things that differs from traditional



(a)



```
//Coherent code
if (a > b) {
        a++;
        someFunction(a)
} else {
        b++;
        someFunction(b)
        someFunction(a)
}
//Coherent code
```

(b)

Figure 6: Bank conflicts and thread divergence. (a) shows conflict free column-wise access of shared memory. The example shows how padding shared memory to be the number of banks plus one gives conflict free access by columns (marked with circles for the first column). Without padding, all elements in the first column would belong to the same bank, and thus give an eight-way bank conflict. By padding the width by one, we ensure that the elements belong to different banks. Please note that the constructed example shows eight banks, whilst current hardware has 32 banks. (b) shows branching on 32-wide SIMD GPU architectures. All threads perform the same computations, but the result is masked out for the dashed boxes.

CPU programming is that the GPU executes instructions in a 32-way SIMD[2] fashion, in which the same instruction is simultaneously executed for 32 different data elements, called a warp. This is illustrated in Figure 6b, in which a branch is taken by only some of the threads within a warp. This means that all threads within a warp must execute both parts of the branch, which in the utmost consequence slows down the program by a factor 32. Conversely, this does not affect performance when all threads in a warp take the same branch.

One technique used to avoid expensive branching within a kernel is to sort the elements according to the branch, and thus make sure the threads within each warp all execute their code without branching. Another way of preventing branching, is to perform the branch once on the CPU instead of for each warp on the GPU. This can be done for example using templates: by replacing the branch variable with a template variable, we can generate two kernels: one for condition true, and one for condition false, and let the CPU perform the branch and from this select the correct kernel. The use of templates, however, is not particularly powerful in this example, as the overhead of running a simple coherent if-statement in the kernel would be small. But when there are a lot of parameters, there can be a large performance gain from using template kernels [7, 5]. Another prime example of the benefit of template kernels, is the ability to specify different shared memory sizes at compile time, thus allowing the compiler to issue warnings for out-of-bounds access. The use of templates can also be used to perform compile-time loop unrolling, which has a great performance impact. By using a switch-case statement,

[2]SIMD stands for single instruction multiple data.

with a separate kernel being launched for different for-loop sizes, performance can be greatly improved.

Memory Guidelines

CPUs have struggled with the memory wall for a long time. The memory wall, in which transferring data to the processor is far more expensive than computing on that data, can also be a problem on GPUs. This means that many algorithms will often be memory bound, making memory optimizations important. The first lesson in memory optimization is to reuse data and keep it in the fastest available memory. For GPUs, there are three memory areas, listed in decreasing order by speed: registers, shared memory, and global memory.

Registers are the fastest memory units on a GPU, and each multiprocessor on the GPU has a large, but limited, register file which is divided amongst threads residing on that multiprocessor. Registers are private for each thread, and if the threads use more registers than are physically available, registers will also spill to the L1 cache and global memory. This means that when you have a high number of threads, the number of registers available to each thread is very restricted, which is is one of the reasons why a high occupancy may actually hurt performance. Thus, thread-level parallelism is not the only way of increasing performance. It is also possible to increase performance by decreasing the occupancy to increase the number of registers available per thread.

The second fastest memory type is *shared memory*, and this memory can be just as fast as registers if accessed properly. Shared memory is a very powerful tool in GPU computing, and the main difference between registers and shared memory is the ability for several threads to share data. Shared memory is accessible to all threads within one block, thus enabling cooperation. It can be thought of as a kind of programmable cache, or scratchpad, in which the programmer is responsible for placing often used data there explicitly. However, as with caches, its size is limited (up-to 48 KB) and this can often be a limitation on the number of threads per block. Shared memory is physically organized into 32 banks that serves one warp with data simultaneously. However, for full speed, each thread must access a distinct bank. Failure to do so leads to more memory requests, one for each *bank conflict*. A classical way to avoid bank conflicts is to use padding. In Figure 6a, for example, we can avoid bank conflicts for column-wise access by padding the shared memory with an extra element, so that neighboring elements in the same column belong to different banks.

The third, and slowest type of memory on the GPU is the *global memory*, which is the main memory of the GPU. Even though it has an impressive bandwidth, it has a high latency, as discussed earlier. These latencies are preferably hidden by a large number of threads, but there are still large pitfalls. First of all, just as with CPUs, the GPU transfers full cache lines across the bus (called coalesced reads). As a rule of thumb, transferring a single element consumes the same bandwidth as transferring a full cache line. Thus, to achieve full memory bandwidth, we should program the kernel such that warps access continuous regions of memory. Furthermore we want to transfer full cache lines, which is done by starting at a quad word boundary (the start address of a cache line), and transfer full quadwords (128 bytes) as the smallest unit. This address alignment is typically achieved by padding arrays. Alternatively, for non-cached loads, it is sufficient to align to word boundaries and transfer words (32 bytes). To fully occupy the memory bus the GPU also uses memory parallelism, in which a large number of outstanding memory requests are used to occupy the bandwidth. This is both a reason for a high memory latency, and a reason for high bandwidth utilization.

Fermi also has hardware L1 and L2 caches that work in a similar fashion as traditional CPU caches. The L2 cache size is fixed and shared between all multiprocessors on the GPU, whilst the L1 cache is per multiprocessor. The L1 cache can be configured to be either 16 KB or 48 KB, at the expense of shared memory. The L2 cache, on the other hand, can be turned on or off on at compile-time, or by using inline PTX assembly instructions in the kernel. The benefit of turning off the L2 cache is that the GPU is now allowed to transfer smaller amounts of data than a full cache line, which will often improve performance for sparse and other random access algorithms.

In addition to the L1 and L2 caches, the GPU also has dedicated caches that are related to traditional graphics functions. The constant memory cache is one example, which in CUDA is typically used for arguments sent to a GPU kernel. It has its own dedicated cache tailored for broadcast, in which all threads in a block access the same data. The GPU also has a texture cache that can be used to accelerate reading global memory. However, the L1 cache has a higher bandwidth, so the texture cache is mostly useful if combined with texture functions such as linear interpolation between elements.

Further Guidelines

The CPU and the GPU are different processors that operate asynchronously. This means that we can let the CPU and the GPU perform different tasks simultaneously, which is a key ingredient of heterogeneous computing: the efficient use of multiple different computational resources, in which each resource performs the taks for which it is best suited. In the CUDA API, this is exposed as *streams*. Each stream is an in-order queue of operations that will be performed by the GPU, including memory transfers and kernel launches. A typical use-case is that the CPU schedules a memory copy from the CPU to the GPU, a kernel launch, and a copy of results from the GPU to the CPU. The CPU then continues to perform CPU-side calculations simultaneously as the GPU processes its operations, and only synchronizes with the GPU when its results are needed. There is also support for independent streams, which can execute their operations simultaneously as long as they obey their own streams order. Current GPUs support up-to 16 concurrent kernel launches [18], which means that we can both have data parallelism, in terms of a computational grid of blocks, and task parallelism, in terms of different concurrent kernels. GPUs furthermore support overlaping memory copies between the CPU and the GPU and kernel execution. This means that we can simultaneously copy data from the CPU to the GPU, execute 16 different kernels, and copy data from the GPU back to the CPU if all these operations are scheduled properly to different streams.

When transferring data between the CPU and the GPU over the PCI express bus, it is beneficial to use so-called page-locked memory. This essentially disables the operating system from paging memory, meaning that the memory area is guaranteed to be continuous and in physical RAM (not swapped out to disk, for example). However, page-locked memory is scarce and rapidly exhausted if used carelessly. A further optimization for page-locked memory is to use write-combining allocation. This disables CPU caching of a memory area that the CPU will only write to, and can increases the bandwidth utilization by up-to 40% [18]. It should also be noted that enabling ECC memory will negatively affect both the bandwidth utilization and available memory, as ECC requires extra bits for error control.

CUDA now also supports a *unified address space*, in which the physical location of a pointer is automatically determined. That is, data can be copied from the GPU to the CPU (or the other way round) without specifying the direction of the copy. While this might not seem like a great benefit at first, it greatly simplifies code needed to copy data between CPU and GPU memories, and enables advanced memory accesses. The unified memory space is particularly powerful when combined with mapped memory. A mapped memory area is a continuous block of memory that is available directly from both the CPU and the GPU simultaneously. When using mapped memory, data transfers between the CPU and the GPU are executed asynchronously with kernel execution automatically.

The most recent version of CUDA has also become thread safe [18], so that one CPU thread can control multiple CUDA contexts (e.g., one for each physical GPU), and conversely multiple CPU threads can share control of one CUDA context. The unified memory model together with the new thread safe context handling enables much faster transfers between multiple GPUs. The CPU thread can simply issue a direct GPU-GPU copy, bypassing a superfluous copy to CPU memory.

4 Profile Driven Development[3]

A famous quote attributed to Donald Knuth is that "premature optimization is the root of all evil" [11], or put another way; make sure the code produces the correct results before trying to optimize it, and optimize only where it will make an impact. The first step in optimization is always to identify the major application bottlenecks, as performance will increase the most when removing these. However, locating the bottleneck is hard enough on a CPU, and can be even more difficult on a GPU. Optimization should also be considered a cyclic process, meaning that after having found and removed one bottleneck, we need to repeat the profiling process to find the next bottleneck in the application. This cyclic optimization can be repeated until the kernel operates close to the theoretical hardware limits or all optimization techniques have been exhausted.

To identify the performance bottleneck in a GPU application, it is important to chose the appropriate performance metrics, and then compare the measured performance to the theoretical peak performance. There are several bottlenecks one can encounter when programming GPUs. For a GPU kernel, there are three main bottlenecks; the kernel may be limited by instruction throughput, memory throughput, or latencies. However, it might also be that CPU-GPU communication is the bottleneck, or that application overheads dominate the run-time.

Locating Kernel Bottlenecks

There are two main approaches to locating the performance bottleneck of a CUDA kernel, the first and most obvious being to use the CUDA profiler. The profiler is a program that that samples different hardware counters, and the correct interpretation of these numbers is required to identify bottlenecks. The second option is to modify the source code, and compare the execution time of the differently modified kernels.

The profiler can be used to identify whether a kernel is limited by bandwidth or arithmetic

[3]Many of the optimization techniques presented in this section are from the excellent presentations by Paulius Micikevisius [16, 15].

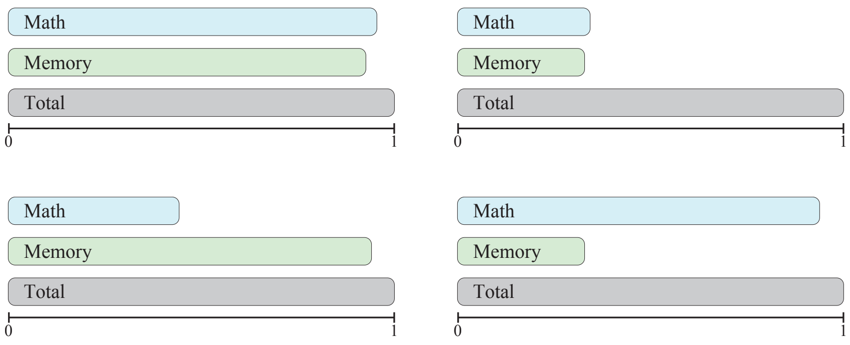


Figure 7: Normalized run-time of modified kernels which are used to identify bottlenecks: (top left) a well balanced kernel, (top right) a latency bound kernel, (bottom left) a memory bound kernel, and (bottom right) an arithmetic bound kernel. "Total" refers to the total kernel time, whilst "Memory" refers to a kernel stripped of arithmetic operations, and "Math" refers to a kernel stripped of memory operations. It is important to note that latencies are part of the measured run-times for all kernel versions.

operations. This is done by simply looking at the instruction-to-byte ratio, or in other words finding out how many arithmetic operations your kernel performs per byte it reads. The ratio can be found by comparing the instructions issued counter (multiplied with the warp size, 32) to the sum of global store transactions and L1 global load miss counters (both multiplied with the cache line size, 128 bytes), or directly through the `instruction/byte` counter. Then we compare this ratio to the theoretical ratio for the specific hardware the kernel is running on, which is available in the profiler as the `Ideal Instruction/Byte ratio` counter.

The profiler does not always report accurate figures because the number of load and store instructions may be lower than the actual number of memory transactions, depending on address patterns and individual transfer sizes. To get the most accurate figures, we can follow another strategy which is to compare the run-time of three modified versions of the kernel: The original kernel, one *Math* version in which all memory loads and stores are removed, and one *Memory* version in which all arithmetic operations are removed (see Figure 7). If the *Math* version is significantly faster than the original and *Memory* kernels, we know that the kernel is memory bound, and conversely for arithmetics. This method has the added benefit of showing how well memory operations and arithmetic operations overlap.

To create the *Math* kernel, we simply comment out all load operations, and move every store operation inside conditionals that will always evaluate to false. We do this to fool the compiler so that it does not optimize away the parts we want to profile, since the compiler will strip away all code not contributing to the final output to global memory. However, to make sure that the compiler does not move the computations inside the conditional as well, the result of the computations must also be used in the condition as shown in Listing 1. Creating the *Memory* kernel, on the other hand, is much simpler. Here, we can simply comment out all arithmetic operations, and instead add all data used by the kernel, and write out the sum as the result.

Note that if control flow or addressing is dependent on data in memory the method becomes less straightforward and requires special care. A further issue with modifying the source code

```
__global__ void main(..., int flag) {
  float result = ...;
  if(1.0f == result * flag)
    output[i] = value;
}
```

Listing 1: Compiler trick for arithmetic only kernel. By adding the kernel argument *flag* (which we always set to 0), we disable the compiler from optimizing away the if-statement, and simultaneously disable the global store operation.

is that register count can change, which again can increase the occupancy and thereby invalidate the measured run-time. This, however, can be is solved by increasing the shared memory parameter in the launch configuration of the kernel,
`someKernel<<<grid_size, block_size, shared_mem_size, ...>>>(...)`,
until the occupancy of the unmodified version is matched. The occupancy can easily be examined using the profiler or the CUDA Occupancy Calculator.

If a kernel appears to be well balanced (i.e., neither memory nor arithmetics appear to be the bottleneck), we must still check whether or not our kernel operates close to the theoretical performance numbers since it can suffer from latencies. These latencies are typically caused by problematic data dependencies or the inherent latencies of arithmetic operations. Thus, if your kernel is well balanced, but operates at only a fraction of the theoretical peak, it is probably bound by latencies. In this case, a reorganization of memory requests and arithmetic operations is often required. The goal should be to have many outstanding memory requests that can overlap with arithmetic operations.

Profiling and Optimizing Memory

Let us assume that we have identified the major bottleneck of the kernel to be memory transactions. The first and least time-consuming thing to try for memory-bound kernels is to experiment with the settings for caching and non-caching loads and the size of the L1 cache to find the best settings. This can have a large impact in cases of register spilling and for strided or scattered memory access patterns, and it requires no code changes. Outside these short experiments, however, there are two major factors that we want to examine, and that is the access pattern and the number of concurrent memory requests.

To determine if the access pattern is the problem, we compare the number of memory instructions with the number of transferred bytes. For example, for global load we should compare the number of bytes requested (`gld_request` multiplied by bytes per request, 128 bytes) to the number of transferred bytes (the sum of `l1_global_load_miss` and `l1_global_load_hit` multiplied by the cache line size, 128 bytes). If the number of instructions per byte is far larger than one, we have a clear indication that global memory loads have a problematic access pattern. In that case, we should try reorganizing the the memory access pattern to better fit with the rules in Section 3. For global store, the counters we should compare are `gst_request` and `global_store_transactions`.

If we are memory bound, but the access patterns are good, we might be suffering from having too few outstanding memory requests: according to Little's Law [13], we need (memory latency $\times$ bandwidth) bytes in flight to saturate the bus. To determine if the number of concurrent memory accesses is too low, we can compare the achieved memory throughput (`glob_mem_read_throughput` and `glob_mem_write_throughput` in the profiler) against the theoretical peak. If the hardware throughput is dramatically lower, the memory bus is not saturated, and we should try increasing the number of concurrent memory transactions by increasing the occupancy. This can be done through adjustment of block dimensions or reduction of register count, or we can modify the kernel to process more data elements per thread. A further optimization path is to move indexing calculations and memory transactions in an attempt to achieve better overlap of memory transactions and arithmetic operations.

An important note when it comes to optimization of memory on the GPU, is that traditional CPU cache blocking techniques typically do not work. This is because the GPUs L1 and L2 caches are not aimed at *temporal* reuse like CPU caches usually are, which means that attempts at cache blocking can even be counter-productive. The rule of thumb is therefore that when optimizing for memory throughput on GPUs, do not think of caches at all. However, fetching data from textures can alleviate pressure on the memory system since these fetches go through a different cache in smaller transactions. Nevertheless, the L1 cache is superior in performance, and only in rare cases will the texture cache increase performance [18].

Profiling and Optimizing Latencies and Instruction Throughput

If a kernel is bound by instruction throughput, there may be several underlying causes. Warp serialization (see Section 3) may be a major bottleneck, or we may similarly have that bank conflicts cause shared memory serialization. The third option is that we have data dependencies that inhibit performance.

Instruction serialization means that some threads in a warp "replay" the same instruction as opposed to all threads issuing the same instruction only once (see Figure 6b). The profiler can be used to determine the level of instruction serialization by comparing the `instructions_executed` counter to the `instructions_issued` counter, in which the difference is due to serialization. Note that even if there is a difference between instructions executed and instructions issued, this is only a problem if it constitutes a significant percentage.

One of the causes for instruction replays is divergent branches, identified by comparing the `divergent_branch` counter to the `branch` counter. We can also profile it by modifying the source code so that all threads take the same branch, and compare the run-times. The remedy is to remove as many branches as possible, for example by sorting the input data or splitting the kernel into two separate kernels (see Section 3).

Another cause for replays is bank conflicts, which can be the case if the `l1_shared_bank_conflict` counter is a significant percentage of the sum of the `shared_loads` and `shared_stores` counters. Another way of profiling bank conflicts is to modify the kernel source code by removing the bank conflicts. This is done by changing the indexing so that all accesses are either broadcasts (all threads access the same element) or conflict free (each thread uses the index `threadIdx.y*blockDim.x+threadIdx.x`). The shared memory variables also need to be declared as volatile to prevent the compiler from storing them in registers in the modified kernel. Padding is one way of removing these bank conflicts (see Section 3), and one can also try rearranging the shared memory layout (e.g., by storing by columns instead of by rows).

If we have ruled out the above causes, our kernel might be suffering from arithmetic operation latencies and data dependencies. We can find out if this is the case by comparing the kernel performance to hardware limits. This is done by examining the `IPC - Instructions/Cycle` counter in the profiler, which gives the ratio of executed instructions per clock cycle. For compute capability 2.0, this figure should be close to 2, whilst for compute capability 2.1 it should approach 4 instructions per cycle. If the achieved instructions per cycle count is very low, this is a clear indication that there are data dependencies and arithmetic latencies that affect the performance. In this case, we can try storing intermediate calculations in separate registers to minimize arithmetic latencies due to register dependencies.

Further Optimization Parameters

Grid- and block-size are important optimization parameters, and they are usually not easy to set. Both the grid size and the block size must be chosen according to the size and structure of the input data, but they must also be tuned to fit the GPU's architecture in order to yield a high performance. Most importantly we need enough total threads to keep the GPU fully occupied in order to hide memory and other latencies. Since each multiprocessor can execute up-to eight blocks simultaneously, choosing too small blocks prevents a high occupancy. Simultaneously, we do not want too large blocks since this may cause register spilling if the kernel uses a lot of registers. The number of threads per block should also, if possible, be a multiple of 32, since each multiprocessor executes full warps. When writing a kernel for the GPU, one also often encounters a situation in which the number of data elements is not a multiple of the block size. In this case, it is recommended to launch a grid larger than the number of elements and use an out-of-bounds test to discard the unnecessary computations.

In many cases, it is acceptable to trade accuracy for performance, either because we simply need a rough estimate, or that modeling or data errors shadow the inevitable floating point rounding errors. The double-precision to single-precision ratio for GPUs is 2:1 [4], which means that a double precision operation takes twice as long as a single precision operation (just as for CPUs). This makes it well worth investigating whether or not single-precision is sufficiently accurate for the application. For many applications, double precision is required for calculating results, but the results themselves can be stored in single precision without loss of accuracy. In these cases, all data transfers will execute twice as fast, simultaneously as we will only occupy half the space in memory. For other cases, single precision is sufficiently accurate for arithmetics as well, meaning we can also perform the floating point operations twice as fast. Remember also that all floating-point literals without the `f` suffix are interpreted as 64-bit according to the C standard, and that when one operand in an expression is 64-bit, all operations must be performed in 64-bit precision. Some math functions can be compiled directly into faster, albeit less accurate, versions. This is enabled using double underscore version of the function, for example `__sin()` instead of `sin()`. By using the `--use_fast_math` compiler flag, all fast hardware math functions that are available will be used. It should also be noted that this compiler flag also treats denormalized numbers as zero, and faster (but less accurate) approximations are used for divisions, reciprocals, and square roots.

Even if an application does none of its computing on the CPU, there still must be some CPU code for setting up and controlling CUDA contexts, launching kernels, and transferring

[4]For the GeForce products this ratio is 8:1.

data between the CPU and the GPU. In most cases it is also desirable to have some computations performed on the CPU. There is almost always some serial code in an algorithm that cannot be parallelized and therefore will execute faster on the CPU. Another reason is that there is no point in letting the CPU idle while the GPU does computations: use both processors when possible.

There are considerable overheads connected with data transfers between the CPU and the GPU. To hide a memory transfer from the CPU to the GPU before a kernel is launched one can use streams, issue the memory transfer asynchronously, and do work on the CPU while the data is being transferred to the GPU (see Section 3). Since data transfers between the CPU and the GPU goes through the PCI Express bus both ways, these transfers will often be a bottleneck. By using streams and asynchronous memory transfers, and by trying to keep the data on the GPU as much as possible, this bottleneck can be reduced to a minimum.

Auto-tuning of GPU Kernels

The above mentioned performance guidelines are often conflicting, one example being that you want to optimize for occupancy to hide memory latencies simultaneously as you want to increase the number of per-thread registers for more per-thread storage. The first of these criteria requires more threads per block, the second requires fewer threads per block, and it is not given which configuration will give the best performance. With the sheer number of conflicting optimization parameters, it rapidly becomes difficult to find out what to optimize. Experienced developers are somewhat guided by educated guesses together with a trial and error approach, but finding the global optimum is often too difficult to be performed manually.

Auto-tuning strategies have been known for a long time on CPUs, and are used to optimize the performance using cache blocking and many other techniques. On GPUs, we can similarly create auto-tuning codes that execute a kernel for each set of optimization parameters, and select the best performing configuration. However, the search space is large, and brute force techniques are thus not a viable solution. Pruning of this search space is still an open research question, but several papers on the subject have been published (see for example [12, 6]).

Even if auto-tuning is outside the scope of a project, preparing for it can still be an important part of profiler guided development. The first part of auto-tuning is often to use template arguments for different kernel parameters such as shared memory, block size, etc. This gives you many different varieties of the same kernel so that you can easily switch between different implementations. A benefit of having many different implementations of the same kernel, is that you can perform run-time auto-tuning of your code. Consider the following example: For a dam break simulation on the GPU, you might have one kernel that is optimized for scenarios in which most of the domain is dry. However, as the dam breaks, water spreads throughout the domain, making this kernel inefficient. You then create a second kernel that is efficient when most of the domain contains water, and switch between these two kernels at run-time. One strategy here is to perform a simple check of which kernel is the fastest after given number of iterations, and use this kernel. Performing this check every hundredth time-step, for example, gives a very small overhead to the total computational time, and ensures that you are using the most efficient kernel throughout the simulation.

Reporting Performance

One of the key points made in early GPU papers was that one could obtain high speedups over the CPU for a variety of different algorithms. The tendency has since been to report ever increasing speedups, and today papers report that their codes run anything from tens to hundreds and thousands times faster than CPU "equivalents". However, when examining the theoretical performance of the architectures, the performance gap is roughly seven times between state-of-the-art CPUs and GPUs (see Figure 2). Thus, reporting a speedup of hundreds of times or more holds no scientific value without further explanations supported by detailed benchmarks and profiling results.

The sad truth about many papers reporting dramatic speedup figures is that the speedup is misleading, at best. Often, a GPU code can be compared to an inefficient CPU code, or a state-of-the-art desktop GPU can be compared to a laptop CPU several years old. Some claims of SSE and other optimizations of the CPU code are often made, giving the impression that the CPU code is efficient. However, for many implementations, this still might not be the case: if you optimize using SSE instructions, but the bottleneck is memory latency your optimizations are worthless in a performance perspective.

Reporting performance is a difficult subject, and the correct way of reporting performance will vary from case to case. There is also a balance between the simplicity of the benchmark and its ease of interpretation. For example, the Top500 list [23], which rates the fastest supercomputers in the world, is often criticized for being too simplistic as it gives a single rating (gigaflops) from a single benchmark (solving a linear system of equations). For GPUs, we want to see an end to the escalating and misleading speedup-race, and rather see detailed profiling results. This will give a much better view of how well the algorithm exploits the hardware, both on the CPU and on the GPU. Furthermore, reporting how well your implementation utilizes the hardware, and what the bottlenecks are, will give insight into how well it will perform on similar and even future hardware. Another benefit here, is that it becomes transparent what the bottleneck of the algorithm is, meaning it will become clear what hardware vendors and researchers should focus on for improving performance.

5 Debugging

As an ever increasing portion of the C++ standard is supported by CUDA and more advanced debugging tools emerge, debugging GPU codes becomes more and more like debugging CPU codes. Many CUDA programmers have encountered the "unspecified launch failure", which could be notoriously hard to debug. Such errors were typically only found by either modification and experimenting, or by careful examination of the source code. Today, however, there are powerful CUDA debugging tools for all the most commonly used operating systems.

CUDA-GDB, available for Linux and Mac, can step through a kernel line by line at the granularity of a warp, e.g., identifying where an out-of-bounds memory access occurs, in a similar fashion to debugging a CPU program with GDB. In addition to stepping, CUDA-GDB also supports breakpoints, variable watches, and switching between blocks and threads. Other useful features include reports on the currently active CUDA threads on the GPU, reports on current hardware and memory utilization, and in-place substitution of changed code in running CUDA application. The tool enables debugging on hardware in real-time, and the only requirement for using CUDA-GDB is that the kernel is compiled with the `-g -G` flags. These flags make the compiler add debugging information into the executable, and the executable to spill all variables to memory.

On Microsoft Windows, Parallel NSight is a plug-in for Microsoft Visual Studio which offers conditional breakpoints, assembly level debugging, and memory checking directly in the Visual Studio IDE. It furthermore offers an excellent profiling tool, and is freely available to developers. However, debugging requires two distinct GPUs; one for display, and one for running the actual code to be debugged.

6 Trends in GPU Computing

GPUs have truly been a disruptive technology. Starting out as academic examples, they were shunned in scientific and high-performance communities: they were inflexible, inaccurate, and required a complete redesign of existing software. However, with time, there has been a low-end disruption, in which GPUs have slowly conquered a large portion of the high-performance computing segment, and three of five of todays fastest supercomputers are powered mainly by GPUs [23]. GPUs were never designed for high-performance computing in mind, but have nevertheless evolved into powerful processors that fit this market perfectly in less than ten years. The software and the hardware has developed in harmony, both due to the hardware vendors seeing new possibilities and due to researchers and industry identifying points of improvement. It is impossible to predict the future of GPUs, but by examining its history and current state, we might be able to identify some trends that are worth noting.

The success of GPUs has partly been due to its price: GPUs are ridiculously inexpensive in terms of performance per dollar. This again comes from the mass production and target market. Essentially, GPUs are inexpensive because NVIDIA and AMD sell a lot of GPUs to the entertainment market. After researchers started to exploit GPUs, these hardware vendors have eventually developed functionality that is tailored for general-purpose computing and started selling GPUs intended for computing alone. Backed by the mass market, this means that the cost for the vendors to target this emerging market is very low, and the profits high.

There are great similarities between the vector machines of the 1990ies and todays use of GPUs. The interest in vector machines eventually died out, as the x86 market took over. Will the same happen to GPUs once we conquer the power wall? In the short term, the answer is no for several reasons: First of all, conquering the power wall is not even on the horizon, meaning that for the foreseeable future, parallelism will be the key ingredient that will increase performance. Also, while vector machines were reserved for supercomputers alone, todays GPUs are available in everything from cell phones to supercomputers. A large number of software companies now use GPUs for computing in their products due to this mass market adaption, and this is one of the key reasons why we will have GPUs or GPU-like accelerator cores in the future. Just as x86 is difficult to replace, it is now becoming difficult to replace GPUs, as the software and hardware investments in this technology are large and increasing.

NVIDIA have just released their most recent architecture, called Kepler [19], which has several updates compared to the Fermi architecture described in this paper. First of all, it has changed the organization of the multiprocessors: whilst the Fermi architecture has up-to sixteen streaming multiprocessors, each with 32 CUDA cores, the new Kepler architecture has only four multiprocessors, but each with 192 CUDA cores. This totals to 1536 CUDA cores for one chip, compared to 512 for the Fermi architecture. A second change is that the clock frequency has been decreased from roughly 1.5 GHz to just over 1 GHz. These two changes are essentially a continuation of the existing trend of increasing performance through more cores running at a decreased clock frequency. Combined with a new production process of 28 nm (compared to 40 nm for Fermi), the effect is that the new architecture has a lower power consumption, yet roughly doubles the gigaflops performance. The bandwidth to the L2 cache has been increased giving a performance boost for applications that can utilize the L2 cache well, yet the main memory bandwidth is the same. NVIDIA has also announced that it plans to build high-performance processors with integrated CPU cores and GPU cores based on the low-power ARM architecture and the future Maxwell GPU architecture.

Competition between CPUs and GPUs is likely to be intense for conquering the high performance and scientific community, and heterogeneous CPU-GPU systems are already on the market. The AMD Fusion architecture [1] incorporates multiple CPU and GPU cores into a single die, and Intel have released their Sandy Bridge architecture based on the same concept. Intel have also developed other highly parallel architectures, including the Larrabee [21] based on simple and power efficient x86 cores, the Single-chip Cloud Computer (SCC) [10], and the 80-core Tera-scale research chip Polaris [24]. These architectures have never been released as commercial products, but have culminated into the Knights Corner co-processor with up-to 1 teraflops performance [9]. This co-processor resides on an adapter connected through the PCI express bus, just as todays graphics adapters, but exposes a traditional C, C++ and Fortran programming environment, in which existing legacy code can simply be recompiled for the new architecture.

Today, we see that the processors converge towards incorporating traditional CPU cores in addition to highly parallel accelerator cores on the same die, such as the aforementioned AMD Fusion and Intel Sandy Bridge. These architectures do not target the high-performance segment. However, NVIDIA have plans for the high performance segment with their work on combining ARM CPU cores and Maxwell GPU cores on the same chip. We thus see it as likely that we will see a combination of CPU and GPU cores on the same chip, sharing the same memory space, in the near future. This will have a dramatic effect on the memory bottleneck that exists between these architectures today, and open for tighter cooperation between fast serial execution and massive parallel execution to tackle Amdahl's law.

7 Summary

In this article, we have given an overview of hardware and traditional optimization techniques for the GPU. We have furthermore given a step-by-step guide to profile driven development, in which bottlenecks and possible solutions are outlined. The focus is on state-of-the-art hardware with accompanying tools, and we have addressed the most prominent bottlenecks: memory, arithmetics, and latencies.

Bibliography

[1] Advanced Micro Devices. AMD Fusion family of APUs: Enabling a superior, immersive PC experience. Technical report, 2010.

[2] G. M. Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities*, chapter 2, pages 79–81. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[3] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, EECS Department, University of California, Berkeley, December 2006.

[4] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, May 2010.

[5] A. R. Brodtkorb, M. L. Sætra, and M. Altinakar. Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. *Computers & Fluids*, 55(0):1–12, 2012.

[6] A. Davidson and J. D. Owens. Toward techniques for auto-tuning GPU algorithms. In *Proceedings of Para 2010: State of the Art in Scientific and Parallel Computing*, 2010.

[7] M. Harris. NVIDIA GPU computing SDK 4.1: Optimizing parallel reduction in CUDA, 2011.

[8] M. Harris and D. Göddeke. General-purpose computation on graphics hardware. `http://gpgpu.org`.

[9] Intel. Intel many integrated core (Intel MIC) architecture: ISC'11 demos and performance description. Technical report, 2011.

[10] Intel Labs. The SCC platform overview. Technical report, Intel Corporation, 2010.

[11] D. E. Knuth. Structured programming with go to statements. *Computing Surveys*, 6:261–301, 1974.

[12] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning gemm for GPUs. In *Proceedings of the 9th International Conference on Computational Science: Part I*, 2009.

[13] J. D. C. Little and S. C. Graves. *Building Intuition: Insights from Basic Operations Management Models and Principles*, chapter 5, pages 81–100. Springer, 2008.

[14] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):1–12, 2002.

[15] P. Micikevicius. Analysis-driven performance optimization. [Conference presentation], 2010 GPU Technology Conference, session 2012, 2010.

[16] P. Micikevicius. Fundamental performance optimizations for GPUs. [Conference presentation], 2010 GPU Technology Conference, session 2011, 2010.

[17] NVIDIA. NVIDIA's next generation CUDA compute architecture: Fermi, 2010.

[18] NVIDIA. NVIDIA CUDA programming guide 4.1, 2011.

[19] NVIDIA. NVIDIA GeForce GTX 680. Technical report, NVIDIA Corporation, 2012.

[20] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[21] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):18:1–18:15, Aug. 2008.

[22] G. Taylor. Energy efficient circuit design and the future of power delivery. [Conference presentation], Electrical Performance of Electronic Packaging and Systems, October 2009.

[23] Top 500 supercomputer sites. `http://www.top500.org/`, November 2011.

[24] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w teraflops processor in 65-nm CMOS. *Solid-State Circuits*, 43(1):29–41, Jan. 2008.

Paper II

Explicit Shallow Water Simulations on GPUs: Guidelines and Best Practices

A. R. Brodtkorb and M. L. Sætra

In proceedings of the XIX International Conference on Computational Methods in Water Resources, 2012

Abstract: Graphics processing units have now been used for scientific calculations for over a decade, going from early proof-of-concepts to industrial use today. The inherent reason is that graphics processors are far more powerful than CPUs when it comes to both floating point operations and memory bandwidth, illustrated by the fact that a growing portion of the top 500 supercomputers in the world now use GPU acceleration. In this paper, we present guidelines and best practices for harvesting the power of graphics processing units for shallow water simulations through stencil computations.

1 Introduction

The shallow water equations are a set of hyperbolic partial differential equations, and as such can be solved using explicit finite difference and finite volume schemes with compact stencils; schemes that map very well to the hardware architecture of graphics processing units (GPUs). In fact, GPUs have been used successfully for solving the shallow water equations since the very beginning of GPU computing (see, e.g., [5]). The most powerful processor in everything from laptops to supercomputers today is typically the GPU, and this is the single most important reason for using GPUs for general purpose computations. In this paper, we present guidelines and best practices for implementing explicit stencil based shallow water solvers on GPUs. Our main focus is on NVIDIA hardware, as this is the most used platform in academia, but most of the techniques described also apply to GPUs from AMD. We briefly introduce the GPU and the shallow water equations in this section, before we describe strategies for mapping the shallow water equations to the GPU in Section 2, and summarize in Section 3.

Graphics processing units: Dedicated processors that accelerated graphics operations were introduced in the 80'ies to offload demanding graphics from the CPU, and in 1999 NVIDIA coined the term GPU with the release of the GeForce 256 graphics card. Around the same time we also saw the first use of GPUs for non-graphics applications [10]. These early graphics cards accelerated a fixed set of graphics operations such as vertex transformations, lighting and texturing, later the fixed functionality has gradually been replaced with fully programmable *shading*. In 2007, NVIDIA released CUDA [4], a language dedicated to GPU computing, which sparked a huge interest in the use of GPUs for scientific applications [9]. Today, we see an ever increasing trend of using the GPU to accelerate high-performance computing, and three of the top five supercomputers on the top 500 list [8] now utilize GPUs.

The major difference between CPUs and GPUs is their architectural design. Current multi-core CPUs are designed for simultaneous execution of multiple applications, and use complex logic and a high clock frequency to execute each application in the shortest possible time. GPUs, on the other hand, are designed for calculating the color of millions of screen pixels in parallel from a complex 3D game world. This essentially means that while CPUs are designed to minimize single thread latency, GPUs are designed for throughput. This is also reflected in how the transistors are used. CPUs use most of their *transistor budget* on huge caches and complex logic to minimize latencies, leaving a very small percentage of transistors for actual computations. GPUs, on the other hand, spend most of their transistor budget on computational units, and have very limited caches and very little of the complex logic found in CPUs. If we compare the state of the art, CPUs such as the Intel Core i7-3960X can have up-to 6 cores $\times$ 8-way SIMD = 48 floating point units, whilst GPUs such as the NVIDIA GeForce 580 GTX can have up-to 16 cores $\times$ 32-way SIMD = 512, an increase of an order of magnitude[5].

The shallow water equations: The shallow water equations are applicable for a wide range of problems, such as dam breaks, inundations, oceanographic currents, avalanches, and other phenomena and scenarios in which the governing flow is horizontal. This system of equations is also representative of the wider class of hyperbolic partial differential equations, and techniques developed for the shallow water equations are also often applicable to other hyperbolic conser-

[5]This comparison assumes single precision operations. It should also be mentioned that a GPU core is quite different from a CPU core, and we refer the reader to [4] for a full overview.

vation laws. In the simplest case, the homogeneous shallow water equations in two spatial dimensions can be written

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

or in vector form, $Q_t + F(Q)_x + G(Q)_y = 0$. Here, Q is our vector of conserved variables, and F and G are flux functions that give rise to gravity waves. In the case of water, h will be the water depth, hu and hv is the momentum along the abscissa and ordinate, respectively, and g is the gravitational acceleration. In this paper, we focus on modern explicit schemes with compact stencils that solve these equations (see, e.g., [13, 7]), as these schemes are often highly suitable for implementation on GPUs.

2 Mapping the Shallow Water Equations to the GPU

Explicit schemes with compact stencils map well to the GPU architecture, since each output element can be computed independently of all other elements, giving rise to a high level of parallelism. In this section, we will illustrate how the classical Lax-Friedrichs finite volume scheme can be mapped to the GPU as an example. It should be noted that the presented code is for illustrative purposes only, and disregard many important optimization parameters. Let us start by writing up the classical Lax-Friedrichs scheme for a volume (i, j):

$$\begin{aligned} Q_{ij}^{n+1} &= \tfrac{1}{4}\left(Q_{i,j+1}^n + Q_{i,j-1}^n + Q_{i+1,j}^n + Q_{i-1,j}^n\right) \\ &- \tfrac{\Delta t}{2\Delta x}\left[F(Q_{i+1,j}^n) - F(Q_{i-1,j}^n)\right] - \tfrac{\Delta t}{2\Delta y}\left[G(Q_{i,j+1}^n) - G(Q_{i,j-1}^n)\right]. \end{aligned} \tag{2}$$

Here, we explicitly calculate the vector Q at the next time step, $(n + 1)\Delta t$, using the stencil containing our four nearest neighbors. A traditional CPU algorithm that evolves the solution one time step can often be similar to the following:

```
for (int j=1; j<ny-1; ++j) {
  for (int i=1; i<nx-1; ++i) {
    int n=(j+1)*nx+i, s=(j-1)*nx+i, e=j*nx+i+1, w=j*nx+i-1;
    h_new[j*nx+i] = 0.25*(h[n]+h[s]+h[e]+h[w])
        - 0.5*dt/dx*(hu[e]-hu[w]) - 0.5*dt/dy*(hv[n]-hv[s]);
  }
}
```

Here we have shown the code for computing h^{n+1} for each internal volume in the discretization, and hu^{n+1} and hv^{n+1} would be computed similarly. Because each volume (i, j) can be computed independently, we may solve for all volumes in parallel, and this is what we exploit when mapping the computations to the GPU. The first thing we do is to identify the independent parallel section of our code, and write it as a GPU *kernel*, shown in the following example:

```
__global__ void LaxFriedrichs(float* h_new,
        float* h, float* hu, float* hv, int nx, int ny) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
```

```
    int j = blockIdx.y*blockDim.y + threadIdx.y;
    if (i > 0 && i < nx-1 && j > 0 && j < ny-1) {
      int n=(j+1)*nx+i, s=(j-1)*nx+i, e=j*nx+i+1, w=j*nx+i-1;
      h_new[j*nx+i] = 0.25*(h[n]+h[s]+h[e]+h[w])
          - 0.5*dt/dx*(hu[e]-hu[w]) - 0.5*dt/dy*(hv[n]-hv[s]);
    }
  }
```

In this kernel, the variables h_new, h etc. are physically located in GPU memory. We use the variables i and j to index this memory. We *launch* this kernel for each finite volume on the GPU using a *grid* of *blocks* to execute it on the GPU (see Figure 1):

```
dim3 block(16, 12);
dim3 grid(ceil(nx/16.0), ceil(ny/12.0));
LaxFriedrichs<<<grid, block>>>(h_new, h, hu, hv, nx, ny);
```

This creates one thread per volume which the GPU executes in parallel. In addition to the code shown above, we also need to allocate memory on the GPU, copy initial conditions from the CPU to the GPU, and copy results back after the desired simulation time has been reached. In both the CPU and the GPU code, we also need to implement boundary conditions, for example using global ghost cells that in general must be updated before every simulation step.

The presented GPU code launches nx $\times$ ny threads organized into blocks of 16×12. On the GPU, each block is assigned to one of many processors, and one processor can hold multiple blocks. This is used to hide memory latencies by rapidly switching between the active blocks. We typically achieve best performance with a large number of blocks, but determining the optimal block size is often a very difficult task. One important parameter here is the 32-way SIMD nature of GPUs, that is, 32 consecutive threads must execute the same instruction on different data for full performance. Another key optimization parameter is *shared memory*. In the code above, each output element is computed from four input elements which have to be read from *global* GPU memory. On the CPU, the input variables would automatically be *cached* for performance. On the GPU, however, we must manually place data in shared memory, a type of programmable cache available to all threads within the same block. We can do this by loading one data element per thread, in addition to the *apron*, into shared memory (see Figure 1). Such a strategy gives us the classical block domain decomposition, in which each CUDA block has an input domain which overlaps with its neighboring blocks, allowing it to execute independently. This means that for a large block size we on average read just over *one* element per thread, compared to eight without the use of shared memory. However, the shared memory is limited in size, thereby limiting our block size. There are several other important optimization parameters, and we refer the reader to [3] for a more thorough discussion of these.

Multi-GPU: The technique presented above for executing CUDA blocks in parallel through the use of domain decomposition can also be used to enable parallel simulations on multiple GPUs. Multi-GPU simulations are highly attractive for simulating very large domains or when performance requirements are very high. Modern computers can be equipped with up-to four dual-GPUs on a single chassis, effectively creating a desktop supercomputer. However, whilst the block decomposition is highly suitable within one GPU because of the limited shared memory size, it can often be better to use a row decomposition for multi-GPU simulations within one

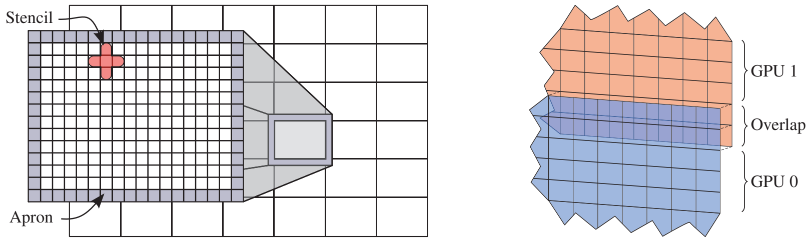


Figure 1: (Left) Illustration of the block and grid concepts with one block hilighted. The *apron* (also called local ghost cells) is used to fulfill the requirements of the stencil, so that all blocks may be executed independently and in parallel by the GPU. (Right) Multi-GPU domain decomposition using overlapping ghost cell regions. The two GPUs exchange the overlapping region after every second time step.

node (see Figure 1). The row decomposition minimizes the overlapping areas of the domain, and also communication costs, as each node has at most two neighbors (as opposed to four for the block decomposition). Before each simulation step, one simply exchanges the overlapping regions between the two GPUs, thereby coupling the two otherwise independent simulations.

The GPU is located on the PCI-express bus, and all communication between different GPUs is therefore slow, especially in terms of latency. Simulations on multiple GPUs may therefore suffer from this slow communication between the GPUs. One way of alleviating this is to use *ghost cell expansion*, in which the size of the overlap between two GPUs is increased. By doing this, we can simulate more than one time step before exchanging information between GPUs, as disturbances will at most travel one cell per time step. Thus, for an overlap of two cells, we would be able to run two time steps before having to exchange the overlapping region. However, the cost is that we must now calculate the flow in the overlapping cells on both GPUs, meaning there is a trade-off. Our experience shows that an overlap of on the order of tens of cells yields highest performance, allowing near-perfect weak *and* strong scaling for representative domain sizes [12]. An extension to this technique is to overlap data transfers with computation, which can be done by solving for the internal cells of the domain simultaneously as the overlap region is being exchanged.

Sparse Simulations: Many real-world scenarios will often have large areas without water, such as in simulation of dam breaks and inundations near riverbanks and coastal regions. These dry areas do not require any computation, as the compact stencil ensures that water will travel at most one cell per time step. Traditional techniques, however, often perform some calculations on these cells before discarding the results if the cell is dry, effectively wasting both memory bandwidth and floating point operations. One particularly effective approach to address this shortcoming on the GPU is to use sparse simulations (see Figure 2) [11]. Sparse simulations are based around the grid concept of CUDA, whereby only blocks requiring computation are launched. At the end of each time step, each block stores “-1” in a buffer if it is guaranteed to be dry the next time step, or its grid position otherwise. At the next time step, this buffer is sorted so that the non-negative indices come first, and a grid with one block per non-negative index is launched. The extra sorting of the buffer incurs a small performance penalty, but being able to skip dry regions altogether yields a great performance increase on GPUs (see Figure 3).

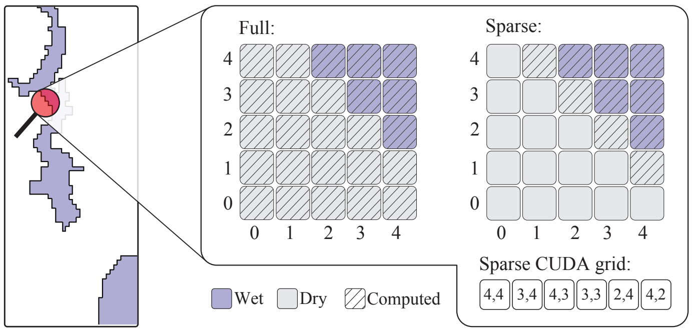


Figure 2: Map of dry blocks during a real-world dam break simulation. By storing whether or not a block contains water, we can create a list of blocks that require computation before each time step, and launch the *kernel* only on these blocks. Since water from a wet block may flow to a neighboring dry block in the next time step, we must also compute these dry blocks. This approach saves both computation and bandwidth, yielding a significant performance improvement on typical domains (see also Figure 3).

The technique of skipping dry parts of the domain can also be extended to the data representation, whereby dry parts of the domain are not represented on the GPU at all. This essentially follows the same procedure, but the data layout is changed so that wet blocks are stored after one another in GPU global memory. Such a sparse memory layout is especially attractive for extremely large domains with little water, and even problems where the full domain would otherwise not fit in GPU global memory. However, the altered data layout makes the process of reading the apron slightly more complicated, as neighboring blocks are no longer neighbors in the physical memory layout.

Accuracy and Performance: A requirement for developing numerical codes on the GPU that accurately captures the physical reality is to choose good verification and validation cases first, and then to optimize the code only after it gives the correct results. A classical problem one soon encounters in this process is that floating point arithmetic is not commutative due to round off errors, that is $a_f + b_f \neq b_f + a_f$. This is important to note when executing parallel code, as the sequence of floating point operations between different execution units often will be non-deterministic.

A further difficulty with numerical codes is that the inevitable floating point errors can blow up, and one often tends to use double precision calculations instead of single precision to counter this. However, single precision may in many cases still be the best choice. First of all, using single precision gives you roughly double the performance, because the size of your data is halved and single precision operations are twice as fast. Furthermore, it is often the case that modeling errors, measurement errors, and other factors shadow the errors imposed by using single precision. For example in [2], the handling of dry states completely masks the errors introduced by using single precision arithmetic for the target scenarios, meaning single precision is sufficiently accurate. However, it is still important to keep floating point errors in mind when implementing all numerical codes. As another example, let us consider a numerical scheme based on the water elevation instead of the water depth (e.g., the Kurganov-Petrova scheme [6]). In such a scheme, it is often tempting to store the water elevation in memory. This, however, will give rise to large relative errors when the water depth is small compared

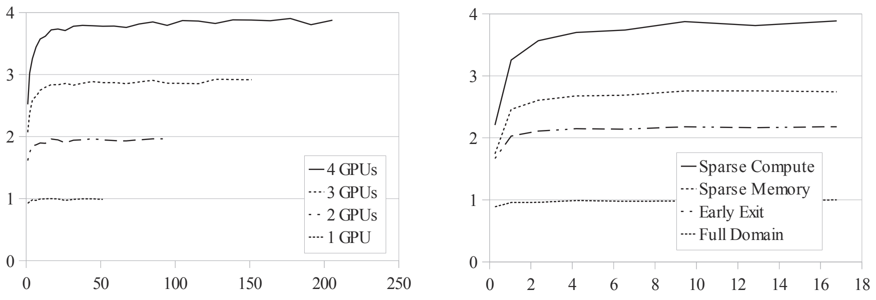


Figure 3: (Left) Performance as a function of domain size in millions of cells for 1 to 4 GPUs. The simulation is a wet bed circular dam break, and shows near-perfect weak and strong scaling for sufficiently large domains. The performance is normalized with respect to the fastest 1 GPU run. (Right) Performance as a function of domain size in millions of cells for different algorithms used to increase the computational throughput for a dry bed circular dam break. The average number of wet cells is approximately 26%. Full domain computes all cells, early exit launches and then exits exits blocks without water, sparse memory stores and computes only on wet blocks, and sparse compute computes only on wet blocks. All graphs are normalized with the fastest full domain run as the reference.

to the elevation. The reason is that floating point numbers are most accurately represented when close to zero. Thus, for a small water depth at a large elevation, the round off errors will often lead to large floating point errors in the results. Therefore, it may be important to store the quantity of interest as a number close to zero in memory, and then reconstruct derived quantities on demand.

Thorough performance assessment for GPUs is often difficult and somewhat neglected. Many papers that are published unfortunately relay overly optimistic speedup figures over CPU codes. This is problematic because when one examines the theoretical performance gap between the architectures, which currently lies at around seven times [1], it is clear that speedup claims of hundreds or more require a thorough explanation. Our view is that a good performance assessment focuses on identifying bottlenecks of the algorithm and reporting the attained percentage of peak performance through careful profiling. This will give the viewer a much more balanced view of the algorithm, and more importantly, clearly identify directions for further research. Profiling can be performed using for example Parallel NSight, which is a superb tool for profiling of GPU codes directly in Visual Studio, and similar tools such as the CUDA Profiler also exist for Linux and OS X.

3 Summary

We have presented general guidelines and best practices for mapping explicit shallow water schemes with compact stencils to graphics processing units. These schemes are naturally suited for the execution model of modern GPUs, and can give unprecedented simulation speeds. We have furthermore discussed strategies for expanding to multi-GPU simulations and strategies for avoiding computing dry areas of the simulation domain. Finally, we have presented floating point considerations with respect to accuracy versus performance, and best practices for performance assessment.

Acknowledgement: Part of this work is supported by the Research Council of Norway's project number 180023 (Parallel3D) and the Norwegian Meteorological Institute. The authors acknowledge the continued support from NVIDIA.

Bibliography

[1] A. R. Brodtkorb. *Scientific Computing on Heterogeneous Architectures*. PhD thesis, University of Oslo, 2010.

[2] A. R. Brodtkorb, T. R. Hagen, K.-A. Lie, and J. R. Natvig. Simulation and visualization of the saint-venant system using GPUs. *Computing and Visualization in Science*, 13:341–353, 2011.

[3] A. R. Brodtkorb, M. L. Sætra, and M. Altinakar. Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. *Computers & Fluids*, 55:1–12, 2012.

[4] R. Farber. *CUDA Application Design and Development*. Morgan Kaufmann. Elsevier Science, 2011.

[5] T. Hagen, J. Hjelmervik, K.-A. Lie, J. Natvig, and M. Henriksen. Visual simulation of shallow-water waves. *Simulation Modelling Practice and Theory*, 13(8):716–726, 2005.

[6] A. Kurganov and G. Petrova. A second-order well-balanced positivity preserving central-upwind scheme for the Saint-Venant system. *Communications in Mathematical Sciences*, 5:133–160, 2007.

[7] R. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2002.

[8] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top 500 supercomputer sites. `http://www.top500.org/`, December 2011.

[9] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[10] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005*, pages 21–51, 2005.

[11] M. L. Sætra. Shallow water simulation on GPUs for sparse domains. In *Proceedings of ENUMATH 2011*, Leicester, UK, 2012.

[12] M. L. Sætra and A. R. Brodtkorb. Shallow water simulations on multiple GPUs. In *Applied Parallel and Scientific Computing*, volume 7134 of *Lecture Notes in Computer Science*, pages 56–66. Springer Berlin / Heidelberg, 2012.

[13] E. Toro. *Shock-Capturing Methods for Free-Surface Shallow Flows*. John Wiley & Sons, Ltd., 2001.

Paper III

Efficient Shallow Water Simulations on GPUs: Implementation, Visualization, Verification, and Validation

A. R. Brodtkorb, M. L. Sætra, and M. Altinakar

Computers & Fluids, **55** (2012), pp. 1–12

Abstract: In this paper, we present an efficient implementation of a state-of-the-art high-resolution explicit scheme for the shallow water equations on graphics processing units. The selected scheme is well balanced, supports dry states, and suits the execution model of graphics processing units well. We verify and validate our implementation and show that use of efficient single precision hardware is sufficiently accurate for real-world simulations. Our framework further supports real-time visualization with both photo-realistic and non-photo-realistic display of the physical quantities. We present performance results showing that we can accurately simulate the first 4000 seconds of the Malpasset dam break case 27 seconds, in which our simulator runs at up-to 700 megacells per second.

1 Introduction

We present a state-of-the-art implementation of a second-order explicit finite-volume scheme for the shallow water equations with bed slope and bed shear stress friction terms. Our implementation is verified against analytical data, validated against experimental data, and we show extensive performance benchmarks. We start by giving an introduction to graphics processing units (GPUs), the shallow water equations, and the use of GPUs for simulation of conservation laws in this section. We continue in Section 2 by presenting the mathematical model and discretization of the selected scheme. In Section 3 we present our implementation, followed by numerical and performance experiments in Section 4, and we give our concluding remarks in Section 5.

Graphics Processing Units

Research on using GPUs for scientific computing started over a decade ago with simple academic tests that demonstrated the use of GPUs for non-graphics applications. Since then, the hardware has evolved from simply accelerating a set of predefined graphics functions for games to being used in the worlds fastest supercomputers [35]. The Chinese Nebulae supercomputer, for example, uses Tesla GPUs from NVIDIA. These cards offer over 500 gigaflops in double precision performance, twice that in single precision, and up-to 6 GB RAM accessible at 148 GB/s. Compared to the state-of-the-art six core Intel Core i7-980X, this is roughly six times the performance and bandwidth. In addition to having a higher peak performance, these GPUs also offer fast, albeit less accurate, versions of trigonometric and other functions, that can be exploited for even higher performance. Researchers have efficiently exploited these GPU features to accelerate a wide range of algorithms, and speedups of 5-50 times over equivalent CPU implementations have been reported (see e.g., [6, 30]).

Today, most scientific articles utilizing GPUs use cards from NVIDIA and program them using CUDA, a C-based programming language for parallel execution on GPUs. CUDA has been widely used by both industrial and academic groups, and over 1100 applications and papers in a wide range of fields have been added to the NVIDIA CUDA Showcase [27] since 2007. Browsing through the CUDA Showcase, it quickly becomes apparent that there is a race to report the largest attained speedup. At the time of writing of the present article, there were 465 papers reporting speedups. Out of these 115 claimed a speedup of 100 or more, 18 claimed a speedup of 500 or more, and one reported a stunning speedup of 2600. With a theoretical performance gap on the order of six times for the fastest available hardware today, we believe many of these claims to be misleading, at best. In fact, Lee et al. [22] support this view in a recent comparison of 14 algorithmic kernels, and report an average speedup of only 2.5 times. Whilst one can argue that a paper featuring Intel-engineers only might favor Intel CPUs, and comparison of a mid 2008 GPU with a late 2009 CPU can be rightfully criticized, their benchmarks clearly illustrate that a 100 times speedup is not automatically achieved by simply moving an algorithm to the GPU. We believe that fair comparisons between same-generation hardware of the same performance class, with reporting of the percent of peak performance, would be considerably more meaningful and useful than escalating the speedup race. Such comparisons make it easier to compare efficiency across algorithms and for different hardware.

Both the strengths and the weaknesses of GPUs come from their architectural differences from CPUs. While CPUs are traditionally optimized for single-thread performance using complex logic for instruction level parallelism and high clock frequencies, GPUs are optimized for net throughput. Today's GPUs from NVIDIA, exemplified by the GeForce GTX 480, consist of up-to 15 SIMD cores called streaming multiprocessors (SM). Each SM holds 2×16 arithmetic-logic units that execute 2×32 threads (two *warps*) in SIMD-fashion over two clock cycles. The SMs can keep multiple warps active simultaneously, and instantly switch between these to hide memory and other latencies. One SM can hold a total of 48 active warps, meaning we can have over 23 thousand hardware threads in flight at the same time [28]. This highly contrasts with the relatively low value of 48 operations for current CPUs (6 cores $\times$ 4-way SIMD $\times$ 2 hardware threads). Thus, the programming model of GPUs is very different from that of CPUs, and to quote Bo Kågström, we especially need to take the algorithm and architecture interaction into account for efficient hardware utilization. For a detailed overview of GPU hardware and software, we refer the reader to [6, 30].

The Shallow Water Equations

The shallow water equations describe gravity-induced motion of a fluid with free surface, and can model physical phenomena such as tidal waves, tsunamis, river flows, dam breaks, and inundation. The system is a set of hyperbolic partial differential equations, derived from the depth-averaged Navier-Stokes equations. As such, solutions of the shallow water equations are only valid for problems where the vertical velocity of the fluid is negligible compared to the horizontal velocities. Luckily, this criteria applies to many situations in hydrology and fluid dynamics, where the shallow water equations are widely used. For hyperbolic equations in general, the domain of dependence is always a bounded set. For the shallow water equations, this means we can solve the system using an explicit scheme, since the waves travel at a finite speed. Furthermore, the intrinsic parallelism of explicit schemes makes them particularly well suited for implementation on modern GPUs.

Conservation Laws on Graphics Processing Units

Using GPUs for the numerical simulation of conservation laws is not a new idea. In fact, the use of GPUs for such simulations was introduced at least as early as 2005, when one had to map the computations to operations on graphical primitives [14]. Since then, there have been multiple publications regarding conservation and balance laws on GPUs [13, 15, 4, 5, 18, 38, 3, 1].

Several authors have published implementations of explicit schemes for the shallow water equations. Hagen et al. [14] implemented multiple explicit schemes on the GPU using OpenGL, including a high-resolution second-order finite volume scheme very similar to the one we examine here, and demonstrated a 15-30 times speedup over an equivalently tuned CPU implementation on same-generation hardware. Liang et al. [25] present a second-order MacCormack scheme including friction described by the Manning equation implemented using OpenGL. They report a speedup of two for their whole algorithm on a laptop, and up-to 37000 for one of their kernels compared to the CPU. They also visualize their results, by first copying the data to the CPU, and then transferring it back to the GPU again. Lastra et al. [21] implemented a first-order, finite-volume scheme using the graphics languages OpenGL and Cg, presenting over 200 times speedup for a 2008 CPU versus a 2004 GPU. de la Asunción et al. [9] explore the same scheme using CUDA, and show a speedup of 5.7 in double precision on same-generation hardware. They report a 43% utilization of peak bandwidth and 13% of peak compute throughput for the whole algorithm. They further compared single versus double precision, reporting numbers indicating that single precision calculations are not sufficiently accurate. The same authors have reported similar findings [8] after extending their implementation to support also two-layer shallow water flows. Brodtkorb et al. [7] implemented three second-order accurate schemes on the GPU, comparing single versus double precision, where single precision was found sufficiently accurate for the implemented schemes. They further report an 80% instruction throughput for computation of the numerical fluxes, which is the most time consuming part of the implementation.

Paper Contribution

We build on the experiences of Brodtkorb et al. [7], and present a completely new state-of-the-art implementation of the Kurganov-Petrova scheme [20]. We have chosen this scheme for several reasons: it is well-balanced, conservative, second order accurate in space, supports dry zones, and is particularly well suited for implementation on GPUs. It has furthermore been shown that single precision arithmetic is sufficiently accurate for this scheme, enabling the use of efficient single precision hardware. Novelties in this paper include: per-block early exit optimization; verification and validation; a 31% decrease in memory footprint; semi-implicit physical friction terms; first order temporal time integration; multiple boundary conditions; simultaneous visualization with multiple visualization techniques; and extensive performance benchmarks.

2 Mathematical Model

The shallow water equations in two dimensions with bed slope and bed shear stress friction terms can be written

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix} + \begin{bmatrix} 0 \\ -gu\sqrt{u^2+v^2}/C_z^2 \\ -gv\sqrt{u^2+v^2}/C_z^2 \end{bmatrix}. \tag{1}$$

Here h is the water depth, and hu and hv are the discharges along the abscissa and ordinate, respectively. Furthermore, g is the gravitational constant, B is the bottom topography measured from a given datum, and C_z is the Chézy friction coefficient (see also Figure 1a). We employ Manning's roughness coefficient, n, in our scheme using the relation $C_z = h^{1/6}/n$. In vector form, we write the system of equations as

$$Q_t + F(Q) + G(Q) = H_B(Q, \nabla B) + H_f(Q), \tag{2}$$

where Q is the vector of conserved variables, F and G represent fluxes along the abscissa and ordinate, respectively, and H_B and H_f are the bed slope and bed shear stress source terms, respectively.

The scheme we consider here is well-balanced, which means that for $u = v = 0$ the free-surface elevation, w, measured with respect to the same datum as B, remains constant. This requires that the numerical fluxes $F+G$ perfectly balance the bed slope source term H_B. Developing a well-balanced scheme is typically difficult when using the physical variables $[h, hu, hv]^T$, but achievable by switching to the set of derived variables $[w, hu, hv]^T$ (see Figure 1a). Thus for

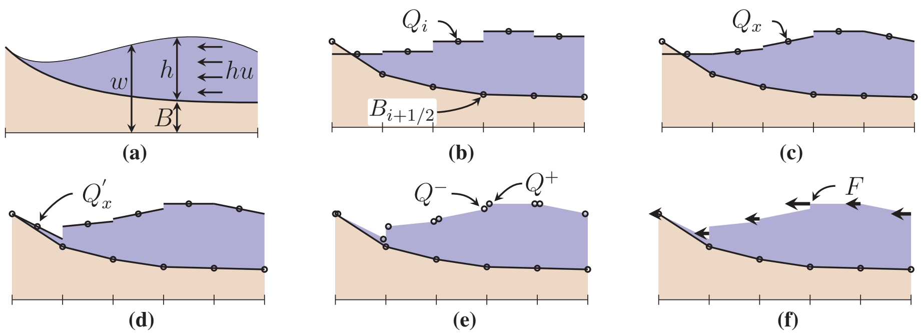


Figure 1: (a) Shallow water flow over a complex bottom topography and definition of variables; (b) conserved variables Q are discretized as cell averages and the bottom topography, B, is represented as a piecewise bilinear function in each cell based on its values at the grid cell intersections; (c) reconstruction of free-surface slopes of each cell using generalized minmod flux limiter; (d) modification of free-surface slopes with wet-dry contact to avoid negative water depth; (e) reconstructed values of the conserved variables on each side of the cell interfaces; and (f) fluxes computed at each cell interface using the central-upwind flux function [19].

the remainder of this article, we use $Q = [w, hu, hv]^T$, in which (1) is rewritten using $h = w - B$ (see [20] for a detailed derivation).

Spatial Discretization

The spatial discretization of the Kurganov-Petrova scheme is based on a staggered grid, where Q is given as cell averages, B is given as a piecewise bilinear surface defined by the values at the four cell corners, and fluxes are calculated at integration points at the midpoint of each grid cell interface (see also Figure 4). The spatial discretization can then be written

$$\begin{aligned}\frac{dQ_{ij}}{dt} &= H_f(Q_{ij}) + H_B(Q_{ij}, \nabla B) - \big[F(Q_{i+1/2,j}) - F(Q_{i-1/2,j})\big] - \big[G(Q_{i,j+1/2}) - G(Q_{i,j-1/2})\big] \\ &= H_f(Q_{ij}) + R(Q)_{ij}.\end{aligned} \tag{3}$$

To compute the fluxes F and G across the cell interfaces, we perform the computations outlined in Figure 1. From the cell averages in Figure 1b we reconstruct a piecewise planar surface for Q in each cell (Figure 1c). A problem with this reconstruction is that we will typically end up with negative values for h at the integration points near dry zones. Without addressing these negative values, our scheme will not handle dry zones, as the eigenvalues of the system are $u \pm \sqrt{gh}$. In the scheme, the slope reconstruction is initially performed using the generalized minmod flux limiter,

$$Q_x = \text{MM}(\theta f, c, \theta b), \qquad \text{MM}(a,b,c) = \begin{cases} \min(a,b,c), & \{a,b,c\} > 0 \\ \max(a,b,c), & \{a,b,c\} < 0 \\ 0, \end{cases} \tag{4}$$

where $\theta = 1.3$ and f, c, and b are the forward, central, and backward difference approximations to the derivative, respectively. This reconstruction, however, does not guarantee non-negativeness of h. To handle dry zones, Kurganov and Petrova propose to simply alter the slope of w so that the value of h at the integration points becomes non-negative (compare Figure 1c with Figure 1d). Because we have a bilinear bottom topography and planar water elevation, we can guarantee non-negativeness for four integration points per cell when the cell average is non-negative. This, however, limits the spatial reconstruction to second-order accuracy.

After having altered the slopes, we reconstruct point values for each cell intersection from the two adjacent cells (Q^+ and Q^- in Figure 1e), and compute the flux shown in Figure 1f using the central-upwind flux function [19]. A difficulty with these computations related to dry zones is that we need to calculate $u = hu/h$. This calculation leads to large round-off errors as h approaches zero, which in turn can lead to very large velocities and even instabilities. Furthermore, as the timestep is directly proportional to the maximal velocity in the domain, this severely affects the propagation of the solution. To avoid these large velocities, Kurganov and Petrova *desingularize* the calculation of u for *shoal* zones ($h < \kappa$) using

$$u = \frac{\sqrt{2}h(hu)}{\sqrt{h^4 + max(h^4, \kappa)}}. \tag{5}$$

This has the effect of dampening the velocities as the water depth approaches zero, making the scheme well-behaved even for shoal zones. Determining the value of κ, however, is difficult. Using too large a value yields large errors in the results, and setting it too low gives very small timesteps, Kurganov and Petrova used $\kappa = \max\{\Delta x^4, \Delta y^4\}$ in their experiments. This approach is insufficient for many real-world applications, such as the Malpasset dam break case (used in Section 4) where $\Delta x = \Delta y = 15\ m$. Thus, we suggest using

$$\kappa = K_0 \max\{1, \min\{\Delta x, \Delta y\}\} \tag{6}$$

as an initial guess, with $K_0 = 10^{-2}$ for single precision calculations, and a smaller constant for double precision. This gives a linear proportionality to the grid resolution, which we find more suitable: for example, for a 15 meter grid cell size our approach desingularizes the fluxes for water depths less than $15\ cm$. Numerical round off errors may also make very small water depths negative, independent of the value of κ. We handle this by setting water depths less than ϵ_m to zero, where ϵ_m is close to machine precision.

To be well-balanced and capture the lake at rest case, the bed slope source term needs to be discretized carefully to match the flux discretization. However, due to the alteration of the slopes of w to guarantee non-negativeness at the integration points, we get nonphysical fluxes emerging from the shores (see Figure 1c and 1d). These fluxes, however, are small, and have minimal effect on the global solution for typical problem setups.

Temporal Discretization

In (3) we have an ordinary differential equation for each cell in the domain. Disregarding the bed shear stress for the time being, we discretize this equation using a standard second-order

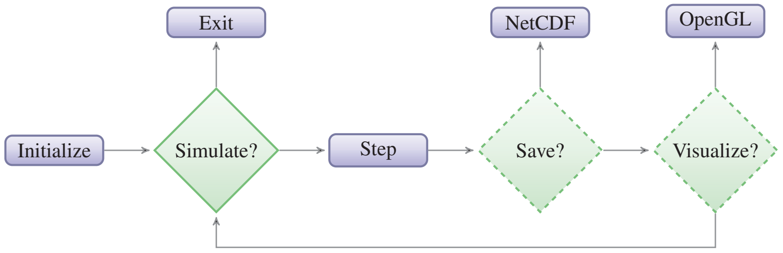


Figure 2: The program flow of our applications. Initialize sets up the simulator class with initial conditions, and step runs one timestep on the GPU. We can also save results in netCDF files and visualize them directly using OpenGL.

total variation diminishing Runge-Kutta scheme [33],

$$\begin{aligned} Q^*_{ij} &= Q^n_{ij} + \Delta t R(Q^n)_{ij} \\ Q^{n+1}_{ij} &= \tfrac{1}{2}Q^n_{ij} + \tfrac{1}{2}\left[Q^*_{ij} + \Delta t R(Q^*)_{ij}\right], \end{aligned} \tag{7}$$

where the timestep, Δt, is restricted by a CFL condition that ensures that disturbances travel at most one quarter grid cell per time step,

$$\Delta t \leq \tfrac{1}{4}\text{min}\Big\{\Delta x/\text{max}_\Omega\big|u \pm \sqrt{gh}\big|, \Delta y/\text{max}_\Omega\big|v \pm \sqrt{gh}\big|\Big\} = \frac{r}{4}. \tag{8}$$

To include the bed shear stress in (7) we use a semi-implicit discretization,

$$\begin{aligned} H_f(Q^*_{ij}) &\approx Q^*_{ij}\tilde{H}_f(Q^n_{ij}), \\ H_f(Q^{n+1}_{ij}) &\approx Q^{n+1}_{ij}\tilde{H}_f(Q^*_{ij}), \end{aligned} \qquad \Bigg| \qquad \tilde{H}_f(Q^k_{ij}) = \begin{bmatrix} 0 \\ -g\sqrt{{u^k_{ij}}^2 + {v^k_{ij}}^2}/h^k_{ij}C^{\,2}_{zij} \\ -g\sqrt{{u^k_{ij}}^2 + {v^k_{ij}}^2}/h^k_{ij}C^{\,2}_{zij} \end{bmatrix} \tag{9}$$

where $\tilde{H}_f(Q^k_{ij})$ is computed explicitly from Q at timestep k. Adding (9) to (7) and reordering, we get

$$\begin{aligned} Q^*_{ij} &= \big[Q^n_{ij} + \Delta t R(Q^n)_{ij}\big]/\big[1 + \Delta t\tilde{H}_f(Q^n_{ij})\big] \\ Q^{n+1}_{ij} &= \Big[\tfrac{1}{2}Q^n_{ij} + \tfrac{1}{2}\big[Q^*_{ij} + \Delta t R(Q^*)_{ij}\big]\Big]/\big[1 + \tfrac{1}{2}\Delta t\tilde{H}_f(Q^*_{ij})\big], \end{aligned} \tag{10}$$

where all terms on the right hand side are explicitly calculated. We can also use a first order accurate Euler scheme, which simply amounts to setting $Q^{n+1}_{ij} = Q^*_{ij}$ in (10).

3 Implementation

We have implemented a cross-platform compatible simulator with visualization using C++, OpenGL [32], and NVIDIA CUDA [28]. The implementation consists of three parts: a C++ interface and CPU code, a set of CUDA kernels that solve the numerical scheme on the GPU, and a visualizer that uses OpenGL to interactively show results from the simulation. The simulation is run solely on the GPU, and data is only transferred to the CPU for file output.

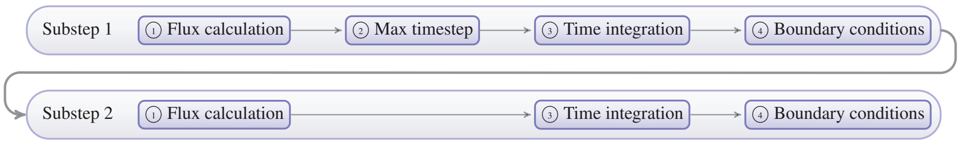


Figure 3: Zoom of the step function in Figure 2 that shows our CUDA kernels. In the first substep, ① calculates R, ② finds the maximum Δt, ③ calculates $\tilde{H}_f$ and Q^*, and ④ enforces boundary conditions on Q^*.

C++ interface and CPU code

Our C++ interface consists of a relatively simple class that handles data allocation, initialization and deallocation; movement of data between the CPU and the GPU; and invoking CUDA kernels on the GPU. The API for the simulator is easy and clean, and it is possible to set up and run a simulation in about 5–10 lines of code without knowledge of implementation details. We have implemented several applications that use this interface, including one that writes results to netCDF [26] files, an open and standardized file format commonly used to store geophysical data, and one that runs real-time visualization of the simulation using OpenGL. Our applications further supports continuing simulations stored in netCDF files. This is done by initializing the simulator with the bottom topography and the physical variables for the last timestep in the file.

Figure 2 shows the program flow for our applications. The applications start by initializing the simulator class, and continues by entering a loop where we perform timesteps. Because each timestep is variable, we do not know, a priori, how many timesteps we need to perform per unit simulation time. Thus, to save or visualize the results at regular simulation time intervals, we check the current simulation time after each timestep and only exit the simulation loop when the desired simulation time has been reached.

Our C++ class allocates the GPU data required to pass information between kernels. Brodtkorb et al. [7] stored approximately 16 values per grid cell: two for B, three for Q, three for Q^*, three for F, three for G, and two for the non-zero source terms. B was represented at the grid intersections and also at the center of cells as a performance optimization. In our approach, however, we have reduced this to only 11 values per grid cell by combining F, G, and the two non-zero source terms into three values for R. This reduces the number of values we have to transfer to and from global memory dramatically, and reduces the memory footprint by approximately 31%. However, this complicates the implementation of the flux calculation, as will be detailed later. We have used single precision in our kernels, as Brodtkorb et al. [7] have previously shown that this is sufficiently accurate. Using single precision over double has several benefits: data transfers and arithmetic operations can execute at least twice as fast, and all data storage in registers, shared and global memory, takes half the space.

CUDA Kernels

Figure 3 shows how our numerical scheme is computed by executing seven CUDA kernels in order. One full time-step with the second-order accurate Runge-Kutta scheme runs through both the first and second substep, whilst running only the first substep reduces to the first order accurate Euler scheme. Thus, for second-order accuracy, we perform around twice the number of calculations. Within each substep we ① calculate the fluxes and bed slope source terms, ② find the maximum timestep, ③ compute bed shear stress source terms and evolve the solution in time, and ④ apply boundary conditions. Inbetween these kernels, we store results in global memory and implicitly perform global synchronization. It would be more efficient if we could perform the full scheme using only a single kernel, however, this is not possible: applying boundary conditions requires the evolved timestep; evolving the solution in time requires the fluxes and maximum timestep; and computing the maximum timestep requires the eigenvalues for all cells that are computed by the flux kernel. One could combine the boundary conditions kernel with either the time integration or the flux kernel, yet this would involve additional branching that quickly outweigh the performance gain from launching one less kernel.

Modern GPUs from NVIDIA consist of a set of SIMD processors that execute *blocks* in parallel. Each block consists of a predefined number of threads, logically organized in a three-dimensional grid. Threads in the same block can cooperate and share data using on-chip *shared memory*. For high performance we need to have a good block partitioning, and choosing a "wrong" block configuration will severely affect kernel performance. However, there are several conflicting criteria for choosing the optimal block size. The optimal block size also varies from one GPU to another, especially between major hardware generations. In the case of the GPU used for the present work, i.e., the NVIDIA GeForce GTX 480, the following factors were observed to affect the performance:

Warp size The hardware schedules the same instruction to 32 threads, referred to as a *warp*, in SIMD fashion. Thus, for optimal performance, we want the number of threads in our block to be a multiple of 32.

Shared memory access Shared memory is organized into 32 *banks*, which collectively service one warp every other clock cycle. If two threads access the same bank simultaneously, however, the transaction takes twice as long. Thus, the width of our shared memory should be a multiple of 33 to avoid bank conflicts both horizontally and vertically.

Shared memory size Keeping frequently used data in shared memory will typically yield performance gains. Thus, we want to maximize the domain kept in shared memory. We also want to keep it as square as possible to maximize the ratio of internal cells to local ghost cells required by the computation (see Figure 4).

Number of warps per streaming multiprocessor Each multiprocessor can keep up-to 48 active warps simultaneously, as long as there are available hardware resources (such as shared memory and registers). Increasing the number of active warps increases the occupancy, which is a measure of how well the streaming multiprocessor can hide memory latencies. The processor uses this occupancy by instantaneously switching to other warps when the current warp stalls, e.g., due to a data dependency. Thus, we want as many warps per multiprocessor as possible.

Global memory access Global memory is moved into the processor in bulks, reminiscent of the way CPUs transfer full cache lines. For maximum performance, the reads from global memory should be 128 byte sequential, and the start address should be aligned on a 128 byte border (called coalesced accesses). This means that our block width should

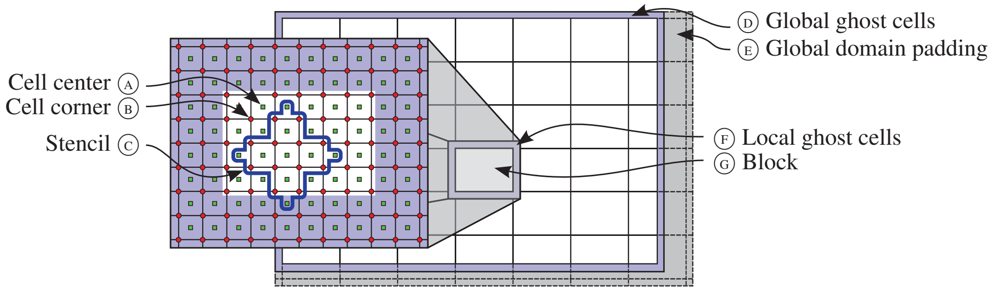


Figure 4: Domain decomposition and variable locations. The global domain is padded Ⓔ to fit an integer number of blocks. Each block Ⓖ has local ghost cells Ⓕ that overlap with other blocks to satisfy the data dependencies dictated by the stencil Ⓒ. Our data variables Q, R, H_B, and H_f are given at grid cell centers Ⓐ, and B is given at grid cell corners Ⓑ.

make sure that global reads start at properly aligned addresses. If we violate these rules, however, we can still benefit from the caching on CUDA compute 2.0 capable cards from NVIDIA.

Flux Calculation The flux kernel is the computationally most expensive kernel in our implementation. Its main task is to compute R_{ij} from (3). This is done by computing the flux across all the cell interfaces, the bed slope source term for all cells, and summing to find the net contribution to each cell. This contrasts the approach of Brodtkorb et al. [7], where the calculation of the net contribution per cell was done in the time integration kernel. Comparing the two, our new approach stores fewer floating-point values per cell, enabling us to reduce the memory footprint on the GPU.

Before we launch our kernel, we start by performing domain decomposition, yielding high levels of parallelism suitable for the execution model of GPUs. Figure 4 illustrates how our global domain is partitioned into blocks that can be calculated independently by using local ghost cells. We use a block configuration of $16 \times 12 = 192$ threads where we have one thread per cell. This size is a compromise between the above mentioned optimization guidelines that has yielded the best performance. Our block size is a multiple of 32 which fits with the warp size. Our shared memory size uses almost all of the 16 KB available, and it is relatively square (20×16 including local ghost cells). It does not, however, ensure that we have no bank conflicts. Making the width 32, thus 28 threads wide block size, would solve this at the expense of other optimization parameters such as warp size. For our flux kernel, we also have the option of configuring the shared memory as 48 KB L1 cache and 16 KB shared memory, or 16 KB L1 cache and 48 KB shared memory. We have found that for our kernel, using 48 KB L1 shared memory yielded the best performance. However, we only use 16 KB per block, meaning we can fit several blocks per streaming multiprocessor. Through experimentation, we have found that using three blocks, corresponding to 18 warps, yielded the best performance. Using too many warps exhausts the register file, meaning that some registers are spilled to local memory. Using too few warps may cause the processor not to have enough warps to select from when some are stalled by latencies. Finally, due to the local ghost cells of our blocks, we are unable to fulfill coalescing rules. Fortunately, however, we do benefit from the 16 KB L1 cache, and 768 KB L2 cache. On older graphics cards, one could use the texture cache for better performance of uncoalesced reads. However, using the L1 and L2 cache on compute 2.0 capable cards is superior in performance to using the texture cache [29].

When launching our kernel, we start by reading from global memory into on-chip shared memory. In addition to the interior cells of our block, we need to use data from two neighbouring cells in each direction to fulfill the data dependencies of the stencil. After having read data into shared memory, we proceed by computing the one dimensional fluxes in the x and y directions, respectively. Using the steps illustrated in Figure 1, fluxes are computed by storing all values that are used by more than one thread in shared memory. We also perform calculations collectively within one block to avoid duplicate computations. However, because we compute the net contribution for each cell, we have to perform more reconstructions and flux calculations than the number of threads, complicating our kernel. This is solved in our code by designating a single *warp* that performs the additional computations; a strategy that yielded a better performance than dividing the additional computations between several warps. This comes at the expense of a more complex flux kernel, but it greatly simplifies, and increases the performance, of the time integration kernel.

The first of our calculations is to reconstruct the value of B at each interface midpoint, so that we have the value of B properly aligned with Q. This calculation is performed every timestep as a performance trade-off; in order to reduce memory and bandwidth usage. We continue by reconstructing the slopes of Q using the branchless generalized minmod slope limiter [13]. The slopes of w are then adjusted to guarantee non-negative values at the integration points, according to the scheme. With the slopes calculated we can reconstruct point values at the cell interfaces, Q^+ and Q^-, and from these values compute the fluxes. We also compute the bed slope source term, sum the contributions for each cell, and write the results to global memory.

The flux kernel is also responsible for computing r in (8) for each integration point, where the global minimum is used to calculate the maximum timestep. A problem with calculating r is that for zero water depths, we get division by zero. We solve this in our code by setting $r = \min\{\Delta x/\max(\epsilon_m, u \pm \sqrt{gh}), \Delta y/\max(\epsilon_m, v \pm \sqrt{gh})\}$, where ϵ_m is close to machine epsilon. Furthermore, instead of storing one value per integration point, we perform efficient shared memory reduction to find the minimum r in the whole block at very little extra cost. This reduces the number of values we need to store in global memory by a factor 192, yielding a two-fold benefit: we transfer far less data, and the maximum timestep kernel needs to consider only one value per block.

Maximum timestep The maximum timestep kernel is a simple reduction kernel that computes the maximum timestep based on the minimum r for each block. Finding the global minimum is done in a similar fashion to the reduction example supplied with the NVIDIA GPU Computing SDK. We use a single block where thread t_{no} *strides* through the dataset, considering elements $t_{\text{no}} + k \cdot n$, where n is the number of threads. This striding ensures fully coalesced memory reads for maximum bandwidth utilization. Once we have considered all values in global memory, we are left with n values that we reduce using shared memory reduction, and we compute the timestep as $\Delta t = 0.25r$. Also here, the block size has a large performance impact. Using fewer threads than the number of elements gives us a sub-optimal occupancy, and using too many threads launches warps where not a single thread is useful. Thus, to launch a suitable number of threads for varying domain sizes we use template arguments to create multiple realizations of the reduction kernel: we generate one kernel for $1, 2, 4, \ldots, 512$ threads, and select the most suitable realization at runtime.

Time integration The time integration kernel performs a domain partitioning similarly to the flux kernel, but we here use a block size of $32 \times 16 = 512$ threads as we are not limited by shared memory: our computations are embarrassingly parallel, and we do not require any local ghost cells. We can thus achieve full coalescing for maximum bandwidth utilization. This is also an effect of storing only the net contribution, R for each cell, as opposed to storing both the fluxes and non-zero source terms. The kernel starts by reading Q and R into per-thread registers, and then computes the bed shear stress source term, $\tilde{H}_f$. The timestep, Δt, is also read into registers, and we evolve the solution in time.

Boundary conditions We have selected to implement boundary conditions using global ghost cells. As our scheme is second-order accurate, we need two global ghost cells in each direction that are set for each substep by the boundary conditions kernel. This makes our flux kernel oblivious to boundary conditions, which means we do not have to handle boundaries differently from cells in the interior of our domain. We have implemented four types of boundary conditions: wall, fixed discharge (e.g., inlet discharge), fixed depth, and free outlet. To implement these conditions, we need to fix both the cell averages and the reconstructed point values, Q^+ and Q^-. To do this, we utilize an intrinsic property of the minmod reconstruction. Recall that this reconstruction uses the forward, backward, and central difference approximation to the derivative, and sets the slope to the least steep of the three, or zero if any of them have opposite signs. Thus, by ensuring that the least steep slope is zero, we can fix the reconstructed point values to any given value. Wall boundary conditions are implemented by mirroring the two cells nearest the boundary, and changing the sign of the normal discharge component. Fixed discharge is implemented similarly, but the normal discharge is set to a fixed value, and fixed depth boundaries set the depth instead of the discharge to the requested value. Finally, free outlet boundaries are implemented by copying the cell nearest the boundary to both ghost cells. It should be noted that our free outlet boundary conditions give rise to small reflections in the domain, as is typical for this kind of implementation (see e.g., [23]). We have further developed our fixed discharge and fixed depth boundary conditions to handle time-varying data. By supplying a hydrograph in the form of time-discharge or time-depth pairs, our simulator performs linear interpolation between points and sets the boundary condition accordingly. These types of boundary conditions can easily be used to perform, e.g., tidal wave, storm surge or tsunami experiments.

The boundary conditions are applied to the four global boundaries in a single kernel call in a very efficient fashion. At compile-time, we generate one kernel for each combination of boundary conditions, which we can automatically select at run-time. We have ten different realizations to optimize for different domain sizes (similarly to the maximum timestep kernel), and five for the different boundary conditions for each of the edges. This totals to $10 \times 5^4 = 6250$ optimized kernel realizations, which exceeds what the linker is capable of handling. To circumvent this compiler issue, we reduce the number of realizations by disregarding optimization for domain sizes where the width and height are both less than 64 cells, leading to only 2500 kernel realizations. Disregarding optimizations for the small domain sizes is not a big loss, since using

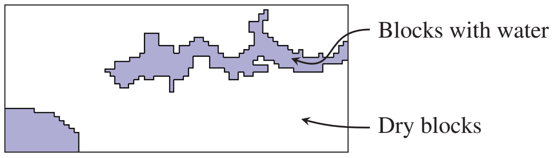


Figure 5: Auxiliary buffer used to store whether blocks contain water or not. The presence of one single wet cell in a block is sufficient to consider the whole block as wet.

the GPU is most efficient for large domain sizes. It is also possible to completely skip boundary conditions, to gain performance. This can safely be done when the domain has dry boundaries.

A weakness in our current implementation is the lack of support for mixed boundary conditions within each boundary. We can implement mixed boundaries using two auxiliary buffers for each edge. The first buffer describes the type of boundary condition, e.g., using an integer, and the second holds a value to be used by the boundary condition (e.g., water depth for fixed depth boundaries). However, such an implementation would be quite complex, require more data accesses, and impose additional branches. Thus, it might be a difficult task to implement a very efficient mixed boundary conditions kernel. On the other hand, the ability to run several kernels simultaneously on compute 2.0 capable cards might enable a different approach, where multiple kernels running simultaneously each handles a part of the boundary conditions.

Early Exit Optimization

The flux and time integration kernels both divide the computational domain into blocks that can be computed independently. However, if a block does not contain water at all, we can simply skip all computations for that block. Unfortunately, we do not a priori know whether or not a block contains water. In our approach, we use the same block size for the time integration kernel and the flux calculations kernel and implement early exit on a per block basis. In the time integration kernel, we let each block perform shared memory reduction to find out if this block contains any water at all, and write to an auxiliary buffer (see Figure 5). In the flux kernel we then read from this buffer, and if the block and its four closest neighbouring blocks are dry, we can skip flux calculations all together. We need to check the neighbouring blocks due to our local ghost cells that overlap with these blocks. This optimization has a dramatic effect on the calculation time for domains with a lot of dry cells, as is typical for flooding applications. We have also developed this technique to mark cells where the source terms perfectly balance the fluxes ($R = 0$) as “dry”. This extends the early exit strategy to also optimize for wet blocks where the steady state property holds. However, there is a small penalty to pay for using early exit, as reading and writing the extra data and performing the shared memory reductions takes a small amount of time to complete. To cope with this problem, we therefore perform runtime analysis of the kernel execution time. Once the kernel execution time for the early exit kernel exceeds the execution time for the regular kernel, we swap over to using the regular kernel. In addition, we also perform a probe every hundredth timestep in order to use the most efficient implementation at all times. This approach can also automatically be used where one has sources and sinks in the interior of a domain by implementing sources and sinks as changes to R, thus violating the lake-at-rest property.

OpenGL Visualizer

We have implemented a direct visualization using OpenGL, which was originally presented in [7]. A new feature in this implementation is the use of a more efficient data path between CUDA and OpenGL, which is enabled by the new graphics interoperability provided by CUDA 3.0. We have also implemented a non-photorealistic visualization that provides a different view of the simulation results. The terrain is rendered by draping it with a user-selectable image (such as a satellite or orthophoto image) and light-set using a static normal map. Several options are available for rendering the water surface. The first method is the photorealistic rendering, where the Fresnel equations are employed to calculate reflection and refraction of rays hitting the water surface as shown in Figure 6. As an alternative, we also provide the possibility of viewing the data using a color transfer function, also shown in Figure 6. We use interpolation between colors in the HSV color space, and the figure shows the water depth where one full color cycle corresponds to $15\ m$ (one Δx in the simulation).

4 Experiments

We have performed several experiments with our implementation. We first present validation against a two-dimensional test problem where there is a known analytical solution. We then present verification against a real-world dam break, and finally we present performance experiments with several different datasets to show performance and scalability. Our benchmarks have been run on a machine with a 2.67 GHz quad core Intel Core i7 920 CPU with 6 GiB RAM. The graphics card is an NVIDIA GeForce GTX 480 with 1.5 GiB RAM in a PCI-express 2.0 $\times 16$ slot in the same system. In the results we present, we have compiled our CUDA source code with CUDA 3.0 using the compiler options

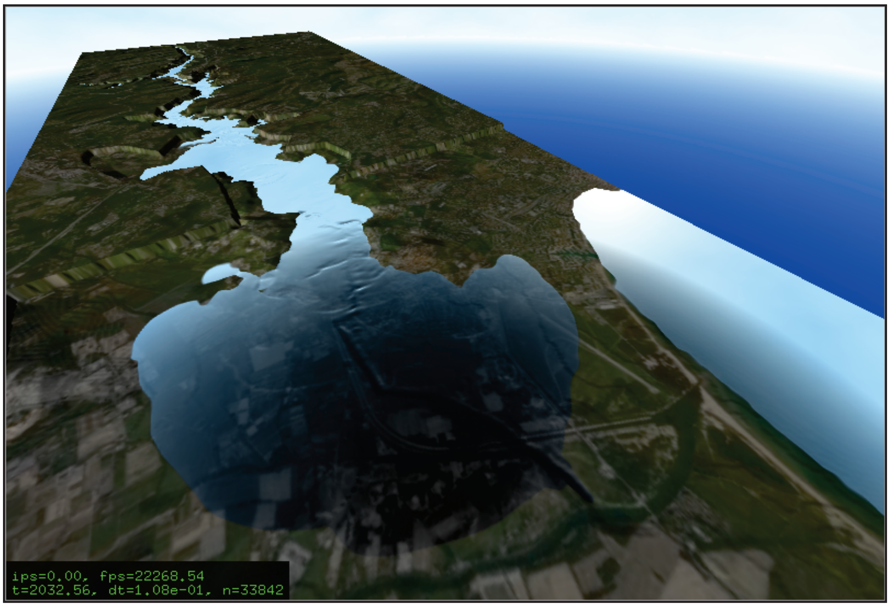
```
-arch=sm_20        //Generate compute 2.0 ptx code
-use_fast_math     //Use fast, but less accurate math functions
-ftz=true          //Flush denormal numbers to zero
-prec-div=false    //Use fast, but inaccurate division
-prec-sqrt=false //Use fast, but inaccurate square root
```

These options sacrifice precision for performance, enabling fast execution on the computational hardware. As will be evident from our verification and validation experiments, however, this does not hinder the models ability to capture analytical solutions and real-world flows.

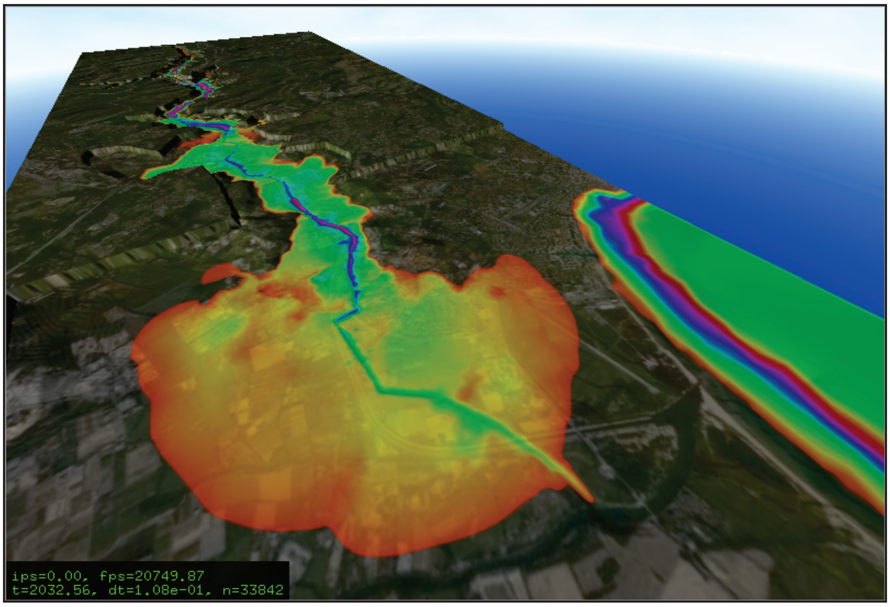
Verification: Oscillations in a Parabolic Basin

The analytical solution of time dependent motion in a friction-less parabolic basin described by Thacker [34] was chosen as the first test case for model verification. In this two-dimensional case, we have a parabolic basin where a planar water surface oscillates. More recently, Sampson et al. [31] extended the solutions of Thacker to include bed friction, however restricted to one dimension. These test problems have been used by several authors previously (see e.g., [24] and references in Sampson [31]) and, thus, can serve as a comparison with other numerical model results. In our test setup, which is identical to that presented by Holdahl et al. [17], we have the parabolic bottom topography given as

$$B(x,y) = D_0\left[(x^2+y^2)/L^2 - 1\right]. \tag{11}$$

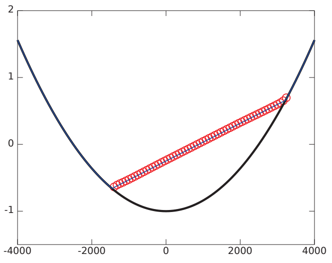


```
ips=0.00, fps=22268.54
t=2032.56, dt=1.08e-01, n=33842
```

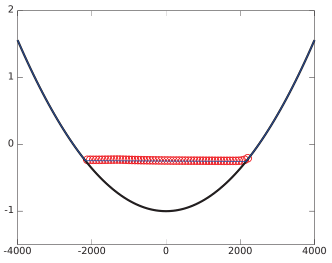


```
ips=0.00, fps=20749.87
t=2032.56, dt=1.08e-01, n=33842
```

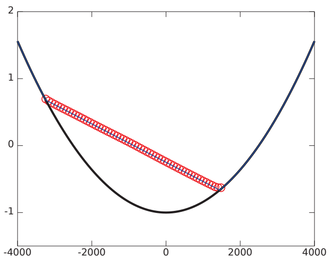
Figure 6: Two visualizations of the Malpasset dam break at $t = 2032.56$ seconds after the breach. We can visualize the results using the Fresnel equations, yielding a water surface with both reflection and refraction (top). We can also visualizes the physical variables using a color transfer function, here show for the water depth (bottom).



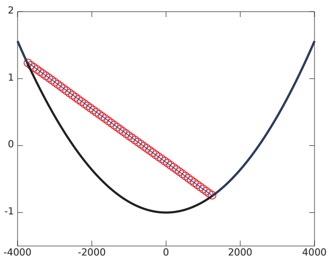
(a) $t \approx 1\pi/8\omega$ $(1392.85\ s)$



(b) $t \approx 2\pi/8\omega$ $(2787.95\ s)$



(c) $t \approx 3\pi/8\omega$ $(4198.64\ s)$



(d) $t \approx 4\pi/8\omega$ $(5559.27\ s)$

Figure 7: Simulation of oscillating water in a parabolic basin, compared to the analytical solution. The solid line is the analytical solution, and the circles are the computed values (only wet cells marked).

The water elevation and velocities at time t are defined as

$$\left.\begin{aligned} w &= 2AD_0(x\cos\omega t \pm y\sin\omega t + LB_0)/L^2 \\ u &= -A\omega\sin\omega t \\ v &= \pm A\omega\cos\omega t. \end{aligned}\right| \qquad \omega = \sqrt{2D_0/L^2} \tag{12}$$

We use the parameters $D_0 = 1$, $L = 2500$, $A = L/2$, and $B_0 = -A/2L$, set the manning coefficient $n = 0$, the gravitational constant $g = 1$, the desingularization epsilon $\kappa = 0.01$, and use 100×100 grid cells with the second-order accurate Runge-Kutta time integrator. Figure 7 shows our results for four snapshots in time compared to the respective analytical solutions. The figure clearly indicates that our implementation captures the analytical solution well for the water surface elevation. For the velocities, however, we see that there is a growing error along the wet-dry boundary that eventually also affects the elevation. This error is difficult to avoid, and is found in many schemes (see e.g., [17]).

Validation: The Malpasset Dam Break

The Malpasset dam, completed in 1954, was a 66.5 meter high double curvature dam with a crest length of over 220 meters that impounded 55 million cubic meters of water in the reservoir. Located in a narrow gorge along the Reyran River Valley, it literally exploded after heavy rainfall in early December 1959. Over 420 casualties were reported due to the resulting flood wave. This dam break is a unique real-world case, in that there exists front arrival time and maximum

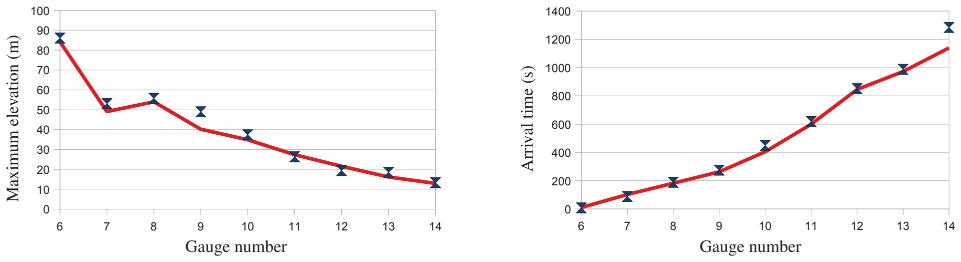
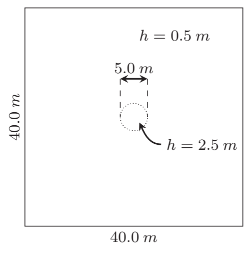
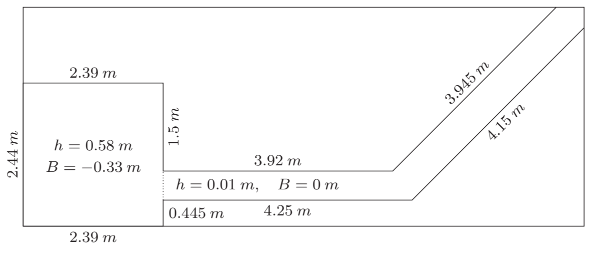


Figure 8: Validation against the maximum water elevation (left) and arrival time (right) from the Malpasset dam break case. The domain consists of 1099×439 cells with $\Delta x = \Delta y = 15$ meters. For both figures, the solid line represents the experimental data, and the symbols are the simulation results.





(a) Case 1 **(b)** Case 2

Figure 9: Dam break test cases used for performance benchmarks. The dotted lines indicate the location of the dams that are instantaneously removed at simulation start. In Case 2 the bottom topography of the area outside the reservoir and the canal is set to $2\ m$.

water elevation data from the original event, in addition to detailed data from a scaled model experiment [12, 10]. However, due to large changes to the terrain as a result of the flood, the digital bottom topography had to be reconstructed from a 1:20000 map dated 1931. The original dataset contains a total of 257622 unstructured points, and our regular Cartesian grid, identical to that of Ying and Wang [39], consists of 1100×440 bottom topography values spaced equally by 15 meters. This results in 1099×439 cells with $\Delta x = \Delta y = 15\ m$. We set the Manning coefficient to $n = 0.033\ m^{1/3}s$, the desingularization epsilon, $\kappa = 40\ cm$, and simulate the first 4000 seconds after the breach using Euler time integration. The desingularization epsilon is set rather high compared to our suggested initial guess of $15\ cm$. This value was chosen through experimentation: setting it to a lower value increased the computational time, while higher values did not dramatically decrease this time. A sensitivity analysis of the parameter would be useful to further justify its setting, but this is outside the scope of the current work.

Figure 8 shows our simulation results compared with the experimental data from a 1:400 scaled model [10]. The largest discrepancy is for gauge 9 for the maximum water elevation, and gauge 14 for the arrival time. These large discrepancies are also found in other published

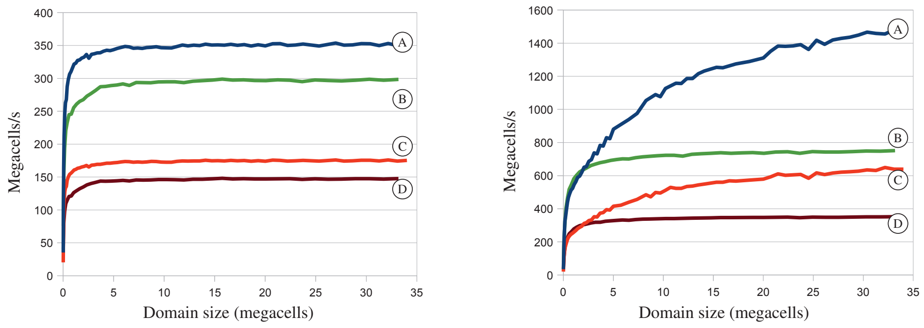


Figure 10: Absolute performance of our implementation as a function of domain size when calculating all cells (left), and when only calculating “wet” cells (right). Ⓐ and Ⓒ correspond to the dam break through 45° bend, and Ⓑ and Ⓓ correspond the to circular dam break case. Ⓐ and Ⓑ use first order accurate Euler time integration, and Ⓒ and Ⓓ use second order accurate Runge-Kutta time integration.

results of this case, and our results compare well with these (see e.g., [2, 16, 37, 39])

Performance Experiments

To assess the performance of our simulator, we have benchmarked our code on a set of well known test cases. Our first case is an idealised circular dam break [36, 23], meaning that the dam collapses instantaneously. The dam is located at the center of a 40×40 metre domain as shown in Figure 9a, where we employ wall boundaries. The second case is a dam break through a 45° bend, as used in the CADAM project [11]. Experimental set up consists of a reservoir connected to a $\sim 0.5\ m$-wide rectangular channel with a horizontal bottom by a $0.33\ m$-high positive step. The channel makes a sharp 45° bend to the left about $4\ m$ downstream from the reservoir and is initially filled with water up to a depth of $0.01\ m$ (see Figure 9b). The south and west boundaries are set as wall boundaries, and the north and east boundaries are free outlet.

Figure 10 shows the absolute performance of our implementation in megacells (millions of cells processed) per second, where we generate both cases for a large number of different domain sizes. We run the simulator to four seconds simulation time for case 1, in which the wave is roughly three metres from the edge having reached 57% of the domain. For the second case, the percentage of the domain covered by wet cells is constant, and we run our simulator for thirty wall clock seconds. Figure 10 (left) clearly illustrates that small domains do not fully utilize GPU’s processing capabilities. However, after five million cells, we see that the GPU has more or less reached the peak performance of up-to 350 million cells per second. For the circular dam break, five million cells corresponds to a domain sized roughly 2200×2200, which should give us almost 26000 blocks. The reason that we need this large number of blocks is not only to fully occupy the GPU hardware, but also in order to make overheads, such as launching kernels, negligible. The largest benchmark we have run is more than 33 million cells (5760×5760 cells for case 1), consuming over 1.4 GB of the 1.5 GB available RAM. We also see that there is a clear difference between the square and rectangular domains, which might seem odd. However, in our tests, we have discovered that this is not in fact due to the size of the

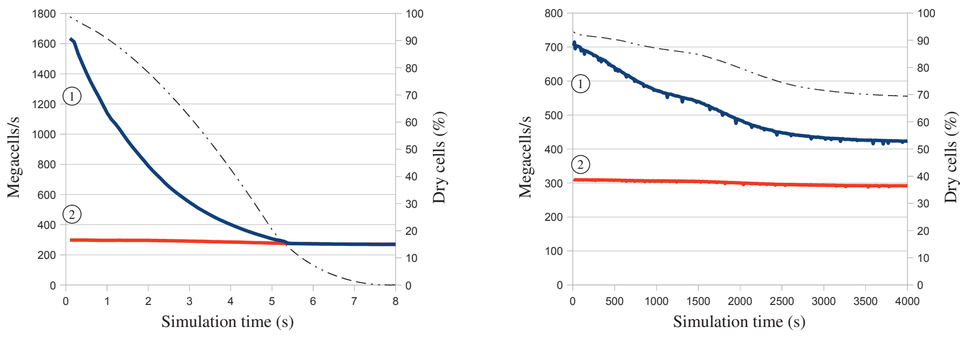


Figure 11: Absolute performance in megacells per second of our implementation for the circular dam break case (left) and the Malpasset dam break case (right). The domains contain 4000×4000 and 1099×439 cells respectively. The graphs show the estimated instantaneous performance as a function of simulation time: ① has enabled the early exit optimization, and ② calculates all cells. The dashed line shows the percentage of dry cells in the domain.

domain, but rather connected to the ratio of wet to dry cells. The reason for the difference is that in the flux function, we do not compute the fluxes for dry cells. This is not related at all to the early exit optimization, but to the fact that we set the values at integration points with $h < \epsilon_m$ to zero. Examining Figure 9, we see that we have water in all cells in case 1, whereas we have large parts where we have no water at all for case 2. However, in the dry parts of the second case we still have to compute all of the reconstructions before we find out that the integration point in fact is dry. We can also see that there is a clear distinction between the first order and second order accurate time integrator schemes: running the second order accurate scheme takes twice as long as the first order scheme, as is expected. In Figure 10 (right) we have enabled the early exit optimization, where dry blocks do not even perform the reconstructions. Compared to Figure 10 (left), we immediately see the effect of this optimization for both cases. For case 1, our performance more than doubles by enabling the early exit optimization, and we soon reach a peak performance of 700 to 750 megacells per second for domain sizes larger than five megacells. For case 2, however, we see a much more rough curve, in addition to it rising to a much higher peak performance. This is because we have a fixed number of cells that are marked as “wet”, independent of time since small waves emerging from the canal walls ensure that the canal is marked as wet within very few timesteps. Thus, by increasing the resolution, we also increase the relative number of blocks that perform early exit. This illustrates that our early exit strategy is highly suited for domains where there are large dry zones.

Figure 11 further shows the effect of the early exit optimization, where we show the absolute performance as a function of simulation time. This illustrates how the propagation of the solution affects the performance. When calculating the full domain, we see that the performance remains at around 300 megacells per second throughout the simulation. When early exit is enabled, we see that the initial performance is much higher, and gradually decreases as the solution progresses. This is because the waves from the dam break spread out, thus marking more and more blocks as “wet”. For case 1 the early exit kernel eventually takes *more* time than the kernel that calculates all cells. However, as we always select the fastest kernel, we are not penalized at all for enabling early exit. For the Malpasset dam break case, on the other hand, we see that the early exit kernel always performs better. This is because the water here follows the valley, thus marking fewer cells as “wet” (see also Figure 5, which is an actual map of “wet” blocks for the Malpasset case). We also here see that for the Malpasset dam break case we reach less than half of the performance of case 1. This is largely due to the domain size which is insufficiently large to mask all overheads.

We have also profiled our code using the CUDA visual profiler, which gives a rough idea of our resource utilization. We run the circular dam break without the early exit optimization using the Runge-Kutta time integration on a 4000×4000 domain, which should make any overheads negligible. For this domain size, we have a 98% utilization of the GPU. Of the 98% percent GPU utilization, our flux kernel is by far most time consuming, using 87.5% of the runtime. Next comes the time integration kernel, with 12%, leaving less than one percent of the time spent on boundary conditions, maximum timestep, and memory copies to set kernel parameters. For our flux kernel, however, we only achieve an instruction throughput of 56%. This relatively low figure comes from the mismatch between the block size and the conflicting optimization parameters. Nevertheless, this is a trade-off where we are able to implement a much more efficient time integration kernel by having a less efficient flux kernel.

Finally, we have tested the performance impact of running real-time visualization of the results. Our implementation ensures a steady frame rate of 30 frames per second visualization, and runs as many simulation steps as possible. For the Malpasset dam break case, for example, our simulator runs a 4000 second simulation in 27 seconds without visualization. Enabling the visualization, however, the same simulation takes only slightly more than 30 seconds, a mere 11% increase. This clearly shows that efficient utilization of GPUs can also be used to offer real-time visualization without a major effect on the simulator performance.

5 Summary

We have presented a highly optimized implementation of the Kurganov-Petrova scheme on GPUs. The implementation has been verified and validated, showing its ability to capture both analytical and real-world shallow water flows, even with first-order accurate time integration. Our implementation contains novel optimization techniques including the especially efficient early exit strategy, clever application of boundary conditions, and a dramatically smaller memory footprint compared to Brodtkorb et al. [7]. Our extensive performance benchmarks show good resource utilization, being able to compute the first 4000 seconds of the Malpasset dam break case in 27 seconds.

Acknowledgements

Part of this work is done under Research Council of Norway project number 180023 (Parallel3D) and 186947 (Heterogeneous Computing). The author from SINTEF would like to acknowledge the continued support from NVIDIA. Partly accomplished at the National Center for Computational Hydroscience and Engineering, this research was also funded by the Department of Homeland Security-sponsored Souteast Region Research Initiative (SERRI) at the U.S. Department of Energy’s Oak Ridge National Laboratory.

Bibliography

[1] M. A. Acuña and T. Aoki. Real-time tsunami simulation on multi-node GPU cluster. Supercomputing, 2009. [Poster].

[2] F. Alcrudo and E. Gil. The Malpasset dam break case study. In *The Proceedings of the 4th CADAM meeting*, 1999.

[3] A. S. Antoniou, K. I. Karantasis, E. D. Polychronopoulos, and J. A. Ekaterinaris. Acceleration of a finite-difference weno scheme for large-scale simulations on many-core architectures. In *Proceedings of the 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, 2010.

[4] T. Brandvik and G. Pullan. Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware. *Journal of Mechanical Engineering Science*, 221(12):1745–1748, 2007.

[5] T. Brandvik and G. Pullan. Acceleration of a 3D Euler solver using commodity graphics hardware. In *46th AIAA Aerospace Sciences Meeting and Exhibit*, number AIAA 2008-607, 2008.

[6] A. Brodtkorb, C. Dyken, T. Hagen, J. Hjelmervik, and O. Storaasli. State-of-the-art in heterogeneous computing. *Journal of Scientific Programming*, 18(1):1–33, 2010.

[7] A. Brodtkorb, T. R. Hagen, K.-A. Lie, and J. R. Natvig. Simulation and visualization of the Saint-Venant system using GPUs. *Computing and Visualization in Science*, 2010. [forthcoming].

[8] M. de la Asunción, J. M. Mantas, and M. J. Castro. Programming CUDA-based GPUs to simulate two-layer shallow water flows. In *Proceedings of the 16th International Euro-Par Conference*, 2010. [accepted for publication].

[9] M. de la Asunción, J. M. Mantas, and M. J. Castro. Simulation of one-layer shallow water systems on multicore and CUDA architectures. *Journal of Supercomputing*, 2010.

[10] S. S. Frazão, F. Alcrudo, and N. Goutal. Dam-break test cases summary. In *The Proceedings of the 4th CADAM meeting*, 1999.

[11] S. S. Frazão, X. Sillen, and Y. Zech. Dam-break flow through sharp bends, physical model and 2D Boltzmann model validation. In *The Proceedings of the 1st CADAM meeting*, 1998.

[12] N. Goutal. The Malpasset dam failure, an overview and test case definition. In *The Proceedings of the 4th CADAM meeting*, 1999.

[13] T. Hagen, M. Henriksen, J. Hjelmervik, and K.-A. Lie. How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational engine. In G. Hasle, K.-A. Lie, and E. Quak, editors, *Geometrical Modeling, Numerical Simulation, and Optimization: Industrial Mathematics at SINTEF*, pages 211–264. Springer Verlag, 2007.

[14] T. Hagen, J. Hjelmervik, K.-A. Lie, J. Natvig, and M. Henriksen. Visual simulation of shallow-water waves. *Simulation Modelling Practice and Theory*, 13(8):716–726, 2005.

[15] T. R. Hagen, K.-A. Lie, and J. R. Natvig. Solving the Euler equations on graphics processing units. In *Proceedings of the 6th International Conference on Computational Science*, volume 3994 of *Lect. Notes Comp. Sci.*, pages 220–227, Berlin/Heidelberg, 2006. Springer Verlag.

[16] J.-M. Hervouet and A. Petitjean. Malpasset dam-break revisited with two-dimensional computations. *Journal of Hydraulic Research*, 37:777–788, 1999.

[17] R. Holdahl, H. Holden, and K.-A. Lie. Unconditionally stable splitting methods for the shallow water equations. *BIT Numerical Mathematics*, 39(3):451–472, 1999.

[18] A. Klöckner, T. Warburton, J. Bridge, and J. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics*, 228(21):7863–7882, 2009.

[19] A. Kurganov, S. Noelle, and G. Petrova. Semidiscrete central-upwind schemes for hyperbolic conservation laws and Hamilton–Jacobi equations. *SIAM Journal of Scientific Computing*, 23(3):707–740, 2001.

[20] A. Kurganov and G. Petrova. A second-order well-balanced positivity preserving central-upwind scheme for the Saint-Venant system. *Communications in Mathematical Sciences*, 5:133–160, 2007.

[21] M. Lastra, J. M. Mantas, C. U. na, M. J. Castro, and J. A. García-Rodríguez. Simulation of shallow-water systems using graphics processing units. *Mathematics and Computers in Simulation*, 80(3):598 – 618, 2009.

[22] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, pages 451–460, New York, NY, USA, 2010. ACM.

[23] R. J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2002.

[24] Q. Liang and F. Marche. Numerical resolution of well-balanced shallow water equations with complex source terms. *Advances in Water Resources*, 32(6):873 – 884, 2009.

[25] W.-Y. Liang, T.-J. Hsieh, M. Satria, J.-P. Chang, Y.-L.and Fang, C.-C. Chen, and C.-C. Han. A GPU-based simulation of tsunami propagation and inundation. In *Algorithms and Architectures for Parallel Processing*, volume 5574 of *Lect. Notes Comp. Sci.*, pages 593–603, Berlin/Heidelberg, 2009. Springer Verlag.

[26] NetCDF (network Common Data Form). `http://www.unidata.ucar.edu/software/netcdf/`. [visited 2010-06-01].

[27] NVIDIA. CUDA community showcase. 2010.

[28] NVIDIA. NVIDIA CUDA programming guide 3.0, 2010.

[29] NVIDIA. Tuning CUDA applications for Fermi, 2010.

[30] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[31] J. Sampson, A. Easton, and M. Singh. Moving boundary shallow water flow above parabolic bottom topography. *Australian and New Zealand Industrial and Applied Mathematics Journal*, 49:666–680, 2006.

[32] D. Shreiner and Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley, 7th edition edition, 2007.

[33] C.-W. Shu. Total-variation-diminishing time discretizations. *SIAM Journal of Scientific and Statistical Computing*, 9(6):1073–1084, 1988.

[34] W. C. Thacker. Some exact solutions to the nonlinear shallow-water wave equations. *Journal of Fluid Mechanics*, 107:499–508, 1981.

[35] Top 500 supercomputer sites. `http://www.top500.org/`, June 2010.

[36] E. F. Toro. *Shock-Capturing Methods for Free-Surface Shallow Flows*. John Wiley & Sons, Ltd., 2001.

[37] A. Valiani, V. Caleffi, and A. Zanni. Case study: Malpasset dam-break simulation using a two-dimensional finite volume method. *Journal of Hydraulic Engineering*, 128:460–472, 2002.

[38] P. Wang, T. Abel, and R. Kaehler. Adaptive mesh fluid simulations on GPU. *New Astronomy*, 15(7):581–589, 2010.

[39] X. Ying and S. Y. Wang. Modeling flood inundation due to dam and levee breach. In *Proceedings of the US-China Workshop on Advanced Computational Modeling in Hydroscience and Engineering*, 2005.

Paper IV

Shallow Water Simulations on Multiple GPUs

M. L. Sætra and A. R. Brodtkorb

Lecture Notes in Computer Science, **7134** (2012), pp. 56-66

Abstract: We present a state-of-the-art shallow water simulator running on multiple GPUs. Our implementation is based on an explicit high-resolution finite volume scheme suitable for modeling dam breaks and flooding. We use row domain decomposition to enable multi-GPU computations, and perform traditional CUDA block decomposition within each GPU for further parallelism. Our implementation shows near perfect weak and strong scaling, and enables simulation of domains consisting of up-to 235 million cells at a rate of over 1.2 gigacells per second using four Fermi-generation GPUs. The code is thoroughly benchmarked using three different systems, both high-performance and commodity-level systems.

1 Introduction

Predictions of floods and dam breaks require accurate simulations with rapid results. Faster than real-time performance is of the utmost importance when simulating these events, and traditional CPU-based solutions often fall short of this goal. We address the performance of shallow water simulations in this paper through the use of multiple graphics processing units (GPUs), and present a state-of-the-art implementation of a second-order accurate explicit high-resolution finite volume scheme.

There has been a dramatic shift in commodity-level computer architecture over the last five years. The steady increase in performance does no longer come from higher clock frequencies, but from parallelism through more arithmetic units: The newest CPU from Intel, for example, contains 24 single precision arithmetic units (Core i7-980X). The GPU takes this parallelism even further with up-to 512 single precision arithmetic units (GeForce GTX 580). While the GPU originally was designed to offload a predetermined set of demanding graphics operations from the CPU, modern GPUs are now fully programmable. This makes them suitable for general purpose computations, and the use of GPUs has shown large speed-ups over the CPU in many application areas [5, 22]. The GPU is connected to the rest of the computer through the PCI Express bus, and commodity-level computers can have up-to two GPUs connected at full data speed. Such solutions offer the compute performance comparable to a small CPU cluster, and this motivates the use of multiple GPUs. In fact, three of the five fastest supercomputers use GPUs as a major source of computational power [19]. However, the extra floating-point performance comes at a price, as it is nontrivial to develop efficient algorithms for GPUs, especially when targeting multiple GPUs. It requires both different programming models and different optimization techniques compared to traditional CPUs.

Related Work: The shallow water equations belong to a wider class of problems known as hyperbolic conservation laws, and many papers have been published on GPU-acceleration of both conservation and balance laws [11, 13, 3, 4, 14, 27, 2]. There have been multiple publications on the shallow water equations as well [12, 18, 17, 8, 9, 6], illustrating that these problems can be efficiently mapped to modern graphics hardware. The use of multiple GPUs has also become a subject of active research. Micikevicius [20] describes some of the benefits of using multiple GPUs for explicit finite-difference simulation of 3D reverse time-migration (the linear wave equation), and reports super-linear speedup when using four GPUs. Overlapping computation and communication for explicit stencil computations has also been presented for both single nodes [24] and clusters [15] with near-perfect weak scaling. Perfect weak scaling was shown by Acuña and Aoki [1] for shallow water simulations on a cluster of 32 GPU nodes, by overlaping computations and communication. Rostrup and De Sterck [25] further present detailed optimization and benchmarking of shallow water simulations on clusters of multi-core CPUs, the Cell processor, and GPUs. Comparing the three, the GPUs offer the highest performance.

In this work, we focus on single-node systems with multiple GPUs. By utilizing more than one GPU it becomes feasible to run simulations with significantly larger domains, or to increase the spatial resolution. Our target architecture is both commodity-level computers with up-to two GPUs, as well as high-end and server solutions with up-to four GPUs at full data speed per node. We present a multi-GPU implementation of a second-order well-balanced positivity preserving central-upwind scheme [16]. Furthermore, we offer detailed performance benchmarks on three different machine setups, tests of a latency-hiding technique called ghost cell expansion, and analyzes of benchmark results.

2 Mathematical Model and Discretization

In this section, we give a brief outline of the major parts of the implemented numerical scheme. For a detailed overview of the scheme, we refer the reader to [16, 7]. The shallow water equations are derived by depth-integrating the Navier-Stokes equations, and describe fluid motion under a pressure surface where the governing flow is horizontal. To correctly model phenomena such as tsunamis, dam breaks, and flooding over realistic terrain, we need to include source terms for bed slope and friction:

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix} + \begin{bmatrix} 0 \\ -gu\sqrt{u^2+v^2}/C_z^2 \\ -gv\sqrt{u^2+v^2}/C_z^2 \end{bmatrix}. \tag{1}$$

Here h is the water depth and u and v are velocities along the abscissa and ordinate, respectively. Furthermore, g is the gravitational constant, B is the bottom topography, and C_z is the Chézy friction coefficient.

To be able to simulate dam breaks and flooding, we require that our numerical scheme handles wetting and drying of cells, a numerically challenging task. However, we also want other properties, such as well-balancedness, accurate shock-capturing without oscillations, at least second order accurate flux calculations, and that the computations map well to the architecture of the GPU. A scheme that fits well with the above criteria is the explicit Kurganov-Petrova scheme [16], which is based on a standard finite volume grid. In this scheme, the physical variables are given as cell averages, the bathymetry as a piecewise bilinear function (represented by the values at the cell corners), and fluxes are computed across cell interfaces (see also Figure 1). Using vectorized notation, in which $Q = [h, hu, hv]^T$ is the vector of conserved variables, the spatial discretization can be written,

$$\begin{aligned} \frac{dQ_{ij}}{dt} &= H_f(Q_{ij}) + H_B(Q_{ij}, \nabla B) - \left[F(Q_{i+1/2,j}) - F(Q_{i-1/2,j})\right] \\ &\qquad - \left[G(Q_{i,j+1/2}) - G(Q_{i,j-1/2})\right] \\ &= H_f(Q_{ij}) + R(Q)_{ij}. \end{aligned} \tag{2}$$

Here $H_f(Q_{ij})$ is the friction source term, $H_B(Q_{ij}, \nabla B)$ is the bed slope source term, and F and G are the fluxes across interfaces along the abscissa and ordinate, respectively. We first calculate $R(Q)_{ij}$ in (2) explicitly, and as in [7], we use a semi-implicit discretization of the friction source term,

$$\tilde{H}_f(Q_{ij}^k) = \begin{bmatrix} 0 \\ -g\sqrt{{u_{ij}^k}^2 + {v_{ij}^k}^2}/h_{ij}^k {C_z}_{ij}^2 \\ -g\sqrt{{u_{ij}^k}^2 + {v_{ij}^k}^2}/h_{ij}^k {C_z}_{ij}^2 \end{bmatrix}. \tag{3}$$

This yields one ordinary differential equation in time per cell, which is then solved using a standard second-order accurate total variation diminishing Runge-Kutta scheme [26],

$$\begin{aligned} Q^*_{ij} &= \big[Q^n_{ij} + \Delta t R(Q^n)_{ij}\big] / \big[1 + \Delta t \tilde{H}_f(Q^n_{ij})\big] \\ Q^{n+1}_{ij} &= \Big[\tfrac{1}{2}Q^n_{ij} + \tfrac{1}{2}\big[Q^*_{ij} + \Delta t R(Q^*)_{ij}\big]\Big] / \big[1 + \tfrac{1}{2}\Delta t \tilde{H}_f(Q^*_{ij})\big], \end{aligned} \tag{4}$$

or a first-order accurate Euler scheme, which simply amounts to setting $Q^{n+1} = Q^*$. The timestep, Δt, is limited by a CFL condition,

$$\Delta t \le \tfrac{1}{4}\min_\Omega\Big\{\big|\Delta x/\lambda_x\big|, \big|\Delta y/\lambda_y\big|\Big\}, \qquad \begin{aligned} \lambda_x &= u \pm \sqrt{gh}, \\ \lambda_y &= v \pm \sqrt{gh} \end{aligned} \tag{5}$$

that ensures that the fastest numerical propagation speed is at most one quarter grid cell per timestep.

In summary, the scheme consists of three parts: First fluxes and explicit source terms are calculated in (2), before we calculate the maximum timestep according to the CFL condition, and finally evolve the solution in time using (4). The second-order accurate Runge-Kutta scheme for the time integration is a two-step process, where we first perform the above operations to compute Q^*, and then repeat the process to compute Q^{n+1}.

3 Implementation

Solving partial differential equations using explicit schemes implies the use of stencil computations. Stencil computations are embarrassingly parallel and therefore ideal for the parallel execution model of GPUs. Herein, the core idea is to use more than one GPU to allow faster simulation, or simulations with larger domains or higher resolution. Our simulator runs on a single node, enabling the use of multithreading, and we use one global *control* thread in addition to one *worker* thread per GPU. The control thread manages the worker threads and facilitates domain decomposition, synchronization, and communication. Each worker thread uses a modified version of our previously presented single-GPU simulator [7] to compute on its part of the domain.

Single-GPU Simulator: The single-GPU simulator implements the Kurganov-Petrova scheme on a single GPU using CUDA [21], and the following gives a brief overview of its implementation. The simulator first allocates and initializes data according to the initial conditions of the problem. After initialization, we repeatedly call a step function to advance the solution in time. The step function executes four CUDA *kernels* in order, that together implement the numerical scheme. The first kernel computes the fluxes across all interfaces, and is essentially a complex stencil computation. This kernel reads four values from global memory, performs hundreds of floating point operations, and writes out three values to global memory again. It is also the most time consuming kernel, with over 87% of the runtime. The next kernel finds the maximum wave speed in the domain, and then computes the timestep size according to the CFL condition. The third kernel simply solves the ordinary differential equations in time to evolve the solution. Finally, the fourth kernel applies boundary conditions by setting the values of global *ghost cells* (see Figure 1).

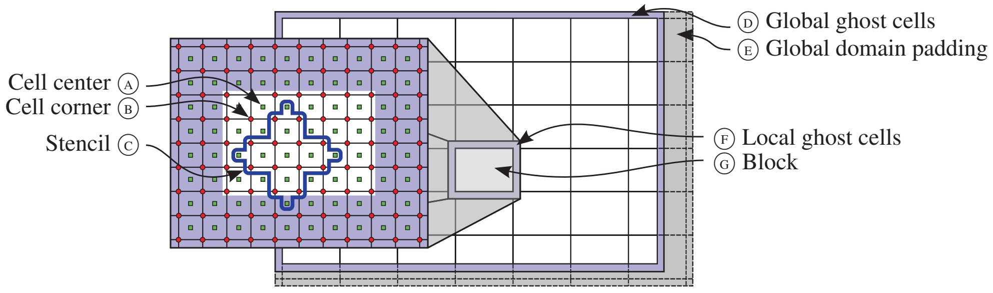


Figure 1: Domain decomposition and variable locations for the single-GPU simulator. The global domain is padded Ⓔ to fit an integer number of CUDA blocks, and global ghost cells Ⓓ are used for boundary conditions. Each block Ⓖ has local ghost cells Ⓕ that overlap with other blocks to satisfy the data dependencies dictated by the stencil Ⓒ. Our data variables Q, R, H_B, and H_f are given at grid cell centers Ⓐ, and B is given at grid cell corners Ⓑ.

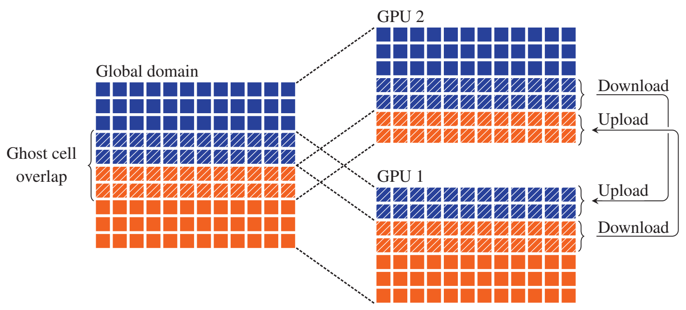


Figure 2: Row decomposition and exchange of two rows of ghost cells. The shaded cells are a part of the overlapping ghost cell region between subdomains.

Threaded Multi-GPU Framework: When initializing our simulator, the control thread starts by partitioning the global domain, and continues by initializing one worker thread per subdomain, which attaches to a separate GPU. We can then perform simulation steps, where the control thread manages synchronization and communication between GPUs. An important thing to note about this strategy is that the control thread handles all multi-GPU aspects, and that each GPU is oblivious to other GPUs, running the simulation on its subdomain similar to a single-GPU simulation.

We use a row domain decomposition, in which each subdomain consists of several rows of cells (see Figure 2). The subdomains form overlapping regions, called ghost cells, which function as boundary conditions that connect the neighbouring subdomains. By exchanging the overlapping cells before every timestep, we ensure that the solution can propagate properly between subdomains. There are several benefits to the row decomposition strategy. First of all, it enables the transfer of continuous parts of memory between GPUs, thus maximizing bandwidth utilization. A second benefit is that we can minimize the number of data transfers, as each subdomain has at most two neighbours. To correctly exchange ghost cells, the control thread starts by instructing each GPU to download its ghost cells to *pinned* CPU memory, as direct GPU to GPU-transfers are currently not possible. The size of the ghost cell overlap is dictated by the stencil, which in our case uses two values in each direction (see Figure 1). This means that we need to have an overlap of four rows of cells, two from each of the subdomains. After having downloaded the ghost cells to the CPU, we need to synchronize to guarantee that all downloads have completed, before each GPU can continue by uploading the ghost cells coming from neighbouring subdomains. Note that for the second-order accurate Runge-Kutta time integration scheme, we have to perform the ghost cell exchange both when computing Q^* and when computing Q^{n+1}, thus two times per full timestep.

The multi-GPU simulator is based on our existing single-GPU simulator, which made certain assumptions that made it unsafe to execute from separate threads. This required us to redesign parts of the code to guarantee thread safety. A further difficulty related to multi-GPU simulation is that the computed timestep, Δt, will typically differ between subdomains. There are two main strategies to handle this problem, and we have investigated both. The simplest is to use a globally fixed timestep throughout the simulation. This, however, requires that the timestep is less than or equal to the smallest timestep allowed by the CFL condition for the full simulation period, which again implies that our simulation will not propagate as fast as it could have. The second strategy is to synchronize the timestep between subdomains for each timestep, and choose the smallest. This strategy requires that we split the step function into two parts, where the first computes fluxes and the maximum timestep, and the second performs time integration and applies boundary conditions. Inbetween these two substeps we can find the smallest global timestep, and redistribute it to all GPUs. This strategy ensures that the simulation propagates at the fastest possible rate, but at the expense of potentially expensive synchronization and more complex code.

Ghost Cell Expansion: Synchronization and overheads related to data transfer can often be a bottleneck when dealing with distributed memory systems, and a lot of research has been invested in, e.g., latency hiding techniques. In our work, we have implemented a technique called *ghost cell expansion* (GCE), which has yielded a significant performance increase for cluster simulations [10, 23]. The main idea of GCE is to trade more computation for smaller overheads by increasing the level of overlap between subdomains, so that they may run more than one timestep per ghost cell exchange. For example, by extending the region of overlap from four to eight cells, we can run two timesteps before having to exchange data. When exchanging ghost cells for every timestep, we can write the time it takes to perform one timestep as

$$w_1 = T(m) + c_T + C(m,n) + c,$$

in which m and n are the domain dimensions, $T(m)$ is the ghost cell transfer time, c_T represents transfer overheads, $C(m,n)$ is the time it takes to compute on the subdomain, and c represents other overheads. Using GCE to exchange ghost cells only every kth timestep, the average time per timestep becomes

$$w_k = T(m) + c_T/k + C(m, n + \mathcal{O}(k)) + c,$$

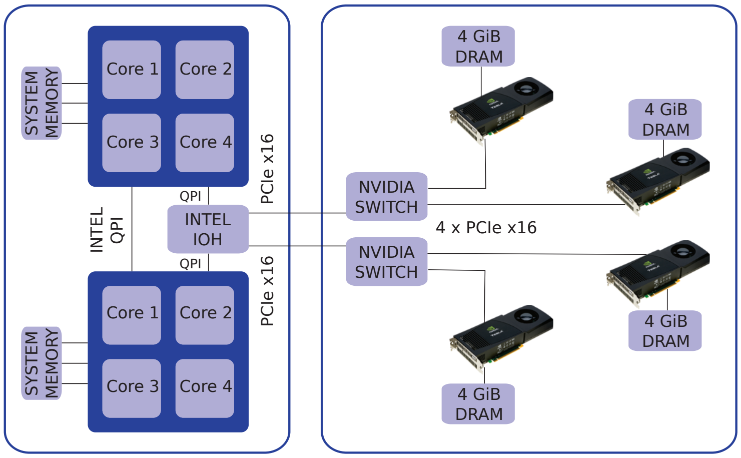


Figure 3: Hardware setup of the Tesla S1070 GPU Computing Server with four Tesla C1060 GPUs (right) connected to an IBM X3550 M2 server (left).

in which we divide the transfer overheads by k, but increase the overlap, and thus the size of each subdomain. This means that each worker thread computes on a slightly larger domain, and we have larger but fewer data transfers.

4 Results and Analysis

To validate our implementation, we have compared the multi-GPU results with the original single-GPU simulator [7], which has been both verified against analytical solutions and validated against experiments. Our multi-GPU results are identical to those produced by the single-GPU implementation, which means that the multi-GPU implementation is also capable of reproducing both analytical and real-world cases.

We have used three different systems for benchmarking our implementation. The first system is a Tesla S1070 GPU Computing Server consisting of four Tesla C1060 GPUs with 4 GiB memory each[6], connected to an IBM X3550 M2 server with two 2.0 GHz Intel Xeon X5550 CPUs and 32 GiB main memory (see Figure 3). The second system is a SuperMicro SuperServer consisting of four Tesla C2050 GPUs with 3 GiB memory each (2.6 available when ECC is enabled)[7], and two 2.53 GHz Intel Xeon E5630 CPUs with 32 GiB main memory. The third system is a standard desktop PC consisting of two GeForce 480 GTX cards with 1.5 GiB memory each and a 2.67 GHz Intel Core i7 CPU with 6 GiB main memory. The first two machine setups represent previous and current generation server GPU nodes, and the third machine represents a commodity-level desktop PC.

As our performance benchmark, we have used a synthetic circular dam break over a flat bathymetry, consisting of a square 4000-by-4000 meters domain with a water column placed in the center. The water column is 250 meters high with a radius of 250 meters, and the water elevation in the rest of the domain is 50 meters. At time $t = 0$, the dam surrounding the water

[6]Connected through two PCIe $\times16$ slots.
[7]Connected through four PCIe $\times16$ slots.

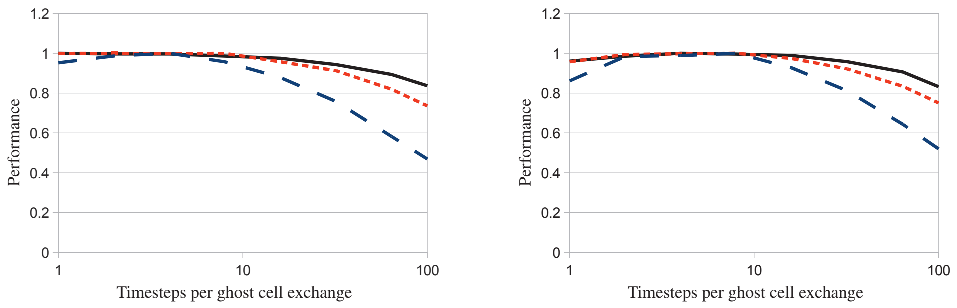


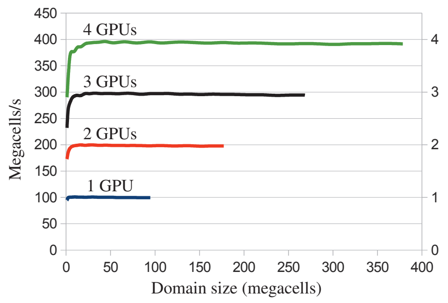
Figure 4: Testing the impact of different numbers of ghost cell expansion rows with four GPUs. Tested on both The Tesla S1070 system (see Figure 3) (left), and the Tesla C2050-based system (right). The domains tested consists of 1024^2 (dashed), 4096^2 (densely dashed) or 8192^2 (solid) cells. The graphs have been normalized relative to the peak achieved performance for each domain size.

column is instantaneously removed, creating an outgoing circular wave. We have used the first-order accurate Euler time integrator in all benchmarks, and the friction coefficient C_z is set to zero. The bed slope and friction coefficient do not affect the performance in this benchmark.

Ghost Cell Expansion We have implemented ghost cell expansion so that we can vary the level of overlap, and benchmarked three different domain sizes to determine the effect. Figure 4 shows that there is a very small overhead related to transferring data for sufficiently large domain sizes, and performing only one timestep before exchanging overlapping ghost cells actually yields the best overall results for the Tesla S1070 system. Expanding with more than eight cells, the performance of the simulator starts decreasing noticeably. From this, we reason that the overhead connected with data transfers between subdomains in these tests is negligible, compared to the transfer and computational time. Increasing the level of GCE only had a positive impact on the smallest domain for the Tesla S1070 system, where the transferred data volume is so small that the overheads become noticeable. On the Tesla C2050-based system, however, we see that the positive impact of GCE is more visible. We expect this is because this GPU is much faster, making the communication overheads relatively larger.

Our results show that ghost cell expansion had only a small impact on the shared-memory architectures we are targeting for reasonably sized grids, but gave a slight performance increase for the Tesla C2050 GPUs. This is due to the negligible transfer overheads. We thus expect GCE to have a greater effect when performing ghost cell exchange across multiple nodes, since the overheads here will be significantly larger, and we consider this a future research direction.

Since our results show that it is most efficient to have a small level of GCE for the Tesla S1070 system, we choose to exchange ghost cells after every timestep in all of our other benchmarks for this system. For the Tesla C2050-based system we exchange data after eight timesteps, as this gave the overall best results. Last, for the GeForce 480 GTX cards, which displayed equivalent behaviour to that of the Tesla C2050-based system, we also exchange ghost cells after performing eight timesteps.



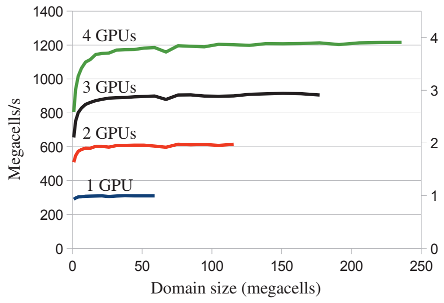


Figure 5: (Left) Performance experiment on a Tesla S1070 system (see Figure 3) with up-to four GPUs. (Right) Performance experiment on a Tesla C2050-based system, using up-to four GPUs. The secondary y-axis on the right-hand side shows scaling relative to the peak achieved performance of a single GPU.

Timestep Synchronization: We have implemented both the use of a global fixed timestep, as well as exchange of the minimum global timestep in our code, and benchmarked on our three test systems to determine the penalty of synchronization. In the tests we compared simulation runs with a fixed $\Delta t = 0.001$ in each subdomain, and runs with global synchronization of Δt. When looking at the results we see that the cost of synchronizing Δt globally has a negligible impact on the performance of the Tesla S1070 system, with an average 0.36% difference for domain sizes larger than five million cells on four GPUs. As expected, the cost is also roughly halved when synchronizing two GPUs compared to four (0.17%). For smaller domain sizes, however, the impact becomes noticeable, but these domains are typically not candidates for multi-GPU simulations. The Tesla C2050- and GeForce 480 GTX-based systems also display similar results, meaning that global synchronization of Δt is a viable strategy for reasonably sized domains.

Weak and Strong Scaling: Weak and strong scaling are two important performance metrics that are used for parallel execution. While varying the number of GPUs, weak scaling keeps the domain size per GPU fixed, and strong scaling keeps the global domain size fixed. As we see from Figure 5, we have close to linear scaling from one to four GPUs. For domains larger than 25 million cells the simulator displays near perfect weak and strong scaling on all three systems. Running simulations on small domains is less efficient when using multiple GPUs for two reasons: First of all, as the global domain is partitioned between more GPUs, we get a smaller size of each subdomain. When these subdomains become sufficiently small, we are unable to fully occupy a single GPU, and thus do not reach peak performance. Second, we also experience larger effects of overheads. However, we quickly get close-to linear scaling as the domain size increases.

The Tesla C1060 GPUs have 4.0 GiB of memory each, which enables very large scale simulations: When using all four GPUs, domains can have up to 379 million cells, computing at 396 megacells per second. Because the most recent Tesla C2050 GPUs from NVIDIA have only 3.0 GiB memory per GPU, our maximum domain size is smaller (235 million cells), but our simulation speed is dramatically faster. Using four GPUs, we achieve over 1.2 gigacells per second. The fastest system per GPU, however, was the commodity-level desktop machine with two GeForce 480 GTX cards. These cards have the highest clock frequency, and we achieve over 400 megacells per second per GPU.

5 Summary and Future Work

We have presented an efficient multi-GPU implementation of a modern finite volume scheme for the shallow water equations. We have further presented detailed benchmarking of our implementation on three hardware setups, displaying near-perfect weak *and* strong scaling on all three. Our benchmarks also show that communication between GPUs within a single node is very efficient, which enables tight cooperation between subdomains.

A possible further research direction is to explore different strategies for domain decomposition, and especially to consider techniques for adaptive domain decompositions.

Acknowledgements: Part of this work is supported by the Research Council of Norway's project number 180023 (Parallel3D), and the authors are grateful for the continued support from NVIDIA. Thanks also to Ole W. Saastad for access to the Teflon GPU compute cluster at the University of Oslo.

Bibliography

[1] M. Acuña and T. Aoki. Real-time tsunami simulation on multi-node GPU cluster. ACM/IEEE conference on Supercomputing, 2009. [poster].

[2] A. Antoniou, K. Karantasis, E. Polychronopoulos, and J. Ekaterinaris. Acceleration of a finite-difference weno scheme for large-scale simulations on many-core architectures. In *Proceedings of the 48th AIAA Aerospace Sciences Meeting*, 2010.

[3] T. Brandvik and G. Pullan. Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware. *Journal of Mechanical Engineering Science*, 221(12):1745–1748, 2007.

[4] T. Brandvik and G. Pullan. Acceleration of a 3D Euler solver using commodity graphics hardware. In *Proceedings of the 46th AIAA Aerospace Sciences Meeting*, number 2008-607, 2008.

[5] A. Brodtkorb, C. Dyken, T. Hagen, J. Hjelmervik, and O. Storaasli. State-of-the-art in heterogeneous computing. *Journal of Scientific Programming*, 18(1):1–33, 2010.

[6] A. Brodtkorb, T. R. Hagen, K.-A. Lie, and J. R. Natvig. Simulation and visualization of the Saint-Venant system using GPUs. *Computing and Visualization in Science*, 2010. [forthcoming].

[7] A. Brodtkorb, M. Sætra, and M. Altinakar. Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. [preprint].

[8] M. de la Asunción, J. Mantas, and M. Castro. Simulation of one-layer shallow water systems on multicore and CUDA architectures. *The Journal of Supercomputing*, pages 1–9, 2010. [published online].

[9] M. de la Asunción, J. Mantas, and M. Castro. Programming CUDA-based GPUs to simulate two-layer shallow water flows. In *Euro-Par 2010 - Parallel Processing*, volume 6272 of *Lecture Notes in Computer Science*, pages 353–364, Berlin/Heidelberg, 2010. Springer Verlag.

[10] C. Ding and Y. He. A ghost cell expansion method for reducing communications in solving PDE problems. In *ACM/IEEE conference on Supercomputing*, pages 50–50, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[11] T. Hagen, M. Henriksen, J. Hjelmervik, and K.-A. Lie. How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational

engine. In G. Hasle, K.-A. Lie, and E. Quak, editors, *Geometrical Modeling, Numerical Simulation, and Optimization: Industrial Mathematics at SINTEF*, pages 211–264. Springer Verlag, 2007.

[12] T. Hagen, J. Hjelmervik, K.-A. Lie, J. Natvig, and M. Henriksen. Visual simulation of shallow-water waves. *Simulation Modelling Practice and Theory*, 13(8):716–726, 2005.

[13] T. Hagen, K.-A. Lie, and J. Natvig. Solving the Euler equations on graphics processing units. In *Proceedings of the 6th International Conference on Computational Science*, volume 3994 of *Lecture Notes in Computer Science*, pages 220–227, Berlin/Heidelberg, 2006. Springer Verlag.

[14] A. Klöckner, T. Warburton, J. Bridge, and J. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics*, 228(21):7863–7882, 2009.

[15] D. Komatitsch, D. Göddeke, G. Erlebacher, and D. Michéa. Modeling the propagation of elastic waves using spectral elements on a cluster of 192 GPUs. *Computer Science - Research and Development*, 25:75–82, 2010.

[16] A. Kurganov and G. Petrova. A second-order well-balanced positivity preserving central-upwind scheme for the Saint-Venant system. *Communications in Mathematical Sciences*, 5:133–160, 2007.

[17] M. Lastra, J. Mantas, C. Ureña, M. Castro, and J. García-Rodríguez. Simulation of shallow-water systems using graphics processing units. *Mathematics and Computers in Simulation*, 80(3):598–618, 2009.

[18] W.-Y. Liang, T.-J. Hsieh, M. Satria, Y.-L. Chang, J.-P. Fang, C.-C. Chen, and C.-C. Han. A GPU-based simulation of tsunami propagation and inundation. In *Algorithms and Architectures for Parallel Processing*, volume 5574 of *Lecture Notes in Computer Science*, pages 593–603, Berlin/Heidelberg, 2009. Springer Verlag.

[19] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top 500 supercomputer sites. `http://www.top500.org/`, November 2010.

[20] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84, New York, NY, USA, 2009. ACM.

[21] NVIDIA. NVIDIA CUDA reference manual 3.1, 2010.

[22] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[23] B. Palmer and J. Nieplocha. Efficient algorithms for ghost cell updates on two classes of MPP architectures. In S. Akl and T. Gonzalez, editors, *Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 192–197, Cambridge, MA, November 2002. IASTED, ACTA Press.

[24] D. Playne and K. Hawick. Asynchronous communication schemes for finite difference methods on multiple GPUs. *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 763–768, May 2010.

[25] S. Rostrup and H. De Sterck. Parallel hyperbolic PDE simulation on clusters: Cell versus GPU. *Computer Physics Communications*, 181(12):2164–2179, 2010.

[26] C.-W. Shu. Total-variation-diminishing time discretizations. *SIAM Journal of Scientific and Statistical Computing*, 9(6):1073–1084, 1988.

[27] P. Wang, T. Abel, and R. Kaehler. Adaptive mesh fluid simulations on GPU. *New Astronomy*, 15(7):581–589, 2010.

PAPER VI

EFFICIENT GPU-IMPLEMENTATION OF ADAPTIVE MESH REFINEMENT FOR THE SHALLOW-WATER EQUATIONS

M. L. Sætra, A. R. Brodtkorb and K.-A. Lie

Journal of Scientific Computing, submitted for 2nd review, 2014

Abstract: The shallow-water equations model hydrostatic flow below a free surface for cases in which the ratio between the vertical and horizontal length scales is small and are used to describe waves in lakes, rivers, oceans, and the atmosphere. The equations admit discontinuous solutions, and numerical solutions are typically computed using high-resolution schemes. For many practical problems, there is a need to increase the grid resolution locally to capture complicated structures or steep gradients in the solution. An efficient method to this end is adaptive mesh refinement (AMR), which recursively refines the grid in parts of the domain and adaptively updates the refinement as the simulation progresses. Several authors have demonstrated that the explicit stencil computations of high-resolution schemes map particularly well to many-core architectures seen in hardware accelerators such as graphics processing units (GPUs). Herein, we present the first full GPU-implementation of a block-based AMR method for the second-order Kurganov–Petrova central scheme. We discuss implementation details, potential pitfalls, and key insights, and present a series of performance and accuracy tests. Although it is only presented for a particular case herein, we believe our approach to GPU-implementation of AMR is transferable to other hyperbolic conservation laws, numerical schemes, and architectures similar to the GPU.

1 Introduction

The shallow-water equations are able to accurately capture the required physics of many naturally occurring phenomena such as dam breaks, tsunamis, river floods, storm surges, and tidal waves. Most of these phenomena typically have some parts of the domain that are more interesting than others. In the case of a tsunami hitting the coast, one is primarily interested in obtaining a high-resolution solution along the coastline where the tsunami hits, whilst a coarser grid may be sufficient to describe the long-range wave propagation out at sea. Similarly, for dam breaks the most interesting part of the domain is downstream from the failed dam, where one wants correct arrival times of the initial wave front and reliable estimates of maximum water height during flooding.

Adaptive mesh refinement (AMR) [3, 2] is a standard technique that was developed to address this particular problem. The basic idea behind AMR is to recursively refine the parts of the domain that require high resolution, and adaptively update the refinement as the simulation progresses. By utilizing AMR and refining only the areas of interest, the required accuracy can be achieved locally at a considerably lower cost than by increasing the resolution of the full domain. To further accelerate the simulation, we propose to move the hierarchical computation of the AMR method to a modern graphics processing (GPU) architecture, which has proved to be particularly efficient for performing the type of stencil computations that are used in high-resolution shallow-water simulators. Herein, our starting point will be a second-order, semi-discrete, non-oscillatory, central-difference scheme that is well-balanced, positivity preserving, and handles wet-dry interfaces and discontinuous bottom topography [18], which we previously have shown can be efficiently implemented on GPUs [8].

Over the last decade or so, GPUs have evolved from being purely graphics co-processors into general-purpose many-core computing engines. Today, a GPU can significantly speed up a wide range of applications in a multitude of different scientific areas, ranging from simulations of protein folding to the formation of black holes [30, 4, 29]. Accelerators like the GPU have been increasingly utilized in supercomputers and adopted by the high-performance computing community as well. If one considers the Top 500 list [24], the first accelerated systems appeared in 2008, and today over 10% use accelerator technology [8]. Compared to the CPU, GPUs generally have much smaller caches and far less hardware logic, and focus most hardware resources on floating point units. This enables execution of thousands to millions of parallel threads, and "scratchpad-type" memory shared between clusters of threads enables fast collaboration. While the CPU is optimized for latency of individual tasks, the GPU is optimized for throughput of many similar tasks. Alongside the development of the GPU hardware, the programming environment has been steadily growing and improving as well. Although GPU development still is somewhat cumbersome and time-consuming, it has been greatly simplified by more expressive high-level languages, tools such as debuggers and profilers, and a growing development and research community utilizing GPUs [6].

Several software packages implement AMR for different problems on the CPU. Some of the most common, free, block-based codes are PARAMESH [23], SAMRAI [15], BoxLib [19], Chombo [10], and AMRClaw [1]. In particular, LeVeque et al. [20] describe in detail the

[8]On the June 2013 list, there were 43 GPU-powered machines, and 12 machines using the Intel Xeon Phi co-processor.

implementation of AMR in the GeoClaw software package to capture transoceanic tsunamis modeled using the shallow-water equations. There are also a few papers that discuss how GPUs can be used to accelerate AMR algorithms [37, 33, 26, 9]. To the best of our knowledge, Wang et al. [37] were the first to map an AMR solver to the GPU based on the Enzo hydrodynamics code [35] in combination with a block-structured AMR. Here, a single Cartesian patch in the grid hierarchy was the unit that is sent to the GPU for computing. Schive et al. [33] present a GPU-accelerated code named "GAMER", which is also an astrophysics simulator implemented in CUDA. Here, the AMR implementation is based on an oct-tree hierarchy of grid patches, in which each patch consists of eight by eight cells. The patches are copied to the GPU for solving, and the results are copied back to the CPU again. However, by using asynchronous memory copies and CUDA streams to solve patches at the same refinement level in parallel, they alleviate some of the overhead connected with the data transfer between the CPU and the GPU. Burstedde et al. [9] discuss a hybrid CPU–GPU version of their elastic wave propagation code, dGea, in which the wave propagation solver runs on the GPU and the AMR operations are executed on the CPU. CLAMR [26] is developed as a testbed for hybrid CPU-GPU codes using MPI and OpenCL [17] for shallow-water simulations.

Existing AMR codes for the GPU tend to handle most of the AMR algorithm on the CPU and only perform the stencil computations on a single or a group of Cartesian subgrids on the GPU. This involves uploading and downloading large amounts of data, when it would be much more efficient to keep the data on the GPU at all times. Herein, we therefore propose to take the development one step further and move all computationally expensive parts of a block-based AMR algorithm to the GPU [9]. Our work is, to the best of our knowledge, the first block-based AMR algorithm that has been *fully implemented* on the GPU, so that all simulation data are kept in GPU memory at all times. Our work is also the first to extend the second-order accurate Kurganov–Petrova scheme [18] to an AMR framework. Although the discussion herein will focus on a specific high-resolution shallow-water solver, most of the choices made in our implementation should be easily transferable to other numerical schemes, other modern accelerators (such as the Intel Xeon Phi), and even other systems of hyperbolic conservation laws.

2 Shallow-Water Simulations on the GPU

We have developed our AMR code based on an existing GPU-accelerated shallow-water simulator [5, 8] that has been thoroughly tested, verified, and validated both on synthetic test cases and against measured data. The simulator is written in C++ and CUDA [27] and has a clean and simple API, which makes it possible to set up and execute a simulation using about 5–10 lines of code. A brief overview of this simulator and its mathematical model will be given in this section. For a complete description, we refer the reader to [8, 32, 31]. A set of best practices for harnessing the power of GPUs for this type of simulation can be found in [7].

Mathematical Model

The shallow-water equations are derived by depth-averaging the Navier–Stokes equations. By adding a bed shear-stress friction term to the standard shallow-water equations, we get the model

[9]It should be noted that some parts of the code, such as launching kernels on the GPU, is necessarily performed on the CPU.

used in our simulator. In two dimensions on differential form it can be written as:

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix} + \begin{bmatrix} 0 \\ -gu\sqrt{u^2+v^2}/C_z^2 \\ -gv\sqrt{u^2+v^2}/C_z^2 \end{bmatrix}. \tag{1}$$

Here, h is the water depth and hu and hv are the discharges along the abscissa and ordinate, respectively. Furthermore, g is the gravitational constant, B is the bottom topography measured from a given datum, and C_z is the Chézy friction coefficient.

Our numerical scheme is based on the semi-discrete, second-order, central scheme by Kurganov and Petrova [18], which has been extended to include physical friction terms [8]. The spatial discretization of the scheme is well-balanced, positivity preserving, and handles wet-dry interfaces and discontinuous bottom topography. In semi-discrete form it can be written as:

$$\begin{aligned} \frac{dQ_{ij}}{dt} &= H_f(Q_{ij}) + H_B(Q_{ij}, \nabla B) - \big[F(Q_{i+1/2,j}) - F(Q_{i-1/2,j})\big] - \big[G(Q_{i,j+1/2}) - G(Q_{i,j-1/2})\big] \\ &= H_f(Q_{ij}) + R(Q)_{ij}. \end{aligned} \tag{2}$$

Here, Q_{ij} is the vector of conserved variables averaged over the grid cell centered at $(i\Delta x, j\Delta y)$, H_B is the bed slope source term, H_f is the bed-shear stress source term, and F and G represent numerical fluxes along the abscissa and ordinate, respectively. The fluxes are calculated explicitly, based on one-sided point-values $Q_{i\pm1/2,j}$ and $Q_{i,j\pm1/2}$ evaluated from a piecewise-linear reconstruction from the cell averages with slopes limited by the nonlinear, generalized minmod function [36, 22, 25, 34]. The bed-slope source term H_B is also calculated explicitly, and the discretization is designed carefully to ensure that the numerical fluxes exactly balance the bed-slope source term for a lake at rest. Finally, to avoid numerical problems with dry states, the scheme uses a mass-conservation equation formulated using water elevation rather than water depth.

To evolve the solution in time, one can choose between the simple first-order, forward Euler method and a second-order, total-variation-diminishing Runge–Kutta method. The friction source term, H_f, is discretized semi-implicitly, which gives rise to the following numerical scheme for forward Euler time integration

$$Q_{ij}^{k+1} = \left(Q_{ij}^k + \Delta t R(Q^k)_{ij}\right) / \left(1 + \Delta t \alpha\right), \tag{3}$$

in which Δt is the time-step size and α is the semi-implicit friction source term. The two steps in the Runge–Kutta method are on the same form. For a detailed derivation of the numerical scheme, we refer the reader to [18, 8].

Shallow-Water Simulations on Cartesian Grids

The execution model of the GPU is perfectly suited for working with structured grids since the GPUs have been designed mainly to calculate the color values of regularly spaced pixels covering the computer screen. Conceptually, we “replace” the screen with a computational domain and the colored points by cells (see Figure 1). By structuring the computations so that every cell can be solved independently, we can solve for all cells in parallel.

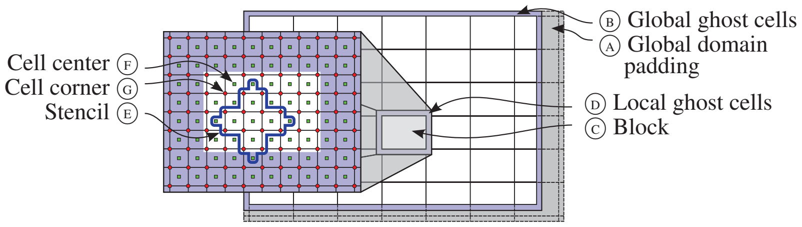


Figure 1: Domain decomposition and variable locations. The global domain is padded Ⓐ to fit an integer number of CUDA blocks, and global ghost cells Ⓑ are used for boundary conditions. Each block Ⓒ has local ghost cells Ⓓ that overlap with other blocks to satisfy the data dependencies dictated by the stencil Ⓔ. Our data variables Q, R, H_B, and H_f are given at grid cell centers Ⓕ, and B is given at grid cell corners Ⓖ.

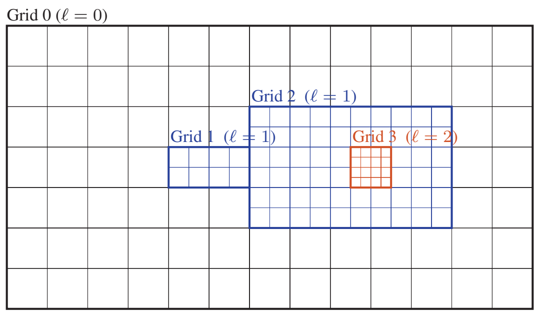
The shallow-water simulator uses four *CUDA kernels* to evolve the solution one time step:

```
1:  while $t < T$ do
2:      for k=1:order do
3:          Flux: reconstruct piecewise continuous solution, compute $F$, $G$, and $H_B(Q_{ij}, \nabla B)$,
4:                and compute upper bound on wave speeds
5:          if k==1 then
6:              Max time step: use parallel reduction to find global limit on time step $\Delta t$
7:          end if
8:          Time integration: evolve solution forward in time
9:          Boundary condition: update all global boundary conditions
10:     end for
11:     $t = t + \Delta t$
12: end while
```

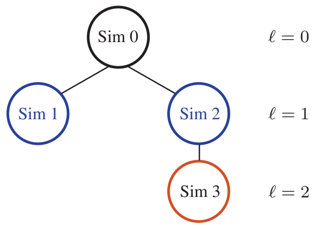
The flux kernel is also responsible for finding an upper bound on the maximum speed of propagation per CUDA *block*, which will be used to limit the time step according to the CFL condition. After the flux kernel has determined an upper bound, the max time-step kernel finds the global limiting factor using a parallel reduction [13]. In the boundary-conditions kernel, each of the four global boundaries may use different boundary conditions. Executing all four CUDA kernels once constitutes one full time step if we use forward Euler for time integration. With second-order Runge–Kutta, all kernels are executed twice, except for the max time-step kernel, which needs only be executed in the first substep. To enable maximal utilization of the GPU accelerator, the simulation data are kept on the GPU at all times. Even when interactive visualization is switched on, the simulation data are simply copied from CUDA memory space to OpenGL memory space, never leaving GPU memory.

3 Adaptive Mesh Refinement

The simulator discussed in the following has two novel components: formulation of the Kurganov–Petrova scheme in an AMR framework, and efficient implementation of this framework on a GPU many-core system. In this section we will give algorithmic and implementation details, focusing mainly on challenges related to the many-core GPU implementation. The basic AMR



(a) Logical grid layout for AMR.



(b) Simulator hierarchy.

Figure 2: The AMR grid hierarchy with two levels of refinement and four grids in total, including the root grid. Each grid node is connected to a separate simulator instance that internally uses the domain decomposition shown in Figure 1.

data structure is a sequence of nested, logically rectangular meshes [2] on which balance law (1) is discretized as in (2). There are two main strategies for local refinement: A *cell-based* strategy will refine each cell based on a given refinement criteria, whereas a *block-based* strategy will group cells together and refine collections of cells. We have chosen a block-based strategy, which we will refer to as *tile-based* since we operate in two dimensions. The tile-based approach has several advantages over the cell-based AMR and is, in particular, a good fit for the GPU architecture with regards to cache locality and memory access patterns. It also fits nicely with using ghost cells to handle boundary conditions. To reduce the overhead of the regridding algorithm, one will typically want to extend the refinement area so that waves can propagate for a few time steps before they reach the boundaries of a newly added patch. In other words, padding of new refined areas is necessary, and this comes automatically when using tiles. An advantage of using cell-based refinement is a finer granularity, but this can also be achieved with tile-based refinement by adjusting the size of the tiles. The tiles could consist of as little as one cell, and the tile size should be adjusted according to the necessary degree of granularity and efficiency for each simulation scenario. It should also be noted that although the refinement is tile-based, criteria for refining are enforced on the cell-level.

To explain the AMR algorithm, we first need to introduce the grid hierarchy (see Figure 2) and establish some terms that will be used consistently for the rest of the paper. Each grid is linked to an instance of the simulator presented in Section 2. For a standard simulation on a single grid, we will have one simulator instance linked to one grid (see Figure 1). This is referred to as the *root grid* in an AMR context, and the root grid is the only grid that covers the full simulation domain. For a two-level AMR simulation, the root will also have a vector of *children*, each of them covering some part of the domain with twice the resolution of the root grid. For more than two levels this becomes recursive, so that all grids, except the root grid and the grids with the highest resolution (leaf node grids), have one parent grid, and one or more children.

In addition to the tree of simulation grids, there are two main components to the AMR algorithm: the time integration and the regridding. The time integration evolves the solution in time on the parent grid, sets the boundary cells, and initiates the time integration on all descendants

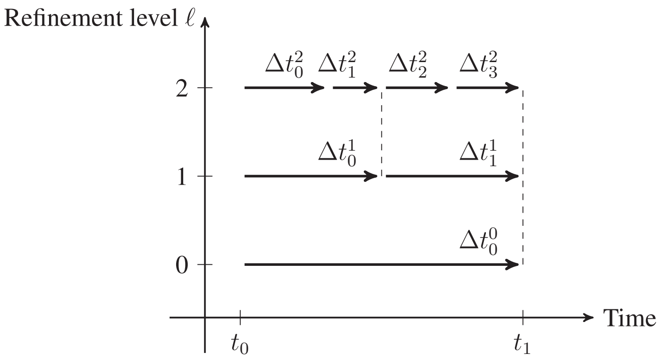


Figure 3: Different time-step sizes Δt_i^ℓ for grids at different refinement levels ℓ. Global time synchronization occurs after every root time step. This means that the simulation time at root level after time step Δt_0^0 will be identical to the simulation time at refinement level 1 after time step Δt_1^1, and simulation time on refinement level 2 after time step Δt_3^2, and so on. Notice that the time-step sizes in the same refinement level are slightly different due to the CFL condition.

of the current simulator instance. The regridding process, on the other hand, adaptively creates and destroys children recursively based on the chosen refinement criteria.

Time integration

The time-integration function evolves the solution on the current grid up to the current time t on the parent grid. In doing so, the last time step must often be reduced to reach t exactly (see Figure 3). No grid on refinement level ℓ can be integrated in time before all grids on levels $0, 1, ..., \ell - 1$ are at least one time step ahead of level ℓ. Except for this restriction, every grid may be integrated in parallel, independently of all other grids. The time-step size is computed independently on each child grid to ensure optimal computational efficiency for the update on each subgrid. That is, subgrids containing slowly propagating waves can be updated using larger time steps than subgrids on the same level that contain fast waves. The resulting AMR time integration can be outlined in five steps (starting with $\ell = 0$):

1. Take one time step of length Δt^ℓ on the grid(s) at level ℓ.

2. Determine ghost-cell values on level $\ell + 1$ grids: Values at time t are known from last time the grid was updated and values at time $t + \Delta t^\ell$ can be reconstructed from the most recent solution on the parent grid. (For spatial interpolation, we reconstruct values in the same way as for calculating the numerical fluxes.) Values at intermediate times $t + \Delta t_0^{\ell+1}, t + \Delta t_0^{\ell+1} + \Delta t_1^{\ell+1}, \ldots$ are computed using linear time interpolation between the known values at time t and $t + \Delta t^\ell$. See also Figure 3.

3. Perform time integration on all level $\ell + 1$ grids to bring these grids up to the current time on the level ℓ grid by running this time integration algorithm recursively.

4. For any grid cell at level ℓ that is covered by a level $\ell + 1$ grid, replace the solution in this cell by an average of the values from the cells in the level $\ell + 1$ grid that covers this cell.

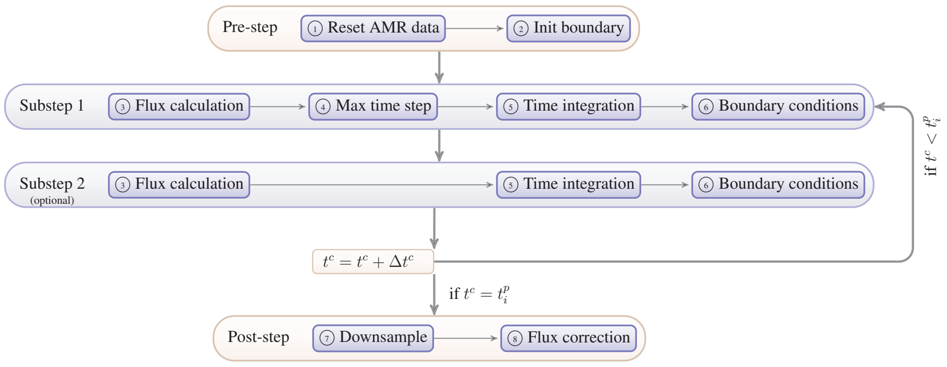


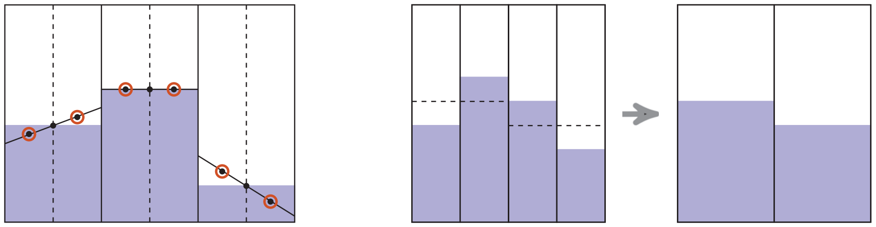
Figure 4: Conceptual overview of all CUDA kernels used to evolve the solution from time t_{i-1}^p to t_i^p. In the pre-step stage, function ① resets all AMR data for the local solve and sets the boundary values for the start of the current time-step series at time t_{i-1}^p to the boundary values used at the end of the previous time-step series. Kernel ② computes the boundary values to be used at the end of the current time-step series using the solution from the parent grid at time t_i^p. In the first substep, kernel ③ calculates F, G, and H_B, kernel ④ finds the maximum Δt^c, kernel ⑤ calculates α and Q^{k+1}, and kernel ⑥ enforces boundary conditions on Q^{k+1}. For all other than the root grid, the boundary values are a linear interpolation between the reconstructed solutions on the parent grid at times t_{i-1}^p and t_i^p. Last, kernel ⑦ replaces the parent solution by the child solution where they overlap, while kernel ⑧ maintains conservation of mass. As indicated in the figure, Substeps 1 and 2 are repeated until the solution on the current grid has been advanced to the same time level $t^c = t_i^p$ as the parent grid.

5. Adjust the values in cells that interface a different grid to maintain conservation of mass.

Figure 4 illustrates one simulation cycle for one grid using this algorithm. Here, function ① represents a reset of AMR data before a new cycle of time steps, while kernels ② to ⑧ represent one CUDA kernel each. We will go through each step, with emphasis on what is special for AMR. A thorough explanation of kernels ③–⑥ for non-AMR simulations can be found in [8].

Before a simulator instance can start updating its grid, the solution on all parent grids must be one time step ahead. That is, for all simulator instances that do not hold the root grid we can assume that there is a coarser solution available from the end-time t_i^p at the parent grid that can be used to interpolate boundary conditions on the child grid. To prepare for the next time steps on the child grid, the *reset AMR data* function (function ①) resets the accumulated fluxes across the child-parent interface and the accumulated time-step sizes to zero. The pointers to the two arrays containing initial and end-time boundary values are swapped so that the solution at the end of the last computed step on the parent grid (at time t_{i-1}^p) becomes the initial solution for the sequence of time steps on the current child grid. The array holding boundary data at time t_i^p is set in the *init-boundaries* kernel (kernel ⑥) by reconstructing the parent solution from time t_i^p and then evaluating the point values at the center points of the child boundary cells (see Figure 5a). For the root grid, the boundary data are obtained from the boundary conditions of the problem.

Once the data structures holding boundary data is properly set, we can evolve all children to time $t^c = t_i^p$: Kernels ③ to ⑥ in the two Runge–Kutta substeps are oblivious to the AMR grid



(a) Kernel ②: 1D minmod reconstruction.

(b) Kernel ⑦: Averaging from level ℓ to level $\ell - 1$ in 1D.

Figure 5: Visualization of the calculations done by the init-boundaries kernel (a) and the downsample kernel (b). The black dots are input, the encircled black dots are output, and the black lines are the reconstructed surface.

hierarchy and treat each grid as if it were a single-grid simulation, with only minor adaptations for AMR. Kernel ③, *flux-calculation*, uses a standard quadrature rule to compute the fluxes across all cell faces. In each quadrature point, a numerical flux function is used to determine the interface flux from one-sided point values reconstructed from cell averages in neighboring cells. The contributions from all quadrature points are accumulated and the result is multiplied with the correct weight in the Runge–Kutta integration[10]. This kernel also finds the limiting factor for the time-step size per CUDA block. Kernel ④, *max time step*, then reduces the per-block limiting factor to a global value, and computes the maximum time step based on the CFL-condition of the system. This time step is then added to the accumulated time t^c for the current child grid. Then, kernel ⑤, *time integration*, advances the solution in time and accumulates the fluxes going across the interface to the parent grid, multiplied with the time-step size. On the root grid, kernel ⑥, *boundary condition*, works as normal, executing the prescribed boundary condition for the domain. For all other grids, this kernel interpolates linearly between the two arrays containing boundary data prepared from the parent solution as described above, using the accumulated time as the input variable. If Euler time integration is used, the second substep is simply omitted, and in kernel ③ we do not multiply the fluxes with 0.5. The time-integration process is repeated until $t^c = t_i^p$ (see Figures 3 and 4). Now, the new solution must be communicated back to the parent grid. In kernel ⑦, *downsample*, the solution in the child grid is averaged down to the resolution of the parent grid (see Figure 5b) and used to replace the solution in the parent grid where this grid overlaps with the child grid.

The *flux-correction* kernels in ⑧ finally ensure conservation of mass by modifying the values of cells interfacing a different grid. For grids at the same refinement level, the flux correction computes the corrected flux across a shared interface using

$$F_{\mathrm{corr}}(Q_{i+1/2,j}) = \frac{1}{2\Delta t}\left(\sum \Delta t^L F(Q^L_{i+1/2,j}) + \sum \Delta t^R F(Q^R_{i+1/2,j})\right),$$
$$\Delta t = \sum \Delta t^L = \sum \Delta t^R.$$

[10]Here, we use a midpoint integration rule, central-upwind numerical flux, and a piecewise-linear reconstruction with slopes limited by a generalized minmod function, see [18] for details. The weight is 0.5 for both steps of the particular stability-preserving, second-order Runge–Kutta scheme used herein. If other schemes are used, this weight must be altered accordingly.

Here, variables with superscript L and R denote that the variable is taken from the left- and right-hand side of the interface, respectively. The sums represent the accumulated fluxes computed on each side of the interface, weighted with their respective time-step sizes, and the corrected flux is then used to adjust the variables in the adjacent cells. For an interface between a parent and a child grid, on the other hand, we assume that the flux computed on the child grid is the most correct. We therefore correct the flux only in the parent grid using

$$F_{\text{corr}}(Q_{i+1/2,j}) = \frac{1}{\Delta t} \sum \left(\Delta t^c F(Q^c_{i+1/2,j}) + \Delta t^c F(Q^c_{i+1/2,j+1}) \right),$$

in which superscript c denotes the child grid. The sums represent the accumulated fluxes for the two cells in the child grid that interface with a single cell in the parent grid. (Note that we have to take into account the difference in grid cell size between the child and parent grid when computing the corrected flux.)

Regridding

The regridding process is performed after a given number of time steps on every grid from the root grid to a prescribed refinement level. The time intervals and grid depth should be set according to the problem type (dam break, river flood, tsunami, etc.) and refinement criteria. We do not perform refinement on a cell-per-cell basis, but rather we consider tiles of 32×32 cells. Tile size may be adjusted to better suit the problem type and domain size. The process starts by running the *refinement-check* kernel that checks all cells in each tile against the given refinement criteria. After a shared-memory reduction, using 32×4 CUDA threads per tile, the kernel outputs the number of cells that fulfilled the refinement criteria per tile to the *refinement map*. This makes it possible to demand that a certain fraction of the cells must fulfill the refinement criteria before a tile is refined. Setting this number low gives better accuracy, but lower efficiency, and vice versa. All new child grids have twice the resolution of the parent grid.

Determining the coordinates of new child grids is the only part of the algorithm performed on the CPU. First, we download the refinement map, which contains one value per tile. Using the vector of existing child grids, we mask out every tile already overlaid with a child grid in the refinement map to avoid generating new grids that include already refined tiles. Nevertheless, we may end up with overlapping grids in some cases, and this needs to be handled after we have made the *proposed bounding boxes* for new child grids. The new bounding boxes are checked against existing bounding boxes, and any overlap is categorized as one of nine different configurations. The first eight are handled as shown in Figure 6, and the ninth corresponds to complete overlap. It should be noted that this step constitutes a negligible portion of the runtime, and though it is possible to implement on the GPU, it is better suited to the serial execution model of the CPU. The refinement map is also used to remove child grids that are no longer required. Since the physical quantities have already been averaged onto the parent grid by the downsample kernel, the child grid can simply be removed and the simulator instance deactivated without sideeffects. It should also be straight forward, though not yet implemented, to include the double-tolerance adaptive strategy proposed recently [21], which aims to optimize the overall numerical efficiency by reducing the number of child grids whilst preserving the quality of the numerical solution.

To initialize a new child grid, we copy and edit the initialization parameters of the parent

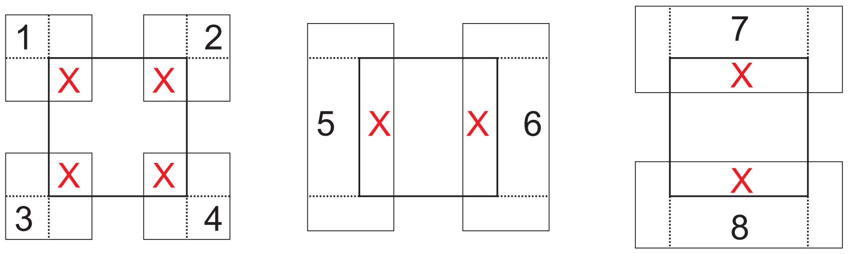


Figure 6: Showing eight different overlapping configurations and how they are treated. For all configurations, the middle bounding box is the existing one. The dotted lines show how new proposed bounding boxes are repartitioned and the overlapping part, marked with 'X', is discarded.

grid; grid size (Δx, Δy), current simulation time, and all four boundary conditions are set to their correct values. All initial conditions (i.e., the physical variables and bathymetry) remain unset, so no data is uploaded to the GPU when the new child-grid simulator instance is constructed. Each new simulator instance also creates its own CUDA stream to enable concurrent execution of grids. The bathymetry can be initialized using texture memory, enabling the use of efficient hardware interpolation. The initial conditions are reconstructed from previous cell-averaged values using the generalized minmod limiter, and the array with initial boundary data is filled by running the init-boundary kernel described earlier[11]. In the case of complete overlap of existing subgrids, the solutions from the existing subgrids are simply copied into the new child grid. Last, we update the neighborhood information for the new simulator instance with any abutting grids at the same refinement level and add it to the vector of children.

To have proper nesting, Berger and Colella [2] require that a child grid starts and ends at the corner of a cell in the parent grid, and that there must be at least one level $\ell - 1$ cell in some level $\ell - 1$ grid separating a grid cell at level ℓ from a cell at level $\ell - 2$ in the north, east, south, and west directions, unless the cell abuts the physical boundary of the domain. These minimum requirements also apply in our GPU code. In addition, we require that no interface between a parent and a child grid, or two child grids at the same refinement level, crosses each other, and on the root grid ($\ell = 0$), we require three cells between the global domain boundary and all level $\ell > 0$ grids. Without a three-cell margin, one of the steps (parent-child flux correction) of the AMR algorithm interferes with the global wall boundary condition. This last requirement is not imposed by the numerical scheme, but is an assumption introduced to simplify the current implementation of global boundary conditions that should be removed when the simulator is moved beyond its current prototype stage.

Optimizations

It is well known that a key challenge of executing an AMR simulation is to ensure proper load balancing when work is distributed unevenly among processors. In our work, we have not addressed scheduling, as this is automatically handled by the underlying CUDA runtime and

[11]Only the array containing boundary values for the end of the time-step series is filled in the initialization. The boundary values for the start of the time-step series is simply set to be the end-values from the previous time-step series.

driver. However, in the next section we report the result of several numerical experiments we have conducted to assess the overhead of our approach. In the rest of the section, we instead focus on other types of optimizations that can improve the computational efficiency.

As we have seen, each grid except the root grid depends on its parent to supply the necessary boundary conditions. If we consider Figure 2, this means each grid only depends on its parent grid to complete one time step before its solution can be evolved up to the same time as its parent. Furthermore, this means that we may run simulations on all grids simultaneously, as long as this one dependency if fulfilled. This task parallelism is handled by using *streams* and *events* in CUDA. Each simulator instance has its own stream, in which kernels and memory transfers are issued and executed in order. The synchronization with other simulators is done by issuing events. This synchronization is implemented in the main AMR step function:

```
1: while $getCurrentTime() < parent\_time$ do
2:     regrid();
3:     step(parent_time);
4:     cudaEventRecord(main_step_complete, stream);
5:     stepChildGrids();
6: end while
```

assuming that '*parent_time*' is passed as an argument from the parent grid. In the code above, `regrid()` checks the current grid hierarchy, and performs regridding based on the current simulation settings of refinement criteria, regridding frequency, etc. After the `step(...)`-function the grid associated with the current simulator instance is advanced in time, and its child grids can then be treated next. If the current grid is the root grid, we only perform one time step. For all other grids than the root grid, we do time stepping until the grid has reached the same advanced time as its parent grid. In the last case, the child grids of the current grid need to be updated between each time step. The integration of child grids also requires similar synchronization:

```
1: function STEPCHILDGRIDS
2:     for $1 \to number\_of\_child\_grids$ do
3:         cudaStreamWaitEvent(child_grids[i]->stream, main_step_complete);
4:         ...
5:         // calling the main AMR step function
6:         child_grids[i]->step(getCurrentTime());
7:         ...
8:         cudaEventRecord(child_grids[i]->child_steps_complete, child_grids[i]->stream);
9:         cudaStreamWaitEvent(stream, child_grids[i]->child_steps_complete);
10:        ...
11:    end for
12: end function
```

Moreover, it is necessary with additional synchronization within each simulation as described above.

To avoid unnecessary computations it is possible to exit early in dry cells, since the solution will remain constant throughout the whole time step unless the cell is neighbor to a wet cell [8]. Likewise, one can reduce the memory footprint if data values are not stored before they actually contribute in the simulation using a sparse-domain algorithm [31]. These code optimizations have not been included in the AMR implementation, and hence all performance results report the time it takes to simulate every cell in the domain. Some minor adaptations have been made to the simulator described in [8], the most noteworthy being the switch from saving the sum of net fluxes and source term as a vector $R(Q)_{ij}$ (see (2)), to saving them separately and postponing the computation of $R(Q)_{ij}$ to the time integration kernel.

Because each child grid has twice the resolution of its parent, one should in principle use two time steps on the child grid for each time step on the parent grid. However, since the maximum time step allowed by the CFL condition is computed for each time step, and estimates of maximum eigenvalues (local wave-propagation speed) tend to increase with increasing resolution, the allowed step sizes tend to decrease with increasing refinement level. Because we need to synchronize the time between grids at different refinement levels (see Figure 3), we limit the size of the last time step so that all grids at level $\ell + 1$ will exactly hit the current time of level ℓ after some number of time steps. Sometimes this leads to very small last time steps. This is unfortunate since very small time steps cost exactly the same as larger time steps, without significantly advancing the solution in time. By reducing the CFL target a few percent below its maximum allowed value, we are able to avoid many of these small time steps, and thus increase the efficiency of the overall scheme (see Results 4).

A feature called dynamic parallelism has been introduced in the most recent versions of CUDA GPUs. Dynamic parallelism enables a kernel running on the GPU to launch further kernels on the GPU without any CPU involvement, thereby improving performance. One proposed use of dynamic parallelism has been AMR [16], as it enables the GPU to adaptively refine the simulation domain. In cell-based AMR codes, the construction and deletion of grids is highly dynamic, and will therefore benefit greatly from dynamic parallelism. However, our simulator is tile-based, and the overhead connected with the regridding process is already only a small fraction of the run time. The impact of using dynamic parallelism will therefore be negligible, and restrict the simulator to only execute on the most up-to-date CUDA GPUs.

4 Results

How to best report the performance and accuracy of a tile-based AMR code is not self-evident. There are many parameters to consider, e.g., refinement criteria, tile size, minimum child grid size, regridding frequency, and maximum levels of refinement. We have constructed eight tests and examples, and the results will be presented in separate sections below.

Our results show that the time integration consumes the majority of the computational time and the time spent on refinement checks and regridding is negligible. The time integration is made up of four kernels, some of them very computationally intensive. The refinement check, on the other hand, is a single, relatively simple kernel and is generally not performed every time step, typically only every 50th or 100th time step. Likewise, the regridding procedure is typically performed on a very small fraction of the global domain. This means that whereas the choice of refinement and regridding parameters will significantly impact the accuracy of the simulation, the hardware utilization of the AMR algorithm will not depend on whether patches are introduced adaptively or statically. Hence, all tests, except those noted, have been run on a statically refined grid with a tile-size of 32×32 cells (measured in the parent grid).

We have run the simulator with both first-order forward Euler time integration and second-

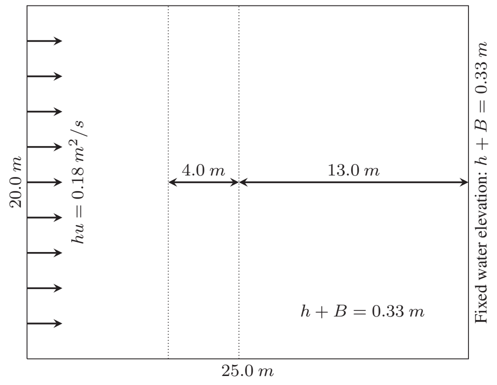


Figure 7: The SWASHES test setup. The dotted lines indicate the bump in the bathymetry.

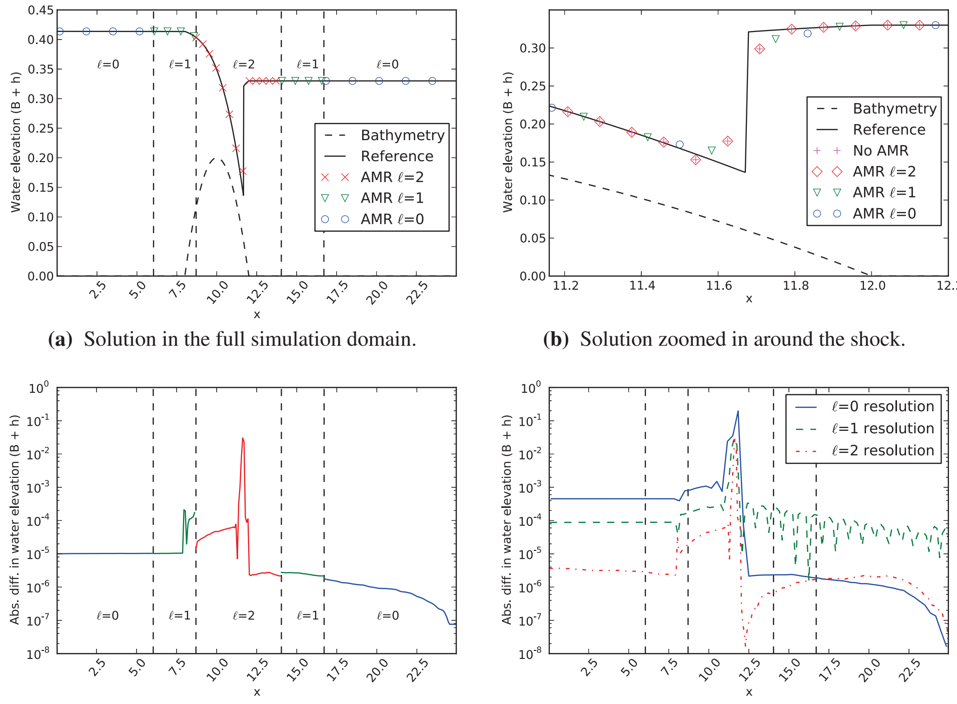
order Runge–Kutta integration. The Runge–Kutta time integration will give overall better hardware utilization and hence improve results on the efficiency tests as we get a higher compute-to-"AMR-overhead" ratio. Friction terms are neglected for simplicity, unless stated otherwise in the description. The performance tests were run on a node with an Intel i7 3930k CPU @ 3.2 GHz, 32 GB RAM, 64-bit Windows 8, and a GeForce 780 GTX graphics card using CUDA 5.5. All other tests were run on a node with an Intel i7 2657M CPU @ 1.6 GHz, 8 GB RAM, 64-bit Linux, and a GeForce GT 540M graphics card using CUDA 5. All simulations are run using single-precision arithmetic.

Verification

In this test, we will use an analytic solution to verify the accuracy of the original simulator and its new AMR version with different levels of refinement. To this end, we will use the SWASHES code [11] to compute a steady-state reference solution for a transcritical flow with a shock over a bathymetry with a single bump [12]. The domain, as depicted in Figure 7, is 25 m $\times$ 20 m with a bathymetry given by:

$$B(x) = \begin{cases} 0.2 - 0.05(x - 10)^2, & \text{if } 8 \text{ m} < x < 12 \text{ m}, \\ 0, & \text{else.} \end{cases} \tag{4}$$

For the AMR code, a new bathymetry is generated for each refinement level to avoid introducing errors due to interpolation or extrapolation. Water elevation is initially set to 0.33 m. Wall boundary conditions are imposed at $y = 0$ m and $y = 20$ m. At $x = 0$ m we have an inflow boundary with a fixed discharge of 0.18 m^2/s in the positive x-direction and at $x = 25$ m we have an outflow boundary with a fixed water elevation at 0.33 m. All simulations are run using first-order Euler time integration until they reach steady state. The AMR algorithm will generate small fluxes in the y-direction across interfaces between grids at different refinement levels, and hence longer simulation times are required before the steady state is reached. Since the SWASHES reference solution is in 1D, we extract an 1D solution from our 2D solution, which is simply a line going through the middle of the domain in the x-direction. From the results, shown in Figure 8, we see that the AMR simulator captures the shock with increasing



(a) Solution in the full simulation domain.

(b) Solution zoomed in around the shock.

(c) Difference between reference and AMR simulation.

(d) Difference between reference and single-grid simulations.

Figure 8: Single-grid simulation and simulations with AMR using up-to two levels of refinement. Every fifth grid point is marked in the full simulation domain solution. The analytic reference solution has been generated using the SWASHES library [11]. Notice that the no-AMR and AMR results match up closely, with an error between 10^{-2} and 10^{-6} in the AMR results. The largest discrepancies are located at the shock itself, and at the bump covered by the level one AMR grid (which has half the resolution of the no-AMR simulation).

accuracy for each level of refinement. Furthermore, the AMR solution, in which we only refine a small area around the shock, is almost identical to the global refinement solution of matching resolution.

Using the same setup, we now compare the computational time for the original simulator and the AMR simulator, with resolution set so that the child grids with highest resolution in the AMR simulation runs have the same resolution as the grid in the no-AMR simulation runs, in which 300×240 cells are used. The level-1 child grid is offset by three cells from the $x = 0$ m boundary, and the level-2 child grid is offset two cells from the boundary of the first child grid. To make the comparison fair, we fit as many tiles as possible into each of the two levels of child grids in the y-direction. In the x-direction, the grid with the highest resolution is just wide enough to capture the bump and the shock, and its parent grid is just wide enough to contain it. All test runs reach steady state. Results are shown in Figure 9. The AMR simulations are clearly faster than the original simulator, and the discrepancy is increasing. This is caused by

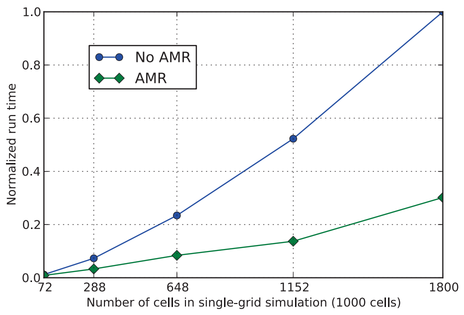


Figure 9: Comparison of wall-clock time for simulation runs with and without AMR. All test runs have been normalized with regards to the single slowest run. The AMR code is over three times faster for larger domain sizes.

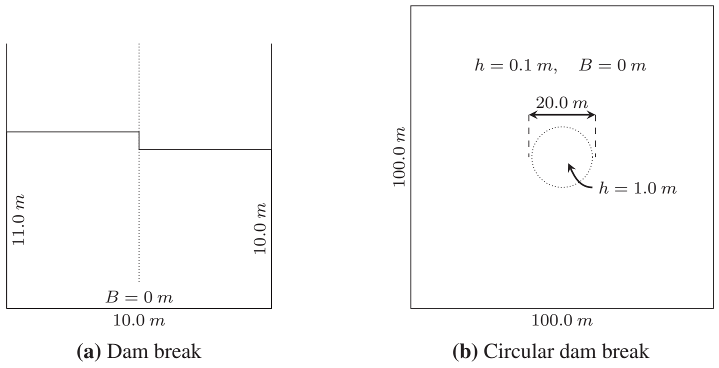


Figure 10: Dam break setups used in the tests: The left figure shows a cross-section of the domain (which is 80 m along the intersected dimension), and the right figure shows a top-view of the domain. The dotted lines indicate the location of the dams that are instantaneously removed at simulation start.

the increasing ratio of cells outside the area of interest (the shock) over the total number of grid cells.

Considering both accuracy and efficiency, we have shown that the AMR-simulation gives the same results for less computation, and several times faster (3.4 times for the highest resolution). In this test, the domain size has been constant at 20 m $\times$ 25 m, and the resolution has been variable. For cases where the area of interest is of fixed size, but the rest of the domain is increased, e.g., if the area of interest is far away from the event origin, the use of AMR will have an even bigger impact on simulation efficiency.

Child grid overhead

Maintaining several simulator instances to represent different subgrid patches may represent a significant computational overhead that may potentially slow down the simulation. In this test, we will therefore measure the added overhead in time stepping connected with an increasing

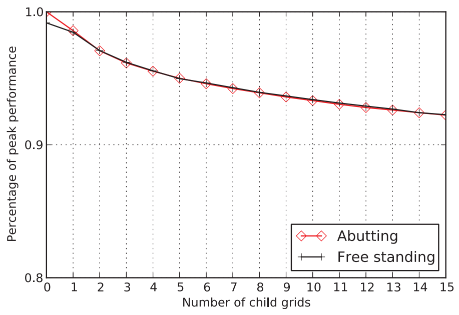


Figure 11: Total time stepping overhead connected with an increasing number of added child grids. For the abutting test runs, each new child grid is added next to the last added child grid so that the two grids interface each other along one full edge. All test runs have been normalized with regards to the single fastest run.

number of child grids. More specifically, we measure the performance in terms of cell updates per second, increase the number of child grids for each test run, and normalize with regards to the single fastest run. To this end, we consider a rectangular domain represented on a root grid with 4096×512 cells. Initial water elevation is set to 11 m in the left half of the domain and 10 m in the right half (see Figure 10a). Wall boundary conditions are used and all tests are run for two minutes of wall-clock time using first-order Euler time integration. The first series of test runs is performed with abutting child grids, centered in the domain, consisting of 256×256 cells (measured in the root grid) laid out so that each new child grid interfaces with the previously added child grid along one full edge. The second set of test runs is performed with a two-cell margin in the root grid between child grids. Results are shown in Figure 11. The difference between the two setups is negligible, showing that the flux correction between neighboring children is efficient. For a total of 15 subgrids, the performance drops to roughly 93% of peak. Kepler-generation GPUs [28] from NVIDIA contain new features for more efficient execution of concurrent kernels, which should decrease this overhead even more.

This test clearly shows that a strategy for minimizing the number of child grids on each level of refinement is important. The current prototype uses a relatively simple strategy for merging neighboring subgrid patches. However, since this is the only operation that is performed on the CPU and it is well separated from the rest of the simulator code, one can easily substitute it by more advanced methods that can be found in generic grid generators. Note that it is the total overhead during time stepping (refinement check is not performed) that is measured in this test. Child grid initialization cost is addressed in the next section.

Child grid initialization cost

While the previous test measured the total overhead connected with child grids, this test measures only the initialization cost. We use a global domain of 100 m $\times$ 100 m and a root grid of 1024×1024 cells. Initial conditions are a circular dam break in the center of the domain with a radius of 10 m, in which the water elevation is set to 1.0 m, and 0.1 m in the rest of the domain (see Figure 10b). Wall boundary conditions are used and we simulate the wave propagation of the dam break using first-order Euler time integration. The wave does not reach the domain

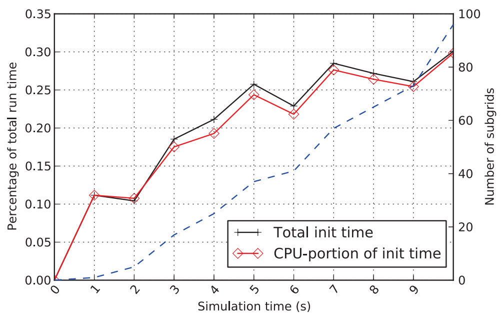


Figure 12: The cost of child grid initialization given in percentage of total run time. This is connected with the total number of added subgrids, shown as the dashed line.

boundaries, thus new child grids are added throughout the simulation run. Refinement check is performed every 50th time step, the minimum child grid size is set to three tiles, and all cells in which $|u| + |v| > 1.0 \times 10^{-4}$ are flagged for refinement. Results are shown in Figure 12.

It is clear from the results that the subgrid initialization cost is low. After close to 100 subgrids have been added, the accumulated subgrid initialization time still constitutes only 0.3% of the total run time. Furthermore, we see that the refinement check kernel is fast, and that the inherently serial overlap testing performed by the CPU constitutes the bulk time usage. The accumulated subgrid initialization time is steadily increasing due to the initial conditions of the particular problem we are simulating; as the radius of the circular wave is increasing, the number of new added subgrids per refinement check is also increasing.

Efficiency

In this test, we compare the efficiency of the original simulator with different simulations with the AMR simulator using a single child grid covering a varying area of the global 100 m $\times$ 100 m domain. The global domain is the same as used in the previous test (see Figure 10b). Initial conditions are a circular dam break in the center of the domain with a radius of 10 m, in which the water elevation is set to 1.0 m, and 0.1 m in the rest of the domain. Wall boundary conditions are used and we simulate the wave propagation of the dam break up to time ten seconds using first-order Euler time integration. Results are shown in Figure 13. As expected, the efficiency of the hardware utilization increases as an increasing percentage of the domain is covered by the child grid. Likewise, increasing the number of cells in the root grid improves the performance and eventually leads to full utilization of the computing potential of the GPU. From this test we conclude that the overhead associated with a single child grid is small (max 5% for one million cells or more) and diminishing as the child grid covers an increasing portion of the root grid and as the number of total cells increases.

For the first of the 39%-coverage runs we have compared the total mass of the initial conditions with the total mass after 10 seconds (389 simulation steps on the root grid) of simulation. The relative change in total mass is less than 10^{-6}. For more details on the mass conservation properties of the original numerical scheme, see [5].

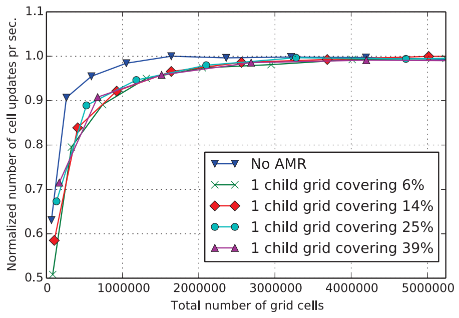


Figure 13: Comparison between single-grid simulation and AMR simulations using a single child grid covering a varying fraction of the global domain. All test runs have been normalized with regards to the single fastest run. There is an overhead connected with using AMR, but the overhead becomes negligible for larger domain sizes.

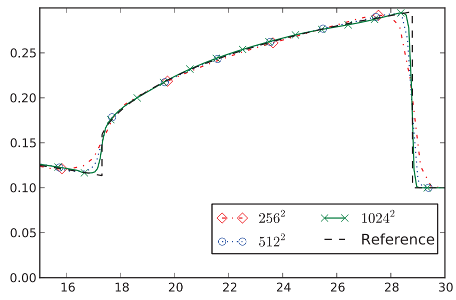
Shock tracking

In this test, we will demonstrate that the regridding algorithm is capable of tracking propagating waves. We mark all cells in which $\max\left(|h_{i+1,j} - h_{i,j}|, |h_{i,j+1} - h_{i,j}|\right) > 0.1$ m for refinement, in which $h_{i,j}$ is the average water depth in the grid cell centered at $(i\Delta x, j\Delta y)$. The domain is as in the previous test (see Figure 10b). Initial conditions are a circular dam break in the center of the domain with a radius of 10 m, in which the water elevation is set to 1.0 m, and 0.1 m in the rest of the domain. The root grid is 512×512 cells, a refinement check is performed every 10th time step, and the minimum child grid size is set to one tile to accurately track the shock. The test is run using second-order accurate Runge–Kutta time integration until it reaches seven seconds simulation time.

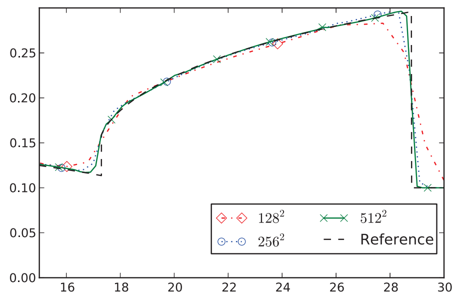
Figure 14 shows a comparison between single-grid simulations and AMR simulations using different grid resolutions. In both cases, we see that the solutions converge toward the front-tracking reference solution as the grid resolution is increased. Furthermore, the use of AMR clearly improves the accuracy of the solution, also on the root grid.

Figure 15a shows that the refinement closely follows the leading shock. The adept reader will notice that there are visible anomalies in the solution. These anomalies are caused by the initialization of new child domains. By initializing a child grid over an existing shock, the inevitable interpolation errors are severely magnified at the location of the shock, which leaves a “ghost” impression of the shock in the solution. These errors are alleviated by simply making sure that child domains are initialized in smooth parts of the solution, thereby minimizing the interpolation errors. One example of this strategy is illustrated in Figure 15b, in which the anomalies are reduced dramatically. Both of these simulations have a large number of child grids. The number of child grids can be dramatically reduced, however, by simply combining multiple neighboring child grids.

Next, we investigate the radial symmetry of the AMR simulator. Figure 16 shows the radial symmetry seven seconds into the simulation. We can see that the single-grid simulation with 512^2 cells, as expected, is more smeared than the reference solution, especially close to the sharp shock. The 1024^2 single-grid simulation, on the other hand, captures the shock much



(a) No AMR.



(b) AMR.

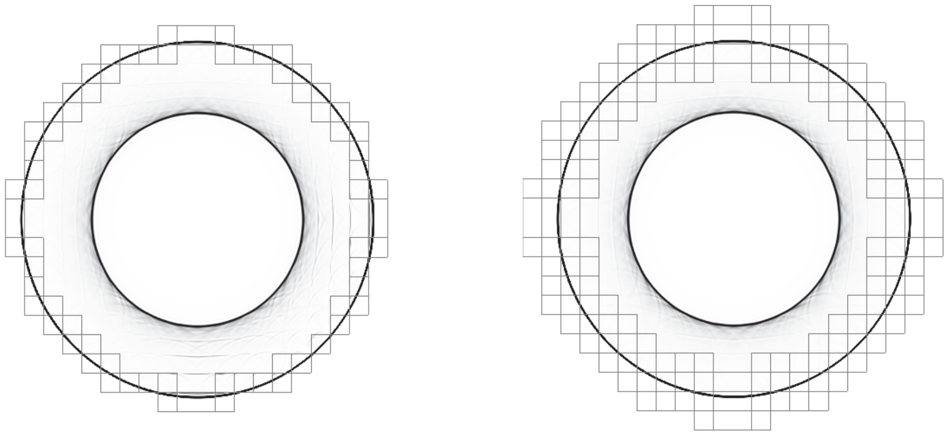
Figure 14: Cross section plot of water elevation values (m) as a function of radius (m) from the center of the circular dam after seven seconds of simulation time. Single-grid simulations and AMR simulations for different grid resolutions are compared. The AMR solutions are given in the root grid resolution. The reference solution is provided by a high-resolution (5000 grid cells) radial simulation using a front-tracking code [14] with double precision.

better. In both simulations, however, we see that there is a slight radial dissymmetry, shown by the vertical spread of the points. When we then go to the AMR simulations, we see that these simulations have a larger radial dissymmetry. This is as expected, as selectively refining parts of the domain will inevitably reduce the radial symmetry. For the simulation in which the child domains are initialized *on* the shock, however, there is a large non-physical dissymmetry (clearly visible approximately 25 meters from the center of the domain). When we initialize child domains before the shock arrives, however, these anomalies disappear. Even though there still is a larger radial dissymmetry than for the non-AMR simulations, we also see that the shock is captured with the resolution of the high-resolution single-grid simulation.

We have also run this test without flux correction (both parent-child and child-child) and with a fixed time-step size in an attempt to reduce the radial dissymmetry even more. However, the results from these test runs showed no significant difference from the AMR-run with padding. After ten seconds of simulation time, the AMR shock-tracking without padding is over 2.2 times faster than using the highest resolution for the whole domain, at the expense of a few minor artifacts in the solution.

Optimization: Effect of reducing Δt

In this test, we will investigate the effect of reducing the time-step size on a parent grid to avoid very small time steps on its children. The computational setup is as in the previous test (see Figure 10b). One child grid covering 25% of the root grid is used, and the shock never travels out of this child grid during the simulation runs, which are run up to time one second using second-order Runge–Kutta time integration. Different factors for limiting Δt, so that the Courant number stays below its maximal value, have been tested. A representative subset of the tested factors are shown in Figure 17. Considering that the areas of the solution with the highest velocities typically are the same areas one wants to refine, these results are promising. However, we have only tested using one level of refinement, and it is not trivial to implement this strategy for more than one level of refinement. This optimization is also highly problem



(a) Without padding. **(b)** With padding.

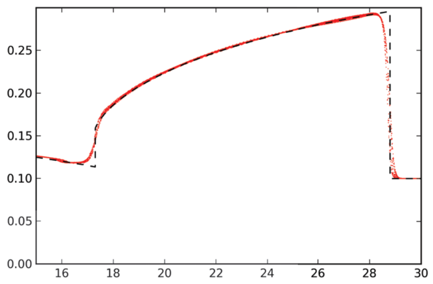
Figure 15: Emulated Schlieren visualization is used to demonstrate how the refinement follows the shock, enabling a more accurate capturing. Both figures are after seven seconds of simulation time, and the root grid resolution is 512 $\times$ 512 cells. Notice the ripples in the solution without padding, which have been significantly reduced when padding is added.

dependent, and further investigation should try to define some factor or range of factors that is proven viable in a broader selection of test setups.

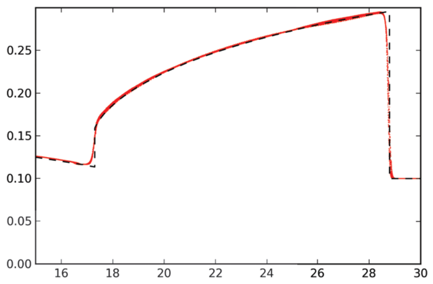
Although further investigation is necessary, some preliminary conclusions can be made. We see that a domain of a certain size (more than 65 000 cells in this example) is necessary before the Δt-limiting produces stable results, and that the stability of the optimization also seems dependent on the size of the limiting factor. The optimal size of the limiting factor would be such that the last very small step on the child grid is completely eliminated, but no larger. Considering that all time-step sizes are varying throughout the simulation, and that some child time-step series will not even have a very small last time step, this is a challenging task to accomplish. It seems likely that this needs to be an auto-tuned parameter, adjusted after a certain number of time steps, analog to the early-exit parameter discussed in [8].

Malpasset dam break

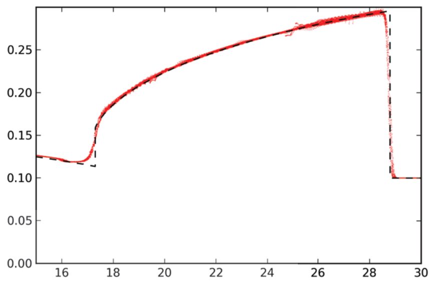
In a previous publication [8], we demonstrated that the non-AMR version of the simulator was capable of accurately simulating the first 4000 seconds of the Malpasset dam break in southern France. In this example, we will use this example to demonstrate that the AMR code is able to cope with such realistic real-world scenarios including complex bathymetry, bed shear-stress friction, and high velocities. For this particular example, increasing the resolution locally will not necessarily lead to more accurate predictions: there is already a high level of uncertainty in the bathymetry, and using the resolution of the mesh that is already available (which has been manually reconstructed from maps) gives simulation results that are close to the measured high water levels and front arrival times. Instead, the purpose is to demonstrate that the AMR method correctly adapts the resolution by adding new child grids to follow the advancing flood wave as it passes through the complex bathymetry.



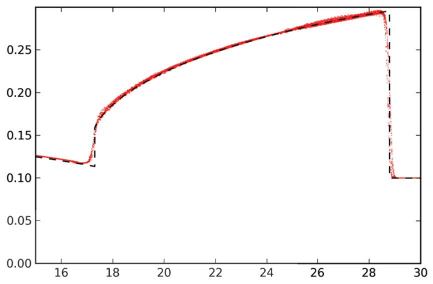
(a) No AMR. Resolution of 512^2.



(b) No AMR. Resolution of 1024^2.



(c) AMR without padding of new child grids.



(d) AMR with padding of new child grids.

Figure 16: Scatter plot of water elevation values (m) as a function of radius (m) from the center of the circular dam after seven seconds of simulation time. The dotted line is a high-resolution reference solution [14], taken through the center of the circular dam, and the dots are the simulation results for each of the different runs. Notice that both AMR simulations capture the shock position accurately and that padding reduces the loss of radial symmetry.

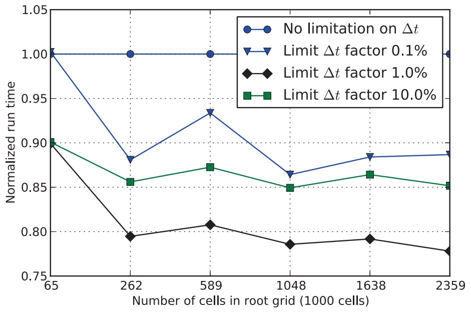


Figure 17: Comparison of wall-clock time for simulation runs with and without limit on root Δt. All time values represent the best performance of five consecutive runs, and are normalized with respect to the simulation run without limit on Δt for each domain size.

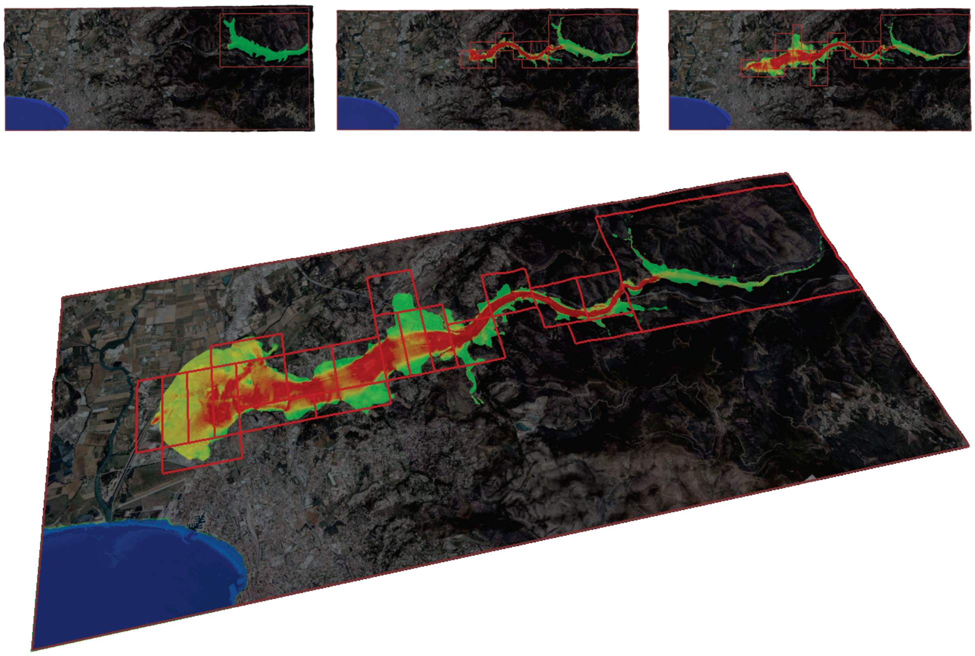
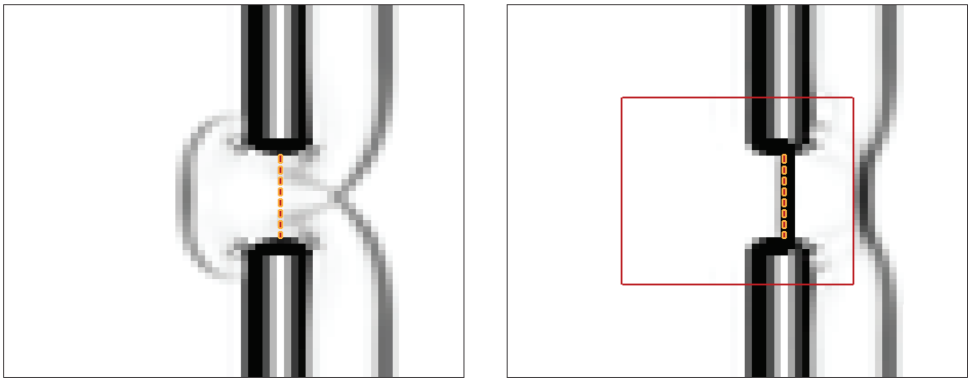


Figure 18: Visualization of water velocity in the 1959 Malpasset dam break case. Child grids are added dynamically to cover the flood as it progresses.

The lower part of Figure 18 shows the simulation when the flooding has reached the city of Fréjus, and as we can see, the child grids completely cover the flood. To ensure refinement once water enters dry areas, we flag cells for refinement when the water level exceeds 1×10^{-4} m. Initially, the water is contained in the dam visible in the upper child grid, wall boundaries are used, the Manning friction coefficient is set to 0.033, and second-order accurate Runge–Kutta time integration is used. A refinement check is performed every 50th time step, the minimum child grid size is set to three tiles, and one level of refinement is used. The speedup varies throughout the simulation run, but never drops below 2.7. If we let the simulation run to time 20 minutes, the AMR solver runs four times faster than the original simulator. After 30 minutes, a quite large fraction of the global domain has been covered by child grids (as depicted in Figure 18), and the speedup has dropped to 3.6 times.

Improved spatial resolution

For many real-world cases, e.g., tsunamis and storm surges, one is interested in following the incoming waves on a hierarchy of scales, from the open ocean to the coastland and connected river systems, swamplands, etc. In such cases, it would be too computationally demanding to compute the long-range wave propagation out at sea with the fine resolution required to provide accurate prediction of what happens when the waves hit the coastline. Use of AMR is one possible solution to this problem. By increasing spatial resolution locally, the simulator is able to capture more details in both the bathymetry and the conserved quantities in areas of



(a) Without AMR. **(b)** With AMR.

Figure 19: The figure represents a small section of a much larger domain. AMR is used to increase resolution in parts of the domain that hold a subgrid feature representing a narrow breakwater. The breakwater structure cannot be represented at the root grid resolution without AMR, which leads to erroneous simulation results. The subgrid, indicated by the red rectangle, covers 32 × 32 cells in the root grid.

particular interest, and at the same time preserve the interaction between local and global wave phenomena in a better and more efficient way than if the two were computed using different models or simulations.

To illustrate this point, we consider the prediction of the local wave pattern around a narrow breakwater whose width straddles the resolution of the model used to predict the incoming waves from the surrounding ocean. That is, we consider the waves inside a small 250 m × 200 m region that represents a small area of a much larger coastline area. The breakwater is represented as a bump in the bathymetry and generated procedurally. The central part narrows to one meter, and is therefore a subgrid feature in the 64×64 cell coarse grid. For the AMR simulation, we use one child grid covering 32 × 32 cells of the root grid, offset by 16 cells from the global boundary in both spatial dimensions.

As we can see in Figure 19, because of insufficient resolution, the original simulator fails to properly model the narrow portion of the breakwater. In the AMR simulation, however, the narrow part is present, and dramatically changes the solution.

5 Conclusions

In this article, we have implemented a tile-based AMR algorithm using a well-balanced, high-resolution finite-volume scheme so that it runs fully on a GPU. The resulting simulator conserves mass and has simple shock-tracking capability. The AMR implementation has been thoroughly tested and verified using analytical solutions, synthetic dam breaks, and real data from the Malpasset dam break. The results show that the code has excellent hardware utilization and that the accuracy on the child grids with the highest resolution (herein, we use at most three levels in total) is close to what would be obtained on a grid with full global refinement. The simulator has been carefully designed using modern software principles so that the code should be easy to use and understand and that simulations should be fast to setup and run.

Acknowledgements

This work is supported in part by the Research Council of Norway through grant number 180023 (Parallel3D), and in part by the Centre of Mathematics for Applications at the University of Oslo (Sætra and Lie).

Bibliography

[1] M. Berger and R. LeVeque. Adaptive mesh refinement for two-dimensional hyperbolic systems and the AMRCLAW software. *SIAM J. Numer. Anal*, 35:2298–2316, 1998.

[2] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:64–84, May 1989.

[3] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484–512, 1984.

[4] A. Brodtkorb, C. Dyken, T. Hagen, J. Hjelmervik, and O. Storaasli. State-of-the-art in heterogeneous computing. *Journal of Scientific Programming*, 18(1):1–33, May 2010.

[5] A. R. Brodtkorb, T. R. Hagen, K.-A. Lie, and J. R. Natvig. Simulation and visualization of the Saint-Venant system using GPUs. *Computing and Visualization in Science, special issue on Hot topics in Computational Engineering*, 13(7):341–353, 2010.

[6] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013.

[7] A. R. Brodtkorb and M. L. Sætra. Explicit shallow water simulations on GPUs: Guidelines and best practices. In *XIX International Conference on Water Resources, CMWR 2012, June 17–22, 2012*. University of Illinois at Urbana-Champaign, 2012.

[8] A. R. Brodtkorb, M. L. Sætra, and M. Altinakar. Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. *Computers & Fluids*, 55(0):1–12, 2012.

[9] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. Wilcox. Extreme-scale AMR. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE Computer Society, 2010.

[10] P. Colella, D. T. Graves, J. N. Johnson, H. S. Johansen, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. McCorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. Van Straalen. Chombo software package for AMR applications design document. Technical report, Lawrence Berkeley National Laboratory, 2012.

[11] O. Delestre, C. Lucas, P.-A. Ksinant, F. Darboux, C. Laguerre, T.-N.-T. Vo, F. James, and S. Cordier. SWASHES: a compilation of shallow water analytic solutions for hydraulic and

environmental studies. *International Journal for Numerical Methods in Fluids*, 72(3):269–300, 2013.

[12] N. Goutal and F. Maurel. Proceedings of the 2nd Workshop on Dam-Break Wave Simulation. Technical report, Groupe Hydraulique Fluviale, Département Laboratoire National d'Hydraulique, Electricité de France, 1997.

[13] M. Harris. NVIDIA GPU computing SDK 4.1: Optimizing parallel reduction in CUDA, 2011.

[14] R. Holdahl, H. Holden, and K.-A. Lie. Unconditionally stable splitting methods for the shallow water equations. *BIT Numerical Mathematics*, 39(3):451–472, 1999.

[15] R. D. Hornung and S. R. Kohn. Managing application complexity in the SAMRAI object-oriented framework. *Concurrency and Computation: Practice and Experience*, 14(5):347–368, 2002.

[16] S. Jones. Introduction to dynamic parallelism. GPU Technology Conference presentation S0338, 2012.

[17] Khronos Group. OpenCL. `http://www.khronos.org/opencl/`.

[18] A. Kurganov and G. Petrova. A second-order well-balanced positivity preserving central-upwind scheme for the Saint-Venant system. *Communications in Mathematical Sciences*, 5:133–160, 2007.

[19] Lawrence Berkeley National Laboratory, Center for Computational Sciences and Engineering. BoxLib. `https://ccse.lbl.gov/BoxLib/index.html`.

[20] R. J. LeVeque, D. L. George, and M. J. Berger. Tsunami modelling with adaptively refined finite volume methods. *Acta Numerica*, 20:211–289, Apr. 2011.

[21] R. Li and S. Wu. H-adaptive mesh method with double tolerance adaptive strategy for hyperbolic conservation laws. *Journal of Scientific Computing*, 56(3):616–636, 2013.

[22] K.-A. Lie and S. Noelle. On the artificial compression method for second-order nonoscillatory central difference schemes for systems of conservation laws. *SIAM Journal on Scientific Computing*, 24(4):1157–1174, 2003.

[23] P. MacNeice, K. M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer. PARAMESH: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330–354, 2000.

[24] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top 500 supercomputer sites. `http://www.top500.org/`, 2013.

[25] H. Nessyahu and E. Tadmor. Non-oscillatory central differencing for hyperbolic conservation laws. *Journal of computational physics*, 87(2):408–463, 1990.

[26] D. Nicholaeff, N. Davis, D. Trujillo, and R. W. Robey. Cell-based adaptive mesh refinement implemented with general purpose graphics processing units. Technical report, Los Alamos National Laboratory, 2012.

[27] NVIDIA. NVIDIA CUDA programming guide 5.0, 2012.

[28] NVIDIA. NVIDIA GeForce GTX 680. Technical report, NVIDIA Corporation, 2012.

[29] NVIDIA. CUDA community showcase. `http://www.nvidia.com/object/cuda_showcase_html.html`, 2013.

[30] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[31] M. L. Sætra. Shallow water simulation on GPUs for sparse domains. In A. Cangiani, R. L. Davidchack, E. Georgoulis, A. N. Gorban, J. Levesley, and M. V. Tretyakov, editors, *Numerical Mathematics and Advanced Applications 2011*, pages 673–680. Springer Berlin Heidelberg, 2013.

[32] M. L. Sætra and A. R. Brodtkorb. Shallow water simulations on multiple GPUs. In K. Jónasson, editor, *Applied Parallel and Scientific Computing*, volume 7134 of *Lecture Notes in Computer Science*, pages 56–66. Springer Berlin Heidelberg, 2012.

[33] H.-Y. Schive, Y.-C. Tsai, and T. Chiueh. GAMER: A graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics. *The Astrophysical Journal Supplement Series*, 186(2):457–484, 2010.

[34] P. K. Sweby. High resolution schemes using flux limiters for hyperbolic conservation laws. *SIAM journal on numerical analysis*, 21(5):995–1011, 1984.

[35] The Enzo Project. Enzo. `http://enzo-project.org/`.

[36] B. van Leer. Towards the ultimate conservative difference scheme. V. A second-order sequel to Godunov's method. *Journal of Computational Physics*, 32(1):101–136, 1979.

[37] P. Wang, T. Abel, and R. Kaehler. Adaptive mesh fluid simulations on GPU. *New Astronomy*, 15(7):581–589, 2010.