Scientific Computing on Heterogeneous Architectures

Thesis by André Rigland Brodtkorb as part of the fulfillment for the degree of Philosophiae Doctor. Defended on the 17th of December 2010 at the Faculty of Mathematics and Natural Sciences, University of Oslo, and approved by the Faculty on the same day based on the recommendation of the Adjudication committee.

Members of the Adjudication Committee

Research Group Leader Robert Strzodka,
Max Planck Institut Informatik,
Saarbrücken, Germany.

Professor Sverker Holmgren,
Department of Information Technology, Uppsala University,
Uppsala, Sweden.

Research Programmer Ola Skavhaug,
Simula Research Laboratory,
Oslo, Norway.

Ph.D. Thesis Advisors

Chief Scientist Tor Dokken,
Department of Applied Mathematics, SINTEF ICT,
Oslo, Norway.

Chief Scientist Knut-Andreas Lie,
Department of Applied Mathematics, SINTEF ICT,
Oslo, Norway.

Professor Knut Mørken,
Centre of Mathematics for Applications, University of Oslo,
Oslo, Norway.

Scientific Computing on Heterogeneous Architectures

André Rigland Brodtkorb

MMX

Abstract: The CPU has traditionally been the computational work horse in scientific computing, but we have seen a tremendous increase in the use of accelerators, such as Graphics Processing Units (GPUs), in the last decade. These architectures are used because they consume less power and offer higher performance than equivalent CPU solutions. They are typically also far less expensive, as more CPUs, and even clusters, are required to match their performance. Even though these accelerators are powerful in terms of floating point operations per second, they are considerably more primitive in terms of capabilities. For example, they cannot even open a file on disk without the use of the CPU. Thus, most applications can benefit from using accelerators to perform heavy computation, whilst running complex tasks on the CPU. This use of different compute resources is often referred to as *heterogeneous computing*, and we explore the use of heterogeneous architectures for scientific computing in this thesis. Through six papers, we present qualitative and quantitative comparisons of different heterogeneous architectures, the use of GPUs to accelerate linear algebra operations in MATLAB, and efficient shallow water simulation on GPUs. Our results show that the use of heterogeneous architectures can give large performance gains.

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 project number 180023/S10), and a three month period as a visiting researcher was partially funded by the NCCHE (National Center for Computational Hydroscience and Engineering).

The thesis consists of two parts, in which Part I contains three chapters: Chapter 1 gives an introduction to scientific computing on heterogeneous architectures; Chapter 2 summarizes our research; and Chapter 3 offers some concluding remarks and future outlooks. In Part II, we present published and submitted scientific papers that constitute the main research in this thesis.

Acknowledgements: This document is the outcome of three years and three months research, and a number of people have influenced my scientific journey. First of all, I would like to thank my three supervisors Tor Dokken, Knut-Andreas Lie, and Knut Mørken for their guidance and encouragement. I especially want to thank the two supervisors who have been my day-to-day contacts at SINTEF: Tor Dokken has counseled me well, was the one who secured funding for this Ph.D. project, and pushed me to publish my first paper; Knut-Andreas Lie has been a much valued advisor since my masters thesis, and his guidance and ruthless red pen has taught me the ways of scientific research.

I thank my closest colleagues at SINTEF: Christopher Dyken, Trond Hagen, Jon Hjelmervik, Jens Olav Nygaard, and Johan Seland. I have learned a lot from you all, and look forward to working closely with you in the future. I also thank my co-authors for our productive collaborations, and hope to continue our cooperation in future projects: Mustafa Altinakar, Christopher Dyken, Trond Hagen, Jon Hjelmervik, Knut-Andreas Lie, Jostein Natvig, Olaf Storaasli, and Martin L. Sætra.

I thank Jon Hjelmervik, my room-mate during a large part of this thesis. Jon has been a good source of information, provided me with sound advice in difficult questions, and included me for a superb visit to Institut polytechnique de Grenoble, France. Lunch at SINTEF was never quite the same after *lapin et vin rouge*. I also thank Martin L. Sætra, my current room-mate, whom I spent three months with in Mississippi at the NCCHE. Martin has been a great friend for many years, and made the stay in Mississippi exceptional: I will never forget our monster margaritas at Casa Mexicana. I thank Sverre Briseid for our conversations and your all-or-nothing attitude: I look forward with great anticipation to our next poker evening, conference, or Krabbefest. I thank Trond Hagen and Roger Bjørgan for their support that enabled me to stay three months at the NCCHE. I also thank all my colleagues at SINTEF, especially Vera Louise Hauge, Oddvar Kloster, Stein Krogstad, Ingeborg Ligaarden, Eivind Nilssen, Atgeirr Rasmussen, Vibeke Skytt, and Jan Thomassen. You have made the office a great place to be, and I look forward to our social and professional interaction in the future.

Finally, I would like to thank my family. The support of my parents and grand parents have enabled my journey into science, for which I am truly grateful. Tine, Kjæresta, you have been a pillar of support, and I cherish your genuine enthusiasm and encouragement. I look forward to spending more time with you without the burden of this document on my shoulders.

July 2010
André Rigland Brodtkorb

Part I
Introduction

Chapter 1

Introduction

There has been a dramatic change to consumer-level hardware in the last five years. CPUs have gone from serial to parallel execution, and we have simultaneously seen a growing use of different types of parallel accelerators. These changes pose new problems and opportunities in computational science, which is the topic of this thesis.

The transition from serial to parallel execution in CPUs has not been by choice. It is caused by the stagnating performance of a single CPU core, which again is caused by physical constraints that strongly limit further developments. The conventional wisdom that serial programs automatically double their performance every 18 months is thus outdated: the main source of performance increase today comes from increased hardware parallelism [6]. Because a lot of existing software is designed to scale with single-core performance, this has a deep impact: a large portion of existing software will not benefit from the increasing performance of todays CPUs.

Concurrently with the shift in CPUs from serial to parallel execution, there has also been a growing use of accelerator cores for computational science. An example of an accelerator is the graphics processing unit (GPU), and the interest in these architectures comes both as a result of the halt in single-core performance, and from the extreme performance they offer. While CPUs are designed to execute a wide span of different tasks, the accelerators focus on select few, in our case floating point operations. As an example, the state-of-the-art Intel Nehalem CPU with six cores has an impressive theoretical peak performance of 170 billion floating point operations per second, which is *one eight* of the GeForce GTX 480 GPU from NVIDIA.

The combination of traditional CPUs and accelerators is called a heterogeneous architecture, and using such architectures for computational science is called heterogeneous computing. The combination of a standard CPU and a modern programmable GPU is today found in over 180 million computers world-wide, and researchers have shown that such architectures can give a speed-up of 5–50 times for a variety of algorithms compared to only using the CPU [10]. It is the widespread deployment, combined with the possible performance gain, that makes heterogeneous architectures very attractive.

Some might consider the use of new multi-core CPUs and accelerators as troublesome, stating the obvious problem that existing software is not designed for these architectures. But instead of viewing these heterogeneous architectures as problematic, we can also see the advantage they bring: there is a plethora of algorithms that are embarrassingly parallel, which we have used serial architectures to execute for over 30 years. There are also many algorithms that

Figure 1: The NVIDIA GTX 480 graphics card. The card is connected to the rest of the computer through a vacant PCI express 2.0 16× slot, and requires extra power directly from the power-supply. The board contains a video BIOS, the graphics processing unit (GPU), and fast dedicated graphics memory.

are neither exclusively serial, nor exclusively parallel, but contain a mixture of both types of work loads. These algorithms benefit especially from the use of heterogeneous architectures, as we can use the most suitable processor for each task.

In this thesis, we commit research into efficient use of heterogeneous architectures for computational science. Our main focus is on the popular combination of a traditional CPU and a modern GPU, and we concentrate on two important algorithmic primitives, namely numerical linear algebra and stencil computations. The two algorithmic primitives are central in scientific computing, and are used in numerical simulation, image processing, signal processing, finance, and search engines, to name a few. We explore the efficient implementation of linear algebra operations on GPUs, and asynchronous execution on both CPUs and GPUs to utilize all available hardware resources. For stencil computations, we focus on efficient implementation of numerical schemes for the shallow water equations.

Linear algebra is the study of linear spaces, in which a key component concept is to solve systems of linear equations. One of the major strengths of linear algebra is that we can formulate a variety of problems as such systems, and use a set of standard methods to find solutions to these. This means that we can spend a lot of effort developing efficient solvers, and apply these in a wide range of application areas. In our work, we have focused on algorithms that are used for solving systems of linear equations.

A stencil computation is to combine multiple input elements into a single output element using a set of (predefined) weights. The set of weights is referred to as the stencil, and the stencil is typically applied to all elements in the computational domain. A major benefit of stencil computations is that they are embarrassingly parallel by design: each output element can be computed independently of all other elements. This makes the use of parallel architectures highly suited for this task. In our work, we have focused on using stencil computations for numerical simulation of the shallow water equations. The shallow water equations are representative of a class of problems called hyperbolic conservation laws, which are central in many different application areas. The shallow water equations are furthermore important in their own right, and are used to model physical phenomena ranging from tsunamis to atmospheric flow.

Efficient use of heterogeneous architectures is non-trivial, both for linear algebra and stencil computations, and generally requires other algorithm design principles than traditional CPUs. When utilizing the heterogeneous combination of a CPU and a GPU there are several things one has to consider. First of all, the execution model of GPUs is dramatically different from CPUs. GPUs employ massive parallelism and wide vector instructions, executing the same instruction for 32 or 64 elements at a time. Without designing algorithms that take this into consideration, the performance will be only a small fraction of what the hardware is capable of. Another thing to consider is that the GPU is on a separate circuit board called the graphics card, shown in Figure 1. The graphics card is connected to the rest of the computer through a vacant PCI express slot, which implies that all data transfer between the CPU and the GPU is relatively slow. These architectural differences are important to note, because it is only through a thorough understanding of the architecture that we can develop and implement efficient algorithms.

In the following, we give an overview of the topics covered by this thesis. First, we give a gentle introduction to numerical linear algebra and stencil computations as seen from a heterogeneous computing perspective in Section 1 and 2, respectively. In Section 3 we present a short overview of the shallow water equations, and how these can be solved using stencil computations. We continue with an instructive programming example in Section 4, where we solve a simple simulation problem on a multi-core and heterogeneous architecture. Then, we motivate our focus on heterogeneous architectures in Section 5, by giving some of the major reasons behind the advent of heterogeneous computing. Finally, we summarize each of the six papers that constitute the main research in this thesis in Chapter 2, and give some concluding remarks and future outlooks in Chapter 3.

1 Linear Algebra

Numerical linear algebra is the corner stone of numerous applications in scientific computing, in which solving systems of linear equations is a key component. We have focused on accelerating such algorithms, and present two papers on the subject [9, 8]. In this section, we will first introduce how numerical simulation can be formulated as systems of linear equations, and continue with an overview of methods for finding solutions to these.

An example of an elementary system of linear equations is

$$\begin{aligned} x + y &= 10 \\ x - y &= 4, \end{aligned} \tag{1}$$

and it is trivial to find that the only solution to this system is $x = 7$ and $y = 3$. We can also write this system using matrix notation,

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 10 \\ 4 \end{bmatrix}, \tag{2}$$

which is convenient when we have a large number of variables. For two and three unknowns, it is not too difficult to find the solution of such problems using pen and paper, but when the number of equations grow, the process becomes complex and error prone. The first electronic digital computer, the Atanasoff-Berry Computer (1942), was designed solely for the purpose of solving such linear systems, and could handle up-to 29 variables. Today, solving systems

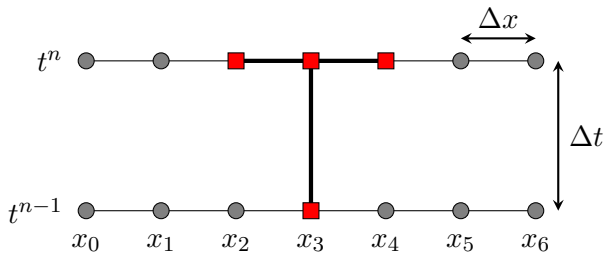


Figure 2: The discrete grid in space and time for the heat equation, with the data dependencies for computing u_3^n marked (red squares). We here show seven discrete points in space ($x_0 - x_6$) for for two timesteps (u^n and u^{n+1}). The spatial cell spacing is Δx, and the temporal spacing is Δt.

of linear equations is one of the basic tasks used in a wide range of applications, and has been extensively used as a performance benchmark. In fact, the worlds fastest supercomputers are ranked according to how fast they can solve such systems [36]. The current record holder is the Jaguar supercomputer at Oak Ridge National Laboratories, which solved a system of 5.474.272 equations in 17 hours and 17 minutes.

The heat equation is a partial differential equation that describes the evolution of heat distribution in a medium, and we will here illustrate how we can find solutions of this equation using linear algebra. Let us consider a thin metal rod, where we model the heat distribution as a one-dimensional problem. In the continuous case, the heat equation is written

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2}, \tag{3}$$

where κ is the thermal conductance of the material, and u is the temperature.

To find solutions of this equation on a computer, we first discretize it by introducing the notion of a discrete grid for our metal rod. Figure 2 shows our chosen grid consisting of seven equally spaced grid points where

$$u_i^n = u(t^n, x_i) = u(n\Delta t, i\Delta x).$$

We continue our discretization by replacing the continuous derivatives in space and time with discrete approximations. We use a central difference for the derivative with respect to x, and a backward difference for the derivative with respect to t. This gives the following implicit finite difference approximation,

$$\frac{1}{\Delta t}(u_i^n - u_i^{n-1}) = \frac{\kappa}{\Delta x^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n),$$

where we can reorder to get the following algebraic equation for each cell,

$$-ru_{i-1}^n + (1+2r)u_i^n - ru_{i+1}^n = u_i^{n-1}, \qquad r = \frac{\kappa \Delta t}{\Delta x^2}. \tag{4}$$

Here, we assume that the temperatures u^{n-1} are known, and we want to find the unknown temperatures u^n. To compute the first grid point, u_0^n, our scheme depends on the value of u_{-1}^n

which is outside our grid, and similarly for u_6^n. We need to modify the equation at these two points to enforce boundary conditions, which are used to describe what the solution should look like outside our computational domain. There are several different boundary conditions we can use. For sake of simplicity, we assume that the heat at both ends of our rod is fixed, also called Dirichlet boundary conditions:

$$u_0^t = \alpha_0, \quad u_6^t = \alpha_1, \quad \forall\, t.$$

The problem is now well-defined, and we can write the system of equations. For the seven grid points in Figure 2, we get

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -r & 1{+}2r & -r & 0 & 0 & 0 & 0 \\ 0 & -r & 1{+}2r & -r & 0 & 0 & 0 \\ 0 & 0 & -r & 1{+}2r & -r & 0 & 0 \\ 0 & 0 & 0 & -r & 1{+}2r & -r & 0 \\ 0 & 0 & 0 & 0 & -r & 1{+}2r & -r \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0^n \\ u_1^n \\ u_2^n \\ u_3^n \\ u_4^n \\ u_5^n \\ u_6^n \end{bmatrix} = \begin{bmatrix} u_0^{n-1} \\ u_1^{n-1} \\ u_2^{n-1} \\ u_3^{n-1} \\ u_4^{n-1} \\ u_5^{n-1} \\ u_6^{n-1} \end{bmatrix}, \tag{5}$$

in which we set $u_0^0 = \alpha_0$ and $u_6^0 = \alpha_1$ to satisfy the chosen boundary conditions.

The set of equations can be written more compactly as

$$Ax = b, \tag{6}$$

where A is the tridiagonal coefficient matrix, $x = u^n$, and $b = u^{n-1}$. Our derivation of the linear system of equations is not dependent on a specific number of cells. For k cells, the matrix A will be a $k \times k$ tridiagonal matrix, and x and b will have k elements each. By solving the system of equations $Ax = b$, we propagate the solution one timestep.

The above example shows how we arrive at a problem on the form $Ax = b$ for the heat equation, and there are a number of different algorithms for solving such systems of linear equations. The prototypical algorithm is classical Gaussian elimination, in which we perform elementary row operations to solve the system. Starting with the problem $Ax = b$,

$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}, \tag{7}$$

we eliminate all elements below the first element, $A_{1,1}$, by subtracting a multiple of the first row,

$$\left[\begin{array}{c|ccc} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ \hline 0 & A_{2,2}^* & A_{2,3}^* & A_{2,4}^* \\ 0 & A_{3,2}^* & A_{3,3}^* & A_{3,4}^* \\ 0 & A_{4,2}^* & A_{4,3}^* & A_{4,4}^* \end{array}\right] \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \left[\begin{array}{c} b_1 \\ \hline b_2^* \\ b_3^* \\ b_4^* \end{array}\right]. \tag{8}$$

Here, * denotes that the element has changed. Now, we can apply the same technique to the lower right part of the system (called the trailing matrix), and continuing this process we eventually end up with an upper triangular matrix where all elements below the diagonal are zero. We then perform the same operation "in reverse", called backward substitution, eliminating all elements above the diagonal. After completing the backward substitution, we end up with a solution on the form

$$\begin{bmatrix} A^*_{1,1} & 0 & 0 & 0 \\ 0 & A^*_{2,2} & 0 & 0 \\ 0 & 0 & A^*_{3,3} & 0 \\ 0 & 0 & 0 & A^*_{4,4} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b^*_1 \\ b^*_2 \\ b^*_3 \\ b^*_4 \end{bmatrix}, \tag{9}$$

assuming the matrix has full rank. However, this algorithm is numerically unstable if the upper left element in the trailing matrix is small. To counter this, a technique known as pivoting is used. By simply exchanging rows within the trailing matrix, so that the upper left element is the largest within its column, we make the algorithm well-behaved for most problems.

Gaussian elimination is an example of a direct method that can be used to find the solution to a system of k equations in $\mathcal{O}(k)$ steps. The algorithm requires $\mathcal{O}(k^3)$ operations, which is highly wasteful for the heat equation example. We know that the heat equation matrix is tridiagonal, and can exploit this fact to create a tridiagonal Gaussian elimination algorithm that only performs computations on elements known to be non-zero. This yields a much more efficient algorithm that can find the solution using only $\mathcal{O}(k)$ operations.

Another class of important algorithms from numerical linear algebra is factorizations. One example is LU factorization. In short, we want to find the matrices L and U such that $LU = A$, in which L is a lower triangular matrix, and U is an upper triangular matrix. This factorization is highly useful in applications where the matrix A remains constant, but the coefficient vector b changes. First, we solve

$$Ly = b \tag{10}$$

for y using forward substitution, and continue solving

$$Ux = y \tag{11}$$

for x using backward substitution. Both solving for y and solving for x has a computational complexity of $\mathcal{O}(k^2)$ operations, which compares favourably to the complexity of $\mathcal{O}(k^3)$ for Gaussian elimination.

The LU factorization can be computed in a similar fashion as Gaussian elimination using the Doolittle algorithm. In the Doolittle algorithm, we perform forward substitution, just as in Gaussian elimination, but store the multipliers used to eliminate elements below the diagonal in the matrix L. Through this modified forward substitution, we populate the lower triangular part of L, and reduce A to an upper triangular form, which gives us an LU factorization.

In addition to the direct methods discussed above, there are also classical iterative methods, such as Jacobi iteration, Gauss-Seidel iteration, and Successive Overrelaxation. These iterative algorithms create approximations to the solution, and one typically stops the iteration when

the approximation has converged to within a given tolerance. Another class of algorithms is Krylov subspace iterative methods, such as the method of Conjugate Gradients and Generalized Minimum Residual (GMRES), which often displays faster convergence rates than the classical iterative methods. For a thorough overview of these and other algorithms, we refer the keen reader to textbooks, for example by Golub and van Loan [18] or Meyer [29].

Because so many different problems can be written as a system of linear equations, there has been an enormous investment in implementing these efficiently on different computational hardware [2, 14, 40, 15, 16, 31, 4, 5, 25]. A benefit of using heterogeneous architectures for linear algebra is that many operations are intrinsically parallel. Take Gaussian elimination as an example. By examining the computational complexity, we see that we on average perform $\mathcal{O}(k)$ operations per element. Most of these operations can be performed in parallel for large parts of the matrix. We can compute each element in the trailing matrix independently of all other elements (see (8)). However, as the elimination progresses, the size of the trailing matrix is reduced, which means that we get decreasing levels of parallelism. A further complication is that there is a lot of global synchronization, which can be expensive on massively parallel architectures. Nevertheless, the use of heterogeneous architectures has great potential due to their high bandwidth and floating point capabilities.

2 Stencil Computations

Stencil computations arise in a wide range of applications, from simple filters in image and signal processing, to complex numerical simulations. In this section, we give an introduction to how stencil computations arise in the solution of partial differential equations. Later in this thesis, we present two papers on the efficient use of GPUs for numerical simulation of a particular type of partial differential equations, called hyperbolic conservation laws [11, 12].

In the previous section, we saw how we could formulate the heat equation as a linear algebra problem. We did this by first introducing a discrete grid, and then deriving a discretization by replacing the derivatives with discrete approximations. We used a centered difference in space, and a backward difference in time, and it is the backward difference that gives us the implicit scheme. The implicit scheme was formed from the discretization

$$\frac{1}{\Delta t}(u_i^n - u_i^{n-1}) = \frac{\kappa}{\Delta x^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n). \tag{12}$$

By changing the backward difference in time to a forward difference, we instead end up with

$$\frac{1}{\Delta t}(u_i^{n+1} - u_i^n) = \frac{\kappa}{\Delta x^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n), \tag{13}$$

where the superscripts on the left hand side have changed. This gives rise to the *explicit* scheme for the heat equation,

$$u_i^{n+1} = ru_{i-1}^n + (1-2r)u_i^n + ru_{i+1}^n, \qquad r = \frac{\kappa \Delta t}{\Delta x^2}. \tag{14}$$

Here, all the values on the right hand side are known. Thus, comparing (4) with the above equation, we see that the implicit scheme gives us a set of algebraic equations, whilst the explicit scheme gives us formulas for the solution.

As with the implicit scheme, we have to handle the end points in a special way since u_0^{n+1} depends on u_{-1}^{n+1} and u_{k-1}^{n+1} depends on u_k^{n+1}, where k is the number of grid points. For the implicit case, we simply changed the first and last equations to implement boundary conditions, and we can follow the same strategy here. However, another way of handling boundary conditions is to extend the domain using *ghost cells*, which means that we add the two points u_{-1} and u_k to the grid. This enables us to use the same stencil throughout the computational domain, and by setting the value of the ghost cells we can enforce boundary conditions.

For the explicit stencil computation to be stable, we need to obey a Courant-Friedrichs-Levy (CFL) condition. Loosely speaking, the CFL condition ensures that the timestep, Δt, is sufficiently small so that the numerical domain of dependence includes the domain of dependence prescribed by the equation. The CFL condition is a requirement (but not a guarantee) for stability of explicit schemes for partial differential equations. For the heat equation, the CFL condition is

$$\frac{1}{2} > \frac{\kappa \Delta t}{\Delta x^2}, \tag{15}$$

which we can use to compute the size of each timestep. We now have a well-defined problem that we can solve numerically, and we can apply the stencil to every cell in the domain to evolve the solution in time. How this is done on a GPU will be shown in Section 4.

For the one-dimensional stencil we have derived here, we can easily calculate the required number of operations we need to perform per element. The calculation of u_i^{n+1} can be done using three multiplications and two additions when κ is constant. We also need to read three values from main memory, and write one value back. Thus, for this stencil, we have a ratio of arithmetic operations to memory accesses of 5 to 4. Because stencil computations typically perform relatively few calculations on large amounts of data, they are often memory bound: the state-of-the-art Nehalem CPU with six cores has a theoretical performance of over 170 billion single-precision operations per second, whilst the memory bandwidth can theoretically transfer only eight billion single-precision values (bidirectional).

CPUs counter the effect of the slow memory bandwidth with huge on-chip caches that are used to hold frequently used data. Traditional CPU techniques to increase throughput of stencil operations exploit this, and are typically based on cache blocking. By dividing the computational domain into sub-domains that each fit in cache, the elements are reused, and the effect of the slow memory bandwidth lessens. A benefit of accelerators such as the GPU, is that they have fast on-chip memory that can be viewed as a programmable cache. Instead of designing our algorithm for an automatic cache that has to guess what part of memory we will use frequently, we can design an algorithm where we ourselves move the data we know we need. By exploiting the predefined data dependency given by our stencil, we can thus maximize the use of on-chip memory. Another benefit of using accelerator cores for stencil computations is that the algorithm is embarrassingly parallel, which means that implementations will scale with future hardware.

3 The Shallow Water Equations

The shallow water equations are a recurring theme throughout this thesis: in [9], we solve the shallow water equations using an implicit scheme, and in [11] and [12], we present the

implementation of explicit schemes. The equations are interesting by themselves, and do not only apply to water cases such as tsunami, dam break, and flood simulation, but also to a wide range of other application areas, including simulation of avalanches and atmospheric flow. Furthermore, the equations are an example of a type of problems called hyperbolic conservation laws. A great number of physical problems within acoustics, astrophysics, geophysics, and multiphase flow, to mention a few, can be modeled using hyperbolic conservation laws. Finding solutions to the shallow water equation shares many commonalities with finding solutions to the aforementioned problems, and our initial interest in the these equations was motivated by its relatively simple and intuitive nature and relation to other types of problems. During the research presented in this thesis, however, a sincere interest in the equations themselves has also developed, and our research has gone beyond discussing solely the efficient implementation of stencil-based schemes. In this section, we briefly describe the equations, and the structure of the types of stencil computations used in this thesis. The discussion here is rather superficial, and the interested reader is referred to text books, for example by LeVeque [28] and Toro [38], for a more thorough introduction to numerical methods for conservation laws and shallow water flow.

The shallow water equations describe gravity-induced motion of a fluid under a free surface, with the assumption of shallow water. Our primary interest in the equations are to simulate flooding and dam-breaks. The equations describe conservation of mass and momentum, and are vertically integrated, thus assuming that the vertical acceleration has a negligible contribution to the pressure. In two space dimensions, the equations read

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = 0, \tag{16}$$

where h is the water depth, hu is the flux along the x-axis, hv is the flux along the y-axis, and g is the gravitational constant (see also Figure 3a). More compactly, we write the equations on vector form,

$$Q_t + F(Q)_x + G(Q)_y = 0. \tag{17}$$

Here, Q is the vector of conserved variables, and F and G are flux functions. The above equations are capable of solving the homogeneous case of the shallow water equations, where we have a flat friction-less bottom topography. These solutions are of limited interest for the use cases we have focused on. For simulation of real-world cases, we need to take the bottom slope and bottom friction into account as well, which can be done by adding *source terms* to the equation,

$$Q_t + F(Q)_x + G(Q)_y = H_B + H_f. \tag{18}$$

Here, H_B is the bed slope source term, and H_f is the bed friction source term.

The shallow water equations are nonlinear, and give rise to discontinuous solutions that are difficult to resolve numerically. Classical explicit methods either excessively smear discontinuities, such as Lax-Friedrichs, or create spurious oscillations near discontinuities, such as

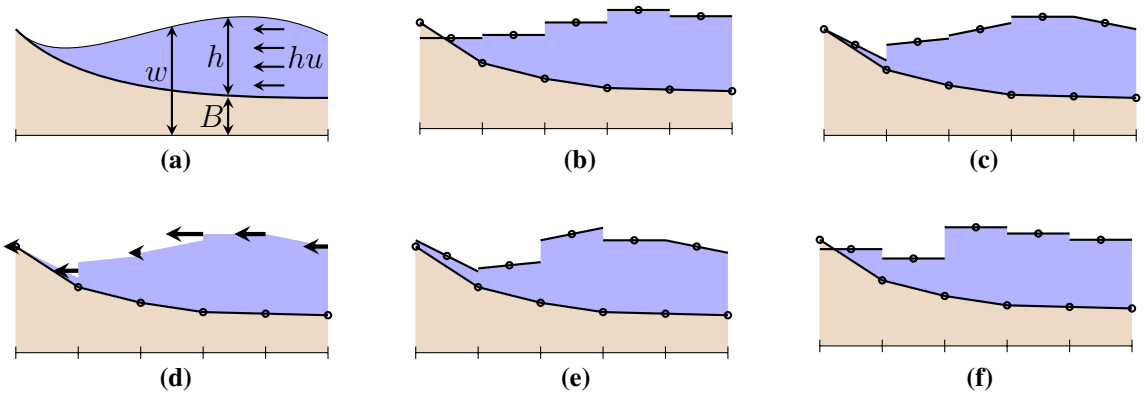


Figure 3: Structure of the explicit high-resolution schemes examined in this thesis (simplified to one dimension). In (a), we have the continuous variables water depth (h), water elevation (w), and bottom elevation (B). The schemes use a *staggered grid*, where B is given as a piecewise linear function defined by values at cell intersections, and the physical variables are given as cell averages, as shown in (b). First, we reconstruct a piecewise linear function for each of the physical variables in (c), and compute a set of fluxes across cell interfaces in (d). Then, we evolve the solution in time by summing the fluxes in (e), and finally find the new average within each cell in (f).

Lax-Wendroff. Modern schemes based on the high-resolution principle [20, 35], however, are capable of capturing both the smooth and discontinuous parts of the solution, and our focus has been on such schemes. The schemes we examine use *staggered grids*, in which the variables are stored as cell averages and we compute fluxes at cell interfaces. In the homogeneous case (without source terms), these types of schemes can be written

$$Q_t = -\big[F(Q_{i+1/2,j}) - F(Q_{i-1/2,j})\big] - \big[G(Q_{i,j+1/2}) - G(Q_{i,j-1/2})\big], \tag{19}$$

and Figure 3 illustrates the operations performed to evolve the solution in time. Using the so-called Reconstruct-Evolve-Average principle, we *reconstruct* a piecewise bilinear function for each variable, compute the numerical fluxes and *evolve* the solution one timestep, and *average* the evolved solution. Compared to the simple heat equation stencil derived earlier, we here perform far more computations, and are thus hardly as memory bound when data can be kept in on-chip memory.

The high-resolution schemes are based on using so-called flux limiters to find the reconstructed variables (Figure 3c). The sole purpose of these limiters is to give high resolution in smooth parts of the domain without smearing discontinuities. One such limiter is the minmod limiter,

$$\text{MM}(a,b,c) = \begin{cases} \min(a,b,c), & \{a,b,c\} > 0 \\ \max(a,b,c), & \{a,b,c\} < 0 \\ 0. \end{cases} \tag{20}$$

For three candidate slopes, a, b, and c, the minmod slope limiter chooses the least steep when all the slopes have the same sign. When the sign of the slopes differ, it returns zero slope. This

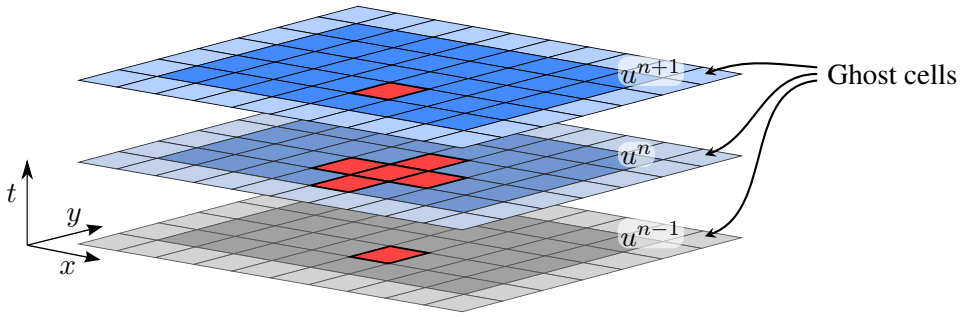


Figure 4: The discrete grid which defines our computational grid for the wave equation. To compute the new timestep (u^{n+1}), we need all the values at the current (u^n) and the previous (u^{n-1}) timestep. The value for each cell is interpreted as a point value in the case of the wave equation. The marked cells constitute the stencil needed to compute the cell at timestep u^{n+1} (see (22)).

strategy is highly successful for capturing both smooth and discontinuous parts of the domain, and falls under a category of methods called shock capturing methods. These methods are oblivious to the presence and location of shocks in the solution, making them highly suitable for architectures such as the GPU.

4 Stencil Computations on a Heterogeneous Architecture

In this section, we examine the linear wave equation using an explicit scheme, and its implementation on the combination of a CPU and a GPU. This example illustrates some of the steps and strategies required to map the more complex shallow water equations to the GPU. The linear wave equation is derived from modeling vibrations of a string, and is used in fields such as acoustics, electromagnetics, and fluid dynamics. It is also an example of a hyperbolic conservation law, and in two dimensions it is written

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right], \tag{21}$$

where c is the wave propagation speed for the medium. We can discretize this equation on a grid (see Figure 4) in a fashion similar to the explicit heat equation scheme. This gives the explicit scheme

$$\begin{aligned} u_{i,j}^{n+1} = 2u_{ij}^n - u_{ij}^{n-1} &+ c1(u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n) \\ &+ c2(u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n), \end{aligned} \qquad c1 = \frac{c^2\Delta t^2}{\Delta x^2}, c2 = \frac{c^2\Delta t^2}{\Delta y^2}, \tag{22}$$

where u_{ij}^{n+1} is a linear combination of five values from u^n and one value from u^{n-1}.

Figure 4 shows our computational domain, which is extended with ghost cells that we need to update every time step. This suits the execution model of graphics cards well since we end up with an algorithm that is oblivious to boundaries. As for the explicit scheme for the heat equation, we are also here bound by a CFL condition,

$$\frac{1}{2} > \max\left\{ \frac{c^2\Delta t}{\Delta x}, \quad \frac{c^2\Delta t}{\Delta y} \right\}, \tag{23}$$

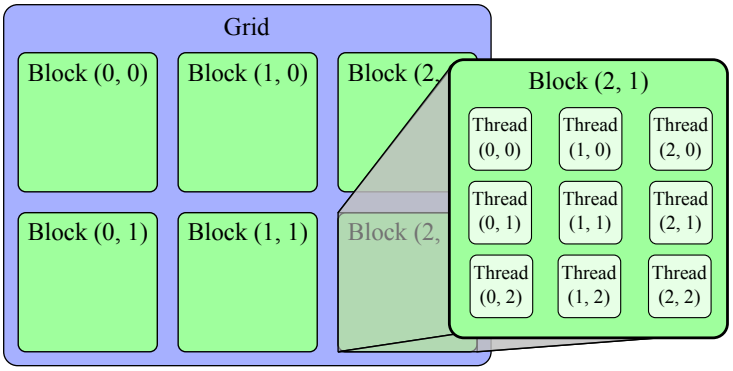


Figure 5: Execution model of CUDA. All blocks consist of the same number of threads, where threads within one block can cooperate and communicate. The hardware automatically schedules blocks to the physical processors on the GPU.

which we can use to compute the size of each timestep.

The wave equation gives us a simple and manageable problem that we can discuss how to implement on a heterogeneous system. It furthermore displays several commonalities with implementing more complex schemes, such as those we have used for simulation of the shallow water equations. In the following, we give an overview of how we can map the computation to a heterogeneous architecture consisting of a CPU and a modern GPU. We start by illustrating how the scheme can be implemented on a multi-core CPU, and then continue by moving part of the computation to the GPU using CUDA. There are alternatives to CUDA for programming GPUs, such as OpenCL [26] and DirectCompute [30], but we have used NVIDIA CUDA throughout this thesis. The programming examples shown in this section are written for clarity, and are obviously not optimized for speed. For brevity, we have also omitted error checking, etc. For a more thorough introduction to programming using CUDA, there are several recent books on the subject, for example by Sanders and Kandrot [34], and Kirk and Hwu [27].

Let us start by examining how we would implement the numerical scheme on a traditional CPU using C++. First of all, we need to allocate memory for our three timesteps, u^{n-1}, u^n, and u^{n+1}. Here, `nx` is the width of the domain including ghost cells, and similarly for `ny`:

```
int main(int argc, char** argv) {
  float* u_m = new float[nx*ny];
  float* u = new float[nx*ny];
  float* u_p = new float[nx*ny];
```

After allocation, we perform initialization of data (not shown), and enter the main simulation loop. The main simulation loop executes the stencil for every cell in the domain, and we use a simple OpenMP [13] pragma that facilitates the use of multi-core CPUs:

```
  while (t < t_end) {
    #pragma omp parallel for
    for (int j=1; j<ny-1; ++j) {
      for (int i=1; i<nx-1; ++i) {
        int north = i+(j+1)*nx;
        int center = i+j*nx;
        int south = i+(j-1)*nx;
```

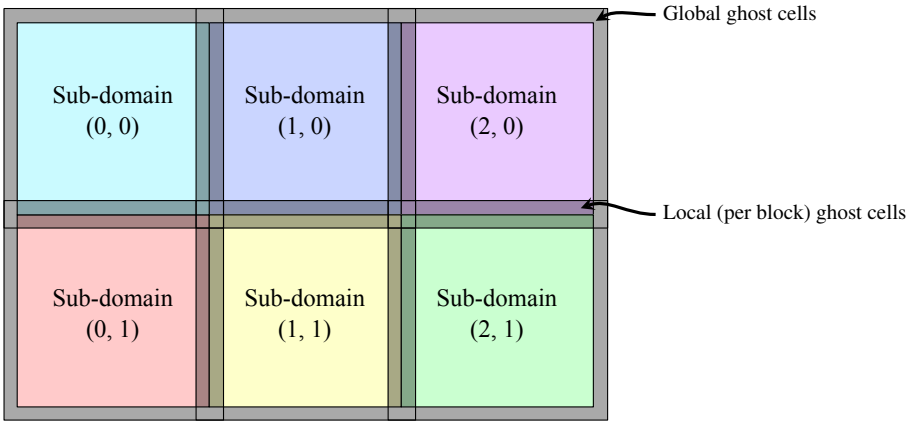


Figure 6: The domain decomposition for the wave equation implementation in CUDA. Each small sub-domain is computed by one CUDA block, and is responsible for 14×14 output elements. To compute these elements, we need local ghost cells, as dictated by our stencil (see (22)). These local ghost cells overlap with other sub-domains, and make the input domain 16×16 elements.

```
            int east = i+1+j*nx;
            int west = i-1+j*nx;
            u_p[center] = 2*u[center] - u_m[center]
                        + c1*(u[east] -2*u[center]+u[west])
                        + c2*(u[north]-2*u[center]+u[south]);
        }
    }
    t += dt;
```

After performing the stencil computation for each cell in the domain, we need to set the boundary conditions (not shown), and shift the pointers so that the newly computed values become input to the next iteration:

```
        float* u_tmp = u_m;
        u_m = u;
        u = u_p;
        u_p = u_tmp;
    }
    return 0;
}
```

This illustrates how we can implement the explicit scheme for the wave equation in very few lines of code, which is a true strength of stencil computations.

Let us now move on to how we can implement this scheme on the GPU using NVIDIA CUDA. We need to explicitly partition the domain to map the computations to the independent processors on the GPU. In CUDA terms, this partitioning consists of a *grid* of *blocks* as shown in Figure 5. Threads within the same block can cooperate and communicate using *shared memory*, which is fast on-chip memory that can be used to store frequently used data for higher performance. Each block is executed independently, and the hardware automatically handles

scheduling of blocks to the processing cores on the GPU, which means that blocks cannot cooperate without expensive global synchronization.

For best performance, we typically first determine a block size that maps well with both the algorithm and the hardware, and then partition the domain using this block size. There are many, often conflicting optimization parameters for choosing a good block size, and the interested reader is referred to CUDA documentation [32, 33] for an overview of these. In our example, we use a block size of 16×16 threads that are responsible for computing 14×14 output elements. Figure 6 illustrates the domain decomposition strategy. The blocks form overlapping input domains because of the stencil dependencies, and we keep each sub-domain in shared memory to increase performance.

We start our CUDA implementation by allocating data on the GPU:

```
int main(int argc, char** argv) {
  float* u_m;
  float* u;
  float* u_p;
  size_t u_m_pitch, u_pitch, u_p_pitch;
  cudaMallocPitch((void**) &u_m, &u_m_pitch, nx*sizeof(float), ny);
  cudaMallocPitch((void**)   &u,   &u_pitch, nx*sizeof(float), ny);
  cudaMallocPitch((void**) &u_p, &u_p_pitch, nx*sizeof(float), ny);
```

The GPU allocation follows C-style malloc, and the above allocates memory to hold `nx`$\times$`ny` floating point numbers. Readers familiar with C might notice the rather strange *pitch* argument supplied to the function. This is the width of the domain in bytes after allocation and is related to efficiency: the allocation pads the domain to ensure that each row starts at a properly aligned address. Next, we can allocate CPU data, initialize it (not shown), and upload it to the GPU. For u^n, this becomes

```
float* u_host = new float[nx*ny];
cudaMemcpy2D(u, u_pitch,
             u_host, nx*sizeof(float),
             nx*sizeof(float), ny, cudaMemcpyHostToDevice);
```

This memory copy also follows C-style copying of data. The first two arguments specify the location and pitch of the destination GPU memory, and the next two the location and pitch of the source CPU memory. The three next arguments specify the size of the 2D domain to copy, and the direction. Now that we have initialized our data, we can enter the main simulation loop,

```
dim3 block = dim3(16, 16);
dim3 grid = dim3((nx-2)/14, (ny-2)/14);
while (t < t_end) {
  wave_stencil<<<grid, block>>>(u_p, u_p_pitch,
      u, u_pitch,
      u_m, u_m_pitch,
      c1, c2);
  t += dt;
```

Here, we have substituted the double for-loop on the CPU with a call to a GPU *kernel* called `wave_stencil`. Essentially, the kernel is a regular function that is executed on the GPU using the predefined grid and block configuration The kernel signature is

```
__global__ void wave_stencil(float* u_p, size_t u_p_pitch,
    float* u, size_t u_pitch,
    float* u_m, size_t u_m_pitch,
    float c1, float c2) {
```

Here, the arguments to the kernel are the pointers to input and output in GPU memory, their respective pitches, and the constants $c1$ and $c2$ (see (22)). Each thread has a unique identifier within each block, and each block has a unique identifier within the global domain. We use these to calculate the position of the current thread within the global domain:

```
int i = threadIdx.x;
int j = threadIdx.y;
int x = blockIdx.x*14 + i;
int y = blockIdx.y*14 + j;
```

Then, we copy from global memory into shared memory. To find the location in global memory, we need to perform some simple pointer arithmetic, since the width of the domain is given in bytes and not elements. In the following, `u_y_ptr` has thus been set to point to row `y` in the domain.

```
__shared__ float smem_u[16][16];
smem_u[j][i]   = u_y_ptr[x];
__syncthreads();
```

We use the intrinsic function `__syncthreads()` to ensures that all threads within the block have completed the memory copy. Up until this point, we have used 16×16 threads, that each have filled their part of shared memory with the correct data. Now, we let the 14×14 internal threads execute the stencil and write the result to global memory,

```
if (i < 1 || i > 14 || j < 1 || j > 14) return;
u_p_y_ptr[x] = 2*smem_u[j][i] - smem_u_m[j][i]
             + c1*(smem_u[j][i+1]-2*smem_u[j][i]+smem_u[j][i-1])
             + c2*(smem_u[j+1][i]-2*smem_u[j][i]+smem_u[j-1][i]);
}
```

This trivial CUDA example contains only 16 lines of code more than the C++ version. Most of these lines are used to copy data between the CPU and the GPU and between global memory and on-chip shared memory on the GPU. While this example shows that it is easy to get started with programming the GPU using CUDA, it is very difficult to make full use of the hardware. The complexity of manual parallelization and data movement, combined with nontrivial hardware limitations makes it a considerable task to design efficient implementations.

5 From Serial to Parallel and Heterogeneous

May 26th, 2005 marks the abrupt end of a 34 year long CPU *frequency race*, where the maximum CPU frequency could be described as an exponential function of time. On this date, Intel released the 3.8 GHz Intel Pentium 4 670, and we have since then not seen a single consumer-level CPU running at a higher clock frequency. Simultaneously, Intel also released the 3.2 GHz dual-core Pentium D 840, which marks the start of the *concurrency race*. We have seen a massive adaption of multi-core CPUs in both consumer-level desktops and laptops, where new CPUs offer higher performance through additional cores instead of higher clock frequencies. This fundamental change in CPU design has a deep impact on scientific computing: existing algorithms designed to scale with increasing single-core performance have now become out-of-date.

What was so special in 2005 that caused this dramatic change in architecture design, affecting the fundamental premises for efficient scientific computing? *Why* would the major CPU vendors suddenly change the highly successful strategy of increasing the clock frequency? The short answer that the power density, P_d, of CPUs is proportional to the cube of the frequency, f:

$$P_d \propto f^3. \tag{24}$$

As a though experiment, let us contemplate on what this means if we want to increase the frequency of the aforementioned Pentium 4 670 by 1 GHz. The processor runs at 3.8 GHz and has a thermal design power (TDP) of 115 watts. Using the cubic relation, we see that the 26% increase in frequency almost doubles the expected TDP to 230 watts for the CPU, which has a die area of 206 mm^2. With current cooling and production techniques, a TDP of 100–150 watts appears to be the maximum achievable for typical CPU designs, which effectively caps the maximum frequency at less than 4 GHz.

The cubic relation between frequency and power explains the end of the road for the frequency race, but it does not explain the shift to parallelism. After all, there are other ways of increasing performance of a single thread. For example, modern CPUs dedicate a lot if the die area to logic for instruction level parallelism (ILP) which increases the number of instructions per clock cycle the processor can execute. Unfortunately, techniques for ILP have been largely explored, and further developments offer diminishing returns [21].

Multi-core is the chosen path of the major silicon vendors to keep within the maximum TDP and still offer higher performance. As a rule of thumb, two cores running at 85% frequency consume the same amount of power as a single core at 100% frequency, yet offer roughly 180% performance. In fact, the first dual-core processors from Intel were nothing more than two independent Pentium 4 cores in the same socket running at a slightly lower frequency.

A benefit of decreasing the frequency is that the memory bandwidth becomes relatively higher per socket. A growing problem with the traditional serial architectures has been that all data in and out of the processor is transferred through a limited number of pins at a limited data rate connecting the CPU to the main memory. This *von Neumann bottleneck* is often referred to as the memory wall, and there has been a growing gap between the rate at which data is transferred to the processor, and the rate at which the processor executes instructions. It is obvious that further increases to processing power becomes useless at some point if you are not able to feed the processor with data fast enough. CPUs have traditionally countered

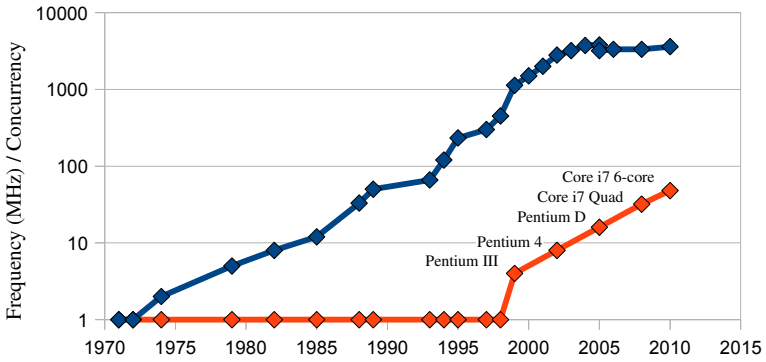


Figure 7: Frequency (upper) and single precision concurrency (lower) of consumer-level Intel CPUs 1971-2010. Streaming SIMD Extensions (SSE) was introduced with the Pentium III; Hyper-Threading Technology was introduced with Pentium 4; Multi-core was introduced with Pentium D; and quad and six-core with Core i7. Data source: Intel ARK [22]

the expanding gap between processor and memory performance by ever-growing caches that keep frequently used data in fast on-chip memory. However, increasing the size of the cache increases the latency, yielding diminishing performance gains [17].

Even though reducing the clock frequency is beneficial for the relative bandwidth, we still suffer from the increasing number of arithmetic units that need to be fed with data through the same von Neumann bottleneck. If multiple processor cores on the same chip have to fight over the same memory bandwidth, each core will experience a continued growth in the memory gap as the number of cores increases. To counter the effect of this gap, we have not only seen ever-growing caches: Intel have also implemented two-way hardware multi-threading in several processors since 2002 under the name of Hyper-Threading Technology (HTT). The processor can use these two threads to mask latencies and stalls. Once one of the threads stalls, for example on data dependency, the processor instantaneously switches to the other thread to better utilize the execution units. Hardware multi-threading in combination with SSE instructions means that the state-of-the-art Nehalem chip with six cores will utilize the hardware best given 48 simultaneous single precision operations. Figure 7 illustrates this explosion in concurrency for consumer-level processors, with an annual 25% increase over the last ten years. Extrapolating the data, this yields almost 500-way concurrency in 2020. And while there are clear challenges that might prevent this level of parallelism in CPUs, we will probably see a steady growth in parallelism of computer systems as a whole.

While CPUs have become highly parallel during the last decade, their parallelism is nothing compared to that of GPUs. GPUs were originally designed to accelerate a set of predefined graphics functions in graphics APIs such as OpenGL and DirectX, and their main task, even today, is to calculate the color of pixels on screen from complex geometries in games. According to figures in the Steam Hardware Survey [39], the average display size contains over 1.4 million pixels, which means we have at least 1.4 million elements we can calculate independently. GPUs exploit this fact, and are massively parallel processors. A minimum frame rate of 30 frames per second is considered acceptable in many games, which is also very close to the number of full frames in standard television formats such as PAL (25) and NTSC (29.97). This means that each second, we have to calculate the value of the red, green, blue, and alpha

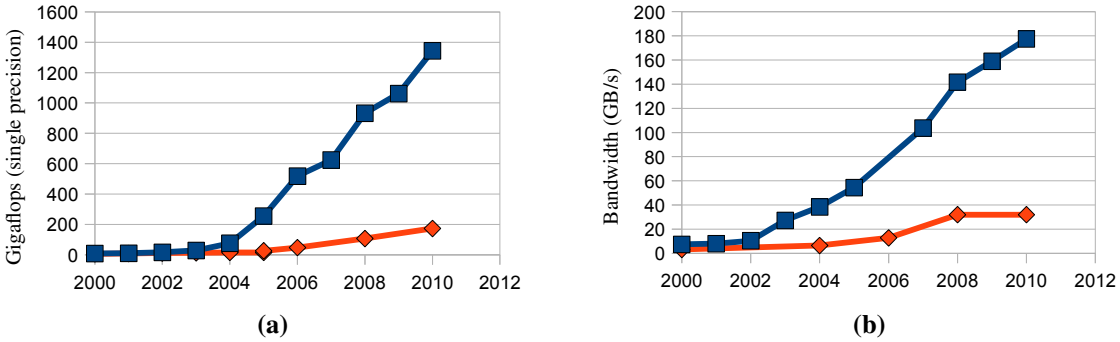


Figure 8: Single precision performance and bandwidth of GPUs (blue squares) and CPUs (red diamonds) 1990-2010. Data sources: Intel microprocessor export compliance metrics [23] and Hardware-Infos:Grafikkarten [19].

component of 1.4 million pixels at least 30 times. This high demand for throughput, and embarrassingly parallel work-load has driven the development of GPUs to their current state. While CPUs optimize for single-thread performance, GPUs optimize for net throughput.

Figure 8 shows the historic development of theoretical performance between GPUs and CPUs, which explains the massive interest in the use of GPUs for scientific computing. The state-of-the-art NVIDIA GeForce GTX 480 GPU offers a single-precision performance of almost 1.4 teraflops, or roughly eight times that of the six-core Nehalem. But this is only part of the explanation. Prior to 2007, GPUs had to be programmed using graphics languages, which required that computation was expressed as operations on graphical primitives. This process was cumbersome and error prone, and graphics restrictions severely affected the use of GPUs for many algorithms. The real revolution came in 2007, when NVIDIA offered CUDA, a C-based programming language for GPU computing. The release of CUDA lifted many restrictions and increased the interest in using GPUs for general purpose computing. When double precision was introduced in 2008, even though at only one eight of the performance of single precision, GPUs could also be used for applications with high demands to accuracy. Today, compute GPUs support double precision at half the speed of single precision, just as in regular CPUs, and they are even used in the second, seventh, 19th, and 64th fastest supercomputers in the world [36].

We have so far introduced background information that covers topics we have studied in this thesis. Starting with linear algebra, we have introduced stencil computations, the shallow water equations, mapping of computations to the GPU, and finally a motivation behind the advent of heterogeneous computing. We will now continue by presenting the main research in this thesis. In the next chapter we present a summary of our published and submitted papers, and in Chapter 3 we offer our concluding remarks.

Chapter 2

Summary of Papers

This thesis consists of research on heterogeneous architectures with a focus on the combination of a commodity CPU and a commodity GPU. We further narrow the field of scientific computing to examine numerical linear algebra and stencil computations. In this section, we give an overview of the papers that constitute the main part of this thesis. Paper I and II give an overview of heterogeneous computing and benefits and drawbacks of different architectures. In Paper III and IV, we address linear algebra on a combination of a multi-core CPU and a modern GPU. Finally, in Paper V and VI we address the efficient mapping of stencil computations to the GPU, and present implementations of different explicit schemes for computing solutions of the shallow water equations.

PAPER I: STATE-OF-THE-ART IN HETEROGENEOUS COMPUTING

A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik and O. O. Storaasli. *In Scientific Programming, IOS Press, 18(1) (2010), pp. 1-33*

This paper gives an overview of the state of the art in heterogeneous computing anno 2009, with a focus on three heterogeneous architectures: the Cell Broadband Engine, the combination of a traditional CPU and a graphics processing unit (GPU), and the combination of CPU and a field programmable gate array (FPGA). We give a review of hardware, available software tools, and an overview of state-of-the-art techniques and algorithms. We further present a qualitative and quantitative comparison of the architectures, and give our view on the future of heterogeneous computing. The following summarizes the article, with slightly updated performance figures for the GPU.

The Cell Broadband Engine is a heterogeneous processor by itself, as shown in Figure 1a. It is used in the worlds third fastest supercomputer (RoadRunner) and it is the main processor in the PlayStation 3 gaming console. It consists of one traditional CPU core, called the Power Processing Element (PPE), and eight accelerator cores called Synergistic Processing Elements (SPEs). The SPEs can run independent programs, and are vector processors that each execute the same instruction on four-long vectors (similar to the SSE instruction set on CPUs). They also have access to the same memory as the CPU core, which facilitates tight cooperation between the CPU and accelerator cores. Each SPE consists of the vector execution unit and a programmable cache, called local store. A memory flow controller is responsible for moving data between the local store and main memory and it can operate independently of the vector execution unit. This is used to overlap data movement with computation, which is a great benefit for algorithms with a predefined data access pattern. One chip running at 3.2 GHz has a peak theoretical performance of 230 gigaflops in single precision, roughly half that in double precision, and can access memory at a rate of 25.6 GB/s. Applications and algorithms such as linear algebra, sequence analysis, molecular dynamics, sorting, and stencil computations have been efficiently mapped to the Cell processor, but programming the Cell BE requires intimate knowledge of the hardware. Nevertheless, the hardware utilization is extreme for suitable algorithms: Alvaro et al. [3] are able to utilize 99.8% of the peak performance of the SPEs for single precision matrix multiplication.

The combination of a CPU and a GPU is shown in Figure 1b. The GPUs we consider are on separate circuit boards, called graphics cards, and these cards are connected to the computer through a vacant PCI express slot in a similar fashion as other peripheral components (e.g., sound cards, network cards and disk controller cards). The GPU consists of tens of vector processors that run the same program on different data, and each processor executes the same instruction on 32-long or 64-long vectors (for NVIDIA and AMD GPUs, respectively). To hide memory access and other latencies, each of the processors can hold many hardware threads active simultaneously, and switch to a ready thread once the current thread stalls. Recall from the programming example presented in Section 4 that the GPU programming model consists of blocks of threads. The blocks are automatically scheduled by the hardware to the physical processors, and cooperation between blocks is only possible through expensive global synchronization. This means that cooperation between processors is not directly supported, but

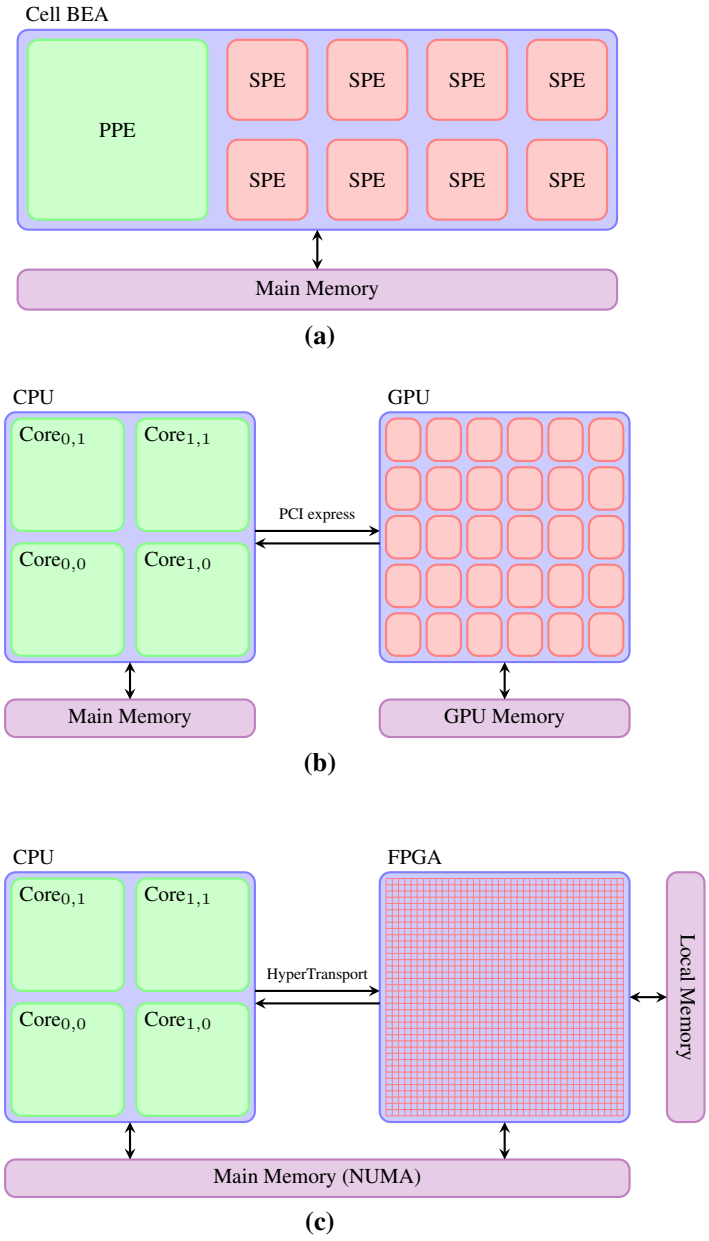


Figure 1: Overview of three heterogeneous architectures: (a) shows the Cell Broadband Engine. The Cell BE consists of nine cores all on the same chip, one Power Processing Element (PPE), and eight Synergistic Processing Elements (SPEs). The PPE is a traditional CPU core, whilst the SPEs are specialized accelerator cores. (b) shows the combination of a traditional CPU and a GPU. The GPU consists of tens of accelerator cores and is designed to hold tens of thousands of active threads. (c) shows the combination of a traditional CPU and an FPGA. The FPGA is dramatically different from the two other architectures in that it does not contain any "cores". When configured, there is a data-path through the FPGA fabric where simple operations are performed along the way.

cooperation between threads within one block is allowed. GPUs designed purely for computation have now been released on the market, and the main difference between these GPUs and those meant for the gaming market is faster double precision calculations and ECC protected memory. The most recent compute GPUs from NVIDIA have a theoretical performance exceeding one teraflops in single precision, half that in double precision, and can access up-to six GiB GPU memory at a rate of 144 GB/s. An implication of the GPU being connected to the rest of the computer system through the PCI express bus is that data transfer between the CPU and the GPU is relatively expensive. The PCI express 2.0 bus can only transfer up-to eight GB/s (bidirectional), which means that the cooperation between the CPU and the GPU is not as trivial as the cooperation on the Cell BE. The GPU has been used with great success in a wide number of application areas, including linear algebra, numerical simulation, sorting, and image processing.

The last heterogeneous architecture we examine is the combination of a CPU and an FPGA, shown in Figure 1c. FPGAs are available in many different formats, including circuit boards that are socket compatible with regular CPUs. This means that one can have a multi-socket machine where one or more sockets have an FPGA instead of a CPU. The FPGA is dramatically different from the Cell BE and the GPU: it consists of up-to 50 000 configurable logic blocks that are connected through an advanced switching network. The FPGA can be programmed using hardware description languages, higher-level C languages, or graphical languages. The resulting program is eventually synthesized to a design which can be viewed as computational data-path through the chip. The switching network determines the data path and the configurable logic blocks perform simple operations along the path. This way of computing is extremely efficient with respect to memory bandwidth, as all data is typically kept in on-chip registers. In addition to having access to main memory at the same rate as CPUs, these FPGAs can also be supplied with additional on-board memory with a slightly higher bandwidth. FPGAs were originally designed for discrete logic, and have been highly successful in streaming application areas such as telecommunications, genome sequencing, and molecular dynamics. In general, the applications performing best on an FPGA have a predefined data flow, and can often benefit from bit, fixed-precision, or mixed-precision algorithms. Floating point operations, on the other hand, are expensive and typically avoided.

Comments

The field of heterogeneous computing is currently in a state of flux: the hardware that dictates the premises for efficient algorithm and application design changes rapidly. For example, in the three and a half years since CUDA was first released in early 2007, NVIDIA have released at least three major hardware generations (Compute 1.0 which were the first GPUs to support CUDA, Compute 1.3 which added the first double precision, and Compute 2.0 which added caches). This means that even though an algorithm scales with future hardware, an implementation that is highly efficient today might be suboptimal in as short time as only a few years. This is a difficult problem, where the use of auto-tuning techniques can become important.

When the article was written, the Cell BE appeared to be a promising architecture, where one could get extremely close to the peak performance through careful programming. However, the programming complexity of this architecture has been criticized by many, and the promised performance increase for the next version has not been as impressive as for GPUs. There are

now several rumors stating that IBM has decided to discontinue work on the next version of the Cell BE, making machines such as the Roadrunner supercomputer a one-off architecture.

Intel originally planned on entering the high-performance graphics market where NVIDIA and AMD are the sole actors today. Their GPU, called Larrabee, was based on using tens of x86-compatible cores based on the Pentium architecture, with additional vector instructions. However, shortly after the article was published, Intel publicly announced the end of the development of the Larrabee as a GPU. However, the technology developed is now used in the experimental Intel Single-chip Cloud Computer [24]. The Single-chip Cloud Computer aims at the high-performance market and will compete with products such as the Tesla compute GPUs from NVIDIA.

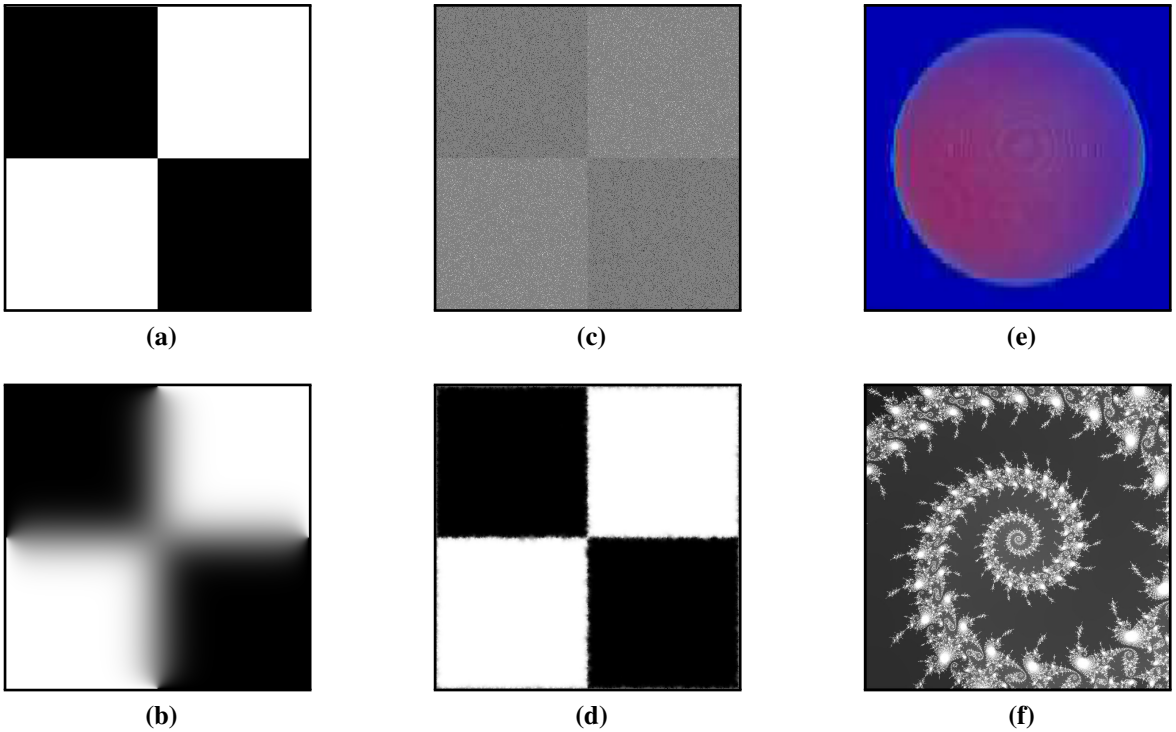


Figure 2: The four algorithms examined in Paper II. Figures (a) and (b) show the heat equation initial conditions and during simulation, respectively. Figures (c) and (d) shows an initial noisy image, and the inpainted version after applying the heat equation to missing pixels. Figure (e) shows a zoom of a single image in an MJPEG compressed movie, displaying compression artifacts. Figure (f) shows a zoom on a portion of the Mandelbrot set.

PAPER II: A COMPARISON OF THREE COMMODITY-LEVEL PARALLEL ARCHITECTURES: MULTI-CORE CPU, CELL BE AND GPU

A. R. Brodtkorb and T. R. Hagen. *In proceedings of the Seventh International Conference on Mathematical Methods for Curves and Surfaces, Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, 5862 (2010), pp. 70–80*

In this paper, we examine the implementation and performance of four algorithms on a multi-core CPU, a GPU, and the Cell BE. The aim of the article is not to present an optimized implementation of each algorithm, but to give an overview of benefits and drawbacks of each architecture. The first algorithm solves the heat equation using an explicit finite difference scheme, the second uses the heat equation to fill in missing pixels in images, the third performs MJPEG movie compression, and fourth computes part of the Mandelbrot set. Figure 2 shows screenshots of each of these four implementations.

The heat equation application shows how efficient each architecture is at streaming data with a predefined and regular data access pattern. This should thus be a memory bound work-load, which displays properties found when solving other PDEs using explicit finite difference/element/volume methods and performing image filtering. In this application, the CPU benefits from its large hardware cache and automatic prefetching, the GPU benefits from the use of shared memory, and the Cell BE benefits from overlapping data movement and computation.

The next application uses the heat equation to fill inn missing pixel values in an image, and shows how well each architecture handles a memory bound and data-dependent work-load. In addition to reading the values in the heat equation, we here also read a mask that determines whether or not an element requires any computation. This application displays properties also found in algorithms with a lot of memory access and few computations, such as image processing algorithms working on masked areas. Here, the CPU saturates the memory bandwidth, and using multiple cores yields only a marginal speedup. The GPU is penalized from the SIMD execution, meaning it takes the same time to solve the heat equation on all elements as it does to compute on only a subset. The Cell BE is heavily penalized for the branching, as the processor does not have a hardware branch predictor.

The third application computes part of the Mandelbrot set. This application requires only one memory access per element, and performs potentially thousands of operations, thus showing how well each architecture performs floating point operations. For two neighbouring pixels, the number of required iterations can be dramatically different, meaning the work-load is data-dependent. Here, the CPU scales perfectly with the number of cores. However, both the GPU and the Cell BE outperform the CPU dramatically as their floating-point capabilities are vastly superior.

The final application performs MJPEG compression of a movie stream. The main part of this application computes the discrete cosine transform, which is a computationally bound uniform work-load with heavy use of trigonometric functions. This is representative for a range of applications such as image registration and Fourier transforms. The GPU benefits from fast, albeit less accurate, trigonometric functions, and the Cell BE benefits from being able to overlap memory transfers and computation.

Of the three architectures examined, the GPU came out with the best performance for the computationally bound algorithms, and the Cell BE performed best for the memory bound algorithms. The CPU scales well for the computationally bound algorithms, but takes a performance hit when the memory bus gets saturated by multiple cores. Furthermore, it does not have floating point capabilities to compete with the GPU and Cell BE. Even though the GPU is the best performing architecture for computationally bound algorithms, it is heavily penalized for the slow PCI express bus connecting GPU memory to main memory.

Comments

The work presented in this article gives an introduction to using the GPU and Cell BE for scientific computing, and illustrates some of the benefits and draw-backs of each architecture. Even though the Cell BE today is a much less interesting architecture than at the time of writing the article, our work clearly illustrates the benefits of its asynchronous memory transfers.

If the topic of the article was to be reexamined today, it would be interesting to focus on the difference between GPUs from NVIDIA and AMD. GPUs from NVIDIA have become the GPU choice for general purpose computation due to their early release of the high-level CUDA language, and high market penetration with CUDA capable GPUs. Today, however, both AMD and NVIDIA support OpenCL, which is an alternative to CUDA that is compatible with GPUs from both vendors. GPUs from both vendors further support DirectCompute, which is a proprietary Microsoft API. Comparing the GPUs using these programming languages would be interesting to exemplify benefits and drawbacks of the different architectures.

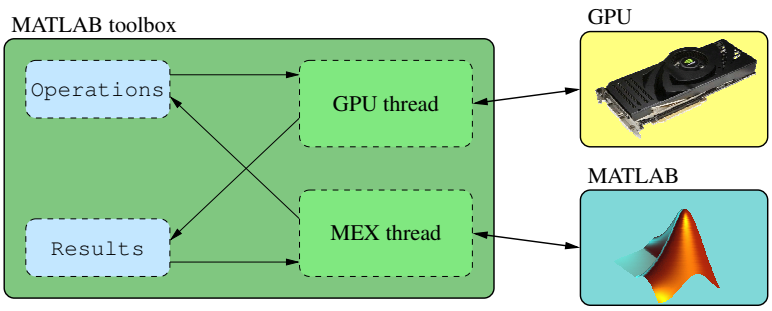


Figure 3: The interface between MATLAB and the GPU that enables asynchronous execution. The MEX thread holds the MATLAB context, and the GPU thread holds the GPU context. The two threads communicate using a queue of operations and a map of computed results. This facilitates the simultaneous use of both the GPU and the CPU.

PAPER III: A MATLAB INTERFACE TO THE GPU

A. R. Brodtkorb. *In proceedings of The Second International Conference on Complex, Intelligent and Software Intensive Systems, IEEE Computer Society, (2008), pp. 822–827*

This paper presents the implementation of several algorithms from linear algebra on GPUs, and how these can be transparently used in MATLAB. The paper continues the work presented in "A MATLAB Interface to the GPU" [7], and we here show how the GPU can be used to accelerate Gaussian elimination, tridiagonal Gaussian elimination, LU factorization, and matrix multiplication in MATLAB without sacrificing the high-level syntax. We further demonstrate asynchronous CPU and GPU execution for full utilization of available hardware resources, and report a maximum speed-up of 31 times over a highly tuned CPU code.

Since this research predates CUDA, it required that the algorithms we implemented were mapped to operations on graphical primitives. In our approach to Gaussian elimination, LU factorization, and matrix multiplication, we thus use two-by-two matrices as the basic unit by packing four elements into the $[r, g, b, a]$ color vector:

$$\begin{bmatrix} r & g \\ b & a \end{bmatrix}.$$

This means that for Gaussian elimination and LU factorization, we factorize in a similar way as panel, or blocked, factorization is performed. For our tridiagonal Gaussian elimination, we store the three non-zero elements in each row of the matrix together with the right hand side, i.e.,

$$\begin{bmatrix} g_1 & b_1 & 0 & 0 & 0 \\ r_2 & g_2 & b_2 & 0 & 0 \\ 0 & r_3 & g_3 & b_3 & 0 \\ 0 & 0 & r_4 & g_4 & b_4 \\ 0 & 0 & 0 & r_5 & g_5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & g_1 & b_1 & a_1 \\ r_2 & g_2 & b_2 & a_2 \\ r_3 & g_3 & b_3 & a_3 \\ r_4 & g_4 & b_4 & a_4 \\ r_5 & g_5 & 0 & a_5 \end{bmatrix}.$$

Solving this system is ultimately a serial process. To benefit from the massive parallelism of GPUs, we can solve many such independent systems in parallel. As one example of an

application that results in a large number of independent tridiagonal matrices, we present the implementation of an alternating direction implicit scheme for the shallow water equations.

Our implementation requires detailed knowledge of graphics hardware and APIs, but all these details are hidden from the user through the high-level MATLAB abstraction. We implement a new class in MATLAB, called `gpuMatrix`, and use operator overloading to automatically execute the implemented algorithms. The traditional way of utilizing GPUs for general purpose computation leaves the CPU idling whilst the GPU is computing. This is highly wasteful, and we address the issue by executing all operations asynchronously on the GPU. This is done using a queue of operations and a map of results, illustrated in Figure 3, and is completely transparent to the user. The only time the CPU and the GPU synchronize is when the user converts from our `gpuMatrix` to a single or double precision MATLAB matrix.

Comments

This work presented the very first use of GPU compute resources from MATLAB. We demonstrated that it was possible to accelerate operations and utilize both the CPU and the GPU simultaneously from MATLAB with minimal changes to the source code. Today, this concept has been commercialized by AccelerEyes through their tool Jacket [1], and their approach shows great resemblance to what we presented. Furthermore, the Mathworks, makers of MATLAB, have also called for participants to a GPU computing beta program, presumably to add GPU computing features into MATLAB in the near future.

At the time when this paper was written, one could only program the GPU using graphics languages. This meant that linear algebra operations had to be mapped to operations on graphical primitives, a delicate and error prone process. Furthermore, the hardware at that time was optimized for operations on 4-long vectors with the rationale that both colors and positions of vertices is represented with four values. This meant that you had to adjust the computations not only to the restrictions imposed by the graphics API being used, but also by the hardware details. Another architectural difficulty with GPUs of that time was missing support for *scattered write*, the ability to write to random locations in memory. GPUs did not support this since it does not make sense when computing the color of a pixel: the output position of the pixel should never be allowed to change.

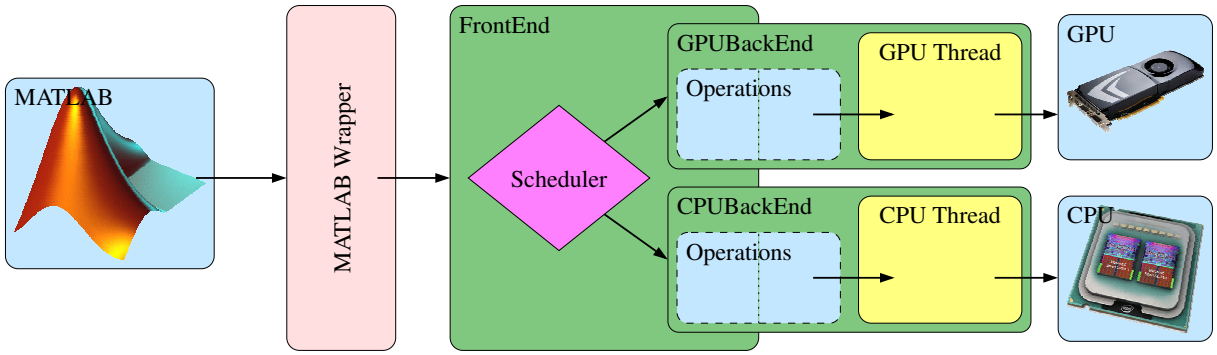


Figure 4: The interface between MATLAB and our different back-ends. The relationship between the front-end and each of the back-ends is done in a similar fashion as shown in Figure 3, using a queue of operations and a map of results. The scheduler selects the most suitable back-end for each incoming operation based on two criteria: processor load, and data location.

PAPER IV: AN ASYNCHRONOUS API FOR NUMERICAL LINEAR ALGEBRA

A. R. Brodtkorb. *In Scalable Computing: Practice and Experience, West University of Timisoara, 9(3) (Special Issue on Recent Developments in Multi-Core Computing Systems) (2008), pp. 153–163.*

In this paper, we continue the work of Paper III, and develop an API to execute linear algebra operators asynchronously in a task-parallel fashion on heterogeneous and multi-core architectures. In this updated approach, we use existing low-level linear algebra libraries, and develop a scheduling technique to best utilize different processing units.

Figure 4 illustrates our approach, where we offer the use of heterogeneous compute resources transparently from MATLAB. The implementation consists of three different parts: a MATLAB wrapper, a frontend, and a backend. The MATLAB wrapper creates the link between the high-level MATLAB syntax and our frontend, and the frontend consists of a scheduler that is connected to several different backends. The backends are designed in a similar way as in our previous approach (see Figure 3), where a queue of operations and a map of results is used for communication.

The scheduler in the frontend is responsible for grouping incoming tasks with its dependencies into a cluster, and to find the optimal backend to schedule this cluster to. We use two criteria to give an estimate of how good a candidate each backend is, and then schedule the cluster to the backend with the best score. The first criterion is based on the load for each backend, where we estimate the load by the sum of the historic processor load and the estimated future load of tasks already in the operations queue. The second criterion considers where existing dependencies have been scheduled: if the operation is to multiply two matrices that reside on the GPU, it makes little sense to transfer these matrices over the PCI express bus to the CPU. Thus, we give a penalty for transferring matrices between processors.

The backends are implemented by utilizing optimized architecture-specific BLAS libraries. For the CPU version, we utilize a single-threaded ATLAS implementation of BLAS, and for the GPU we use CUBLAS. The use of a standard API with efficient implementations for a wide range of architectures makes our approach applicable for a wide range of heterogeneous architectures.

Our benchmarking results show that the implementation only imposes small overheads related to scheduling, and we achieve near perfect weak scaling when going from one to two CPU cores. Using the GPU yields even higher performance, but our scheduler makes suboptimal decisions when scheduling to both the GPU and the CPU. We also present double precision results, and compare results computed by the CPU with results computed by the GPU. Our experiments show that the difference between naive matrix multiplication on the CPU and CUBLAS is negligible and does not grow with the matrix size.

Comments

This paper presented, to our knowledge, the first approach to using an asynchronous execution model for linear algebra operations on heterogeneous architectures. We further presented our experiences with double precision calculations on early engineering samples of the first GPUs to support double precision. Before double precision was supported in GPU hardware, there was a great scepticism towards using GPUs, but today there is an increasing acceptance for the use of GPUs for scientific computing.

The proof-of-concept scheduler presented in this paper has one flaw that we have yet to resolve. When faced with a set of heterogeneous backends, the scheduler makes suboptimal choices by scheduling too many tasks to the slower processor. However, we believe that the solution to this issue can be found by tweaking the scheduling parameters, and that the presented scheduler uses a suitable set of criteria. The use of asynchronous execution on todays architectures is potentially a very good idea. For algorithms where one can maintain a queue of operations, our approach can be used to ensure a high utilization of compute resources.

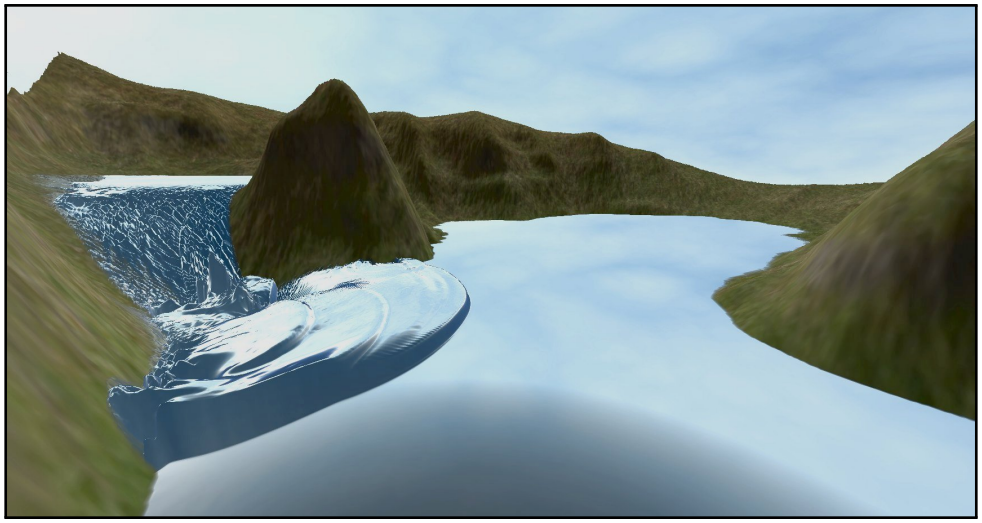


Figure 5: Simulation of a dam-break in an artificial terrain with direct visualization of simulation results.

PAPER V: SIMULATION AND VISUALIZATION OF THE SAINT-VENANT SYSTEM USING GPUS

A. R. Brodtkorb, T. R. Hagen, K.-A. Lie, and J. R. Natvig. *In Computing and Visualization in Science, Springer-Verlag Berlin Heidelberg, (special issue on Hot Topics in Computational Engineering), (2010). [in press]*

This paper presents the efficient implementation of three explicit finite difference schemes for the shallow water equations. The three schemes are second-order accurate, well-balanced, and support dry states. This makes them highly suitable for use in simulation of physical phenomenon such as flooding, tsunamis, river flow, and dam breaks. They also perform many arithmetic operations per memory access, and are thus less memory bound than many other schemes. They are furthermore embarrassingly parallel, making them suitable for GPU implementation. We analyze the performance of the schemes, and the effect of rounding errors imposed by single versus double precision.

Handling of dry zones in the shallow water equations is difficult for numerical schemes because a single negative water depth will ruin the whole simulation. Two of the schemes, Kurganov-Levy and modified Kurganov-Levy, are based on switching between different reconstructions to handle dry zones. For the dry zones and areas with small water depths, a reconstruction that trades the well-balancedness for support for dry zones is used. The last scheme, Kurganov-Petrova, uses a different strategy, where the reconstruction is altered only for problematic cells. This scheme also handles discontinuous bottom topographies gracefully, whereas the two former have difficulties simulating on such domains.

Our implementations of the three schemes use a block domain decomposition, where we partition the domain into equally sized blocks and let CUDA automatically handle the scheduling. The blocks are executed independently, but all the cells within one block can communicate and cooperate. We exploit this by storing data required by the stencil for all cells within the same block in on-chip shared memory.

We present benchmarks of the three schemes where we both analyze the performance and the effect of single precision rounding errors. Our experiments show that the Kurganov-Petrova scheme is the scheme of choice both with respect to numerical accuracy and performance. Our experiments with double precision further show that the error caused by the handling of dry zones is far greater than the error caused by single precision arithmetics. This means that single precision is sufficiently accurate for the types of cases that these schemes were designed to handle, and initial experiments with mixed-precision, where only parts of the algorithm are calculated in double, show promising results.

In addition to implementing the numerical schemes, we also present direct visualization of the results using OpenGL. By copying data from CUDA to OpenGL, we can offer visualization of the simulation without the data ever leaving the graphics card. This is a major benefit as the bandwidth between the GPU and the CPU is limited. Our visualization, shown in Figure 5, uses the Fresnel equations to render the water surface. The water reflection is colored using environment mapping, yielding a mirror-like surface where discrepancies are easily spotted.

Comments

This is a proof-of-concept simulator which is capable of simulating shallow-water flow in real-time. However, the implementation lacks physical friction terms, and is thus not suitable for real-world terrains. This means that we can only handle toy problems. However, the focus has been on accuracy and on implementation efficiency, and we clearly demonstrate that these schemes are well suited for GPU implementation and single precision arithmetics. We continue the work presented in this article in Paper VI.

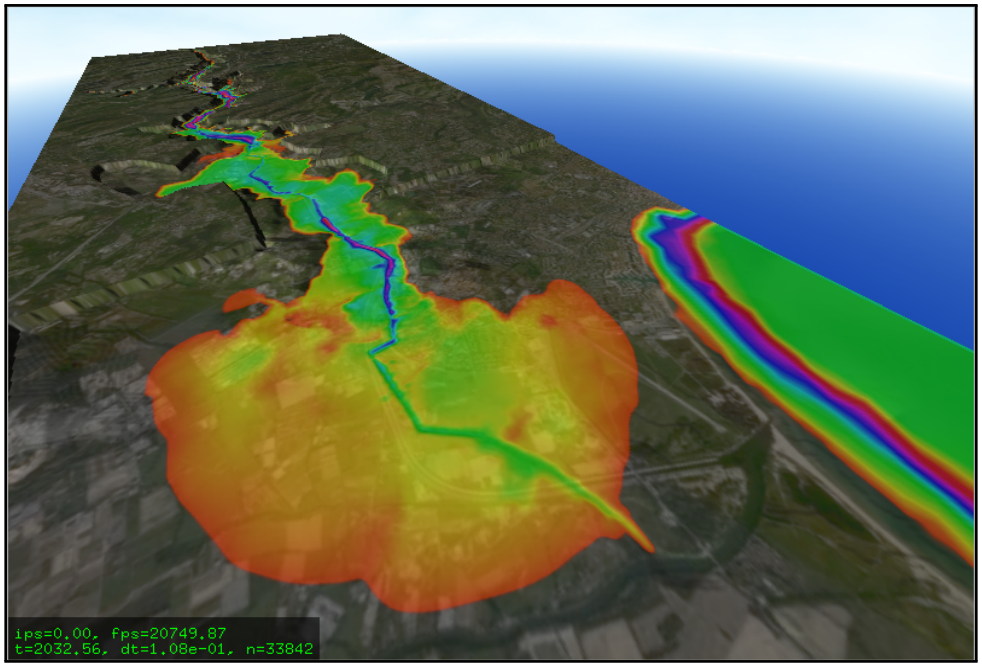


Figure 6: Simulation of the Malpasset dam break in south-eastern France where the initial flood wave caused over 420 casualties.

PAPER VI: EFFICIENT SHALLOW WATER SIMULATIONS ON GPUS: IMPLEMENTATION, VISUALIZATION, VERIFICATION, AND VALIDATION

A. R. Brodtkorb, M. L. Sætra, and M. Altinakar. *In review, 2010*

This paper builds on Paper V, and we here concentrate on the Kurganov-Petrova scheme. We have chosen to focus on this scheme as it was found to be the most promising of the schemes examined in Paper V. We present novel optimizations, verification and validation, a dramatic decrease in memory footprint, physical friction terms, multiple time integrator schemes, efficient implementation of boundary conditions, and a more efficient data path for visualization.

The most important optimization presented is an early exit strategy, where parts of the domain without water do not perform computation. We exploit the explicit data dependencies of the stencil, the blocking execution model of CUDA, and a small auxiliary buffer to dramatically increase performance. The auxiliary buffer stores whether or not a block contains water, and if the current block and its four closest neighbouring blocks are dry, we can safely exit without performing any of the complex computations.

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 6). Our implementation accurately predicts both the wave arrival time and maximum water depth for this real-world case.

Comments

This paper addresses the shortcomings of our first paper on shallow water simulations, and our new simulator is capable of accurately capturing real-worlds flows. We have several interesting ideas for future research.

We have already developed a prototype multi-GPU implementation based on the simulator presented in this paper. The implementation uses row decomposition, where the exchange of information between domains is shown to have a minimal performance penalty. Our implementation further shows near perfect weak scaling. Extending this multi-GPU solution to a truly heterogeneous implementation, with a domain decomposition between multiple GPUs *and* CPU cores is a relevant research direction. This will require further research into efficient load balancing between the heterogeneous compute resources.

Another extension of this work is to implement efficient mixed boundary conditions per edge. Our current boundary conditions can be used to represent phenomenon such as storm surges and tsunamis, but are unsuitable for representing a river inlet, for example. This is potentially a difficult task, and we sketch some preliminary ideas in the paper.

Chapter 3

Summary and Outlook

In the previous two chapters, we introduced the main research topics and summarized the contributions of the scientific papers. This chapter gives a brief summary, and our view on the future of heterogeneous computing.

Our work has focused on the efficient use of heterogeneous architectures for scientific computing, and we have presented six papers within this topic. Our focus has been on the combination of a CPU and a GPU, and we have shown that this heterogeneous architecture can dramatically increase the speed of linear algebra and stencil computations. We have given a thorough review of several heterogeneous architectures, and exemplified architectural differences using four algorithms. We have further illustrated that GPUs can accelerate the process of solving systems of linear equations, and that asynchronous execution can be used to increase performance. Finally, we have shown that careful implementation of stencil computations on GPUs can yield fast and efficient shallow water simulations, capable of capturing real-world flows.

The Future of Heterogeneous Computing

The field of heterogeneous computing has become increasingly popular over the last ten years. From the initial academic proof-of-concept applications using graphics APIs, we have today industrial, government and military applications that run on heterogeneous architectures. Even some of the fastest supercomputers in the world are heterogeneous, including three in the top ten. If we rank the worlds fastest supercomputers according to their performance per watt, eight of the top ten are heterogeneous, and these heterogeneous systems use approximately one third of the power of traditional CPU-based systems [37]. Since power consumption is a major constraint for supercomputers, it is likely that we will see a growing number of such architectures in high-performance computing.

The use of heterogeneous architectures is not only reflected in supercomputers. The hardware and programming platform is already deployed to millions of users, and the software ecosystem is slowly adapting. A significant number of applications currently use GPUs to accelerate general purpose computations, including commercial video editing, image processing, mathematical, and medical software, to name a few. We expect that the use of GPUs in standard software will increase dramatically over the next few years, especially now that OpenCL and DirectCompute can be used to implement algorithms on both AMD and NVIDIA GPUs.

The first work we present on using GPUs for general purpose computing in this thesis re-

quired the use of graphics programming languages. At that time, support for floating point arithmetic on GPUs was still new, and code written for one GPU would typically not work on any other GPUs. When asking graphics card manufacturers about double precision support in GPUs, we were told that it was not even on the horizon: single precision is more than sufficiently accurate to calculate the color of pixels in games. We did not even hope to think that a general purpose language such as CUDA would be released any time soon. The development of GPUs has thus far exceeded our expectations, and we have seen restrictions rapidly disappear. With this rapid development, it is difficult to predict where the field of heterogeneous computing will go in the future.

The use of GPUs for scientific applications has so far benefited greatly the symbiotic and self-amplifying development of computer games and GPUs. Recently, however, we have seen GPUs incorporate features that do not make sense for games, such as double precision. A prerequisite for the future development of GPU hardware for scientific and general purpose computing is high production numbers to justify development costs. With the increasing number of applications that today start supporting GPU acceleration, we might end up in a situation where commodity computers, not only those aimed at the gaming market, will incorporate powerful GPUs. No matter how the hardware evolves, it will be an exciting future for heterogeneous computing that requires continuous research: on hardware, software, and algorithms.

Bibliography

[1] Accelereyes. Jacket for MATLAB. `http://www.accelereyes.com/`. [visited 2010-07-21].

[2] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1):012037, 2009.

[3] W. Alvaro, J. Kurzak, and J. Dongarra. Fast and small short vector SIMD matrix multiplication kernels for the Synergistic Processing Element of the Cell processor. In *Intl. Conf. on Computational Science*, pages 935–944, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] AMD. AMD core math library (ACML) version 4.4.0. `http://www.amd.com/acml`. [visited 2010-07-21].

[5] AMD. AMD core math library for graphic processors user guide for version 1.1. `http://developer.amd.com/gpu/acmlgpu/`, July 2010. [visited 2010-07-21].

[6] 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.

[7] A. R. Brodtkorb. A MATLAB interface to the GPU. Master's thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, May 2007.

[8] A. R. Brodtkorb. An asynchronous API for numerical linear algebra. *Scalable Computing: Practice and Experience: special issue on Recent Developments in Multi-Core Computing Systems*, 9(3):153–163, 2008.

[9] A. R. Brodtkorb. The graphics processor as a mathematical coprocessor in MATLAB. In *The Second International Conference on Complex, Intelligent and Software Intensive Systems*, pages 822–827. IEEE CS, 2008.

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

[11] 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*, to appear, 2010.

[12] A. R. Brodtkorb, M. L. Sætra, and M. Altinakar. Efficient shallow water simulations on gpus: Implementation, visualization, verification, and validation, 2010. [In review].

[13] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[14] J. Dongarra. Basic Linear Algebra Subprograms Technical forum standard. *High Performance Applications and Supercomputing*, 16:1–111, 2002.

[15] J. Dongarra and J. Wasniewski. High performance linear algebra package - lapack90. In *Parallel and Distributed Processing*, volume 1388/1998. Springer Berlin / Heidelberg, 1998.

[16] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. Stewart. *LINPACK Users' Guide*. SIAM, 1979.

[17] U. Drepper. What every programmer should know about memory. `http://people.redhat.com/drepper/cpumemory.pdf`, November 2007. [visited 2009-03-20].

[18] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[19] Hardware-Infos. Grafikkarten. `http://hardware-infos.com/grafikkarten_nvidia.php`. [visited 2010-07-30].

[20] A. Harten. High resolution schemes for hyperbolic conservation laws. *Journal of Computational Physics*, 49(3):357–393, 1983.

[21] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition edition, 2007.

[22] Intel. Ark | your source for information on intel products. `http://ark.intel.com/`. [visited 2010-07-30].

[23] Intel. Intel microprocessor export compliance metrics. `http://www.intel.com/support/processors/sb/CS-017346.htm`. [visited 2010-07-30].

[24] Intel. Tera-scale computing research program. `http://www.intel.com/go/terascale`. [visited 2010-07-30].

[25] Intel. Intel math kernel library (Intel MKL) 10.2. `http://software.intel.com/en-us/intel-mkl/`, 2009. [visited 2010-07-21].

[26] Khronos OpenCL Working Group. The OpenCL specification 1.0. `http://www.khronos.org/registry/cl/`, 2008. [visited 2009-03-20].

[27] D. B. Kirk and W.-M. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

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

[29] C. D. Meyer. *Matrix Analysis and Applied Linear Algebra*. Society for Industrial and Applied Mathematics, 2001.

[30] Microsoft. DirectX: Advanced graphics on windows. `http://msdn.microsoft.com/directx`. [visited 2009-03-31].

[31] NVIDIA. CUDA CUBLAS library version 3.1, May 2010.

[32] NVIDIA. NVIDIA CUDA C best practices guide version 3.1, May 2010.

[33] NVIDIA. NVIDIA CUDA C programming guide version 3.1, May 2010.

[34] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[35] P. K. Sweby. High resolution schemes using flux limiters for hyperbolic conservation laws. *Siam Journal of Numerical Analysis*, 21(5):995–1011, 1984.

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

[37] Top green500 list. `http://www.green500.org/`, June 2010. [visited 2010-07-31].

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

[39] Valve Corporation. Steam hardware survey. `http://store.steampowered.com/hwsurvey/`, 2010 June. [visited 2010-07-21].

[40] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98*, pages 1–27. IEEE CS, 1998.

Part II
Scientific Papers

PAPER I

STATE-OF-THE-ART IN HETEROGENEOUS COMPUTING

A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik and O. O. Storaasli

In Scientific Programming, IOS Press, 18(1) (2010), pp. 1-33

Abstract: Node level heterogeneous architectures have become attractive during the last decade for several reasons: compared to traditional symmetric CPUs, they offer high peak performance and are energy and cost efficient. With the collapse of the serial programming paradigm for high-performance computing, there is an acute need for a good overview and understanding of these architectures. We give an overview of the state-of-the-art in heterogeneous computing, focusing on three commonly found architectures: the Cell Broadband Engine Architecture, graphics processing units (GPUs), and field programmable gate arrays (FPGAs). We give a review of hardware, available software tools, and an overview of state-of-the-art techniques and algorithms. We further present a qualitative and quantitative comparison of the architectures, and give our view on the future of heterogeneous computing.

1 Introduction

The goal of this article is to provide an overview of node-level heterogeneous computing, including hardware, software tools, and state-of-the-art algorithms. Heterogeneous computing refers to the use of different processing cores to maximize performance, and we focus on the Cell Broadband Engine Architecture (CBEA) and CPUs in combination with either graphics processing units (GPUs) or field programmable gate arrays (FPGAs). These new architectures break with the traditional evolutionary processor design path, and pose new challenges and opportunities in high-performance computing.

Processor design has always been a rapidly evolving research field. The first generation of digital computers were designed using analog vacuum tubes or relays and complex wiring, such as the Atanasoff-Berry Computer [1]. The second generation came with the digital transistor, invented in 1947 by Bardeen, Brattain, and Shockley, for which they were awarded the 1956 Nobel prize in physics. The transistor dramatically reduced space requirements and increased speed of logic gates, making computers smaller and more power efficient. Noyce and Kilby independently invented the integrated circuit in 1958, leading to further reductions in power and space required for third generation computers in the early 1960-ies. The integrated circuit rapidly led to the invention of the microprocessor by Intel engineers Hoff, Faggin, and Mazor [2] in 1968, the TMS1802NC from Texas Instruments [3], and the Central Air Data Computer CPU by Holt and Geller [4] working for AiResearch. The trend to decrease space and power requirements continues even today, with smaller feature sizes with every new production technique. As size and power consumption per logic gate have been reduced, there has been a proportional growth in computational power. The two main contributors are the number of transistors and the frequency they run at.

In 1965, Moore predicted that the number of transistors inexpensively placed on a single chip would double every two years [5], a trend followed for over thirty years [6]. With a state-of-the-art 32 nm production process, it is now possible, though not inexpensive, to place 16 *billion* transistors on a single chip [7, p. 92], and there are currently no signs suggesting that this exponential growth will stop.

Traditionally, the processor frequency closely follows Moore's law. However, physical constraints have stopped and even slightly reversed this exponential frequency race. A key physical constraint is power density, often referred to as the *power wall*. Kogge et al. [7, p. 88] state the relationship for power density as

$$P = C\rho f V_{dd}^2, \tag{1}$$

where P is the power density in watts per unit area, C is the total capacitance, ρ is the transistor density, f is the processor frequency, and V_{dd} is the supply voltage. The frequency and supply voltage are related, as higher supply voltages allow transistors to charge and discharge more rapidly, enabling higher clock frequencies. It should be noted that the formula ignores leakage power, which typically amounts to 30-40% of the total power consumption [7, p. 93]. The power density in processors has already surpassed that of a hot plate, and is approaching the

Communicating author: André R. Brodtkorb <Andre.Brodtkorb@sintef.no>
Part of this work is done under Research Council of Norway project number 180023 (Parallel3D) and 186947 (Heterogeneous Computing). Dr. Storaasli's research contributions were sponsored by the Laboratory Directed Research & Development Program of Oak Ridge National Laboratory managed by UT-Battelle for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725.

physical limit of what silicon can withstand with current cooling techniques. Continuing to ride the frequency wave would require new cooling techniques, for example liquid nitrogen or hydrogen, or new materials such as carbon nanotubes. Unfortunately, such solutions are currently prohibitively expensive.

The combination of Moore's law and the power wall restraints processor design. The frequency cannot be increased with today's technology, so performance is primarily boosted by increased transistor count. The most transistor-rich CPU yet, the Intel Itanium Tukwila [8], uses two *billion* transistors. It is increasingly difficult to make good use of that many transistors. It is the trade-off between performance gain and development cost that has evolved processors into their current design, and most transistors are currently devoted to huge caches and complex logic for instruction level parallelism (ILP). Increasing the cache size or introducing more ILP yields too little performance gain compared to the development costs.

The "rule of thumb" interpretation of (1) is that if you decrease the voltage and frequency by one percent, the power density is decreased by three percent, and the performance by 0.66 percent. Thus, dual-core designs running at 85% of the frequency with 85% of the supply voltage offer 180% better performance than single-core designs, yet consume approximately the same power. Consequently, major silicon vendors now spend their transistor budget on symmetric multi-core processors for the mass market. This evolutionary path might suffice for two, four, and perhaps eight cores, as users might run that many programs simultaneously. However, in the foreseeable future we will most likely get hundreds of cores. This is a major issue: if silicon vendors and application developers cannot give better performance to users with new hardware, the whole hardware and software market will go from selling new products, to simply maintaining existing product lines [9].

Todays multi-core CPUs spend most of their transistors on logic and cache, with a lot of power spent on non-computational units. Heterogeneous architectures offer an alternative to this strategy, with traditional multi-core architectures in combination with accelerator cores. Accelerator cores are designed to maximize performance, given a fixed power or transistor budget. This typically implies that accelerator cores use fewer transistors and run at lower frequencies than traditional CPUs. Complex functionality is also sacrificed, disabling their ability to run operating systems, and they are typically managed by traditional cores to offload resource-intensive operations.

Algorithms such as finite-state machines and other intrinsically serial algorithms are most suitable for single-core CPUs running at high frequencies. Embarrassingly parallel algorithms such as Monte Carlo simulations, on the other hand, benefit greatly from many accelerator cores running at a lower frequency. Most applications consist of a mixture of such serial and parallel tasks, and will ultimately perform best on heterogeneous architectures. The optimal type and composition of processors, however, will vary from one application to another [10, 11].

With the recent emphasis on green computing, it becomes essential to use all possible resources at every clock cycle, and the notion of *green algorithms* is on the doorstep. Both academia and industry realize that serial performance has reached its zenith, leading to an increased focus on new algorithms that can benefit from parallel and heterogeneous architectures. Further insight into these different architectures, and their implications on algorithm performance, is essential in algorithm design and for application developers to bridge the gap between peak performance and experienced performance. The field of heterogeneous computing covers

a large variety of architectures and application areas, and there is currently no unified theory to encompass it all. Fair comparisons of the architectures are therefore difficult. Our goal is to give thorough comparisons of the CBEA and CPUs in combination with GPUs or FPGAs, and to contribute new insight into the future of heterogeneous computing.

We begin with an overview of traditional hardware and software in Section 2, followed by a review of the three heterogeneous architectures in Section 3. In Section 4 we discuss programming concepts, followed by an overview of state-of-the-art algorithms and applications in Section 5. Finally, we conclude the article in Section 6, including our views on future trends.

2 Traditional Hardware and Software

In this article, we use the term chip to denote a single physical package, and core to denote a physical processing core. The term processor has, unfortunately, become ambiguous, and may refer to a chip, a core, or even a virtual core. We begin by describing parallelism and memory hierarchies, and then give an overview of relevant CPU architectural details, and shortfalls revealing why they cannot meet future performance requirements. Finally, we give a short summary of programming concepts and languages that form a basis of knowledge for heterogeneous computing.

There are multiple layers of parallelism exposed in modern hardware, including the following:

Multi-chip parallelism is having several physical processor chips in a single computer sharing resources, in particular system memory, through which relatively inexpensive communication is done.

Multi-core parallelism is similar to multi-chip parallelism, but the processor cores are contained within a single chip, thus letting the cores share resources like on-chip cache. This makes communication even less expensive.

Multi-context (thread) parallelism is exposed within a single core when it can switch between multiple execution contexts with little or no overhead. Each context requires a separate register file and program counter in hardware.

Instruction parallelism is when a processor can execute more than one instruction in parallel, using multiple instruction units.

Current CPUs use several of these techniques to decrease cycles per instruction and to hide memory latency. Examples include hardware pipelining, where multiple instructions are simultaneously in the pipeline; vector parallelism, where an instruction is replicated over an array of arithmetic units; and superscalar processors, where multiple instructions are dispatched to different execution units either automatically in hardware, or explicitly using very long instruction words (VLIWs). Techniques that target the memory latencies are also plentiful. Out-of-order execution reorders an instruction stream to minimize the number of processor stalls caused by latencies of data dependencies. Hardware multi-threading lets a set of execution contexts share the same execution units. The CPU instantaneously switches between these contexts when memory requests stall, decreasing the impact of latencies. This should not be confused with software threads, where the different execution contexts typically are stored in main memory.

Such techniques are usually combined, and make program execution complex and hard to predict.

Traditionally, floating-point operations were considered expensive, while retrieving data was practically free. However, this conventional wisdom has changed [11], and memory access has grown to become the limiting factor in most applications. Data is moved in or out of the processor using a limited number of pins running at a limited frequency, referred to as the *von Neumann bottleneck* [12]. In addition, there is a significant latency to initiate data transfers. From 1980 to 1996, the gap between processor and memory performance grew annually by 50% [13]. To bridge this gap, large memory hierarchies have been developed, addressing both the von Neumann bottleneck and memory latency by copying requested data to faster memory. This enables rapid access to recently used data, increasing performance for applications with regular memory access. In the following, we classify a memory hierarchy according to latency:

Registers are the closest memory units in a processor core and operate at the same speed as the computational units.

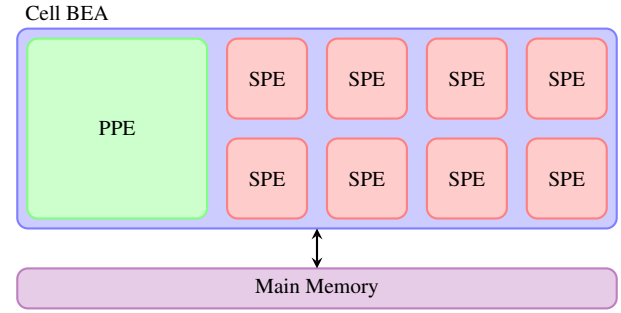
Local store memory, or scratchpad memory, resembles a programmable cache with explicit data movement. It has a latency of tens of clock cycles.

Caches have rapidly grown in size and number. On modern CPUs, there is typically a hierarchy of two or three layers that operate with a latency of tens of clock cycles. Off-chip caches, also found in some computers, have a latency somewhere between on-chip cache and main memory.

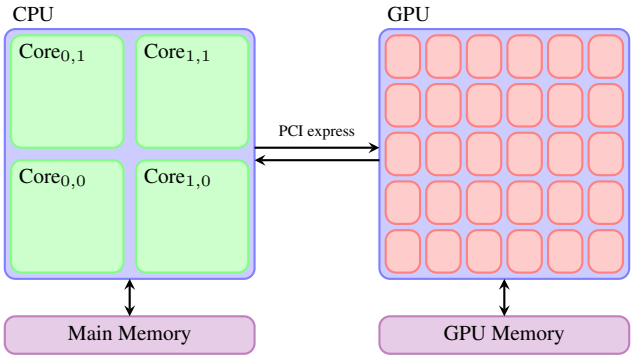
Main memory has a latency of hundreds of clock cycles, and is limited in bandwidth by the von Neumann bottleneck.

The recent end of the frequency race, mentioned in the introduction, has caused the relative increase in latency to halt, a positive side effect. The von Neumann bottleneck, however, continues to be a burden. It can be alleviated, somewhat, by using non-uniform memory access (NUMA) on shared-memory machines. On NUMA machines, memory is physically distributed between cores, and the access time depends on where the memory is physically located relative to the core. All major silicon vendors have now started producing chip-level NUMA processors, making placement of data in the memory hierarchy important. Another way to address the von Neumann bottleneck, is simply to increase cache size until it exceeds the working set size. However, increasing the cache size directly corresponds to increased latency, only countered by smaller feature sizes [14, p.16 – 17]. Hardware caches are very efficient for programs with predictable and regular memory access, but cannot speed up programs with random memory access. Even worse, using caches for random access applications can degrade performance, as the cache transfers full cache lines across the memory bus when only a single element is needed without reuse [14, p. 34– 35, p. 47].

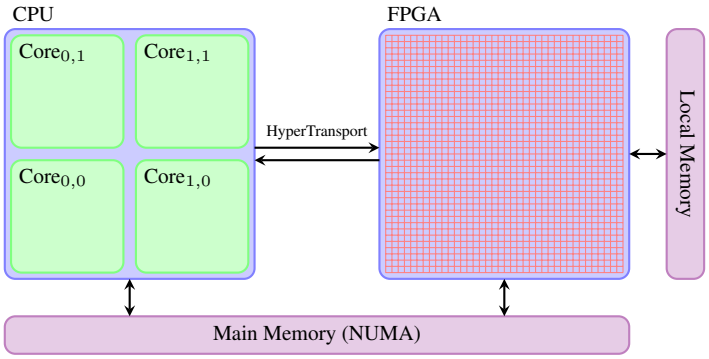
Data movement is not only a limiting factor in terms of execution times, but also seriously affects the energy consumption [7, p. 225 – 227]. The energy to move one bit between two levels in the memory hierarchy is around 1-3 pico joule [7, p. 211], and energy consumed by memory operations is therefore becoming a major constraint for large clusters.



(a) CBEA



(b) CPU in combination with GPU



(c) CPU in combination with FPGA

Figure 1: Schematic of heterogeneous architectures we focus on: The Cell Broadband Engine is a heterogeneous chip (a), a CPU in combination with a GPU is a heterogeneous system (b), and a CPU in combination with an FPGA is also a heterogeneous system (c). The GPU is connected to the CPU via the PCI express bus, and some FPGAs are socket compatible with AMD and Intel processors.

Parallel computing has a considerable history in high-performance computing. One of the key challenges is to design efficient software development methodologies for these architectures. The traditional methodologies mainly focus on task-parallelism and data-parallelism, and we emphasize that the two methodologies are not mutually exclusive.

Task-parallel methodology roughly views the problem as a set of tasks with clearly defined communication patterns and dependencies. A representative example is pipelining. Fortran M [15] was one of the early approaches to task-parallelism in Fortran, and MPI [16] is another prime example of traditional task-parallelism.

Data-parallel methodology, on the other hand, roughly views the problem as a set of operations carried out on arrays of data in a relatively uniform fashion. Early approaches use a global view methodology, where there is one conceptual thread of execution, and parallel statements such as FORALL enable parallelism. Fortran D [17] and Vienna Fortran [18] are early examples of the global view methodology. High Performance Fortran (HPF) [19, 20] followed, and HPF+ [21] further introduced task-parallelism, as well as addressing various issues in HPF. OpenMP [22] is another example that exposes global view parallelism, and it supports both task- and data-parallel programming in C, C++, and Fortran. Co-Array Fortran [23], Unified Parallel C [24], and Titanium [25] on the other hand, expose a Partitioned Global Address Space (PGAS) programming model, where each conceptual thread of execution has private memory in addition to memory shared among threads.

The problem of extending C or Fortran is that modern programming language features, such as object orientation, possibilities for generic programming and garbage collection, are missing. However, there are examples of parallel programming languages with these features. Titanium [25] is based on Java without dynamic threads, Chapel [26] exposes object orientation and generic programming techniques, and X10 [27] integrates reiterated concepts for parallel programming alongside modern programming language features. These traditional and modern languages form the basis for new languages, where existing ideas, for example from the glory days of vector machines, are brought back to life for heterogeneous architectures.

3 Heterogeneous Architectures

The introduction described the current tendency to increase performance by parallelism instead of clock frequency. Our focus is on parallelism within a single node, where instruction-level parallelism is nearly fully exploited [28, p. 154 – 192]. This means that increased performance must come from multi-chip, multi-core, or multi-context parallelism. Flynn's taxonomy [29] defines four levels of parallelism in hardware:

1. single instruction single data (SISD),
2. single instruction multiple data (SIMD),
3. multiple instruction single data (MISD), and
4. multiple instruction multiple data (MIMD).

In addition, two subdivisions of MIMD are single program multiple data (SPMD), and multiple program multiple data (MPMD). We use these terms to describe the architectures.

The single-chip CBEA, illustrated in Figure 1a, consists of a traditional CPU core and eight SIMD accelerator cores. It is a very flexible architecture, where each core can run separate programs in MPMD fashion and communicate through a fast on-chip bus. Its main design criteria has been to maximise performance whilst consuming a minimum of power. Figure 1b shows a GPU with 30 highly multi-threaded SIMD accelerator cores in combination with a standard multi-core CPU. The GPU has a vastly superior bandwidth and computational performance, and is optimized for running SPMD programs with little or no synchronization. It is designed for high-performance graphics, where throughput of data is key. Finally, Figure 1c shows an FPGA consisting of an array of logic blocks in combination with a standard multi-core CPU. FPGAs can also incorporate regular CPU cores on-chip, making it a heterogeneous chip by itself. FPGAs can be viewed as user-defined application-specific integrated circuits (ASICs) that are reconfigurable. They offer fully deterministic performance and are designed for high throughput, for example in telecommunication applications.

In the following, we give a more detailed overview of the hardware of these three heterogeneous architectures, and finally sum up with a discussion, comparing essential properties such as required level of parallelism, communication possibilities, performance and cost.

Graphics Processing Unit Architecture

The GPU was traditionally designed for use in computer games, where 2D images of 3D triangles and other geometric objects are *rendered*. Each element in the output image is referred to as a pixel, and the GPU uses a set of processors to compute the color of such pixels in parallel. Recent GPUs are more general, with rendering only as a special case. The theoretical peak performance of GPUs is now close to three teraflops, making them attractive for high-performance computing. However, the downside is that GPUs typically reside on the PCI express bus. Second generation PCI express $\times 16$ allows 8 GB/s data transfers between CPU and GPU memory, where 5.2 GB/s is attainable on benchmarks.

A GPU is a symmetric multi-core processor that is exclusively accessed and controlled by the CPU, making the two a heterogeneous system. The GPU operates asynchronously from the CPU, enabling concurrent execution and memory transfer. AMD, Intel, and NVIDIA are the three major GPU vendors, where AMD and NVIDIA dominate the high-performance gaming market. Intel, however, has disclosed plans to release a high-performance gaming GPU called Larrabee [31]. In the following, we give a thorough overview of the NVIDIA GT200 [32] and AMD RV770 [33] architectures, followed by a brief description of the upcoming Intel Larrabee. The first two are conceptually quite similar, with highly multi-threaded SIMD cores, whereas the Larrabee consists of fewer, yet more complex, SIMD cores.

Current NVIDIA hardware is based on the GT200 architecture [32], shown in Figure 2. The GT200 architecture is typically programmed using CUDA [34] (see Section 4), which exposes an SPMD programming model using a large number of threads organized into *blocks*. All blocks run the same program, referred to as a *kernel*, and threads within one block can synchronize and communicate using *shared memory*, a kind of local store memory. Communication between blocks, however, is limited to atomic operations on global memory.

The blocks are automatically split into *warps*, consisting of 32 threads. The blocks are scheduled to the streaming multiprocessors (SMs) at runtime, and each warp is executed in SIMD fashion. This is done by issuing the same instruction through four consecutive runs on

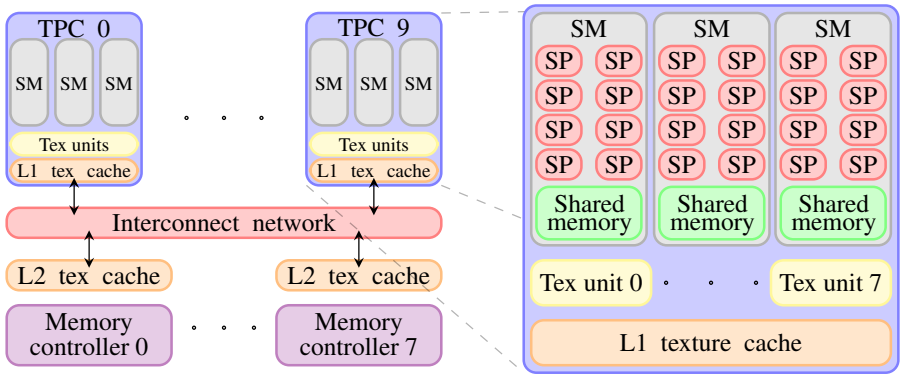


Figure 2: NVIDIA GT200 GPU architecture. The abbreviations have the following meaning: TPC: Texture Processing Cluster; SM: Streaming Multiprocessor; Tex unit: Texture Unit; Tex cache: Texture Cache; SP: Scalar Processor.

the eight scalar processors (SPs). Divergent code flow between threads is handled in hardware by automatic thread masking and serialization, and thus, divergence within a warp reduces performance, while divergence between warps has no impact on execution speed. In addition to eight scalar processors, the streaming multiprocessor also contains a double precision unit, two special function units, 16 KiB of shared memory, 8 KiB constant memory cache, and 16384 32-bit registers. Each scalar processor is a fully pipelined arithmetic-logic unit capable of integer and single precision floating point operations, while the special function unit contains four arithmetic-logic units, mainly used for vertex attribute interpolation in graphics and in calculating transcendentals. The streaming multiprocessor is a dual-issue processor, where the scalar processors and special function unit operate independently.

All the threads within a block have, as mentioned, access to the same shared memory. The shared memory is organized into 16 memory banks that can be accessed every other clock cycle. As one warp uses four clock cycles to complete, it can issue two memory requests per run, one for each *half warp*. For full speed, it is vital that each thread in a half warp exclusively accesses one memory bank. Access to shared memory within a warp is automatically synchronized, and barriers are used to synchronize within a block.

The streaming multiprocessors are designed to keep many active warps in flight. It can switch between these warps without any overhead, which is used to hide instruction and memory latencies. A single multiprocessor executes all warps within a block, but it can also run multiple blocks. The number of blocks each multiprocessor can run is dictated by the register and shared memory use of their respective warps, which cannot exceed the physically available resources. The ratio of active warps to the maximum number of supported warps is referred to as *occupancy*, which is a measure indicating how well the streaming multiprocessor may hide latencies.

The GT200 has eight 64-bit memory controllers, providing an aggregate 512-bit memory interface to main GPU memory. It can either be accessed from the streaming multiprocessors through the texture units that use the texture cache, or directly as *global memory*. Textures are a computer graphics data-format concept, and can be though of as a read-only 2D image. Texture access is thus optimized for 2D access, and the cache holds a small 2D neighbourhood, in contrast to the linear caching of traditional CPUs. The texture units can perform simple filtering

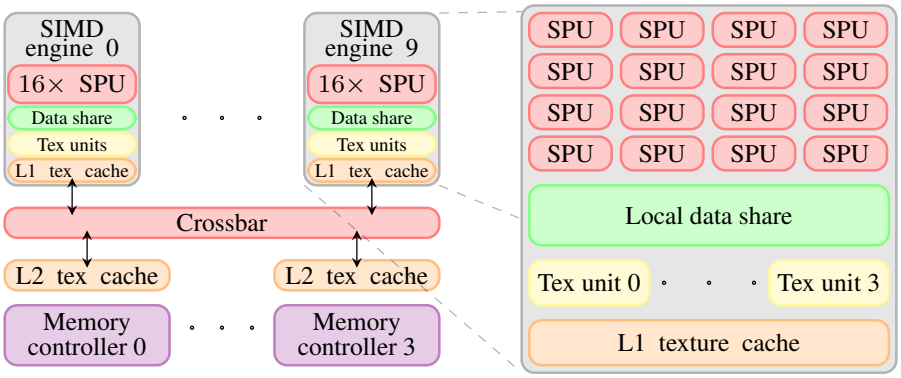


Figure 3: AMD RV770 GPU architecture. The abbreviations have the following meaning: SPU: Shader Processing Unit; Tex unit: Texture Unit; Tex cache: texture cache.

operations on texture data, such as interpolation between colors. Three and three streaming multiprocessors are grouped into *texture processing clusters*, that share eight such texture units and a single L1 texture cache. Global memory access has a high latency and is optimized for linear access. Full bandwidth is achieved when the memory requests can be *coalesced* into the read or write of a full memory segment. In general, this requires that all threads within one half-warp access the same 128 bit segment in global memory. When only partial segments are accessed, the GT200 can reduce the segment size, reducing wasted bandwidth. Per-thread arrays, called *local memory*, are also available. In contrast to previous array concepts on GPUs, these arrays can be accessed dynamically, thus not limited to compile-time constants. However, local memory resides in an auto-coalesced global memory space, and has the latency of global memory. The threads can also access main CPU memory via *zero-copy*, where data is moved directly over the PCI express bus to the streaming multiprocessor. A great benefit of zero-copy is that it is independent of main GPU memory, thus increasing total bandwidth to the streaming multiprocessors.

NVIDIA have also recently released specifications for their upcoming GT300 architecture, codenamed Fermi [35]. Fermi is based around the same concepts as the GT200, with some major improvements. First of all, the number of Scalar Processors has roughly doubled, from 240 to 512. The double precision performance has also improved dramatically, now running at half the speed of single precision. All vital parts of memory are also protected by ECC, and the new architecture has cache hierarchy with 16 or 32 KiB L1 data cache per Streaming Multiprocessor, and a 768 KiB L2 cache shared between all Streaming Multiprocessors. The memory space is also unified, so that *shared memory* and *global memory* use the same address space, thus enabling execution of C++ code directly on the GPU. The new chip also supports concurrent kernel execution, where up-to 16 independent kernels can execute simultaneously. The Fermi is currently not available in stores, but is expected to appear shortly.

The current generation of AMD FireStream cards is based on the RV770 [33] architecture, shown in Figure 3. Equivalent to the NVIDIA concept of a grid of blocks, it employs an SPMD model over a *grid* of *groups*. All groups run the same kernel, and threads within a group can communicate and synchronize using local data store. Thus, a group resembles the concept of blocks on NVIDIA hardware.

Each group is an array of threads with fully independent code-flow, and groups are sched-

uled to *SIMD engines* at runtime. The SIMD engine is the basic core of the RV770, containing 16 shader processing units (SPUs), 16 KiB of local data share, an undisclosed number of 32-bit registers, 4 texture units, and an L1 texture cache. Groups are automatically split up into *wavefronts* of 64 threads, and the wavefronts are executed in SIMD fashion by four consecutive runs of the same instruction on the 16 shader processing units. The hardware uses serialization and masking to handle divergent code flow between threads, making divergence within wavefronts impact performance, whereas divergence between wavefronts runs at full speed. As such, wavefronts are equivalent to the concept of warps on NVIDIA hardware. However, unlike the scalar design of the NVIDIA stream processor, the shader processing unit is super-scalar and introduces instruction level parallelism with five single precision units that are programmed using very long instruction words (VLIW). The super-scalar design is also reflected in registers, which use four-component data types. The fifth unit also handles transcendental and double-precision arithmetic.

All threads within a group have access to the same *local data share*, which is somewhat similar to the shared memory of NVIDIA. Data is allocated per thread, which all threads within the group can access. However, threads can only write to their own local data share. Synchronization is done using memory barriers, as on NVIDIA hardware. The RV770 also exposes shared registers, which are persistent between kernel invocations. These registers are shared between all wavefronts processed by a SIMD engine, in which thread i in wavefront A can share data with thread i in wavefront B, but not with thread j. Thus, any operation on shared registers is atomic.

Thread grids are processed in either pixel shader or compute mode. The pixel shader mode is primarily for graphics, and restricts features to that of the R6xx architecture without local data share. Using the compute shader mode, output is required to be a *global buffer*. Global buffers are accessed directly by the threads, allowing arbitrary read and write operations. As global buffers are not cached, *burst writes* are used for consecutive addresses in a fashion similar to coalesced memory write on NVIDIA hardware. On current hardware, however, global buffers are inferior to regular buffers with respect to performance. Threads can also access part of main CPU memory; the GPU driver reserves a small memory segment that the SIMD engines can access directly over the PCI express bus. The RV770 also contains instruction and constant memory caches, an L2 texture cache, and a global data share that are shared between the SIMD engines. The global data share enable the SIMD engines to share data, but is currently not exposed to the programmer.

The upcoming Larrabee [31] GPU from Intel can be considered a hybrid between a multi-core CPU and a GPU. The Larrabee architecture is based on many simple in-order CPU cores that run an extended version of the x86 instruction set. Each core is based on the Pentium chip with an additional 16-wide SIMD unit which executes integer, single-precision, or double-precision instructions. The core is dual-issue, where the scalar and SIMD units can operate simultaneously, and it supports four threads of execution. The Larrabee also features a coherent L2 cache shared among all the cores, which is divided into local subsets of 256 KiB per core. Cores can communicate and share data using on-chip communication, consisting of a 1024-bit bi-directional ring network. A striking design choice is the lack of a hardware rasterizer, which forces the Larrabee to implement rasterization in software. The first Larrabee GPUs are expected to arrive in 2010.

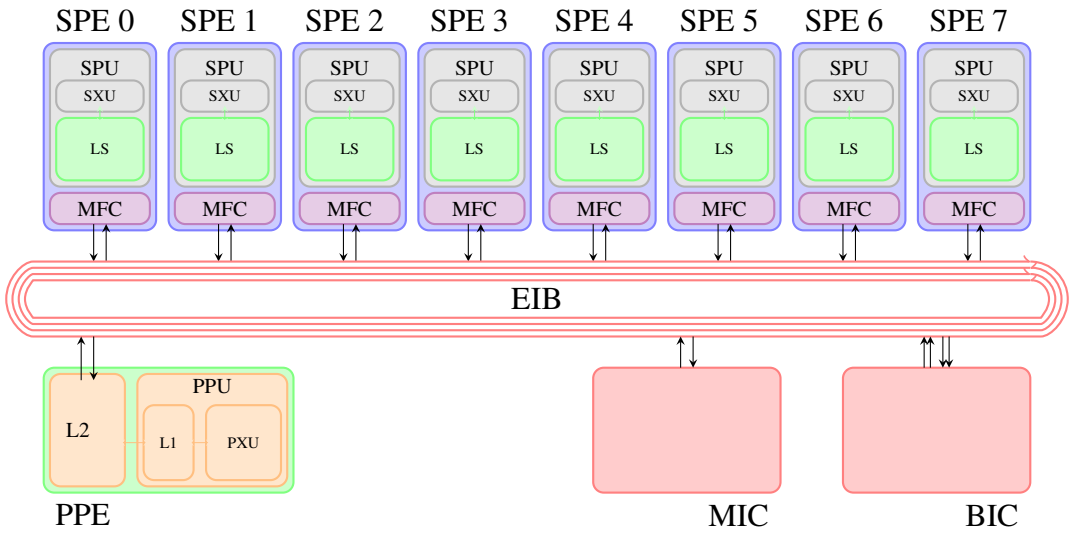


Figure 4: PowerXCell 8i architecture. The abbreviations have the following meaning: SPE: Synergistic Processing Element; SPU: Synergistic Processing Unit; SXU: Synergistic Execution Unit; LS: Local Store; MFC: Memory Flow Controller; EIB: Element Interconnect Bus; PPE: Power Processing Element; PPU: Power Processing Unit; PXU: Power Execution Unit; MIC: Memory Interface Controller; BIC: Bus Interface Controller.

The architectures from NVIDIA and AMD are, as mentioned earlier, conceptually similar. They operate with the same execution model, but have differences in their hardware setups. AMD has twice the SIMD width of NVIDIA hardware, runs at about half the frequency, and employs a superscalar architecture. However, they both feature similar peak performance numbers at around one teraflops in single precision. The memory bandwidth is also strikingly similar, just surpassing 100 GB/s, even though they have different hardware setups. When utilizing texture sampling units, an interesting note is that NVIDIA has three execution units per texture sampler, whereas AMD has four. This can impact performance of algorithms with heavy use of texture sampling.

Cell BEA

The CBEA is a heterogeneous processor with a traditional CPU core and eight accelerator cores on the same chip, as shown in Figure 4. It is used in the first petaflops supercomputer, called the Roadrunner [36]. The Roadrunner is the worlds fastest computer on the June 2009 Top500 list [37], and the seven most energy-efficient supercomputers on this list use the CBEA as the main processor [38]. Commercially, it is available as 1U form factor or blade servers, as PCI express plug-in cards, and in the PlayStation 3 gaming console. The state-of-the-art server solutions use the PowerXCell 8i version of the CBEA. For example, the Roadrunner supercomputer consists of two 3.2 GHz CBEAs per dual-core 1.8 GHz Opteron, in which the CBEAs contribute to 96% of the peak performance [39]. The PCI express plug-in cards also use the PowerXCell 8i processor, but are primarily intended for development and prototyping. The CBEA in the PlayStation 3 is inexpensive due to its high production numbers, but it is a different chip than the PowerXCell 8i. It offers low double precision performance and a very limited amount of main memory. Furthermore, one of the accelerator cores is disabled in hardware to increase production yields, and another accelerator core is reserved for the Hypervisor virtualization

layer when running Linux. We focus on the PowerXCell 8i implementation that consists of the Power Processing Element (PPE), eight Synergistic Processing Elements (SPEs), and the Element Interconnect Bus (EIB).

The PPE is a traditional processor that uses the Power instruction set. It contains a 512 KiB L2 cache, an in-order dual-issue RISC core with two-way hardware multi-threading, and a VMX [40] engine for SIMD instructions. Compared to modern CPUs, the PPE is a stripped-down core with focus on power efficiency. It is capable of outputting one fused multiply-add in double precision, or one SIMD single precision fused multiply-add per clock cycle. Running at 3.2 GHz, this yields a peak performance of 25.6 or 6.4 gigaflops in single and double precision, respectively.

The SPE consists of a memory flow controller (MFC) and a Synergistic Processing Unit (SPU). The SPU is a dual-issue in-order SIMD core with 128 registers of 128 bit each and a 256 KiB local store. The SPU has a vector arithmetic unit similar to VMX that operates on 128-bit wide vectors composed of eight 16-bit integers, four 32-bit integers, four single-precision, or two double-precision floating point numbers. The even pipeline of the dual-issue core handles arithmetic instructions simultaneously as the odd pipeline handles load, store, and control. The SPU lacks a dynamic branch predictor, but exposes hint-for-branch instructions. Without branch hints, branches are assumed not to be taken. At 3.2 GHz, each SPU is capable of 25.6 or 12.8 gigaflops in single and double precision, respectively.

The SPU uses the local store similarly to an L2 cache, and the MFC uses DMA requests to move data between the local store and main memory. Because the MFC operates asynchronously from the SPU, computations can take place simultaneously as the MFC is moving data. The MFC requires 16-byte alignment of source and destination data addresses and transfers multiples of 128 bytes. Smaller transfers must be placed in the *preferred slot* within the 128-byte segment, where unwanted data is automatically discarded. The MFC also has mailboxes for sending 32-bit messages through the element interconnect bus, typically used for fast communication between the SPEs and the PPE. Each SPE has one outbound mailbox, one outbound interrupt mailbox, and one inbound mailbox capable of holding four messages.

At the heart of the CBEA is the EIB, a ring bus with two rings in each direction that connects the PPE, the SPEs, the Memory Interface Controller (MIC), and the Bus Interface Controller (BIC). The MIC handles main memory and the BIC handles system components such as the HyperTransport bus. The EIB is capable of moving an aggregate 128 byte $\times\, 1.6$ GHz = 204.8 GB/s, and 197 GB/s has been demonstrated [41].

Field Programmable Gate Array Architecture

The first commercially viable FPGA was developed by Xilinx co-founder Ross Freeman in 1984, and he was entered into the National Inventors Hall of Fame for this accomplishment in 2006. FPGAs were initially used for discrete logic, but have expanded their application areas to signal processing, high-performance embedded computing, and recently as accelerators for high-performance computing. Vendors such as Cray, Convey, SGI, HP, and SRC all offer such high-performance solutions. While early FPGAs had sufficient capability to be well suited for special-purpose computing, their application for general-purpose computing was initially restricted to a first-generation of low-end reconfigurable supercomputers, for which PCI interfaces were sufficient for CPU communication. However, this situation has recently changed

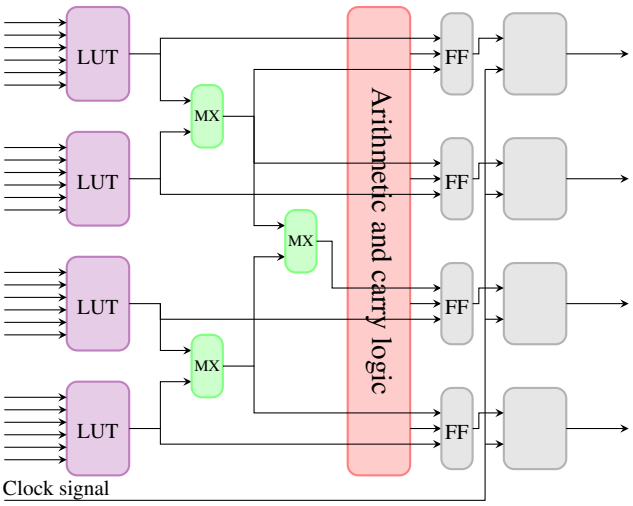


Figure 5: Simplified diagram of a Virtex-5 FPGA slice. The abbreviations have the following meaning: LUT: Look-up table; MX: Multiplexer; FF: Flip-flop.

with the adoption of high-speed IO standards, such as QuickPath Interconnect and HyperTransport. The major FPGA vendors, Altera and Xilinx have collaborated with companies such as DRC Computer, Xtreme Data, Convey, and Cray, who offer FPGA boards that are socket compatible with Intel or AMD processors, using the same high-speed bus. These boards also offer on-board memory, improving total bandwidth.

An FPGA is a set of configurable logic blocks, digital signal processor blocks, and optional traditional CPU cores that are all connected via an extremely flexible interconnect. This interconnect is reconfigurable, and is used to tailor applications to FPGAs. When configured, FPGAs function just like application specific integrated circuits (ASICs). We focus on the popular Xilinx Virtex-5 LX330, consisting of 51 840 *slices* and 192 digital signal processing slices that can perform floating point multiply-add and accumulate. Heterogeneous FPGAs, such as the Virtex-5 FX200T, offer up-to two traditional 32-bit PowerPCs cores on the same chip. Each slice of a Virtex-5 consists of four 6-input look-up tables and four flip-flops, as shown in Figure 5. The multiplexers are dynamically set at runtime, whereas the flip-flops are statically set at configuration. Two slices form a configurable logic block, and programmable interconnects route signals between blocks. Configuring the interconnect is part of FPGA programming, and can be viewed as creating a data-path through the FPGA. This can make FPGAs fully deterministic when it comes to performance. A key element to obtaining high-performance on FPGAs is to use as many slices as possible for parallel computation. This can be achieved by pipelining the blocks, trading latency for throughput; by data-parallelism, where data-paths are replicated; or a combination of both. As floating point and double-precision applications rapidly exhausted the number of slices available on early FPGAs, they were often avoided for high-precision calculations. However, this situation has changed for current FPGAs which have sufficient logic to compute thousands of adds, or about 80 64-bit multiplies per clock cycle. The conceptually simple design of FPGAs make them extremely versatile and power-efficient, and they offer high computational performance.

Table 1: Summary of architectural properties. The numbers are of current hardware, and are reported per physical chip. The CPU is an Intel Core i7-965 Quad Extreme, the AMD GPU is the FireStream 9270, the NVIDIA GPU is the Tesla C1060, the CBEA is the IBM QS22 blade, and the FPGA is the Virtex-6 SX475T.

	CPU	GPU (AMD/NVIDIA)	CBEA	FPGA
Composition	Symmetric Multicore	Symmetric Multicore	Heterogeneous Multicore	Heterogeneous Multicore
Full cores	4	-	1	2
Accelerator cores	0	10 / 30	8	2016 DSP slices
Intercore communication	Cache	None	Mailboxes	Explicit wiring
SIMD width	4	64 / 32	4	Configurable
Additional parallelism	ILP	VLIW / Dual-issue	Dual-issue	Configurable
Float operations per cycle	16	1600 / 720	36	520
Frequency (GHz)	3.2	0.75 / 1.3	3.2	<0.55
Single precision gigaflops	102.4	1200 / 936	230.4	550
Double : single precision performance	1:2	1:5 / 1:12	∼1:2	∼1:4
Gigaflops / watt	0.8	5.5 / 5	2.5	13.7
Megaflops / USD	70	800 / 550	>46	138
Accelerator Bandwidth (GB/s)	N/A	109 / 102	204.8	N/A
Main Memory Bandwidth (GB/s)	25.6	8	25.6	6.4
Maximum memory size (GiB)	24	2 / 4	16	System dependent
ECC support	Yes	No	Yes	Yes

Performance, Power Consumption, and Cost

Due to the lack of a unified theory for comparing different heterogeneous architectures, it is difficult to give a fair comparison. However, in Table 1, we summarize qualitative and quantitative characteristics of the architectures, giving an overview of required level of parallelism, performance and cost. The performance numbers per watt and per dollar are measured in single precision. However, comparing the different metrics will ultimately be a snapshot of continuous development, and may change over time. There are also differences between different hardware models for each architecture: in particular, there are less expensive versions of all architectures, especially for the CPU and GPU, that can benefit from mass production. However, we wish to indicate trends, and argue that the differences we distinguish are representable, even though there are variations among models and in time.

The SIMD units of the state-of-the-art Intel Core i7-965 Quad Extreme processor is capable of 16 single precision multiply-add instructions per clock cycle at 3.2 GHz, that is, 102.4 gigaflops single precision performance, and half of that in double precision. The i7-965 has a thermal design power rating of 130 watt, indicating around 0.8 single precision gigaflops per watt. It has a manufacturer suggested retail price of 999 USD, giving around 100 megaflops per USD.

The NVIDIA Tesla C1060 has 30 streaming multiprocessors that each can execute eight single precision multiply-adds on the scalar processors, and either eight single precision multiplies or one double precision multiply-add on the special function unit. This totals to 720 single precision operations per clock cycle at 1.3 GHz, yielding 936 gigaflops single precision perfor-

mance, or one twelfth of that in double precision. The C1060 draws a maximum of 187.8 watt, yielding around 5 gigaflops per watt. The manufacturer suggested retail price of 1699 USD gives around 550 megaflops per USD. The C1060 has 4 GiB of GDDR3 memory, and with a 512-bit memory interface at 800 MHz, the peak memory bandwidth is 102 GB/s.

The up-coming Fermi architecture from NVIDIA doubles the number of stream processors in the chip, and increases the double precision performance to half the speed of single precision. Assuming a similar clock frequency as the C1060, one can expect roughly double the performance in single precision, and a dramatic improvement in double precision performance. The Fermi will be available with up-to 6 GB memory on a 384-bit wide GDDR5 bus. This roughly gives a 50% bandwidth increase compared to the C1060, again assuming a similar frequency.

The AMD FireStream 9270 contains 160 shader processing units. Assuming full utilization of instruction level parallelism, each shader processing unit can execute five multiply-add instructions per clock cycle resulting in a total of 1600 concurrent operations. Running at 750 MHz this yields a performance of 1.2 teraflops in single precision, and one fifth of that in double precision. The card draws a maximum of 220 watt, implying approximately 5.5 gigaflops per watt. It has a manufacturer suggested retail price of 1499 USD, resulting in 800 megaflops per USD. The 9270 has 2 GiB of GDDR5 memory on a 256-bit memory bus. Running at 850 MHz, the bus provides a peak internal memory bandwidth of 109 GB/s.

AMD recently released their first DirectX11 compatible GPUs, based on the R800 [42] architecture. The R800 is quite similar to the R700 series GPUs, even though the exact architectural details are unpublished. The Radeon HD 5870 uses the R800 architecture, with a total of 320 shader processing units. This chip runs at 850 MHz, and has a peak performance of 2.7 teraflops, again assuming full utilization of instruction level parallelism. The new card has 1 GB memory on a 256-bit wide GDDR5 bus. Running at 1200 MHz this results in a bandwidth of 153 GB/s.

The IBM QS22 blade [43] consists of two PowerXCell 8i CBEAs connected through a HyperTransport bus. Each of the CBEAs is capable of 230 gigaflops in single precision, and slightly less than half that in double precision. The CBEA consumes 90 watt, yielding 2.5 gigaflops per watt. The QS22 blade with 8 GiB memory has a manufacturer suggested retail price of 9995 USD, implying 46 megaflops per USD. However, this rather pessimistic figure includes the cost of the whole blade server.

On FPGAs, floating point operations are often traded away due to their real-estate demands. However, if one were to implement multiply-add's throughout the FPGA fabric, the 32-bit floating point performance attains 230 gigaflops for the Virtex-5 LX330, 210 gigaflops for the SX240T and 550 gigaflops for the Virtex-6 SX475 [44, 45]. The SX475T performance is due to the 2016 DSP slices, twice that of the SX240T and ten times as many as the LX330. Double precision multiply-add units consume four times the real-estate, and twice the memory, implying about one fourth double precision performance. FPGA cost is about twice processor cost. The LX330, SX240T and SX475T cost about 1700, 2000, and 4000 USD, respectively, but deliver approximately 17, 18 and 14 gigaflops per watt. An example of an FPGA board that is socket compatible with AMD Opteron CPUs is the Accelium AC2030. It can access main memory at 6.4 GB/s, has three HyperTransport links running at 3.2 GB/s, and has 4.5 GiB on-board RAM with a bandwidth of 9.6 GB/s. Equipped with the LX330, the board uses 40 watt.

Numerical Compliance and Error Resiliency

Floating point calculations most often contain errors, albeit sometimes small. The attained accuracy is a product of the inherent correctness of the numerical algorithm, the precision of input data, and the precision of intermediate arithmetic operations. Examples abound where double precision is insufficient to yield accurate results, including Gaussian elimination, where even small perturbations can yield large errors. Surprising is Rump's example, reexamined for IEEE 754 by Loh and Walster [46], which converges towards a completely wrong result when precision is increased. Thus, designing numerically stable algorithms, such as convex combinations of B-spline algorithms, is essential for high-precision results. Floating point operations and storage of any precision have rounding errors, regardless of algorithmic properties, and even the order of computations significantly impacts accuracy [47].

Using high precision storage and operations is not necessarily the best choice, as the accuracy of an algorithm might be less than single or double precision [48, 39, 49]. Using the lowest possible precision in such cases has a two-fold benefit. Take the example of double versus single precision, where single precision is sufficiently accurate. Both storage and bandwidth requirements are halved, as single precision consumes half the memory. Further, single precision increases compute performance by a factor 2–12, as shown in Table 1. Single precision operations can also be used to produce high precision results using mixed precision algorithms. Mixed precision algorithms obtain high-precision results for example by using lower precision intermediate calculations followed by high-precision corrections [50, 51, 52].

GPUs have historically received scepticism for the lack of double precision and IEEE-754 floating point compliance. However, current GPUs and the CBEA both support double precision arithmetic and conform to the floating point standard in the same way as SIMD units in CPUs do [53]. FPGAs can ultimately be tuned to your needs. Numerical codes executing on these architectures today typically yield bit-identical results, assuming the same order of operations, and any discrepancies are within the floating point standard.

Spontaneous bit errors are less predictable. Computer chips are so small that they are susceptible to cosmic radiation, build-up of static charge on transistors, and other causes which can flip a single bit. Several technologies can detect and correct such errors. One is ECC memory where a parity bit is used to detect and correct bit errors in a word. Others perform ECC logic in software [54], or duplicate computations and check for consistency. There is also research into algorithmic resiliency [7, p. 231], possibly the best solution. Using a few extra computations, such as computing the residual, the validity of results can be determined.

4 Programming Concepts

Heterogeneous architectures pose new programming difficulties compared to existing serial and parallel platforms. There are two main ways of attacking these difficulties: inventing new, or adapting existing languages and concepts. Parallel and high-performance computing both form a basis of knowledge for heterogeneous computing. However, heterogeneous architectures are not only parallel, but also quite different from traditional CPU cores, being less robust for non-optimal code. This is particularly challenging for high-level approaches, as these require new compiler transformation and optimization rules. Porting applications to heterogeneous architectures thus often requires complete algorithm redesign, as some programming patterns, which in principle are trivial, require great care for efficient implementation on a heterogeneous plat-

form. As Chamberlain et al. [55] note, heterogeneous architectures most often require a set of different programming languages and concepts, which is both complex from a programmers perspective, as well as prone to unforeseen errors. Furthermore, a programming language that maps well to the underlying architecture is only one part of a productive programming environment. Support tools like profilers and debuggers are just as important as the properties of the language.

In this section, we describe programming languages, compilers, and accompanying tools for GPUs, the CBEA, and FPGAs, and summarize with a discussion that compares fundamental characteristics such as type of abstraction, memory model, programming model, and so on. We also report our subjective opinion on ease of use and maturity.

Multi-architectural Languages

OpenCL [56] is a recent standard, ratified by the Khronos Group, for programming heterogeneous computers. Khronos mostly consists of hardware and software companies within parallel computing, graphics, mobile, entertainment, and multimedia industries, and has a highly commercial incentive, creating open standards such as OpenGL that will yield greater business opportunities for its members.

Khronos began working with OpenCL in June 2008, in response to an Apple proposal. The Khronos ratification board received the standard draft in October, ratifying it in December 2008. Most major hardware and many major software vendors are on the ratification board, giving confidence that OpenCL will experience the same kind of success as OpenMP. Apple included OpenCL in Mac OS X Snow Leopard, and both NVIDIA and AMD have released beta-stage compilers. It is also probable that CBEA compilers will appear, as the CBEA team from IBM has participated in the standard. Support for FPGAs, however, is unclear at this time.

OpenCL consists of a programming language for accelerator cores and a set of platform API calls to manage the OpenCL programs and devices. The programming language is based on C99 [57], with a philosophy and syntax reminiscent of CUDA (see Section 4) using SPMD kernels for the data-parallel programming model. Task-parallelism is also supported with single-threaded kernels, expressing parallelism using 2–16 wide vector data types and multiple tasks. Kernel execution is managed by command queues, which support out-of-order execution, enabling relatively fine-grained task-parallelism. Out-of-order tasks are synchronized by barriers or by explicitly specifying dependencies. Synchronization between command queues is also explicit. The standard defines requirements for OpenCL compliance in a similar manner as OpenGL, and extensively defines floating-point requirements. Optional extensions are also defined, including double precision and atomic functions.

RapidMind [58] is a high-level C++ abstraction to multi-core CPU, GPU, and CBEA programming. It originated from the research group around Sh [59] in 2004, and is a now a commercial product. It has a common abstraction to architecture-specific back-ends. Low-level optimization and load balancing is handled by the back-ends, allowing the programmer to focus on algorithms and high-level optimizations. They argue that for a given development time, a programmer will create better performing code using the RapidMind compared to lower level tools [60]. RapidMind exposes a streaming model based on arrays, and the programmer writes kernels that operate on these arrays. Their abstraction is implemented as a C++ library and thus requires no new tools or compilers. Debugging is done with traditional tools using a debugging

back-end on the CPU, and performance statistics can also be collected.

A more high-level approach is HMPP, the Hybrid Multi-core Parallel Programming environment [61]. Their approach is to annotate C or Fortran source code, similar to OpenMP, to denote the parts of the program that should run on the device. In addition, special pragmas give hints about data movement between the host and device. The HMPP consists of a meta compiler and runtime library, that supports CUDA and CAL-enabled GPUs. A similar approach is taken by the Portland Group with their Accelerator C and Fortran compilers [62], currently supporting CUDA enabled GPUs. Such approaches have the advantage of easy migration of legacy code, yielding good performance for embarrassingly parallel code. However, many algorithms require redesign by experts to fully utilize heterogeneous architectures. Relying on high-level approaches in these cases will thus yield sub-optimal performance.

Graphics Processing Unit Languages

Initially, GPUs could only be programmed using graphics APIs like OpenGL [63]. General purpose stream-computing was achieved by mapping stream elements to pixels. The obvious drawback was that the developer needed a thorough understanding of computer graphics. As a result, higher-level languages were developed that ran on top of the graphics API, such as Brook [64] and RapidMind [58]. To provide a more direct access to the GPU, AMD has released Stream SDK [65] and NVIDIA has released CUDA [34]. Both Stream SDK and CUDA are tied to their respective vendor's GPUs, which is an advantage when considering performance, but a problem for portability. Direct3D [66] is a graphics API from Microsoft targeting game developers. It standardizes the hardware and is supported by both NVIDIA and AMD. Direct3D 11, which is expected to appear shortly, includes a *compute shader*, whose main goal is to tightly integrate GPU accelerated 3D rendering and computation, such as game physics. The Direct3D compute shader uses ideas similar to NVIDIA CUDA.

The CUDA Toolkit [67] provides the means to program CUDA-enabled NVIDIA GPUs, and is available on Windows, Linux, and Mac OS X. CUDA consists of a runtime library and a set of compiler tools, and exposes an SPMD programming model where a large number of threads, grouped into blocks, run the same kernel.

The foundation of the CUDA software stack is CUDA PTX, which defines a virtual machine for parallel thread execution. This provides a stable low-level interface decoupled from the specific target GPU. The set of threads within a block is referred to as a co-operative thread array (CTA) in PTX terms. All threads of a CTA run concurrently, communicate through shared memory, and synchronize using barrier instructions. Multiple CTAs run concurrently, but communication between CTAs is limited to atomic instructions on global memory. Each CTA has a position within the grid, and each thread has a position within a CTA, which threads use to explicitly manage input and output of data.

The CUDA programming language is an extension to C, with some C++-features such as templates and static classes. The new Fermi architecture enables full C++ support, which is expected to appear in CUDA gradually. The GPU is programmed using CUDA kernels. A kernel is a regular function that is compiled into a PTX program, and executed for all threads in a CTA. A kernel can directly access GPU memory and shared memory, and can synchronize execution within a CTA. A kernel is invoked in a CUDA context, and one or more contexts

can be bound to a single GPU. Multiple contexts can each be bound to different GPUs as well, however, load balancing and communication between multiple contexts must be explicitly managed by the CPU application. Kernel execution and CPU-GPU memory transfers can run asynchronously, where the order of operations is managed using a concept of streams. There can be many streams within a single GPU context: all operations enqueued into a single stream are executed in-order, whereas the execution order of operations in different streams is undefined. Synchronization events can be inserted into the streams to synchronize them with the CPU, and timing events can be used to profile execution times within a stream.

CUDA exposes two C APIs to manage GPU contexts and kernel execution: the runtime API and the driver API. The runtime API is a C++ abstraction where both GPU and CPU code can be in the same source files, which are compiled into a single executable. The driver API, on the other hand, is an API that requires explicit handling of low-level details, such as management of contexts and compilation of PTX assembly to GPU binary code. This offers greater flexibility, at the cost of higher code complexity.

There are several approaches to debugging CUDA kernels. NVCC, the compiler driver that compiles CUDA code, can produce CPU emulation code, allowing debugging with any CPU debugger. NVIDIA also provides a beta-release of a CUDA capable GNU debugger, called cuda-gdb, that enables interactive debugging of CUDA kernels running on the GPU, alongside debugging of CPU code. Cuda-gdb can switch between CTAs and threads, step through code at warp granularity, and peek into kernel variables. NVIDIA has also announced an up-coming Visual Studio toolset called Nexus that integrates debugging and performance analysis [68].

CUDA also contains a non-intrusive profiler that can be invoked on an existing binary. The profiler can log kernel launch attributes such as grid-size, kernel resource usage, GPU and driver execution times, memory transfers, performance counters for coalesced and non-coalesced loads and stores, local memory usage, branch divergence, occupancy, and warp serialization. NVIDIA also provides the CUDA Visual Profiler, which is a GUI front-end for profiling CUDA applications, and the CUDA Occupancy Calculator for experimenting with how register and shared memory use affects the occupancy on different GPUs.

NVIDIA GPUs native instruction sets are proprietary, but many details have been deciphered through differential analysis of compiled GPU kernels. In particular, decuda [69], a third party disassembler for compiled CUDA kernels, is a valuable tool to analyze and optimize kernels.

The AMD Stream SDK [65], available on Windows and Linux, is the successor of ATI's Close To the Metal initiative from 2006, and provides the means to program AMD GPUs directly. The software stack consists of the AMD Compute Abstraction Layer (CAL), which supports the R6xx-series and newer AMD GPUs, and the high-level Brook+ language. CAL exposes an SPMD programming model, in which a program is executed by every thread in a large grid of threads. Threads are initialized with their grid positions, which can be used for explicit input and output of data. The output of threads can also be defined implicitly by using the pixel shader mode, which can improve performance. Threads are clustered into groups, where threads within a single group can synchronize using barriers and communicate using local data store.

CAL has two major components, the CAL runtime and the CAL compiler. The CAL compiler is a C interface used to compile GPU assembly or CAL intermediate language, a GPU-

agnostic assembly language similar to CUDA PTX, into GPU binaries. The CAL runtime is a C-interface reminiscent of the CUDA driver API with respect to programming complexity. It is used to create GPU contexts, load and execute kernels on the GPU, and manage GPU resources. A context is tied to a single GPU, and holds a queue of operations that are asynchronously executed in order. Multiple contexts can be created on a single GPU, and they can share resources. However, resource sharing and load balancing between different GPUs has to be explicitly managed by the application. New features are added to CAL through an extension mechanism. One extension enables resources sharing with DirectX, and another enables performance counters of cache hit rate and SIMD engine idle cycles.

Brook+ is AMD's extension of Brook [64], and is a high-level programming language and runtime, with CPU and CAL back-ends. Brook+ is a higher-level language compared to CUDA. The programmer defines kernels that operate on input and output data streams, and multiple kernels operating on the same streams create an implicit dependency graph. Data is explicitly copied from the CPU to input streams, and from output streams back to the CPU after kernel execution. Copying data into streams and execution of kernels run asynchronous, enabling concurrent CPU computations. Reading data back to the CPU from a stream is blocking by default. Brook+ allows streams and kernels to be assigned to different GPUs, however, streams used by a kernel are required to reside on the GPU running the kernel, i.e., inter-GPU communication has to be explicitly managed by the application.

The basic Brook+ kernel maps input stream elements one-to-one to output stream elements. Reduction kernels reduce the dimensionality of a stream along one axis using an associative and commutative binary operator. Kernels can also have explicit communication patterns, providing random-access gather operations from input streams, scatter write to a single output stream, as well as access to the local data share.

A Brook+ application contains both code running on the CPU, and kernels running on the GPU. The Brook+ kernel language is an extension of C, with syntax to denote streams and keywords to specify kernel and stream attributes. Kernels are processed by the Brook+ compiler, which produces a set of C++-files that implement the CPU interface of the kernel, and the kernel itself as either CAL intermediate language, Direct3D HLSL, or CPU emulation code. The generated C++-files can then be included into any C++ project.

Both intermediate language code and Brook+ code can be analyzed using the Stream KernelAnalyzer [70]. It is a graphical Windows application that shows the native GPU assembly code, kernel resource use, prediction on whether arithmetic or memory transfer is the bottleneck, and performance estimates for a full range of GPUs.

Both NVIDIA and AMD offer assembly level programming of their GPUs. NVIDIA also offers CUDA and AMD offers Brook+, both based on C to write kernels. However, CUDA offers an abstraction closer to the hardware, whereas Brook+ abstracts away most hardware features. This enables experienced CUDA programmers to write highly tuned code for a specific GPU. Brook+, on the other hand, enables rapid implementation of programs, giving compilers responsibility for optimizations.

Cell BEA Languages

Recall from Section 3 that the CBEA consists of one PPE and eight SPEs connected by the Element Interconnect Bus. While regular CPUs come with logic for instruction level parallelism

and hardware managed caches, the CBEA focuses on low power consumption and deterministic performance, sacrificing out-of-order execution and dynamic branch prediction. This shifts responsibility for utilizing the SPE hardware resources to the programmer and compiler, and makes execution time fully deterministic as long as memory contention is avoided.

The IBM Cell SDK [71] is the de facto API for programming the CBEA. The PPE is typically programmed using Fortran, C, or C++, where the SDK offers function calls to manage the SPEs. These functions include ways of creating, starting, and stopping SPE contexts, communicating with the SPEs, etc. Each SPE can run a separate program and communicate with all other nodes on the EIB, such as the PPE and main memory. This enables techniques such as SPE pipelining. The SPEs are also programmed using Fortran, C, or C++, but with certain restrictions; e.g., using `iostream` in C++ is prohibited. The SPE compilers also support SIMD intrinsics, similar to VMX, and intrinsics for managing the memory flow controller. Part of the strength of the SPE is the memory flow controller, which gives the programmer full control over data movement. This enables techniques such as software-managed threads [72, 73, p. 67 – 68], which can hide memory latency using the same philosophy as hardware threads. The SDK also includes a software cache that can hide the details of data movement at the cost of a small overhead.

There are currently two compilers for the CBEA architecture, GCC and IBM XL. The latter is often given credit for yielding the best performance for moderately to highly tuned code. To compile and link code for the CBEA, the SPE source code is compiled and linked to an SPE executable, which is then embedded into a PPE object file. This object file is finally linked with the rest of the PPE code to produce a CBEA executable. The limited size of local store is overcome by using manually or automatically generated code overlays for large codes. The code overlay mechanism enables arbitrary large codes to run on an SPE by partitioning the SPE program into segments that each fit in local store. Segments are then transparently transferred from main memory at run-time, however, switching between segments is expensive, and should be avoided.

The CBEA also includes solid support for program debugging, profiling, and analysis, all included in the IBM Cell SDK. The compiler can output static timing analysis information that shows pipeline usage for the SPEs. Profiling information can be collected using OProfile. Hardware counters can be accessed through the Performance Counter Tool, and runtime traces can be generated using the Performance Debugging Tool. Vianney et al. [74] give a step-by-step guide to porting serial applications to the CBEA using these tools, which can be accessed directly, or through the Eclipse-based Visual Performance Analyzer. Further, the IBM Full-System Simulator for the CBEA [75] is a cycle accurate simulator, capable of simulating the PPE and SPE cores, the memory hierarchy, and disk and bus traffic. The simulator can output raw traces and has a graphical user interface to visualize statistics.

Programming with the IBM Cell SDK requires detailed knowledge of the hardware, and thus, there has been research into higher-level abstractions. Currently, there are four main abstractions; OpenMP, MPI, CellSs and Sequoia. The OpenMP approach uses a PPE-centric shared-memory model, where the program runs on the PPE and data-intensive parts are offloaded to the SPEs. The other three approaches, on the other hand, employ an SPE-centric distributed memory model, in which the PPE simply manages the work and executes support functions for the SPEs.

Table 2: Summary of programming languages properties. Under abstraction, "compiler" implies that an extra compiler has to be integrated into the toolchain. VHDL and Verilog are described as "C-like", even though they capture much more than typical sequential C. Under kernel language, CUDA supports some features as templates and function overloading, and libspe supports full C++ except parts of the STL. Several of the languages have a memory model similar to Partitioned Global Address Space (PGAS). Under task parallelism, "explicit" implies explicit communication, "streams" implies multiple in-order asynchronous streams, and "full" implies out-of-order execution with automatic handling of given dependencies. Ease of use and maturity refers to our subjective opinion on the programming languages.

	OpenMP	**MPI**	**OpenCL**	**CUDA**	**Brook+**	**libspe**	**VHDL/Verilog**	**Mitrion-C**
Target platform	CPU CBEA	CPU	CPU GPU CBEA	GPU (NV)	GPU (AMD)	CBEA	FPGA	FPGA
Availability	Win Linux Mac	Win Linux Mac	Win Linux Mac	Win Linux Mac	Win Linux	Linux	Win Linux Mac	Win Linux Mac
Abstraction	Pragmas	API	API	API, compiler	API, compiler	API, compiler	API	Compiler
Host language	C, C++, Fortran	C, C++, Fortran	C	C, C++	C++	C, C++, Fortran	"C-like"	C
Kernel language	—	—	C99-based	C99-based, some C++	C99-based	C, Fortran, almost C++	—	C
Memory model	Shared	Distributed	∼ PGAS	∼ PGAS	Data streams	∼ PGAS	all	all
Data-parallelism	Global view	—	SPMD, SIMD	SPMD	SPMD	MPMD	all	all
Task-parallelism	—	Explicit	Full	Streams	Streams	Explicit	all	all
Ease of use	⋆⋆⋆	⋆	⋆⋆	⋆⋆	⋆⋆	⋆	⋆	⋆
Maturity	⋆⋆⋆	⋆⋆⋆	⋆	⋆⋆⋆	⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆⋆

OpenMP [22] is a set of pragmas used to annotate parallel sections of the source code. The compiler uses these annotations to generate parallel binary code. OpenMP is implemented in most modern compilers for multi-core architectures, and the IBM XL includes an implementation that leverages the computational power of the SPEs [77, 78, 79].

MPI [16] is the de facto API for parallel programming on clusters. It exposes a distributed memory model with explicit communication. Ohara et al. [80] were the first to use the MPI programming model to abstract the CBEA. Currently, there are two main implementations of MPI for the CBEA; one by Kumar et al. [81] and Krishna et al. [82]; and The Cell Messaging Layer [83]. Whilst the former employs a traditional MPI message passing philosophy, the latter focuses on receiver-initiated message passing to reduce latencies. Of the MPI abstractions, the Cell Messaging Layer appears to be the most mature technology.

Cell superscalar (CellSs) [84, 85] is a source-to-source compiler for the CBEA. CellSs is similar to OpenMP, as the source code is annotated, but instead of annotating parallel sections, functions that can be offloaded to the SPEs are annotated. These functions are then scheduled to the SPEs according to their dependencies.

Sequoia [86, 87] is a programming language for the CBEA and traditional CPU clusters. It targets vertical data movement in the memory hierarchy, such as moving data between main memory and the local store of an SPE. This contrasts the horizontal data movement found in programming models such as MPI. Vertical data movement is exposed to the programmer as a tree of tasks, where the tasks are typically used as wrappers for optimized native functions. An

automatic tuning framework for Sequoia also exists [88]. However, it should be noted that there has been no public release of Sequoia since 2007.

Field Programmable Gate Array Languages

The major drawback of FPGAs has been the time and expense to program them. Using FPGAs has clearly been cost-effective for communications, high-performance embedded computing, military, and space applications, but problematical for typical high-performance computing. Not all applications can benefit from FPGAs, and even for those that do, nothing is automatic: obtaining speedups may be time-consuming and arduous work. The ideal candidate for FPGA acceleration contains a single computational kernel that comprises most of the program runtime. This computational kernel should be possible to divide into hundreds of tasks or data-parallel operations. One example of such a program is the Smith-Waterman algorithm for local sequence alignment, which illustrates the potential for FPGA acceleration [89]. As FPGAs were developed by logic designers, they are traditionally programmed using circuit design languages such as VHDL [90] and Verilog [91]. These languages require the knowledge and training of a logic designer, take months to learn and far longer to code efficiently. Even once this skill is acquired, VHDL or Verilog coding is strenuous, taking months to develop early prototypes and often years to perfect and optimize. FPGA code development, unlike high-performance computing compilers, is slowed by the additional steps required to synthesize, place and route the circuit. These steps often take hours, or overnight, to complete. However, once the time is taken to code applications efficiently in VHDL, its FPGA performance is excellent. Since 2000, the severe programming limitation has been tended to by dozens of C-like programming languages such as Mitrion-C [92], Impulse-C [93], System-C [94] and Celoxica [95], and graphical languages such as DSPlogic [96] and Viva [97]. There is also an increasing use of shared libraries offered by Xilinx [98] and OpenFPGA [99].

Starbridge Systems and DSPlogic provide graphical icon-based programming environments. Similar to Labview, Viva allows FPGA users to write and run scientific applications without having to deal with esoteric timing and pinout issues of digital logic that require much attention in VHDL and Verilog. Viva coding is a two-step process: first code is written, debugged and tested using the graphical interface, and then automatic place and route synthesis is performed for the target FPGA system. This graphical coding process alleviates the need to know VHDL or Verilog, while the second step simplifies development, enabling users to focus on developing and debugging their algorithms. Viva has been used at NASA with great success.

An innovative approach for users to program FPGAs without circuit design skills, is provided by Mitrion-C and other "C to gate" languages. Users can program the Mitrion Virtual Processor in Mitrion-C, a C-based programming language with additions for FPGA memory and data access. The first step when using Mitrion-C is, just as Viva, to design, debug and test the program on a regular computer. The second step involves place and route synthesis, often a time consuming task.

Discussion

With heterogeneous architectures, care is required by the programmer to fully utilize hardware. One problem in sharing hardware is that execution units may starve each other, for example fighting over the same set of cache lines. Unfortunately, the future holds no promise of an easy way out, so programmers must write algorithms suitable for heterogeneous architectures

Listing 1: Comparison of matrix addition using different programming languages. The CPU shows the explicit double for-loop, where the out-most loop is run in parallel. The FPGA code shows a matrix adder that can add two matrices in each clock cycle. In addition to the code shown, around five lines of CPU code is required to open the FPGA device, load its registers with pointers to input and output data, start execution, and finally to close the device. In addition, the FPGA needs VHDL code that can read and write the matrices directly by using the pointers. The GPU, on the other hand is invoked in parallel over a grid, where each thread within a block has an implicitly given position. Only the GPU code is shown, and around ten lines of CPU code is required to allocate GPU memory, upload data to the GPU, execute the kernel, and read data back to main memory. Finally, the CBEA code shows how DMA requests load data into the local store prior to computation, and write back after completion. Notice that functions that begin with "my" have been renamed for readability. In addition to the SPE code shown, the program requires around 20 lines of code to create a context for each SPE, load the program into the SPEs, set arguments, create a CPU thread for each SPE context, and wait for the SPE programs to complete.

(a) CPU (OpenMP)

```
void add(float* c, float* a, float* b, int w, int h) {
#pragma omp parallel for
  for (int j=0; j<h; ++j) {
    for (int i=0; i<w; ++i) {
      c[j*h+i] = a[j*h+i] + b[j*h+i];
    }
  }
}
```

(b) FPGA (VHDL)

```
architecture behave of maxtrix_adder is
  constant w : integer := 10;
  constant h : integer := 10;
  signal a, b : array(0 to w-1, 0 to h-1)
                of std_logic_vector(7 down to 0);
  signal c : array(0 to w-1, 0 to h-1)
             of std_logic_vector(8 down to 0);
begin
  w_set : for i in 0 to w-1 generate
  begin
    h_set : for j in 0 to h-1 generate
    begin
      c(i, j) <= a(i,j) + b(i,j);
    end generate h_set;
  end generate w_set;
end behave;
```

Listing 1: (continued)

(c) GPU (CUDA)

```
__global__ void addKernel(float* c, float* a, float* b) {
  int i = blockIdx.i*blockDim.i+threadIdx.i;
  int j = blockIdx.j*blockDim.j+threadIdx.j;
  int w = gridDim.i*blockDim.i;
  c[j*w+i] = a[j*w+i] + b[j*w+i];
}
```

(d) CBEA (libspe)

```
int main(unsigned long long speid,
    unsigned long long argp,
    unsigned long long envp) {

  MyAddContext ctx __attribute__((aligned(128)));
  vector float *a, *b, *c;
  int nbytes;

  myDMAReadAsync(&ctx, argp, sizeof(MyAddContext));
  myDMAFlush();

  nbytes = ctx.n*sizeof(float);
  a = (vector float*) malloc_align(nbytes);
  b = (vector float*) malloc_align(nbytes);
  c = (vector float*) malloc_align(nbytes);

  myDMARead(b, ctx.b_addr, nbytes);
  myDMARead(c, ctx.c_addr, nbytes);
  myDMASync();

  for (int i=0; i<ctx.n/4; ++i)
    c[i] = spu_add(a[i], b[i]);

  myDMAWrite(ctx.c_addr, c, nbytes);
  myDMASync();

  return 0;
}
```

to achieve scalable performance. Table 2 shows representative languages mentioned in this section, comparing abstraction levels, memory models, and our subjective rating of their ease of use and maturity. In addition, Listing 1 shows actual code for each architecture.

All of the architectures we have described use memory latency hiding techniques. The GPU and CBEA both employ memory parallelism, where multiple outstanding memory requests are possible. On the GPU this is implicit by the use of hardware multi-threading, whereas it is explicitly managed on the CBEA in terms of DMA queues. Furthermore, the CBEA can lessen the effect of latencies by using software threads, or overlapping communication and computation with multi-buffering techniques. The FPGA typically employs pipelined designs in which on-chip RAM is used to store intermediate results. This efficiently limits off-chip communication. In contrast, traditional CPUs use power-hungry caches to automatically increase experienced memory performance. However, as previously noted, automatic caches can also worsen performance for many algorithms.

Traditional CPU cores impose special requirements to alignment when using SIMD instructions. Data typically has to be aligned to quadword boundaries, and full quadwords are loaded into the SIMD registers of the processors. The concept of coalesced and burst memory access on GPUs, and the requirements of alignment on the CBEA is strikingly similar. Thus, for high-performance code, there is little difference in the complexity of efficient memory access.

There is a wide difference between the architectures we have described when it comes to programming complexity and flexibility. All of the architectures may eventually be possible to program using OpenCL, and RapidMind already offers a common platform for the CPU, GPUs, and the CBEA. However, the architecture-specific languages expose certain differences. Programming the GPU for general purpose problems has become much easier with CUDA and Brook+. Nevertheless, writing code that fully utilizes the hardware can still be difficult. One remark for GPU programming is that optimization can be strenuous, where slight changes to the source code can have dramatic effects on the execution time. Another issue with GPUs is that they are best suited for streaming applications, because global communication and synchronization is particularly expensive. The CBEA, on the other hand, offers software and hardware where communication and synchronization is efficient. However, the programmer must explicitly manage scheduling and load balancing for maximum performance. Finally, for efficiency, FPGAs are typically programmed using low level languages that require a detailed knowledge of the hardware design, with an especially time consuming place and route stage which generates a valid design for a target FPGA. Nevertheless, this also implies that FPGAs can be extremely efficient for well-suited algorithms.

5 Algorithms and Applications

As noted in the introduction, most algorithms can benefit from heterogeneous architectures, where the application itself determines the best distribution of traditional and accelerator cores. Codes which lack compute intensity and large I/O requirements might prove challenging, as exemplified by the evaluation of Van Amesfoort et al. [100], where they compare multi-core CPU, the CBEA, and the GPU for a data-intensive radio-astronomy application. Large I/O requirements also make assessing performance problematic as individual kernel performance can deviate from overall application performance. Comparing individual performance can in general be difficult, as data movement times between host and device memory may vary. For

some applications it is valid to assume data resides in device memory, while for others it is not.

Asanovic et al. [11, 9] identify 13 distinct computing bottlenecks they term *motifs*. Che et al. [101] have qualitatively examined three of these motifs, dense linear algebra, combinatorial logic, and dynamic programming. Comparing development costs and performance on an FPGA, multi-core CPU and a GPU, they conclude that the FPGA excels at combinatorial logic, where efficient use of bit operations can be used, and GPUs excel at parallel workloads with deterministic memory access. The complex data flow of linear algebra, however, does not adapt well to FPGAs. The same is true for floating point, where fixed point arithmetic is more efficient. GPUs, on the other hand, have no problem with floating point, but perform poorly when there is limited exposed parallelism, and many memory accesses, such as that of the dynamic programming motif. Flynn et al. [102] state that FPGAs benefit most when used with deep pipelines. By storing intermediate data in registers or local memory, the von Neumann bottleneck is efficiently bypassed. However, as Beeckler and Gross [103] note in their work on particle simulations, development on FPGAs can still be costly when using languages such as VHDL. Using higher abstractions to the FPGA, such as Mitrion-C increases productivity dramatically. Mitrion-C code can compute 112 million particles per second with three Xilinx Virtex-4 LX200 FPGAs. FPGAs have also been used for tasks such as genome sequencing, molecular dynamics, weather/climate and linear algebra, with a 10-100 times speedup compared to CPUs [104]. Pico Computing, an FPGA board manufacturer, have also accelerated genome sequencing. They report a speedup of over 5000 times over CPUs using 112 FPGA devices, and the FPGAs consume less than 300 watts of power [105, 106]. Leveraging the CBEA for sequence analysis has been explored by Sachdeva et al. [107], where compute intensive functions are performed on the SPEs, and logic kept on the PPE. They conclude that the CBEA is attractive from an energy perspective. On the CBEA based Roadrunner supercomputer [36], Swaminarayam et al. [39] take an existing molecular dynamics code, and restructure it to suit the heterogeneous architecture. They achieve 368 teraflops, which corresponds to 28% of peak performance. In addition to speed increase, their implementation increased load balancing possibilities, as well as freeing up Opteron resources, enabling simultaneously analysis, visualization, or checkpointing.

There has been an exponential interest and use of node-level heterogeneous architectures in recent years. The maturity of heterogeneous computing has in the same time period gone from a proof-of-concept state to industrially interesting technology. With the ever-increasing application areas of heterogeneous architectures, an exhaustive survey is impossible. Therefore, we focus on a select few algorithms and applications that together form a basis for understanding state-of-the-art approaches.

Numerical Linear Algebra

Numerical linear algebra is a main building block in many numerical algorithms, making efficient execution essential. Basic Linear Algebra Subprograms (BLAS) [147] is the de facto API for low-level linear algebra operations. Implementations of BLAS exist for a variety of different architectures, and much work has been done to optimize these due to their frequent use. Both NVIDIA and AMD provide GPU-accelerated BLAS implementations, namely CUBLAS [148] and ACML-GPU [149]. BLAS is divided into three *levels*, where the first two operate on vectors, typically making them bandwidth limited. Utilizing more processor cores on such

Table 3: Summary of state-of-the-art approaches in heterogeneous computing. The table is not an exhaustive list, but gives an overview of state-of-the-art problems and techniques.

Application	Approach
Miscellaneous	Use of different programming languages (FPGA) [103]; Qualitative comparison (FPGA, CPU, GPU) [101]; Bioinformatics (CBEA) [107]; Restructuring of code for heterogeneous architectures (CBEA) [39]. Molecular Dynamics (FPGA) [104];
Dense Linear Algebra	Achieving peak performance (CBEA) [41, 108, 109]; Matrix multiplication, LU decomposition (FPGA) [110]; Use of registers instead of shared memory (GPU) [111]; Linpack (CBEA) [112]; Mixed precision (FPGA) [52].
Sparse Linear Algebra	Blocking (CBEA) [113]; Data structures (GPU) [114];
Fast Fourier Transform	Auto-tuning (GPU) [115]; Hierarchically decomposed (GPU) [116]; Iterative out-of-place (CBEA) [117]; Communication overheads (CBEA) [118]; Power and resource consumption (FPGA) [119]; Double precision performance (FPGA) [120].
Stencil computations	Cluster implementation (GPU) [121]; Varying stencil weights (CBEA, GPU) [122]; Domain blocking, register cycling (GPU) [123]; Domain blocking, loop unrolling (CBEA) [124]; Auto-tuning (CBEA, GPU) [125]; Time skewing (CBEA) [126, 127]
Random numbers	Sub-word operations (FPGA) [128]; Generating initial state [129]; Stateless (GPU) [130]; Tausworthe (GPU) [131]; Architectural constraints (GPU, CPU, FPGA) [132].
Scan	Up and down sweep (GPU) [133, 134]; Linked lists (CBEA) [72]; Histogram pyramids (GPU) [135, 136].
Sorting	Bitonic sort (GPU) [137]; Hybrid radix-bitonic, out-of-core (GPU) [138]; Radix (GPU) [139]; AA-sort (CBEA) [140].
Image Processing	Canny edge detection (GPU, FPGA) [141, 142]; OpenCV (GPU, CBEA) [143, 144, 145]; Computer Vision library (GPU) [146].

bandwidth limited problems has no effect as long as the bandwidth remains constant. On the other hand, using the vast bandwidth of accelerators, such as the GPU and CBEA, can increase performance over CPU implementations. FPGAs can also increase performance via pipelined designs where latency is traded for throughput. To take full advantage of the floating point performance of accelerators, applications must exhibit a high ratio of arithmetic to memory operations. Level 3 BLAS operates on matrices, and typically performs $\mathcal{O}(n^3)$ operations on $\mathcal{O}(n^2)$ data elements, thus well suited for heterogeneous architectures.

A key operation to solve dense linear systems is matrix multiplication. Matrix multiplication exhibits high arithmetic density, has regular memory access patterns and control flow, making it one of the very few operations where experienced and theoretical floating-point performance roughly match. Larsen and McAllister [150] developed the first algorithm for matrix multiplication using a graphics API, before GPU floating-point arithmetic were available. Matrix multiplication on GPUs has since been a popular research topic. CUBLAS 2.0 uses the matrix multiplication algorithm developed by Volkov and Demmel [111], optimized not only to hide memory latency between main graphics memory and shared memory, but also to reduce movement from shared memory to registers. Contrary to general guidelines, their approach consumes a considerable amount of registers, reducing occupancy (see Section 3), yet provides unprecedented performance. A subset of BLAS has also been implemented on the CBEA. Chen et al. [41], Hackenberg [108], and Alvaro et al. [109] present highly optimized matrix multiplication implementations, where the latter uses 99.8% of the floating-point capacity of the SPEs for $C = C - A \times B$. On FPGAs, both fixed and floating point matrix multiplication has been proposed. Zhuo and Prasanna [110] present blocked matrix multiplication. Using *processing elements* consisting of one floating-point multiplier, one floating point adder and a set of registers, they optimize performance with respect to clock speed and memory bandwidth use. The use of more processing elements negatively affects attainable clock speed, whilst increased block size positively affects memory bandwidth use. Their design is further extendible to use several FPGAs, each with a separate local memory, and they attain 4 gigaflops performance on a Virtex-II

XC2VP100.

Matrix multiplication is also a key element in the Linpack benchmark, used to classify computers on the Top500 list. Kistler et al. [112] implemented the benchmark on the CBEA-based Roadrunner supercomputer [36]. Using a matrix multiplication algorithm similar to that of Alvaro et al. [109], they achieved 75% of theoretical peak performance for the full benchmark.

LAPACK is a commonly used library for high-level algorithms from numerical linear algebra, that relies on parallel BLAS implementations to benefit from multi-core and heterogeneous architectures. To take full advantage of parallel BLAS implementations, however, LAPACK must be adapted to avoid unnecessary memory allocations and copying of data. PLASMA [151] is a new initiative to develop a linear algebra package that is more suitable for multi-core and heterogeneous architectures. Buttari et al. [152] describe how operations like Cholesky, LU, and QR factorizations can be broken into sequences of smaller tasks, preferably Level 3 BLAS operations. A dependency graph is used to schedule the execution of tasks, so individual tasks can be assigned to one or more accelerator cores. On the FPGA, LU decomposition has been implemented by Zhuo and Prasanna [110]. They use a linear array of processor elements consisting of a floating point multiplier and a floating point adder. The first processor element also includes a floating point divider, requiring large amounts of real-estate. The whole design is limited by the frequency of the floating point divider for a small number of processing elements. A problem with using many processing elements is that the complex wiring degrades clock frequency. Going from one to 18 processing elements, the clock frequency is degraded by 25%. A mixed precision LU decomposition was developed by Sun et al. [52], where the FPGA calculates a low-precision approximation to the solution using the Dolittle algorithm, followed by an iterative refinement step on the CPU. Using lower precision for most calculations on the FPGA lessens latencies of floating point units and uses less bandwidth, thus speeding up computations.

Whilst LAPACK is a highly efficient library, it requires low-level programming. MATLAB addresses this by offering a high-level interactive interface. There have been several efforts to accelerate MATLAB using GPUs, where native MATLAB syntax is used to transparently execute computations on the GPU. The first approach was given by Brodtkorb [153], providing a MATLAB extension calling an OpenGL-based backend to carry out the computations. A similar approach is taken in Jacket [154], a commercial product from Accelereyes.

Sparse linear algebra is most often bandwidth limited, as random sparse matrices are typically stored using a format such as compressed row storage (CRS). The CRS format consists of one vector with the values of all non-zero entries, stored row wise, and a second vector containing the column index of each entry. The start index of each row is stored in a third vector, yielding a very compact storage format. However, such storage formats require three lookups to find the value of an element: first the row, then the column, and finally the value. Due to such indirections, it is common with less than 10% bandwidth utilization, and less than 1% compute utilization when computing with sparse matrices on the CPU [155]. An important algorithm in sparse linear algebra is matrix-vector multiply, often used in matrix solvers such as the conjugate gradients algorithm. Williams et al. [113] explore a range of blocking strategies to avoid cache misses and improve performance of sparse matrix-vector multiply on multi-core platforms. They use asynchronous DMA transfers on the CBEA to stream blocks of matrix elements in and blocks of vector elements out of each SPE. A branchless technique

similar to that of Blelloch et al. [156] is used to avoid mis-predictions, and SIMD instructions are are used to further increase performance. They are, on average, able to utilize 92.4% of the peak bandwidth for a wide range of sparse matrices. This is far greater than the bandwidth utilization of multi-core CPUs when similar techniques are employed. On the GPU, Bell and Garland [114] explore sparse matrix-vector product using CUDA. They investigate a wide range of data structures including variants of CRS. However, the variants of CRS suffers either from non-coalesced memory reads or unbalanced workloads within warps. To overcome this, they developed a hybrid format, where structured parts of the matrix are stored using a dense format, and the remaining non-zero elements in a CRS format. Using this technique, a single NVIDIA GTX 280 GPU was twice as fast as two CBEAs running the aforementioned algorithm of Williams et al. [113].

The Fast Fourier Transform

The fast Fourier transform (FFT) is at the heart of many computational algorithms, such as solving differential equations in a Fourier basis, digital signal processing, and image processing. The FFT is usually among the first of algorithms ported to a new hardware platform, and efficient vendor-specific libraries are widely available.

NVIDIA provides the CUFFT library [157] for CUDA-enabled GPUs, modeled on the FFTW library [158]. Version 2.3 supports single and double precision complex values, while real values are supported for convenience only, as the conjugate symmetry property is not used. Nukada and Matsuoka [115] present an auto-tuning 3D FFT implementation that is 2.6–8 times faster than CUFFT version 2.1, and with only small performance differences between power-of-two and non-power-of-two FFTs. Their implementation is 2.7–16 times faster on a GTX 280, compared to FFTW running on a 2.2 GHz AMD quad core Phenom 9500. Govindaraju et al. [116] have implemented three different FFT approaches using CUDA; shared memory, global memory, and a hierarchically decomposed FFT. At runtime, they select the optimal implementation for the particular FFT. They report surpassing CUFFT 1.1 on both performance and accuracy. On a GTX280, they report 8–40 times speedup over a 3.0 GHz Intel Core2 Quad Extreme using Intel MKL. Volkov and Kazian [159] employ the approach taken by Volkov and Demmel [111], where register use in CUDA is increased, and report results equivalent to that of Govindaraju et al. [116]. Lloyd, Boyd and Govindaraju [160] have implemented the FFT using DirectX, which runs on GPUs from different vendors, and report speeds matching CUFFT 1.1.

IBM has released FFT libraries for the CBEA that can handle both 1D, 2D and 3D transforms [161, 162]. The FFTW library includes an implementation for the CBEA [158]. Bader and Agarwal [117] present an iterative out-of-place FFT implementation that yields 18.6 gigaflops performance, several gigaflops faster than FFTW, and claim to be the fastest FFT implementation for 1–16000 complex input samples. Xu et al. [118] have another approach where they address communication latencies. By using indirect swap networks, they are able to halve communication overheads.

Both Altera and Xilinx provide FFT IP cores [163, 164] that can be set up for both single precision and fixed point FFT computations of vectors of up-to 65536 elements. McKeown and McAllister [119] optimize FFT performance with respect to power and resource consumption, and show over 36% reduction in power consumption, and 61% reduction in slice use compared to the 64-point Xilinx IP core. Hemmert and Underwood [120] have given some remarks on

double precision FFTs on FPGAs, discussing pipelined versus parallel designs. They report that the optimal design depends on both the FFT size, as well as the FPGA fabric size.

Stencil Computations

Stencil computations are central in many computational algorithms, including linear algebra, solving partial differential equations, and image processing algorithms. Basically, a stencil computation is a weighted average of a neighbourhood, computed for every location in a grid. The stencil determines the size and weighing of the neighbourhood, and all grid points can be computed in parallel by using a separate input and output buffer. The computational complexity of stencil computations is often low, making them memory bound, and thus, leveraging the high bandwidth of accelerators can yield performance gains. As an extreme example, Mattson et al. [165] demonstrated one teraflops performance whilst using a mere 97 watts of power on the Intel Teraflops Research Chip [166].

Stencil computations can be used efficiently to solve partial differential equations that are discretized using an explicit finite difference approach. Hagen et al. [48, 167] explored the suitability of early GPUs for complex finite difference schemes to solve the shallow water and Euler equations. Speed-ups of 10-30 times compared to equivalent CPU implementations were presented, even on non-trivial domains. The Asian Disaster Preparedness Center, and the Japan Meteorological Agency solve the shallow water equations in their prototype tsunami simulator [121]. Their single node GPU implementation is 62 times faster than the corresponding CPU version, and they estimate that a simulation of a 6000×6000 mesh will run in less than 10 minutes using a cluster of 32 GPUs. Christen et al. [122] explore stencil computations in a bio-medical simulation where the stencil weights vary in space. This makes the computation far more complicated compared to a static stencil, yet they report 22 gigaflops performance using two CBEAs on a QS22 blade, and 7.8 gigaflops on the NVIDIA GeForce 8800 GTX. For comparison, a dual quad-core Intel Clovertown achieves 2 gigaflops for the same computation.

Araya-Polo et al. [124] and Micikevicius [123] solve the linear wave equation in three dimensions on the CBEA and GPU, respectively. The wave equation is key to reverse time migration in seismic processing, and they solve it using an explicit finite difference scheme that ends up in a stencil computation. On the CBEA, Araya-Polo et al. divide the computational domain into blocks, where the SPEs independently compute on separate blocks. They traverse each block in 2D planes, and are able to hide memory transfers by explicitly overlapping communication with computations. They also employ extensive loop unrolling enabled by the large number of registers, and report 21.6 GB/s and 58.3 gigaflops performance, corresponding to 85% and 25% of peak performance for bandwidth and arithmetics, respectively. Their implementation runs eight times faster than an equally tuned CPU counterpart on a 2.3 GHz PowerPC 970MP. On the GPU, Micikevicius describes another approach, where the volume is similarly divided into blocks. Within each block, the volume is traversed in a single pass by cycling over 2D slices. Shared memory is used to store the 2D slice, including the apron as defined by the stencil. The depth dimension is stored in per-thread registers, which are used as a cyclic buffer. Consistent results of 45 to 55 GB/s, and up-to 98 gigaflops floating point performance are reported. This corresponds to around 50% and 9% of peak bandwidth and arithmetic performance, respectively, and the bottleneck appears to be global memory latency. A GPU cluster implementation is also presented. Super-linear speedup over one GPU is achieved as each card

(a) Time-skewing **(b)** Cache Oblivious

Figure 6: Cache-aware and Cache-oblivious blocking algorithms.

uses less memory, thus causing fewer translation lookaside buffer (TLB) misses.

Datta et al. [125] employ an auto-tuning approach to optimize performance of a seven-point 3D stencil computation on multi-core CPUs, the CBEA, and a GPU. In double precision they achieve 15.6 gigaflops on a QS22 blade, and 36.5 gigaflops on the NVIDIA GTX 280 GPU. However, whilst they report that the CBEA is memory bound, they conclude that the GPU is computationally bound for double precision. In total the GPU is 6.8 times faster than their best auto-tuned code on a traditional symmetric multi-core platform, a dual socket Sun UltraSparc T2+ solution with a total of 16 cores.

Traditional cache based architectures can employ algorithmic optimizations such as time skewing [168] to increase cache reuse. Time skewing requires the user to explicitly partition the domain into space-time parallelograms, as shown in Figure 6a. After partitioning, the stencil kernel is executed on each parallelogram in turn. This concept has been further developed into recursive cache-oblivious algorithms that subdivide the trapezoids until they contain only one time-step [169], as illustrated in Figure 6b. Both cases can be extended to parallel execution by processor overlap. Time skewing was mapped to the CBEA by Kamil et al. [127] and Datta et al. [126]. In their early work, they used four step time skewing with overlapping parallelograms between SPEs to solve a three dimensional heat diffusion problem using a finite difference scheme. They demonstrate 65.8 gigaflops in single precision for the 3.2 GHz CBEA. For double precision calculations, the early CBEA chips were computationally bound, and thus, do not benefit from memory optimizations.

Random Number Generation

Monte Carlo integration methods are based on function evaluations at randomly sampled points. The idea is to run the same simulation numerous times continuously drawing new random numbers. In principle, this approach is embarrassingly parallel. However, the production of random numbers, which fuels the process, is usually not embarrassingly parallel.

Pseudorandom number generators approximate truly random numbers, and are often formulated as forms of recurrence relations, maintaining a state of the m most recently produced numbers and tapping into this state to produce a new pseudorandom number. Linear congruential generators are defined by the simple recurrence relation $x_i = (ax_{i-1} + c) \mod m$, producing sequences with a modest maximum period of m. Multiple recursive generators combine the output of multiple generators, extending the overall period to the smallest common multiplier of the periods of the combined generators. The Mersenne Twister [170] is based on a linear recurrence function tapping into a state of 624 32-bit integers. Its output is of high quality, and with a very long period, it is often the generator of choice.

Manipulation at word level is the natural primitive for most processors, and thus, most pseudorandom number generators work on the level of words as well. A variant of the Mersenne Twister is the SIMD-oriented Fast Mersenne Twister [171], which utilizes the 128-bit operations provided by SSE2 and AltiVec, and roughly doubles the performance. On the other hand, FPGAs provide very fine-grained binary linear operations, which opens possibilities for sub-word operations, investigated by e.g., Thomas and Luk [128].

Recurrence relations are serial in nature, but a common strategy is to run a set of pseudorandom number generators in parallel, taking care when initializing the generators to avoid correlations. For the Mersenne Twister, Matsumoto and Nishimura [129] proposed a scheme to generate the initial state for an array of independent generators. Further, the size of the generator's state may pose a problem: On NVIDIA GPUs, a single warp of 32 threads, each running the Mersenne Twister, would require 78 KiB of memory just for the state, almost five times the amount of shared memory.

The output of pseudorandom number generators is usually uniformly distributed. There are various approaches to convert uniformly distributed numbers to other distributions. One example is a rejection method, which rejects samples that do not fit the distribution. This could introduce code path divergences on GPUs: The Ziggurat method has only 1% risk of divergent behaviour, but with 32 threads in a warp, the risk of a divergent warp is 30% [132]. On FPGAs, such a varying amount of outputs per invocation requires the implementation to handle pipeline bubbles. A more direct approach is to use the inverse cumulative distribution function. For the normal distribution, a closed expression for the inverse cumulative distribution function does not exist, so one must approximate using piecewise rational polynomials. The Box-Muller transform uses trigonometric operations and logarithms, which are generally considered expensive. On GPUs, however, fast versions of these instructions are built in, as they are very common in computer graphics.

Most random number generators can be used directly on the CBEA, and the CBEA SDK [71] contains a Monte Carlo library with implementations of the Mersenne Twister, the SIMD-oriented Fast Mersenne Twister, Kirkpatrick-Stoll pseudorandom number generators, as well as the Box-Muller transform and transformation using the inverse cumulative normal distribution. On GPUs, the early approach of Sussman et al. [130] to implementing random number generation used stateless combined explicit inverse congruential generator. Howes and Thomas [131] give an overview of implementing pseudorandom number generators in CUDA, pointing to the problems of large state vectors and that integer modulo is expensive on current GPUs, and propose a hybrid generator that combines a Tausworthe generator [172] with linear congruential generators. The CUDA SDK [173] contains an implementation running an array of Mersenne Twisters in parallel, storing the states in local memory. The Brook+ SDK [65] contains an implementation of the SIMD-oriented Fast Mersenne Twister, suitable for the superscalar architecture of the AMD GPUs. Further, Thomas et al. [132] evaluate strategies for generating pseudorandom numbers on CPUs, GPUs, FPGAs, and massively parallel processor arrays, focusing on algorithmic constraints imposed by the respective hardware architectures. They give an interesting benchmark, comparing both raw performance and energy efficiency for their chosen approaches. An Intel Core 2 CPU produces on average 1.4 gigasamples/s at an energy efficiency of 5 megasamples/joule, an NVIDIA GTX 280 produces 14 gigasamples/s at 115 megasamples/joule, and a Xilinx Virtex-5 FPGA produces 44 gigasamples/s at 1461 megasam-

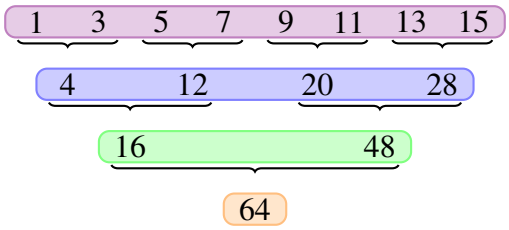


Figure 7: Reduction to sum n elements requires $\log(n)/\log(p)$ passes, where p is the reduction factor per pass.

ples/joule.

Data-parallel primitives and sorting

Data-parallel primitives are building blocks for algorithm design that execute efficiently on data-parallel architectures. Particularly common patterns are reduce, partial sums, partition, compact, expand, and sorting.

The reduce pattern reduces an array of values into a single result, e.g., calculating the sum of an array of numbers. In this case, the direct approach of adding numbers one-by-one, $(((a_0 + a_1) + a_2) \cdots + a_{n-1})$ has both work and step efficiency of $\mathcal{O}(n)$. However, if we do pairwise recursive additions instead, shown in Figure 7, the step efficiency is reduced to $\mathcal{O}(\log_2 n)$ if we can carry out n additions in parallel. Pairwise recursive addition is an example of the binary reduce pattern, which works with any binary associate operator.

Prefix-sum scan, initially proposed as part of the APL programming language [174], computes all partial sums, $[a_0, (a_0+a_1), (a_0+a_1+a_2), \ldots, (a_0+\cdots+a_{n-1})]$, for any binary associative operator. The approach of Hillis and Steele [175] computes scan in $\mathcal{O}(\log_2 n)$ steps using $\mathcal{O}(n \log_2 n)$ arithmetic operations. This approach was implemented by Horn [176] in OpenGL, producing the first efficient implementation of scan on the GPU. Another implementation is provided in the Brook+ SDK [65]. Blelloch [177] presents a two-phase approach of scan using an up-sweep phase and a down-sweep phase. With this approach, the number of arithmetic operations is reduced to $\mathcal{O}(n)$. Sengupta et al. [178] proposed an OpenGL implementation of this approach, and Greß et al. [134] proposed another variant based on 2D-arrays and 4-way reductions, also using OpenGL. Harris et al. [179] proposed an implementation optimized for CUDA, which has been incorporated in the CUDA Data Parallel Primitives Library [180]. Blelloch [177] presents a range of uses for scan, complemented by Sengupta et al. [133] with particular emphasis on GPU implementation. On the CBEA, Bader et al. [72] investigate prefix-sum scan of arbitrary linked lists. They partition the lists into a set of sublists, and find the local prefix-sum of each sublist in parallel, using software-managed threads (see Section 4). In a subsequent step, the prefix-sum of the sums of the sublists are computed, which is then used to update local prefix-sums forming the global prefix-sum.

The partition pattern can be implemented using scan. For example, Blelloch [177] suggests implementing the split phase of radix sort using scan. In step i of radix sort, the data is partitioned into two sets depending on whether bit i is set or not. By creating an array containing 1 for every element with bit i not set and 0 otherwise, the $+$-scan of this array produces the offsets in the resulting partitioned array for the 0-bit elements. The $+$-scan is simply scan using

the addition operator. Running a $+$-scan in reverse order produces the offsets from the end of the output array for the 1-bit elements, and these two scans can be carried out in parallel.

Compaction is a variation of partition in which the output is a compact array containing only a subset of the elements in the input array, and expansion is the situation where an input stream element is expanded to multiple elements in the output stream. Both can be done using scan with a predicate that associates input stream elements with the output stream element count for that element. For compaction, this amounts to associating 1 with elements that should be kept and 0 with the elements to be discarded. Running scan on the result of the predicate produces the offset into the compacted array for each input array element to be kept, and by iterating over the input elements, consulting the results of the predicate function and scan, elements to be kept can be written to its location in the compacted array. This operation requires writing to arbitrary memory locations, that is, scatter writes, which was unavailable on early GPUs. Horn [176] circumvented this by iterating over the compacted elements, using binary search on the scan output to find the corresponding input array element. Ziegler et al. [135] proposed a method for this problem in OpenGL using histogram pyramids. Histogram pyramids use 4-way reduction to produce a pyramid of partials sums, similar to the up-sweep phase of Greß et al. [134], however, the down-sweep phase is omitted. The algorithm iterates over the compacted array and the position of the corresponding input element is found by traversing the histogram pyramid from top to bottom. An extension to the OpenGL histogram pyramid algorithm allowing stream expansion was proposed by Dyken et al. [136], along with a variant adapted to the CUDA architecture.

Another important algorithmic primitive is sorting a one-dimensional vector, a common task in a wide range of applications. However, traditional sorting algorithms developed for the CPU do not map directly to new architectures. Early GPGPU sorting algorithms implemented using graphics APIs could not output data to arbitrary memory locations. Instead, Purcell et al. [137] implemented bitonic sort, where the basic operation is to compare two values and output the smallest or largest element. Govindaraju et al. [138] implemented a hybrid radix-bitonic sort to sort out-of-core data sets, allowing the sorting key to be larger than 32 bit. In their implementation, the GPU mainly performs a bitonic sort.

The introduction of shared memory opened up the possibility to implement a wider range of algorithms directly on GPUs. Generally, the algorithms are based on first sorting a block of data that fits in shared memory, and then merge blocks. This strategy is also suited for other parallel architectures, such as multi-core CPUs and the CBEA, and is similar to algorithms used for traditional distributed memory systems. On the GPU, Satish et al. [139] base their implementation on radix sort, and use the aforementioned data parallel primitives in what they claim is the fastest sorting implementation written in CUDA. AA-sort by Inoue et al. [140] target CBEA and multi-core processors with SIMD capabilities. In their implementation, data blocks fitting in cache/local store are sorted using a SIMD version of comb sort. Chhugani et al. [181] presented a sorting algorithm suitable for parallel architectures with a shared and coherent L2 cache and wide SIMD capabilities. The upcoming Larrabee GPU is an example of such an architecture. First, they divide the data into blocks that fit in L2 cache and sort each block. Each block is split once more, to allow each core to work on one sub-block. Then the cores cooperate on merging the sorted sub-blocks. This structure allows the blocks to reside in L2 cache until the entire block is sorted.

Image Processing and Computer Vision

Many algorithms within image processing and computer vision are ideal candidates for heterogeneous computing. As the computational domain often is regular and two-dimensional, the algorithms often include embarrassingly parallel stages, in addition to high-level reasoning. One example is Canny edge detection [182], which itself is often used as input to other algorithms. In Canny edge detection, a blurred gradient image is created by convolving the input with a Gauss filter, followed by a gradient convolution filter. Local maxima in the gradient direction are considered candidate edges. To distinguish between false and true edges, a two-step hysteresis thresholding called non-maximum suppression is performed. Magnitudes larger than the upper threshold are considered true edges, and magnitudes lower than the lower threshold are considered non-edges. The remaining edges are then classified as edges only if they can be directly connected to existing true edges. This is an iterative process, where the real edges grow out along candidate edges.

Luo and Duraiswami [141] present a full implementation of Canny edge detection on the GPU. The convolution steps are fairly straightforward, and are efficiently implemented using techniques described in the NVIDIA CUDA Programming Guide. However, they operate on RGB images with eight bits per channel, whereas the memory bus on the GPU is optimized for 32-bit coalesced reads. They solve this issue by using 25% fewer threads during memory read, but where each thread reads 32 bits. These 32 bits are subsequently split into the respective eight bit color channels using efficient bit shifts. Thus, they require no redundant memory reads, and the access is coalesced. For the final part of the algorithm, they use a four-pass algorithm to connect edges across CUDA block boundaries. Within each block, the threads cooperate using a breadth first search, and edges that cross boundaries are connected between the passes. On an NVIDIA 8800 GTX, the implementation executed over three times faster compared to an optimized OpenCV implementation on an 2.4 GHz Intel Core 2 Duo for different test images. The bottleneck is the edge connection step, which is a function of the number of edges, and dominates 70% of the computational time. On the FPGA, Neoh and Hazanchuk [142] utilize a pipelining approach to find edges. Both convolutions are implemented using symmetric separable filters in a pipeline where latency is traded for throughput. Connecting candidate edges is done using a FIFO queue initially containing all classified edges. For each element in the FIFO queue, candidate edges in the three by three neighbourhood are found, marked as true edges and added to the FIFO queue. On the Altera Stratix II FPGA, the implementation is capable of processing 4000 256×256 images per second.

Canny edge detection is, as mentioned, often an input to other algorithms for high-level reasoning, commonly referred to as computer vision. OpenCV [183] is an open source computer vision library, focusing on real-time performance with a highly tuned CPU implementation. OpenCV has also been mapped to both GPUs and the CBEA. GpuCV [143] uses GLSL, CUDA, or a native OpenCV implementation, and can automatically switch to the best implementation at run-time with a small overhead. Operators such as Erode and Sobel are benchmarked, and show speedups of 44 to 193 times on an NVIDIA GTX 280 compared to a 2.13 GHz Core2 Duo. CVCell is a project that consists of CBEA accelerated plugins for OpenCV [144]. Speedups of 8.5 to 37.4 times compared to OpenCV on a 2.66 GHz Intel Core 2 Duo are reported for morphological operations [145]. Finally, MinGPU [146], is a computer vision library built

on top of OpenGL. This preserves portability, but often sacrifices performance, as OpenGL initiation can be more costly compared to CUDA.

It becomes increasingly important to design image processing algorithms for heterogeneous architectures, for example using total variation methods instead of graph cut methods [184] for applications such as optical flow computation, image registration, 3D reconstruction and image segmentation. Michel et al. [185] use the GPU to enable a humanoid robot to track 3D objects. Color conversion, Canny edge detection, and pose estimation is run on the GPU, and iterative model fitting is run on the CPU. This enables real-time performance, and leaves enough CPU resources to calculate footsteps and other tasks. Another example is Boyer et al. [186], who use the GPU to speed up automatic tracking of white blood cells, important in research into inflammation treatment drugs. Currently, this is a manual process, where the center of each white blood cell is marked manually. They employ a Gradient Inverse Coefficient of Variation algorithm to locate the cells initially, and in subsequent frames track the boundary. They isolate a rectangle around the original cell position, and compute a snake algorithm on rectangles in parallel. They report 9.4 and 27.5 times speedup on an NVIDIA GTX 280 compared to an OpenMP implementation running on a 3.2 GHz Intel Core 2 Quad Extreme. Finally, Baker et al. [187] compared FPGAs, GPUs, and the CBEA for matched filter computations, used to analyze hyperspectral images. The algorithm is partitioned onto the CPU and accelerator core, and they report the NVIDIA 7900 GTX to be most cost-efficient, and the CBEA to give the best speedup and speedup per watt.

6 Summary

We have described hardware, software, and state-of-the-art algorithms for GPUs, FPGAs, and the CBEA in the previous sections. In this section, we summarize the architectures, give our view on future trends, and offer our concluding remarks.

Architectures

The three architectures we have reviewed have their distinct benefits and drawbacks, which in turn affect how well they perform for different applications. The major challenge for application developers is to bridge gap between theoretical and experienced performance, that is, writing well balanced applications where there is no apparent single bottleneck. This, of course, varies from application to application, but also for a single application on different architectures. However, it is not only application developers that have demands to architectures: economic, power, and space requirements in data-centers impose hard limits to architecture specifications.

The GPU is the best performing architecture when it comes to single precision floating point arithmetic, with an order of magnitude better performance compared to the others. When considering the price and performance per watt, the GPU also comes out favourably. However, the GPU performs best when the same operation is independently executed on a large set of data, disfavouring algorithms that require extensive synchronization and communication. GPU double precision performance is now available, with one fifth of the single precision performance for AMD, and one half for NVIDIA. The new programming environments, such as CUDA and Brook+, enable efficient development of code, and OpenCL further offers hope for code portability.

The CBEA is still twice as fast as the state-of-the-art CPU, even though it is now three

years old. Furthermore, it offers a very flexible architecture where each core can run a separate program. Synchronization and communication is fast, with over 200 GB/s bandwidth on the element interconnect bus. This makes the CBEA extremely versatile, yet somewhat more difficult to program than the GPU. It is also a very well performing architecture when it comes to double precision. However, it is rather expensive.

FPGA performance is difficult to quantify in terms of floating point operations, as floating point is typically avoided. FPGAs are much more suited for algorithms where fixed point, integer, or bit operations are key. For such tasks, the FPGA has an outstanding raw performance, and an especially good performance per watt ratio. If one needs floating point operations, FPGA implementations benefit from tuning the precision to the minimum required level, independent of the IEEE-754 standard. However, FPGAs have a discouraging price tag. They can also be very difficult to program, but new languages such as Mitrion-C and Viva offer promising abstractions.

Emerging Features and Technology

We believe that future applications will require heterogeneous processing. However, one must not overlook legacy software that has existed for decades. Such software is economically unfeasible to redesign, and must use library calls to benefit from heterogeneous processing. This has the drawback captured in Amdahl's law, where the serial part of the code quickly becomes the bottleneck. New applications and algorithms, however, can be designed for existing, and future, architectures.

In the future, we see it as likely that GPUs will enter the HyperTransport bus, similarly to current FPGAs. We believe that such GPU cards on the HyperTransport bus will contain separate high-speed memory, and target server markets at a higher premium. Using NUMA technology, we even see the possibility of a shared address space, where both the CPU and GPU can transparently access all memory. Such an architecture would indeed be extremely useful, alleviating the PCI express bottleneck, but more importantly the issue of distinct memory spaces. Furthermore, it will be interesting to see the impact of the Larrabee on the future of GPU computing.

The CBEA roadmap schedules a chip with two Power processor cores and 32 synergistic processing elements for 2010. The state of these plans is uncertain, but we find it likely that similar designs will be used in the future. The major strengths of the CBEA is the versatility of the synergistic processing units, and the use of local store memory in conjunction with DMA. We believe such features will become increasingly used as power constraints become more and more pressing.

The obvious path of FPGAs is to simply continue increasing the clock frequency, and decrease the production techniques. However, the inclusion of power processors in hardware is an interesting trend. We believe that the trend of an increased number of special-purpose on-chip hardware, such as more floating point adders, will continue. This will rapidly broaden the spectrum of algorithms that are suitable for FPGAs.

Concluding remarks

Heterogeneous computing has in very few years emerged as a separate scientific field encompassing existing fields such as GPGPU. One of the reasons that heterogeneous computing has succeeded so well until now has been the great success of GPGPU. However, the field of

GPGPU is focused only on the use of graphics cards. In the future, we believe that algorithms cannot be designed for graphics cards alone, but for general heterogeneous systems with complex memory hierarchies. We do not believe that symmetric multiprocessing has a future in the long term, as it is difficult to envision efficient use of hundreds of traditional CPU cores. The use of hundreds of accelerator cores in conjunction with a handful of traditional CPU cores, on the other hand, appears to be a sustainable roadmap.

We have little belief in the one-size-fits-all for scientific computing. The three architectures we have described are currently addressing different needs, which we do not see as transient. The GPU maximises highly parallel stream computing performance, where communication and synchronization is avoided. The CBEA offers a highly versatile architecture, where each core can run a separate program with fast inter-core communication. Finally, the FPGA offers extreme performance for applications relying on bit, integer, logic and lower precision operations.

Acknowledgements

The authors would like to thank Gernot Ziegler at NVIDIA Corporation, Knut-Andreas Lie and Johan Seland at SINTEF ICT, and Praveen Bhaniramka and Gaurav Garg at Visualization Experts Limited for their fruitful comments and feedback. We also appreciate the valuable input from the anonymous reviewers, and the continued support from AMD, IBM, and NVIDIA.

Bibliography

[1] J. Gustafson, "Reconstruction of the Atanasoff-Berry computer," in *The first computers: history and architectures*, R. Rojas and U. Hashagen, Eds. Cambridge, MA, USA: MIT Press, 2000, ch. 6, pp. 91–106.

[2] W. Aspray, "The Intel 4004 microprocessor: What constituted invention?" *Hist. of Computing*, vol. 19, no. 3, pp. 4–15, 1997.

[3] G. Boone, "Computing systems CPU," [United States Patent 3,757,306], August 1971.

[4] R. Holt, "LSI technology state of the art in 1968," September 1998.

[5] G. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, April 1965.

[6] E. Mollick, "Establishing Moore's law," *Hist. of Computing*, vol. 28, no. 3, pp. 62–75, July-Sept. 2006.

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

[8] B. Stackhouse, B. Cherkauer, M. Gowan, P. Gronowski, and C. Lyles, "A 65nm 2-billion-transistor quad-core Itanium processor," in *Intl. Solid-State Circuits Conf.*, Feb. 2008, pp. 92–598.

[9] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, E. Lee, N. Morgan, G. Necula, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "The parallel computing laboratory at u.c. berkeley: A research agenda based on the berkeley view," EECS Department, University of California, Berkeley, Tech. Rep., December 2008.

[10] M. Hill and M. Marty, "Amdahl's law in the multicore era," *IEEE Computer*, vol. 41, no. 7, pp. 33–38, July 2008.

[11] 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," EECS Department, University of California, Berkeley, Tech. Rep., December 2006.

[12] J. Backus, "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[13] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, 1997.

[14] U. Drepper, "What every programmer should know about memory," http://people.redhat.com/drepper/cpumemory.pdf, November 2007, [visited 2009-03-20].

[15] I. Foster and K. Chandy, "Fortran M: A language for modular parallel programming," *Parallel and Distributed Computing*, vol. 26, 1992.

[16] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*, 2nd ed. Cambridge, MA, USA: MIT Press, 1999.

[17] C. Koelbel, U. Kremer, C.-W. Tseng, M.-Y. Wu, G. Fox, S. Hiranandani, and K. Kennedy, "Fortran D language specification," Tech. Rep., 1991.

[18] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald, "Vienna Fortran — a language specification version 1.1," Austrian Center for Parallel Computation, Tech. Rep., 3 1992.

[19] H. Richardson, "High Performance Fortran: history, overview and current developments," 1.4 TMC-261, Thinking Machines Corporation, Tech. Rep., 1996.

[20] K. Kennedy, C. Koelbel, and H. Zima, "The rise and fall of High Performance Fortran: an historical object lesson," in *Conf. on History of Programming Languages*, 2007.

[21] S. Benkner, E. Laure, and H. Zima, "HPF+: An extension of HPF for advanced applications," The HPF+ Consortium, Tech. Rep., 1999.

[22] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[23] R. Numrich and J. Reid, "Co-Array Fortran for parallel programming," Fortran Forum, Tech. Rep., 1998.

[24] "UPC language specification v1.2," UPC Consortium, Tech. Rep., 2005.

[25] P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, B. Liblit, G. Pike, J. Su, and K. Yelick, "Titanium language reference manual," U.C. Berkeley, Tech. Rep., 2005.

[26] "Chapel language specification 0.782," Cray Inc., Tech. Rep.

[27] "Report on the experimental language X10, draft 0.41," IBM, Tech. Rep., 2006.

[28] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[29] M. Flynn, "Some computer organizations and their effectiveness," *Transactions on Computing*, vol. C-21, no. 9, pp. 948–960, 1972.

[30] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[31] 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," *Transactions on Graphics*, vol. 27, no. 3, pp. 1–15, August 2008.

[32] NVIDIA, "NVIDIA GeForce GTX 200 GPU architectural overview," May 2008.

[33] AMD, "R700-family instruction set architecture," March 2009.

[34] NVIDIA, "NVIDIA CUDA reference manual 2.0," June 2008.

[35] ——, "NVIDIA's next generation CUDA compute architecture: Fermi," October 2009.

[36] K. Barker, K. Davis, A. Hoisie, D. Kerbyson, M. Lang, S. Pakin, and J. Sancho, "Entering the petaflop era: The architecture and performance of Roadrunner," in *Supercomputing*, November 2008.

[37] "Top 500 supercomputer sites," http://www.top500.org/, June 2009.

[38] "Top green500 list," http://www.green500.org/, November 2008.

[39] S. Swaminarayan, K. Kadau, T. Germann, and G. Fossum, "369 tflop/s molecular dynamics simulations on the Roadrunner general-purpose heterogeneous supercomputer," in *Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–10.

[40] IBM, "PowerPC microprocessor family: Vector/SIMD multimedia extension technology programming environments manual," 2005.

[41] T. Chen, R. Raghavan, J. Dale, and E. Iwata, "Cell Broadband Engine Architecture and its first implementation: a performance view," *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 559–572, 2007.

[42] AMD, "ATI Radeon HD 5870 GPU feature summary," http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-specifications.aspx, [visited 2009-10-28].

[43] IBM, "IBM BladeCenter QS22," http://www-03.ibm.com/systems/bladecenter/hardware/servers/qs22/, [visited 2009-04-20].

[44] D. Strenski, J. Simkins, R. Walke, and R. Wittig, "Evaluating fpgas for floating-point performance," in *Intl. Workshop on High-Performance Reconfigurable Computing Technology and Applications*, Nov. 2008, pp. 1–6.

[45] D. Strenski, 2009, personal communication.

[46] E. Loh and G. Walster, "Rump's example revisited," *Reliable Computing*, vol. 8, no. 3, pp. 245–248, 2002.

[47] N. Higham, "The accuracy of floating point summation," *Scientific Comp.*, vol. 14, no. 4, pp. 783–799, 1993.

[48] T. Hagen, J. Hjelmervik, K.-A. Lie, J. Natvig, and M. Henriksen, "Visual simulation of shallow-water waves." *Simulation Modelling Practice and Theory*, vol. 13, no. 8, pp. 716–726, 2005.

[49] E. Elsen, P. LeGresley, and E. Darve, "Large calculation of the flow over a hypersonic vehicle using a GPU," *Comput. Phys.*, vol. 227, no. 24, pp. 10 148 – 10 161, 2008.

[50] D. Göddeke, R. Strzodka, and S. Turek, "Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations," *Parallel, Emergent and Distributed Systems*, vol. 22, no. 4, pp. 221–256, Jan. 2007.

[51] D. Göddeke and R. Strzodka, "Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations (part 2: Double precision GPUs)," Technical University Dortmund, Tech. Rep., 2008.

[52] J. Sun, G. Peterson, and O. Storaasli, "High-performance mixed-precision linear solver for fpgas," *Computers, IEEE Transactions on*, vol. 57, no. 12, pp. 1614–1623, Dec. 2008.

[53] M. Harris, "Parallel computing with CUDA," http://sa08.idav.ucdavis.edu/NVIDIA.CUDA.Harris.pdf, SIGGRAPH Asia 2008 presentation, [visited 2009-04-28].

[54] Tokyo Tech, http://www.voltaire.com/assets/files/Case%20studies/titech_case_study_final_for_SC08.pdf, November 2008, booth #3208 at Supercomputing '08, [visited 2009-04-28].

[55] R. Chamberlain, M. Franklin, E. Tyson, J. Buhler, S. Gayen, P. Crowley, and J. Buckley, "Application development on hybrid systems," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–10.

[56] Khronos OpenCL Working Group, "The OpenCL specification 1.0," http://www.khronos.org/registry/cl/, 2008, [visited 2009-03-20].

[57] ISO/IEC, "9899:TC3," International Organization for Standardization, September 2007.

[58] M. McCool, "Data-parallel programming on the Cell BE and the GPU using the Rapidmind development platform," November 2006, gSPx Multicore Applications Conference.

[59] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule, "Shader algebra," in *SIGGRAPH*. New York, NY, USA: ACM, 2004, pp. 787–795.

[60] Rapidmind, "Cell BE porting and tuning with RapidMind: A case study," http://www.rapidmind.net/case-cell.php, 2006, [visited 2009-03-20].

[61] F. Bodin, "An evolutionary path for high-performance heterogeneous multicore programming," 2008.

[62] Portland Group, "PGI accelerator compilers," http://www.pgroup.com/resources/accel.htm, [visited 2009-08-05].

[63] OpenGL Architecture Review Board, D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, 6th ed. Addison-Wesley, 2007.

[64] I. Buck, T. Foley, D. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," 2004.

[65] AMD, "ATI stream software development kit," http://developer.amd.com/gpu/ATIStreamSDK/, [visited 2009-04-28].

[66] Microsoft, "DirectX: Advanced graphics on windows," http://msdn.microsoft.com/directx, [visited 2009-03-31].

[67] NVIDIA, "CUDA Zone," http://www.nvidia.com/cuda, [visited 2009-03-20].

[68] ——, "Developer Zone," http://developer.nvidia.com/, [visited 2009-09-07].

[69] W. van der Laan, "Cubin utilities," http://www.cs.rug.nl/~wladimir/decuda/, 2007, [visited 2009-03-20].

[70] AMD, "Stream KernelAnalyzer," http://developer.amd.com/gpu/ska/.

[71] IBM, "Software development kit for multicore acceleration version 3.1: Programmers guide," August 2008.

[72] D. Bader, V. Agarwal, and K. Madduri, "On the design and analysis of irregular algorithms on the Cell processor: A case study of list ranking," in *Intl. Parallel and Distributed Processing Symp.*, 2007.

[73] J. Hjelmervik, "Heterogeneous computing with focus on mechanical engineering," Ph.D. dissertation, University of Oslo and Grenoble Institute of Technology, 2009, [Thesis accepted. Defence 2009-05-06].

[74] D. Vianney, G. Haber, A. Heilper, and M. Zalmanovici, "Performance analysis and visualization tools for Cell/B.E. multicore environment," in *Intl. Forum on Next-generation Multicore/Manycore Technologies*. New York, NY, USA: ACM, 2008, pp. 1–12.

[75] J. Peterson, P. Bohrer, L. Chen, E. Elnozahy, A. Gheith, R. Jewell, M. Kistler, T. Maeurer, S. Malone, D. Murrell, N. Needel, K. Rajamani, M. Rinaldi, R. Simpson, K. Sudeep, and L. Zhang, "Application of full-system simulation in exploratory system design and development," *IBM J. Res. Dev.*, vol. 50, no. 2/3, pp. 321–332, 2006.

[76] A. Varbanescu, H. Sips, K. Ross, Q. Liu, L.-K. Liu, A. Natsev, and J. Smith, "An effective strategy for porting C++ applications on Cell," in *Intl. Conf. on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2007, p. 59.

[77] A. Eichenberger, J. O'Brien, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind, "Optimizing compiler for the Cell processor," in *Intl. Conf. on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 161–172.

[78] A. Eichenberger, J. O'Brien, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. Gschwind, R. Archambault, Y. Gao, and R. Koo, "Using advanced compiler technology to exploit the performance of the Cell Broadband Engine Architecture," *IBM Syst. J.*, vol. 45, no. 1, pp. 59–84, 2006.

[79] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang, "Supporting OpenMP on Cell," *Parallel Programming*, vol. 36, no. 3, pp. 289–311, 2008.

[80] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani, "MPI microtask for programming the Cell Broadband Engine processor," *IBM Syst. J.*, vol. 45, no. 1, pp. 85–102, 2006.

[81] A. Kumar, G. Senthilkumar, M. Krishna, N. Jayam, P. Baruah, R. Sharma, A. Srinivasan, and S. Kapoor, "A buffered-mode MPI implementation for the Cell BE processor," in *Intl. Conf. on Computational Science*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 603–610.

[82] M. Krishna, A. Kumar, N. Jayam, G. Senthilkumar, P. Baruah, R. Sharma, S. Kapoor, and A. Srinivasan, "A synchronous mode MPI implementation on the Cell BE Architecture," in *Intl. Symp. on Parallel and Distributed Processing with Applications*, 2007, pp. 982–991.

[83] S. Pakin, "Receiver-initiated message passing over RDMA networks," in *Intl. Parallel and Distributed Processing Symp.*, Miami, Florida, April 2008.

[84] P. Bellens, J. Perez, R. Badia, and J. Labarta, "CellSs: a programming model for the Cell BE architecture," in *Supercomputing*. New York, NY, USA: ACM, 2006, p. 86.

[85] J. Perez, P. Bellens, R. Badia, and J. Labarta, "CellSs: making it easier to program the Cell Broadband Engine processor," *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 593–604, 2007.

[86] K. Fatahalian, D. Horn, T. Knight, L. Leem, M. Houston, J. Park, M. Erez, M. Ren, A. Aiken, W. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *Supercomputing*. New York, NY, USA: ACM, 2006, p. 83.

[87] T. Knight, J. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. Dally, and P. Hanrahan, "Compilation for explicitly managed memory hierarchies," in *Symp. on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2007, pp. 226–236.

[88] M. Ren, J. Park, M. Houston, A. Aiken, and W. Dally, "A tuning framework for software-managed memory hierarchies," in *Intl. Conf. on Parallel Architectures and Compilation Techniques*. New York, NY, USA: ACM, 2008, pp. 280–291.

[89] O. Storaasli, W. Yu, D. Strenski, and J. Maltby, "Performance evaluation of fpga-based biological applications," in *Cray User Group*, 2007.

[90] EDA Industry Working Groups, http://www.vhdl.org/, [visited 2009-04-28].

[91] "Verilog website," http://www.verilog.com/, [visited 2009-04-28].

[92] "Mitronics website," http://www.mitrion.com/, [visited 2009-04-28].

[93] "Impulse accelerated technologies," http://impulseaccelerated.com/, [visited 2009-04-28].

[94] "Open systemc initiative," http://www.systemc.org/, [visited 2009-04-28].

[95] "Celoxica website," http://www.celoxica.com/, [visited 2009-04-28].

[96] "Dsplogic website," http://www.dsplogic.com/, [visited 2009-04-28].

[97] "Starbridge systems website," http://www.starbridgesystems.com/, [visited 2009-04-28].

[98] "Xilinx website," http://www.xilinx.com/, [visited 2009-04-28].

[99] "OpenFPGA website," http://www.openfpga.org/, [visited 2009-04-28].

[100] A. van Amesfoort, A. Varbanescu, H. Sips, and R. van Nieuwpoort, "Evaluating multi-core platforms for HPC data-intensive kernels," in *Conf. on Computing frontiers*. New York, USA: ACM, 2009, pp. 207–216.

[101] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," in *Application Specific Processors, 2008. SASP 2008. Symposium on*, June 2008, pp. 101–107.

[102] M. Flynn, R. Dimond, O. Mencer, and O. Pell, "Finding speedup in parallel processors," in *Intl. Symp. on Parallel and Distributed Computing*, July 2008, pp. 3–7.

[103] J. Beeckler and W. Gross, "Particle graphics on reconfigurable hardware," *Reconfigurable Technology and Systems*, vol. 1, no. 3, pp. 1–27, 2008.

[104] O. Storaasli and D. Strenski, "Beyond 100x speedup with FPGAs: Cray XD1 i/o analysis," in *Cray Users Group*, 2009.

[105] Pico Computing, "Accelerating bioinformatics searching and dot plotting using a scalable FPGA cluster," November 2009, [visited 2009-11-14].

[106] HPCWire, "FPGA cluster accelerates bioinformatics application by 5000x," November 2009, [visited 2009-11-14].

[107] V. Sachdeva, M. Kistler, E. Speight, and T.-H. Tzeng, "Exploring the viability of the Cell Broadband Engine for bioinformatics applications," *Parallel Comput.*, vol. 34, no. 11, pp. 616–626, 2008.

[108] D. Hackenberg, "Fast matrix multiplication on Cell (SMP) systems," http://www.tu-dresden.de/zih/cell/matmul/, July 2007, [visited 2009-02-24].

[109] W. Alvaro, J. Kurzak, and J. Dongarra, "Fast and small short vector SIMD matrix multiplication kernels for the Synergistic Processing Element of the Cell processor," in *Intl. Conf. on Computational Science*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 935–944.

[110] L. Zhuo and V. K. Prasanna, "High-performance designs for linear algebra operations on reconfigurable hardware," *IEEE Trans. Comput.*, vol. 57, no. 8, pp. 1057–1071, 2008.

[111] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.

[112] M. Kistler, J. Gunnels, D. Brokenshire, and B. Benton, "Petascale computing with accelerators," in *Symp. on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2008, pp. 241–250.

[113] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–12.

[114] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, Dec. 2008.

[115] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT library for CUDA GPUs," in *Supercomputing*, 2009.

[116] N. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete Fourier transforms on graphics processors," in *Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[117] D. Bader and V. Argwal, "FFTC: Fastest Fourier transform for the IBM Cell Broadband Engine," in *Intl. Conf. on High Performance Computing*, 2007.

[118] M. Xu, P. Thulasiraman, and R. Thulasiram, "Exploiting data locality in FFT using indirect swap network on Cell/B.E." in *Intl. Symp. on High Performance Computing Systems and Applications*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 88–94.

[119] S. McKeown, R. Woods, and J. McAllister, "Algorithmic factorisation for low power fpga implementations through increased data locality," in *Int. symp. on VLSI Design, Automation and Test*, April 2008, pp. 271–274.

[120] K. Hemmert and K. Underwood, "An analysis of the double-precision floating-point FFT on FPGAs," in *Symp. on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 171–180.

[121] T. Aoki, "Real-time tsunami simulation," 2008.

[122] M. Christen, O. Schenk, P. Messmer, E. Neufeld, and H. Burkhart, "Accelerating stencil-based computations by increased temporal locality on modern multi- and many-core architectures," in *Intl. Workshop on New Frontiers in High-performance and Hardware-aware Computing*, 2008.

[123] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Workshop on General Purpose Processing on Graphics Processing Units*. New York, NY, USA: ACM, 2009, pp. 79–84.

[124] M. Araya-Polo, F. Rubio, R. de la Cruz, M. Hanzich, J. Cela, and D. Scarpazza, "3D seismic imaging through reverse-time migration on homogeneous and heterogeneous multicore processors," *Scientific Prog.*, vol. 17, no. 1-2, pp. 185–198, 2009.

[125] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[126] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Review*, vol. 51, no. 1, pp. 129–159, 2009.

[127] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Implicit and explicit optimizations for stencil computations," in *Workshop on Memory System Performance and Correctness*. New York, NY, USA: ACM, 2006, pp. 51–60.

[128] D. Thomas and W. Luk, "High quality uniform random number generation using LUT optimised state-transition matrices," *VLSI Signal Processing Systems*, vol. 47, no. 1, pp. 77–92, 2007.

[129] M. Matsumoto and T. Nishimura, "Dynamic creation of pseudorandom number generators," in *Monte Carlo and Quasi-Monte Carlo Methods 1998*. Springer, 2000, pp. 56–69.

[130] M. Sussman, W. Crutchfield, and M. Papakipos, "Pseudorandom number generation on the GPU," in *Graphics Hardware*, Sep. 2006, pp. 87–94.

[131] L. Howes and D. Thomas, "Efficient random number generation and application using CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, Aug. 2007.

[132] D. Thomas, L. Howes, and W. Luk, "A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation," in *FPGA*, 2009.

[133] S. Sengupta, M. Harris, Y. Zhang, and J. Owens, "Scan primitives for GPU computing," in *EUROGRAPHICS*, 2007, pp. 97–106.

[134] A. Greß, M. Guthe, and R. Klein, "GPU-based collision detection for deformable parameterized surfaces," *Computer Graphics Forum*, vol. 25, no. 3, pp. 497–506, September 2006.

[135] G. Ziegler, A. Tevs, C. Theobalt, and H.-P. Seidel, "GPU point list generation through histogram pyramids," Max-Planck-Institut für Informatik, Tech. Rep. MPI-I-2006-4-002, 2006.

[136] C. Dyken, G. Ziegler, C. Theobalt, and H.-P. Seidel, "High-speed marching cubes using histogram pyramids," *Computer Graphics Forum*, vol. 27, no. 8, pp. 2028–2039, 2008.

[137] T. Purcell, C. Donner, M. Cammarano, H. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," in *EUROGRAPHICS*. Eurographics Association, 2003, pp. 41–50.

[138] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: high performance graphics co-processor sorting for large database management," in *Intl. Conf. on Management of Data*. New York, NY, USA: ACM, 2006, pp. 325–336.

[139] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," NVIDIA, NVIDIA Technical Report NVR-2008-001, Sep. 2008.

[140] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani, "AA-sort: A new parallel sorting algorithm for multi-core SIMD processors," in *Intl. Conf. on Parallel Architecture and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 189–198.

[141] Y. Luo and R. Duraiswami, "Canny edge detection on NVIDIA CUDA," in *Computer Vision and Pattern Recognition Workshops*, June 2008, pp. 1–8.

[142] H. Neoh and A. Hazanchuk, "Adaptive edge detection for real-time video processing using FPGAs," http://www.altera.com/literature/cp/gspx/edge-detection.pdf, March 2005, [visited 2009-03-20].

[143] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan, "GpuCV: A GPU-accelerated framework for image processing and computer vision," in *Intl. Symp. on Advances in Visual Computing*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 430–439.

[144] Fixtars, "OpenCV on the Cell," http://cell.fixstars.com/opencv/, [visited 2009-03-20].

[145] H. Sugano and R. Miyamoto, "Parallel implementation of morphological processing on Cell/BE with OpenCV interface," in *Intl. Symp. on Communications, Control and Signal Processing*, March 2008, pp. 578–583.

[146] P. Babenko and M. Shah, "MinGPU: a minimum GPU library for computer vision," *Real-Time Image Processing*, vol. 3, no. 4, pp. 255–268, December 2008.

[147] J. Dongarra, "Basic Linear Algebra Subprograms Technical forum standard," *High Performance Applications and Supercomputing*, vol. 16, pp. 1–111, 2002.

[148] NVIDIA, "CUDA CUBLAS library version 2.0," March 2008.

[149] AMD, "AMD Core Math Library for graphic processors," http://developer.amd.com/gpu/acmlgpu/, March 2009, [visited 2009-04-20].

[150] E. Larsen and D. McAllister, "Fast matrix multiplies using graphics hardware," in *Supercomputing*. New York, NY, USA: ACM, 2001, pp. 55–55.

[151] "Parallel linear algebra for scalable multi-core architectures (PLASMA) project," http://icl.cs.utk.edu/plasma/, [visited 2009-04-20].

[152] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Comput.*, vol. 35, no. 1, pp. 38–53, 2009.

[153] A. Brodtkorb, "The graphics processor as a mathematical coprocessor in MATLAB," in *Intl. Conf. on Complex, Intelligent and Software Intensive Systems*. IEEE CS, 2008, pp. 822–827.

[154] Accelereyes, "Jacket user guide," February 2009.

[155] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. Buijssen, M. Grajewski, and S. Turek, "Exploring weak scalability for FEM calculations on a GPU-enhanced cluster," *Parallel Comput.*, vol. 33, no. 10–11, pp. 685–699, Nov. 2007.

[156] G. Blelloch, M. Heroux, and M. Zagha, "Segmented operations for sparse matrix computation on vector multiprocessors," School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep., 1993.

[157] NVIDIA, "CUDA CUFFT library version 2.1," March 2008.

[158] "Fftw website," http://www.fftw.org, [visited 2009-04-28].

[159] V. Volkov and B. Kazian, "Fitting FFT onto the G80 architecture," http://www.cs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6_report.pdf, [visited 2009-08-10].

[160] B. Lloyd, C. Boyd, and N. Govindaraju, "Fast computation of general Fourier transforms on GPUs," in *Intl. Conf. on Multimedia & Expo*, 2008, pp. 5–8.

[161] IBM, "Fast Fourier transform library: Programmer's guide and API reference," August 2008.

[162] ——, "3d fast Fourier transform library: Programmer's guide and API reference," August 2008.

[163] Altera, "FFT megacore function user guide," March 2009.

[164] ——, "Logicore ip fast Fourier transform v7.0 user guide," June 2009.

[165] T. Mattson, R. Van der Wijngaart, and M. Frumkin, "Programming the Intel 80-core network-on-a-chip Terascale processor," in *Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.

[166] 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*, vol. 43, no. 1, pp. 29–41, Jan. 2008.

[167] T. Hagen, K.-A. Lie, , and J. Natvig, "Solving the euler equations on graphics processing units," in *Intl. Conf. on Computational Science*, ser. LNCS, V. N. Alexandrov, G. D. van Albada, P. M. Sloot, and J. Dongarra, Eds., vol. 3994. Springer, 2006, pp. 220–227.

[168] J. McZalpin and D. Wonnacott, "Time skewing: A value-based approach to optimizing for memory locality," Rutgers School of Arts and Sciences, Tech. Rep. dcs-tr-379, 1999.

[169] M. Frigo and V. Strumpen, "The memory behavior of cache oblivious stencil computations," in *Supercomputing*, vol. 39, no. 2. Hingham, MA, USA: Kluwer Academic Publishers, 2007, pp. 93–112.

[170] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.

[171] M. Saito and M. Matsumoto, "SIMD-oriented fast Mersenne Twister: a 128-bit pseudorandom number generator," in *Monte Carlo and Quasi-Monte Carlo Methods*. Springer, 2008.

[172] P. L'Ecuyer, "Maximally equidistributed combined Tausworthe generators," *Mathematics of Computation*, vol. 65, no. 213, pp. 203–213, 1996.

[173] NVIDIA, "CUDA SDK version 2.0," 2008.

[174] K. Iverson, *A programming language*. New York, NY, USA: John Wiley & Sons, Inc., 1962.

[175] W. Hillis and G. Steele Jr, "Data parallel algorithms," *Commun. ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.

[176] D. Horn, "Stream reduction operations for GPGPU applications," in *GPU Gems 2*, M. Pharr and R. Fernando, Eds. Addison-Wesley, 2005, pp. 573–589.

[177] G. Blelloch, "Prefix sums and their applications," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-90-190, Nov. 1990.

[178] S. Sengupta, A. Lefohn, and J. Owens, "A work-efficient step-efficient prefix sum algorithm," in *Workshop on Edge Computing Using New Commodity Architectures*, May 2006, pp. D–26–27.

[179] M. Harris, S. Sengupta, and J. Owens, "Parallel prefix sum (scan) with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, Aug. 2007.

[180] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson, "CUDPP: CUDA data parallel primitives library," http://www.gpgpu.org/developer/cudpp/, [visited 2009-03-20].

[181] J. Chhugani, A. Nguyen, V. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient implementation of sorting on multi-core SIMD CPU architecture," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1313–1324, 2008.

[182] J. Canny, "A computational approach to edge detection," *Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–698, 1986.

[183] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*. Cambridge, MA: O'Reilly, 2008.

[184] T. Pock, M. Unger, D. Cremers, and H. Bischof, "Fast and exact solution of total variation models on the GPU," in *Computer Vision and Pattern Recognition Workshops*, June 2008, pp. 1–8.

[185] P. Michel, J. Chestnut, S. Kagami, K. Nishiwaki, J. Kuffner, and T. Kanade, "GPU-accelerated real-time 3D tracking for humanoid locomotion and stair climbing," in *Intl. Conf. on Intelligent Robots and Systems*, 29 2007-Nov. 2 2007, pp. 463–469.

[186] M. Boyer, D. Tarjan, S. Acton, and K. Skadron, "Accelerating leukocyte tracking using cuda: A case study in leveraging manycore coprocessors," in *Int. Parallel and Distributed Processing Symp.*, 2009.

[187] Z. Baker, M. Gokhale, and J. Tripp, "Matched filter computation on FPGA, Cell and GPU," in *Symp. on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 207–218.

PAPER II

A COMPARISON OF THREE COMMODITY-LEVEL PARALLEL ARCHITECTURES: MULTI-CORE CPU, CELL BE AND GPU

A. R. Brodtkorb and T. R. Hagen

In proceedings of the Seventh International Conference on Mathematical Methods for Curves and Surfaces, Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, 5862 (2010), pp. 70–80

Abstract: We explore three commodity parallel architectures: multi-core CPUs, the Cell BE processor, and graphics processing units. We have implemented four algorithms on these three architectures: solving the heat equation, inpainting using the heat equation, computing the Mandelbrot set, and MJPEG movie compression. We use these four algorithms to exemplify the benefits and drawbacks of each parallel architecture.

1 Introduction

The gain in performance of computer programs has typically come from increased processor clock frequency and increased size of system memory. New computers have been able to handle larger problems in the same timeslot, or the same problem at greater speed. Recently, however, this trend has seemed to stop, and in the most recent years we have actually seen a *decrease* in clock frequency. The new trend is instead to increase the *number* of processor cores. There are several different multi-core designs in commodity hardware: your typical multi-core CPU consists of a few *fat* cores with a lot of complex logic; graphics processing units (GPUs) consist of several hundred *thin* processors with reduced functionality; and the Cell BE [1] consists of a mixture of fat and thin cores.

A key point to designing algorithms for these architectures is to understand their hardware characteristics. The aim of this work is to give a good understanding of how the hardware performs, with respect to a set of four example algorithms: solving the heat equation, inpainting missing pixels using the heat equation, computing the Mandelbrot set, and MJPEG movie compression. These four algorithms are chosen because they display different characteristics found in many real-world applications.

2 Related work

There has been a lot of research into high-level programming abstractions to parallel architectures. All abstractions attempt to create an intuitive and simple API with as low as possible performance penalty. APIs such as OpenMP [2] and MPI [3] show good performance and have become de facto standards in shared memory and cluster systems, respectively. These two APIs have also been exploited to create abstractions to programming the Cell BE [4, 5]. CellSs [6] is another abstraction for the Cell BE, consisting of a source to source compiler and runtime system. There have also been several high-level abstractions to programming the GPU for general purpose computation (GPGPU [7]). The most active languages today are CUDA [8] and Brook [9], where CUDA is a vendor specific language from NVIDIA, and Brook is an API with an extended version for AMD GPUs called Brook+ [10]. RapidMind [11] is a high-level C++ abstraction offering a common platform for different back-ends: multi-core CPUs, the Cell BE, and GPUs. This enables the programmer to easily run the same code on different hardware setups by simply changing the back-end. The back-ends themselves are responsible for low-level optimization and tuning, letting the programmer focus on high-level optimization of algorithms. OpenCL [12] is an API ratified by the Khronos group, in the same family of standards as OpenGL [13]. OpenCL offers a common low-level interface to program architectures such as multi-core CPUs, Cell BE and GPUs. Approved in December 2008, the first compilers have now appeared for GPUs, and the number of supported architectures is expected to rise. Such an open standard with support for multiple platforms is a great step in unifying heterogeneous programming efforts. In this work, we do not use or discuss the aforementioned higher-level abstractions, but instead present implementations created using lower-level tools for each architecture.

The algorithms we examine have been implemented earlier on all three architectures. The heat equation and other partial differential equations (PDEs) have been implemented on the Cell BE [14] and on the GPU [15]. Inpainting using PDE-based techniques has also been im-

plemented on the GPU [16], but not, to our knowledge, on the Cell BE. A Mandelbrot generator is part of the NVIDIA CUDA SDK code samples, and writing an optimized Mandelbrot set generator for the Cell BE was the topic for the linux.conf.au 2008 hackfest. The main building block of MJPEG movie compression is also part of the NVIDIA CUDA SDK, and MJPEG 2000 has been implemented on the Cell BE [17]. We emphasize that our aim is not to compete with these implementations, but rather to show how implementations with a similar amount of optimization can uncover differences.

3 Architectures

Multi-core CPUs, the Cell BE, and GPUs all consist of several processors, with varying type and composition. This section briefly describes these architectures and the low-level languages and tools we have used to program them. We have chosen to compare models that were released in the same time period, and argue that the intrinsic differences we identify do not change dramatically with newer generations.

Modern CPUs incorporate a multi-core design that consists of multiple fat cores on a single chip, where each core has a large cache and a lot of logic for instruction level parallelism (ILP). More than half of the chip area is typically used for cache and logic, leaving a relatively small number of transistors for pure arithmetic operations. To utilize all cores in the multi-core processor we use OpenMP, a C, C++ and Fortran API for multi-threading in a shared memory system. In C++, the API consists of a set of compiler pragmas that for example can execute loops in parallel.

The Cell BE is a heterogeneous processing unit, consisting of one power processor element (PPE) and eight synergistic processing elements (SPEs). The PPE is a regular fat CPU core, while the SPEs are thin cores capable of SIMD execution on 128 bits of properly aligned memory. The next version of the Cell BE is planned to have two PPEs and 32 SPEs. In the current version, each SPE has a small *local store* to hold their program and data, and is connected to the PPE through a fast on-chip interconnect. As opposed to the PPE, the SPEs have little or no ILP or cache, dedicating almost all transistors to floating point calculations. The API that we have used to access and program the SPEs is the SPE Runtime Management Library version 2 (libspe 2) [18]. In our use of libspe 2, the PPE program typically creates a context for each of the SPEs, and loads an SPE program (written in a subset of C++) into each context. The PPE then sets the SPEs to execute the programs. The SPEs are responsible for explicitly transferring data between local store and main memory using direct memory access (DMA). DMA executes asynchronously, meaning we can overlap memory transfer with computation. This effectively hides memory latency and transfer time. While the SPEs are computing, the PPE mainly acts as a supervisor handling support functions for the SPEs. This use is often referred to as the SPE-centric programming model, where the SPEs run the main part of the program.

The gaming industry is the main driving force behind the development of GPUs. Traditionally, the GPU only had a fixed set of graphics operations implemented in hardware, but recent generations have become programmable. Current high-end graphics cards have several hundred *stream processors* where almost all transistors are dedicated to floating point arithmetics. The NVIDIA GeForce 9800 GX2 for example, has 256 stream processors. The last generations of GPUs from NVIDIA can be programmed using CUDA, an extension to C that views the GPU as a general stream processor. CUDA typically uses the GPU to execute the same program,

referred to as the *kernel*, over a large set (or stream) of data. To execute the kernel, the input data is first transferred to the GPU, and then the computational domain is divided into equally sized blocks. Each block has one virtual thread per element, and the blocks are computed independently. When the kernel has completed execution on all blocks, the data is typically read back to the CPU.

We have benchmarked the four algorithms on two different commodity level hardware setups. The CPU and GPU implementations were run on the same system, consisting of an Intel Core 2 Duo 2.4 GHz processor with 4 MiB cache, 4 GiB system memory, and an NVIDIA GeForce 8800 GTX card with 768 MiB graphics memory. The Cell BE implementations were run on a PlayStation 3, containing the 3.2GHz Cell BE processor with 6 available SPEs and 256 MiB system memory. The CPU has a theoretical performance of about 20 GFLOPS per core, and each SPE on the Cell BE has a theoretical performance of about 25 GFLOPS. The GPU has a theoretical performance of about 520 GFLOPS. All the implementations have been compiled using level three optimization and the *fast-math* option enabled. They also contain approximately the same amount of platform specific optimizations. The time and effort needed to program the different architectures, however, differs somewhat. Implementing the CPU version required less time and debugging efforts than the GPU version and the Cell BE version. The difference in effort comes from two main contributors: prior experience with the architecture and quality of programming tools, debuggers and profilers. If we attempt to disregard the impact of prior experience, we can give a few general remarks on programming effort. Of the three architectures, we find that the CPU implementation uses the highest level API, and requires the least programming effort. Both the GPU and Cell BE implementations on the other hand, use lower level APIs that require detailed knowledge of the hardware to avoid performance pitfalls. Thus, they require more programming effort to reach the same level of optimization as the CPU.

4 Algorithms and Results

To compare the architectures, we have implemented four different algorithms: solving the heat equation with a finite difference method, inpainting missing pixels using the heat equation solver, computing the Mandelbrot set, and MJPEG movie compression. We have selected these algorithms because they exhibit different computational and memory access properties that are representative for a large range of real-world problems: the first two are memory streaming problems with regular and irregular memory access patterns, respectively, and the last two are number crunching problems with uniformly and nonuniformly distributed computation. All four algorithms can easily be parallelized and should thus fit the architectures well.

The following sections describe the algorithms and some details for the Cell BE and GPU implementations. Our CPU implementations use OpenMP to parallelize the execution, using a static scheduler for the two algorithms with a predetermined work-load and a dynamic scheduler for the two with a data-dependent work-load. We employ the `omp parallel for` pragma on the outmost loops, thus minimizing OpenMP overheads.

The Heat Equation

The heat equation describes how heat dissipates in a medium. In the case of a 2D homogeneous and isotropic medium, it can be written as $u_t = a(u_{xx} + u_{yy})$, where a is a material specific constant. Using an explicit finite difference scheme, the unknown solution $u_{i,j}^{n+1}$ in grid point

Listing 1: CellHeat.cpp

```
for (i=1; i<=height; ++i) {
  //Increment front, back, up, center, down, next
  DMA_REQUEST_ROW_FROM_MEMORY(input[next]);
  DMA_WAIT(input[down]);
  DMA_WAIT(output[front]);
  for(j=1; j<width-1; ++j)
    output[front][j] = 0.125f * (input[up][j]
                               + input[center][j-1]
                               + 4.0f*input[center][j]
                               + input[center][j+1]
                               + input[down][j]);
  DMA_REQUEST_ROW_TO_MEMORY(output[front]);
}
```

(ih, jh) at time $(n+1)k$ is given by

$$u_{i,j}^{n+1} = a\frac{k}{h^2}\left(u_{i-1,j}^{n} + u_{i+1,j}^{n} + 4u_{i,j}^{n} + u_{i,j-1}^{n} + u_{i,j+1}^{n}\right), \quad (1)$$

where h and k determine the spatial and temporal resolution, respectively.

Characteristics - The heat equation benchmark shows how efficient each architecture is at streaming data. Solving the heat equation using this explicit finite difference scheme requires approximately the same number of memory accesses as arithmetic instructions. As the processor clock speeds are much higher than the memory clock speeds on our architectures, the performance should be limited by memory bandwidth. The memory access pattern, however, is regular and thus enables efficient hardware pre-fetching techniques on the multi-core CPU. Thus, the multi-core CPU should get full use of its large cache as long as the two cores keep from fighting over the same set of cache lines. On the Cell BE we can try to hide memory access by overlapping this with computation, and on the GPU explicitly gather the data into fast on-chip memory called *shared memory*.

This algorithm displays properties found when solving other PDEs using explicit finite difference/element/volume methods. Image filtering, such as convolution, is another class of algorithms with the same properties.

Implementation - By examining the computational molecule for $u_{i,j}^{n+1}$, we see that it depends on the value at grid-point (ih, jh) and the four closest neighbours in the previous time-step. Because of this dependency, we cannot simply update the value of $u_{i,j}^{n}$ in-place. By having two buffers, we store the result of odd time-steps in the "odd" buffer, reading from the "even" buffer, and conversely for even time-steps. This way we can update all elements in parallel at each time-step.

On the Cell BE, we divide the domain into blocks with an equal number of rows, and let each SPE calculate its sub-domain. Listing 1 shows the Cell BE algorithm in pseudo code, and Figure 1 is a graphical representation. For each row we want to compute in the output, we need to explicitly gather the needed data from main memory to local store and write the result back.

Using asynchronous memory transfer, we can overlap with computation, and only keep four rows of input in local store at any given time. We have used intrinsic SIMD instructions in the actual implementation, as the SPEs are SIMD processors.

On the GPU, we divide the computational domain into blocks with one virtual thread per element. For each block, each thread reads one element from the GPUs main memory into shared memory. The shared memory is fast on-chip memory accessible to all virtual threads in the same block. We also have to read the *apron* (also called ghost cells) into shared memory. The apron consists of the data-elements outside the block that we need for our computational molecule. Figure 1 shows the block and its apron. When all data has been read into shared memory, each thread computes one output element and writes the result to GPU main memory.

Results - Figure 2 shows the runtime of computing one step of the heat equation on the different architectures. For the GPU, this includes the time it takes to read the result back from graphics memory. The other two architectures place the result in main memory without the explicit read-back. Table 1 breaks the GPU and Cell BE run-times up into time spent computing and time spent waiting for memory transfers to complete. The table shows that the GPU suffers from an expensive read-back of data from the GPU to the CPU, whilst the Cell BE is able to hide much of the memory reads and writes by overlapping with computation. This explains why the GPU barely beats the multi-core CPU implementation. The GPU, however, will outperform the other architectures if the data is allowed to remain on the GPU throughout multiple time-steps.

Inpainting

Noisy images (e.g., from poor television reception) can be repaired by *inpainting*. Here we use the heat equation on masked areas as a naïve example of inpainting. Using this approach, information from the area surrounding a block of noisy pixels will be diffused into the block and fill in the missing values. Technically, each masked element is updated using Equation (1), whereas we set $u_{i,j}^{n+1} = u_{i,j}^{n}$ for unmasked elements.

Characteristics - The inpainting benchmark shows how efficient each architecture is at streaming data, executing conditionals, and computing on a subset of the data. Inpainting using the heat equation requires even fewer arithmetic instructions per memory access than solving the heat equation. This is because we have to read the mask for all elements, but only run computations on a few of them. This memory access pattern will further make matters worse, as the

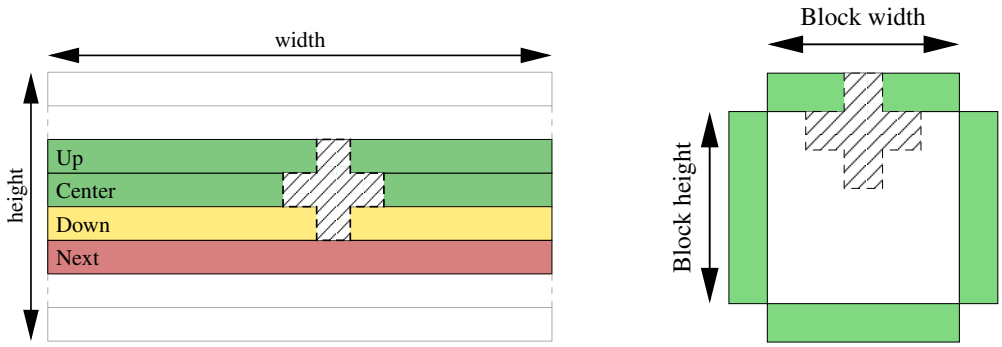


Figure 1: The heat equation computed on the Cell BE (left) and on the GPU (right).

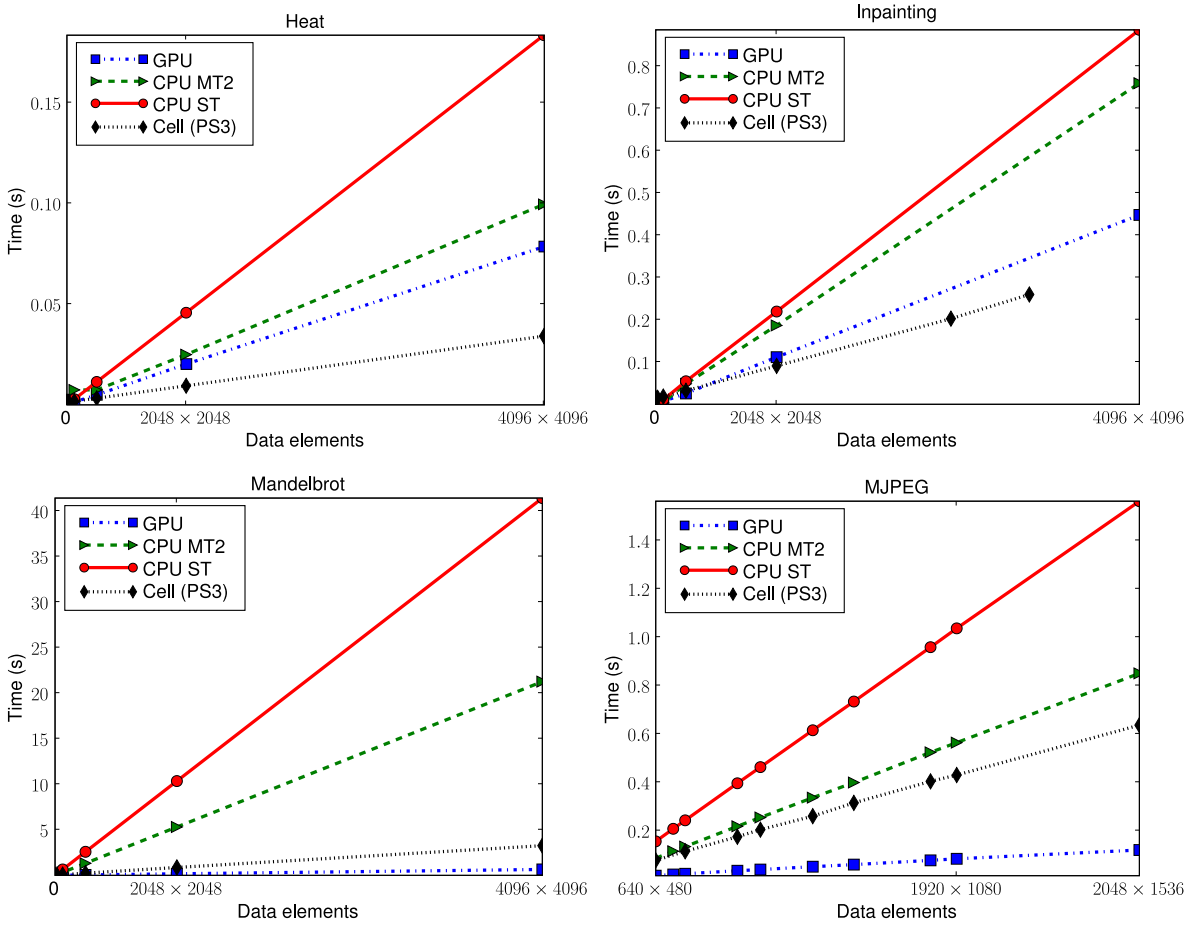


Figure 2: Runtime of the four different algorithms: the heat equation (top left), inpainting (top right), the Mandelbrot set (bottom left), MJPEG (bottom right). CPU ST represents the serial CPU version, and CPU MT2 represents the multi-threaded CPU version running on two CPU cores.

CPU will underutilize its cache. The GPU will have problems with the added conditional as all processors in one *warp* (currently 32 stream processors) are forced to run commands synchronously. The effect is that even if only one processor in the warp branches differently from the others, the whole warp must evaluate both sides of the branch.

This algorithm has properties that are also found in other algorithms with a lot of memory access and little computation, e.g., image processing algorithms working on masked areas.

Implementation - Both the Cell BE and GPU implementations of this algorithm are very similar to the heat equation solver. The major difference is that the Cell BE and GPU must explicitly gather the mask into local store and shared memory, respectively.

Results - For each image resolution, we used a noise mask to inpaint approximately 5% of the image using ten time-steps of the heat equation. Figure 2 shows that the GPU performs relatively better compared to the heat equation solver. This is because the GPU is able to keep the data in graphics memory throughout the ten passes. This lessens the effect of the single read-back, as shown in Table 1. Because the GPU executes one warp synchronously, it uses about the same time to complete one pass of inpainting, as it does to complete one pass of the whole heat equation. The Cell BE computes four elements synchronously using intrinsic SIMD

Table 1: Breakdown of running times. For the GPU, the percentages represent kernel execution time and GPU-CPU transfer time. For the Cell BE, the time represents time spent computing, and stall time while waiting for DMA requests to complete.

		Heat	Inpainting	Mandelbrot	MJPEG
GPU	Memory	75%	55%	0%	80%
	Computation	25%	45%	100%	20%
Cell BE	Memory	10%	10%	0%	5%
	Computation	90%	90%	100%	95%

instructions. However, it only runs slightly faster per pass than the heat equation. This can be explained by the fact that the SPEs do not contain a conditional branch predictor, making the branch almost as expensive as computing the heat equation itself. Using a computationally more demanding algorithm would diminish the effect of the branch on the Cell BE. There was also not enough physical system memory to benchmark domains larger than approximately 3600×3600 for the Cell BE. The single threaded CPU version runs much faster per pass than the heat equation, whilst the multi-threaded version only has a marginal speedup. This can be explained by the increased load on the memory bus, as the additional mask has to be loaded into cache, compared to the heat equation solver alone. Using multiple cores does not increase performance when the bottleneck is the memory bandwidth.

The Mandelbrot Set

The Mandelbrot set is a fractal that has a very simple recursive definition:

$$M = \left\{ c \in \mathbb{C} : z_0 = c, \quad z_{n+1} = z_n^2 + c, \quad \sup_{n \in \mathbb{N}} |z_n| < \infty \right\}. \tag{2}$$

Informally, it is the set of all complex numbers that do not tend towards infinity when computing z_{n+1}. When computing the Mandelbrot set, one uses the fact that c belongs to M if and only if $|z_n| < 2$ for all $n \geq 0$. Typically, one picks a set of discrete complex points C and a fixed m, and then the point $c \in C$ is assumed to be in the set if $|z_n| < 2$ for all $n \leq m$.

Characteristics - This benchmark shows how well each architecture performs floating point operations, and how it copes with dynamic workloads. Computing whether a complex coordinate belongs to the Mandelbrot set requires a lot of computation, and only a single memory write. For coordinates that are in the set the program has to compute all m iterations while coordinates outside often compute only a few. This means that neighbouring pixels often have very different workloads, as the boundary has a highly complex geometry.

Computing the Mandelbrot set exhibits properties also found in algorithms such as ray-tracing and ray-casting, as well as many other iterative algorithms. Algorithms with a lot of computation per memory access, such as protein folding, also show these properties.

Implementation - Using the abscissa as the real part and the ordinate as the imaginary part we create a set of complex numbers C. For each point $c \in C$, a while-loop computes z_n until $|z_n| > 2$ or $n > m$. The pixel is colored using the value of n when the loop terminates, yielding a gray-scale image where all white pixels are assumed to be part of the Mandelbrot set.

In our Cell BE implementation, we partition the domain into lines, where only the real part of c varies between pixels. We know that the computational time for each line can differ drastically, so we use a dynamic load distribution. The PPE simply fills a fixed length queue with line-numbers for each SPE, and the SPEs then start processing their queue. The PPE continues to enqueue line-numbers in non-full queues until the whole domain has been enqueued. The actual computation is done using SIMD instructions to utilize the hardware.

Results - Table 1 shows that computing whether a complex coordinate belongs to the Mandelbrot set or not is computationally intensive, and Figure 2 shows that both the GPU and Cell BE perform very well. Using multiple CPU cores scales perfectly. On the GPU, it does not drastically affect the results that two pixels close to each other can have very different workloads. Even though each warp is executed synchronously, most warps simply contain only pixels within the set, or outside it. Thus, the number of warps with a mixture of pixels within and outside the set is often far less than the number of warps with a relatively homogeneous workload.

MJPEG

MJPEG is an "industry standard" for compression of a movie stream. The main part of the algorithm consists of dividing each frame into 8×8 blocks, computing the discrete cosine transform (DCT), and then quantizing each block:

$$\begin{aligned} p_{u,v} &= \alpha(u)\alpha(v) \sum_{i=0}^{7} \sum_{j=0}^{7} g_{i,j} \cos\left[\frac{\pi}{8}(i+0.5)\,u\right] \cos\left[\frac{\pi}{8}(j+0.5)\,v\right], \\ \alpha(n) &= \begin{cases} \sqrt{1/8} & , n = 0 \\ \sqrt{2/8} & , n \neq 0 \end{cases}, \\ r_{u,v} &= \text{round}\left(p_{u,v}/q_{u,v}\right). \end{aligned}$$

Here, $g_{i,j}$ is the element (i, j) from the 8×8 block, $p_{u,v}$ is the amplitude of frequency (u, v), $q_{u,v}$ is element (u, v) of the quantization matrix, and $r_{u,v}$ is the result.

Characteristics - The MJPEG results show how efficient each architecture is with a typical data flow, where the computationally intensive part of the code is accelerated, and the rest of the code runs on a single CPU core. Computing the DCT and then quantizing is an algorithm that is both cache friendly and requires a lot of computation per memory access. The performance should therefore be limited by processing power rather than memory access. Computing the cosines is typically also very expensive.

The MJPEG algorithm is representative for many image and movie compression algorithms, as they all do a lot of computation per memory access. These properties are also displayed in computationally heavy image processing algorithms, such as computing the FFT and image registration.

Implementation - To optimize for DMA transfer, our Cell BE version computes a row of blocks. Since the total amount of work is constant in each block-line, we use a static load distribution, dividing the domain into an equal number of block-lines per SPE. The SPE then computes each block in each block-line until there are no more block-lines to compute.

MJPEG compression fits the GPU perfectly, as CUDA already assumes that computation

should be split up into blocks. We simply use a block-size of 8×8 and let CUDA automatically schedule the blocks in our GPU implementation.

Results - Figure 2 shows the time it takes to compute the DCT, quantize, and Huffman code one image consisting of an intensity channel and two chrominance channels with half the horizontal and vertical resolution of the intensity image. Table 1 breaks down the time spent on DCT and quantization, as these are the parts that have been accelerated. The GPU is heavily penalized for overhead connected with starting each memory transfer. It has to upload the three images and download them again after the computations. The overhead with starting these six memory transfers is substantial. The Cell BE does not suffer from such overheads, but is able to overlap almost all memory access by computation. However, Huffman coding using the PPE is very slow. The PPE takes 50% more time to complete compared to the CPU, even though it is the exact same code.

5 Summary

We have examined how a set of four algorithms perform on three sets of commodity level parallel hardware. All the algorithms we displayed, except inpainting, scaled well on the CPU. The inpainting algorithm saturates the memory bus, which again limits performance. The Cell BE performs well on algorithms where memory access can be hidden by computations, and when it comes to raw floating point performance. However, it has a relatively slow PPE core that can limit performance. The limited system memory on the PlayStation 3 can certainly also be a problem. The GPU is, by far, the best performing architecture for computationally intensive operations, but transferring memory to and from the graphics card can be very expensive.

This work has focused on four applications that show different properties of the three architectures. This is a small comparison, and it would be of great use to broaden the number of algorithms, architectures, and programming languages to give a broader understanding of commodity level parallel architectures.

The authors would like to thank the anonymous reviewers for their thorough comments and feedback.

Bibliography

[1] IBM, Sony, Toshiba: Cell Broadband Engine programming handbook version 1.1 (2007)

[2] OpenMP Architecture Review Board: OpenMP application program interface version 2.5 (2005)

[3] Message Passing Interface Forum: MPI-2: Extensions to the message-passing interface (2003)

[4] Ohara, M., Inoue, H., Sohda, Y., Komatsu, H., Nakatani, T.: MPI microtask for programming the Cell Broadband Engine processor. IBM Systems Journal **45** (2006) 85–102

[5] Eichenberger, A., O'Brien, J., O'Brien, K., Wu, P., Chen, T., Oden, P., Prener, D., Shepherd, J., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., Gschwind, M.: Optimizing compiler for the Cell processor. In: Intl. Conf. on Parallel Architectures and Compilation Techniques, Washington, DC, USA, IEEE Computer Society (2005) 161–172

[6] Bellens, P., Perez, J., Badia, R., Labarta, J.: CellSs: a programming model for the Cell BE architecture. In: SuperComputing'06. (2006)

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

[8] NVIDIA corporation: NVIDIA CUDA programming guide version 1.1 (2007)

[9] Buck, I., Foley, T., Horn, D., Sugerman, J., Houston, M., Hanrahan, P.: Brook for GPUs: Stream computing on graphics hardware (2004)

[10] AMD Corporation: AMD stream computing revision 1.3.0 (2008)

[11] McCool, M.: Data-parallel programming on the Cell BE and the GPU using the rapidmind development platform (2006) GSPx Multicore Applications Conference.

[12] Khronos OpenCL Working Group: The OpenCL specification 1.0 (2008)

[13] OpenGL Architecture Review Board, Shreiner, D., Woo, M., Neider, J., Davis, T.: OpenGL Programming Guide: The Official Guide to Learning OpenGL. 6th edn. Addison-Wesley (2007)

[14] Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The potential of the Cell processor for scientific computing. In: Computing Frontiers '06. (2006)

[15] Hagen, T., Henriksen, M., Hjelmervik, J., Lie, K.A.: How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational engine. Geometric Modelling, Numerical Simulation and Optimization: Industrial Mathematics at SINTEF, Springer verlag (2007)

[16] Hagen, T., Rahman, T.: GPU-based image inpainting using a TV-Stokes equation. Preprint (2008)

[17] Muta, H., Doi, M., Nakano, H., Mori, Y.: Multilevel parallelization on the Cell / B.E. for a motion JPEG 2000 encoding server. In: MULTIMEDIA '07. (2007)

[18] IBM: Software development kit for multicore acceleration version 3.0 (2007)

PAPER III

A MATLAB INTERFACE TO THE GPU

A. R. Brodtkorb

In proceedings of The Second International Conference on Complex, Intelligent and Software Intensive Systems, IEEE Computer Society, (2008), pp. 822–827

Abstract: We present an interface to the graphics processing unit (GPU) from MATLAB, and four algorithms from numerical linear algebra available through this interface; matrix-matrix multiplication, Gauss-Jordan elimination, PLU factorization, and tridiagonal Gaussian elimination. In addition to being a high-level abstraction to the GPU, the interface offers background processing, enabling computations to be executed on the CPU simultaneously. The algorithms are shown to be up-to 31 times faster than highly optimized CPU code. The algorithms have only been tested on single precision hardware, but will easily run on new double precision hardware.

1 Introduction

The graphics processing unit (GPU) is the processor on graphics cards, dedicated to rendering images on screen. The rendered images typically consist of millions of pixels that can be computed in parallel. The GPU exploits this fact, and exhibits high levels of parallelism with up-to several hundred processors. Recent generations of off-the-shelf GPUs have become programmable, enabling the use of GPUs for general purpose computations. This has opened a new field of research called GPGPU.

The reason for interest in GPUs is their massive floating point performance. They offer far higher peak performance than CPUs, and the performance gap is increasing. While the processing power of CPUs has followed Moore's law closely, doubling every 18-24 months, the processing power of the GPU has doubled every 9 months [1]. An argument against using GPUs, however, has been the lack of double precision. This is now outdated with new double precision hardware.

Numerical linear algebra includes many computationally heavy operations that are central in many fields, ranging from search engines to games, cryptology and solving partial and ordinary differential equations numerically. Using the GPU to speed up such computations is important for all these applications. Harvesting the raw power of the GPU, however, is nontrivial and not even possible for some problems. In order to solve a problem using the GPU, it has to fit the GPU programming model and have a highly parallel nature. Traditionally, the GPU had to be accessed via a graphics API such as OpenGL [2] and DirectX [3], requiring that the problem is rewritten in terms of operations on graphical primitives. New vendor-specific APIs such as "Close To the Metal" (CTM) [4] from AMD and "Computer Unified Device Architecture" (CUDA) [5] from NVIDIA, however, offer access to the hardware without going through the graphics API. There also exists two free APIs for GPGPU: Brook [6] and Sh [7]. These two, however, do not seem to be actively developed, as Sh has been commercialized as RapidMind [8], and Brook has been commercialized as Brook+ from AMD [9].

This article presents four selected operators from numerical linear algebra implemented on the GPU. We have used OpenGL to access the GPU, as this was our best alternative before CUDA was released. The algorithms are chosen because of their importance and how they fit the GPU programming model: Full matrix-matrix multiplication is one of the building blocks in numerical linear algebra; Gauss-Jordan elimination is a direct solver that fits the GPU programming model well; PLU factorization is another direct solver, efficient for solving a system for multiple right hand sides; tridiagonal Gaussian elimination solves a tridiagonal system of equations efficiently. The four operators are accessed via MATLAB using familiar MATLAB syntax, and the MATLAB interface uses a processing queue and background processing on the GPU. This enables the use of the GPU *and* the CPU simultaneously for maximum performance.

2 Related work

Coupling the GPU and MATLAB has been shown by NVIDIA, who presented a MATLAB plug-in for 2D FFT using CUDA [10]. This speeded up simulation of 2D isotropic turbulence by almost a factor 16 on a 1024×1024 grid.

Larsen and McAllister [11] were the first to present matrix-matrix multiplication on the GPU, prior to the arrival of programmable hardware. Hall, Carr and Hart [12] presented a

way of blocking the computation by using several passes, and reported 25% less data transfer and executed instructions compared to the former. Jiang and Snir [13] showed a way of tuning matrix multiplication automatically to the underlying hardware, and reported 13 GFLOPS for a hand-tuned version, compared to 9 GFLOPS for the automatically tuned version on an NVIDIA GeForce 6800 U. For other hardware setups, the automatically tuned version outperformed hand-tuned equivalents. The efficiency of using blocking techniques for the matrix-matrix product was analyzed by Govindaraju et al. [14], where they reported less than 6% cache misses when using efficient block-sizes, and 17.6 GFLOPS on an NVIDIA GeForce 7900 GTX. Peercy, Segal and Gerstmann [15] presented an implementation of matrix-matrix multiplication using CTM on an ATI X1900 XTX graphics card, where they reported their implementation to perform 110 GFLOPS.

Galoppo et al. [16] presented PLU factorization using the Doolittle algorithm on the GPU, and single-component textures to store the data. Using consecutive passes, they first located the pivot element. Then the rows (and columns for full pivoting) were interchanged in yet another pass, and finally the matrix was reduced. This process was repeated until the whole matrix was decomposed. They claimed their algorithm to be 35% faster than ATLAS [17] for PLU factorization with partial pivoting, and an order of magnitude faster than the Intel Math Kernel Library (MKL) [18] for full pivoting. However, the benchmarks were highly synthetic, and assumed no cache misses.

Bolz et al. [19] presented a conjugate gradient algorithm using the GPU, where they stored their sparse matrices using two textures. The first texture simply contained all nonzero elements in the matrix packed row by row. The second texture contained pointers to the first element in each row. Their conjugate gradient algorithm was implemented using the GPU to compute sparse matrix-vector multiplication and sparse vector-vector inner product. They reported overhead connected with pixel buffer switching as the limiting factor. This overhead is now far less when using framebuffer objects instead of pixel buffers. Krüger and Westermann [20] used another storing strategy for banded sparse matrices, where each diagonal-vector was stored in a separate texture. They reported precision issues, and claimed a speedup over their reference CPU implementation. Vectorized SSE implementations, however, are supposed to be $2-3\times$ faster than their CPU implementation.

Part of the material presented here is a result of the master's thesis "A MATLAB Interface to the GPU" [21]. The reader is encouraged to consult the master's thesis for implementation and other details.

3 A GPU toolbox for MATLAB

MATLAB is a standard tool for scientists and engineers all over the world. Utilizing the GPU as a mathematical coprocessor will not only offload the CPU, but possibly also increase performance.

MATLAB supports user-defined classes and operator overloading for these classes. This enables us to implement a *gpuMatrix* class, which can be programmed to execute custom code for all standard MATLAB operators and functions. These functions can be programmed in C/C++ as a MATLAB executable (MEX) file. By programming the MEX file to use the GPU as the computational engine, we can utilize the GPU in MATLAB. However, using the GPU most often excludes use of the CPU simultaneously because many calls to OpenGL are *blocking*.

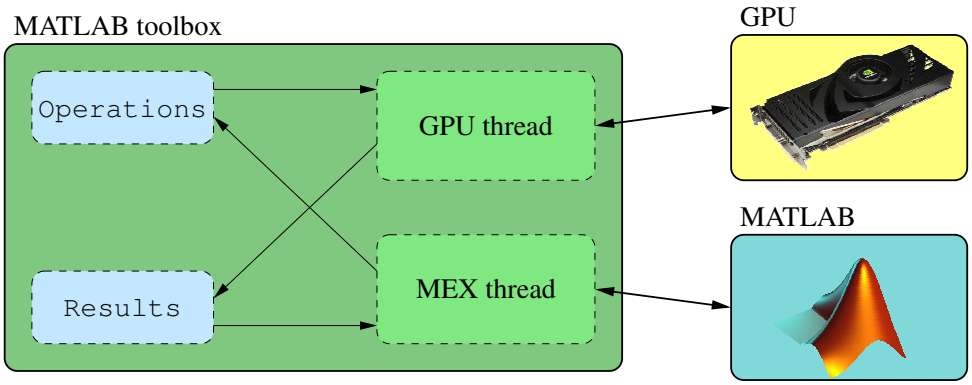


Figure 1: Splitting of program execution into one part dealing with MATLAB, and one part dealing with the GPU

Synchronous data transfer to or from the CPU are examples of blocking calls where both the CPU and the GPU stop executing while data is transferred. Using the GPU to compute results in the background, however, will enable the use of the CPU simultaneously.

To use the GPU as a mathematical coprocessor, working in the background, we utilize threads that execute code independently from each other. Because neither MATLAB [22] nor most OpenGL driver implementations are thread-safe, we cannot arbitrarily call MATLAB and OpenGL functions from different threads. To circumvent this, the program execution is split into two separate threads, as shown in Figure 1. The *MEX thread* holds a MATLAB context and communicates with MATLAB, while the *GPU thread* holds an OpenGL context and communicates with the GPU. The two threads communicate with each other via a queue of operations, and a map of results.

When MATLAB operates on a gpuMatrix object, the MEX thread is called. It then creates the wanted operation, adds it to the operations queue, notifies the GPU thread of a change, and returns a unique ID to the operation. The GPU thread receives the notification, and executes all elements in the operations queue. Every completed operation is moved from the operations queue to the results map, where the ID of the operation is the key. When the result is requested by the user via a new call from MATLAB, the MEX file simply waits until the correct ID appears in the results map. When it is found, the result returned to MATLAB. The conversion between an operation ID and the corresponding matrix is transparent for the user, and an operation ID can be enqueued in further operations even before the result is computed.

4 Operators on the GPU

Data transfer between the GPU and the CPU has to pass through the 4GB/s full duplex PCI Express $16\times$ bus [23]. To prevent the need to repack data (and transfer it over the PCI Express bus again) for reuse in other GPU computations, we have implemented all algorithms using a consistent data-structure. We have used the 2×2 packing scheme proposed by Hall, Carr and Hart [12], where 2×2 sub-matrices are packed into the four color vectors in each pixel (red, green, blue and alpha). This packing utilizes the vectorized arithmetic found in many GPUs, and offers good performance for most applications, even though other packing schemes might fit specific algorithms better.

Full matrix-matrix multiplication

We have implemented two versions of matrix multiplication. One single-pass, and one multi-pass. Hall, Carr and Hart [12] presented a multi-pass algorithm that views the matrix-matrix product as a sum of individual multiplications. But because the matrix is packed using the 2×2 schema, the algorithm computes the "inner product" of two 2×2 matrices as

$$C_{i,j}^{k+1} = C_{i,j}^{k} + \begin{bmatrix} a_{i,2k+1} & a_{i,2k+2} \\ a_{i+1,2k+1} & a_{i+1,2k+2} \end{bmatrix} \begin{bmatrix} b_{2k+1,j} & b_{2k+1,j+1} \\ b_{2k+2,j} & b_{2k+2,j+1} \end{bmatrix}, \tag{1}$$

where $C_{i,j}^{k+1}$ is the result buffer, and $C_{i,j}^{k}$ is an intermediate accumulation buffer. The result is computed in $n/2$ passes, so that the product $AB = C^{n/2}$. Here the role of the accumulation and destination buffers are swapped each pass.

This algorithm forms the basis for our implementation of the multi-pass algorithm. Instead of using an extra accumulation buffer, we accumulate using a single buffer. Writing to a texture which is also input to the computation is undefined [16], because the order of computation is unknown, i.e., you do not know which pixels are computed first. Nevertheless, our empirical tests on the NVIDIA GeForce 7800 GT show that writing to the same buffer works as long as the input and output texels are at the exact same position. Utilizing this eliminates the need for a separate accumulation buffer in our algorithm, thus significantly lessening memory requirements.

Fatahalian, Sugerman and Hanrahan [24] presented a single-pass matrix multiplication algorithm that corresponds to viewing the matrix multiplication as a series of vector-vector inner products. Each output element is then computed as

$$(AB)_{i,j} = \sum_{k=1}^{n/2-1} \begin{bmatrix} a_{i,2k+1} & a_{i,2k+2} \\ a_{i+1,2k+1} & a_{i+1,2k+2} \end{bmatrix} \begin{bmatrix} b_{2k+1,j} & b_{2k+1,j+1} \\ b_{2k+2,j} & b_{2k+2,j+1} \end{bmatrix}. \tag{2}$$

The main difference from the multi-pass algorithm is that the for-loop is moved from the CPU to the GPU, eliminating the need for several passes. Our algorithm is implemented as a GPU program that runs once, where one 2×2 sub-matrix of the result matrix is computed for each pixel.

Jiang and Snir [13] reported the single-pass algorithm as faster than the multi-pass algorithm for all the hardware setups they benchmarked on. Nevertheless, they did not benchmark on the hardware used here, the NVIDIA GeForce 7800 GT and 8800 GTX. Since we are operating on new hardware setups, we have implemented both the single- and multi-pass algorithms.

Gauss-Jordan elimination

We have implemented Gauss-Jordan elimination with partial pivoting. Gauss-Jordan elimination fits the GPU programming model better than standard Gaussian elimination because only *half* the number of passes are needed. Because we have packed 2×2 sub-matrices into each pixel, we exchange rows of 2×2 elements. This optimization increases performance since we only need *half* the number of passes compared to exchanging single rows, but will possibly create larger numerical errors than standard partial pivoting.

Finding the largest element of our pivot candidates requires a measure for each candidate.

We use the value of the diagonal-elements after forward substitution,

$$\begin{bmatrix} r & g \\ b & a \end{bmatrix} \xrightarrow{\text{Subst.}} \begin{bmatrix} r & g \\ 0 & a - \frac{b}{r}g \end{bmatrix}. \tag{3}$$

This gives us the diagonal-elements $q_{1,1} = r$ and $q_{2,2} = a - \frac{b}{r}g$, where we compute

$$k = \frac{q_{1,1} \cdot q_{2,2}}{q_{1,1} + q_{2,2}} \tag{4}$$

similarly to the harmonic mean. We have experimentally found k to be a good measure for our application.

Finding the pivot element is done using a multi-pass reduction shader (GPU program) that first computes k for each element, and then reduces the vector down to one element, the maximum. In addition to finding the largest element, we also need to find the corresponding coordinate. It is not trivial to compute both the maximum *and* its norm effectively in one shader on the GPU. The naïve approach of using if-tests is a possibly expensive task, as all processors in the same single instruction multiple data (SIMD) group have to execute the same instructions; if one of the processors branches differently from the others, all processors have to evaluate both sides of the branch. We can, however, rewrite the branches into implicit if-tests, e.g., `float(a == b)*result`. This gives us the maximum, as well as the correct coordinate. If two elements have identical norms, the largest coordinate is selected.

When the reduction is complete, we are left with the greatest coordinate of the largest norm. If the largest norm is sufficiently close to zero, the matrix is assumed to be singular or near-singular.

After we have located the pivot element we need to swap the top row with the pivot row, convert the leading element to a leading one, and reduce all elements above and below to zero. Since we are using 2×2 packing, we normalize two rows, and eliminate two columns. This is done in a ping-pong fashion reading from the previously computed values, writing to the destination buffer. The pivot row is normalized when it is written to the position of the top row. The top row is simultaneously written at the position of the pivot row, and eliminated. The rest of the matrix is then eliminated in the next pass. This process is repeated until the matrix is reduced to the identity in the left part of the matrix, with the solution to the right. It should be noted that the algorithm easily can be extended to full pivoting as well.

PLU factorization

The Doolittle algorithm for computing the PLU factorization is a small alteration of Gaussian elimination. The pivoting order is used to construct P, the multipliers used in the elimination are stored to create the unit lower triangular matrix L, and the matrix resulting from pivoting and forward substitution is the upper triangular U. The algorithm is executed as follows:

1. Find the pivot element.
2. Calculate two rows of U from the pivot row. Render the top row at the position of the pivot row simultaneously, thus swapping the two rows.
3. Eliminate below the top row, using the normalized pivot row.

The pivoting strategy used is the same as described for Gauss-Jordan elimination, and the pivoting order is stored on the CPU to construct P. When using the 2×2 packing scheme, we have to calculate two rows of U simultaneously. We do not need to alter the top row in the 2×2 row, but we have to reduce the bottom row in the same fashion as shown in Eq. (3). The last step of the algorithm calculates the multipliers needed to eliminate the rest of the column, and reduces the lower right part of the matrix accordingly. This process is repeated until the whole matrix is factorized.

When the matrix is factorized, we have L and U stored on the GPU as one texture. After transferring back to the CPU, L is constructed by adding the lower part of the texture with the identity matrix, and U is simply the upper triangular part of the texture.

Tridiagonal Gaussian elimination

Tridiagonal systems of equations arise e.g., when solving PDEs and ODEs numerically. Solving these systems using a full matrix solver is highly inefficient, as most elements are known to be zero. We can exploit the structure of the matrix to provide an efficient tridiagonal solver. We store the non-zero diagonals and the right hand side of the system in the four color channels red, green, blue and alpha:

$$\left[\begin{array}{ccccc|c} g_1 & b_1 & 0 & 0 & 0 & a_1 \\ r_2 & g_2 & b_2 & 0 & 0 & a_2 \\ 0 & r_3 & g_3 & b_3 & 0 & a_3 \\ 0 & 0 & r_4 & g_4 & b_4 & a_4 \\ 0 & 0 & 0 & r_5 & g_5 & a_5 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} 0 & g_1 & b_1 & a_1 \\ r_2 & g_2 & b_2 & a_2 \\ r_3 & g_3 & b_3 & a_3 \\ r_4 & g_4 & b_4 & a_4 \\ r_5 & g_5 & 0 & a_5 \end{array}\right].$$

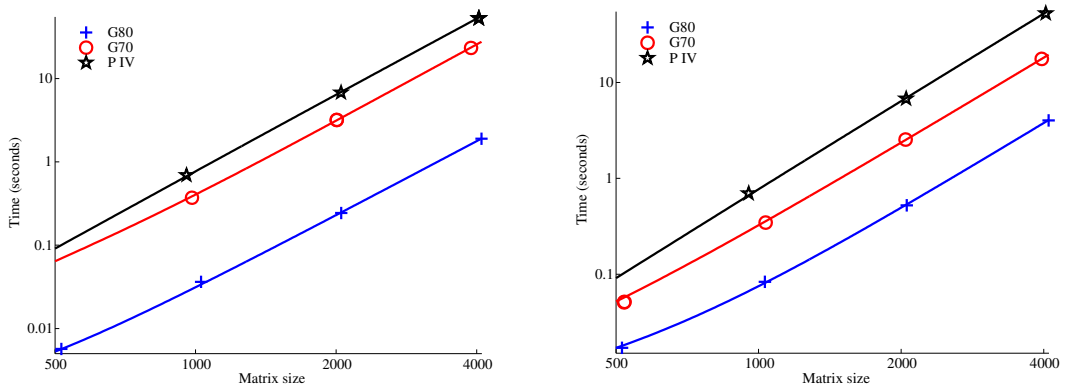
We perform $n-1$ passes where we forward substitute, thus eliminating r_{i+1} at pass i, followed by $n-1$ passes where we backward substitute, eliminating b_{n-i} in pass i.

This is, however, a highly serial computation, as only one row of the matrix is updated in each pass. In order to benefit from the parallel execution mode of the GPU, we solve many such systems in parallel. Our tridiagonal solver is created specifically to solve many tridiagonal systems of equations, such as those that arise in the semi-implicit alternating direction discretization of the shallow water equations [25]. It is also possible to solve a single tridiagonal system in parallel (see e.g., [26]), but this is not the focus of our approach. To solve many systems in parallel, we stack our systems beside each other so that system i is in column i of the texture. This allows us to solve up-to 8192 tridiagonal systems of up-to 8192 equations in parallel on the NVIDIA GeForce 8800 GTX, which is the maximum texture size.

5 Results

The algorithms presented in the previous section all have native MATLAB implementations as well. MATLAB uses ATLAS [17], LAPACK [27], and BLAS [28] routines for its numerical linear algebra algorithms [29, 30]. The routines offered by these libraries are regarded as highly optimized, and the MATLAB interface to them is considered efficient as well [31]. It should be noted, however, that the GPU does not offer fully IEEE-754 compliant floating point (there are some anomalies with e.g. denormals), which might be discerning for some uses.

The following compares the time used by the native MATLAB algorithm and the GPU algorithm. The benchmark times are measured using MATLABs internal timing mechanism.



(a) Single-pass **(b)** Multi-pass

Figure 2: The weighted least squares approximation to the execution time for full matrix multiplication, and a subset of the measured execution times. Notice that the G70 GPU performs best using single-pass algorithm, and the G80 using the multi-pass algorithm.

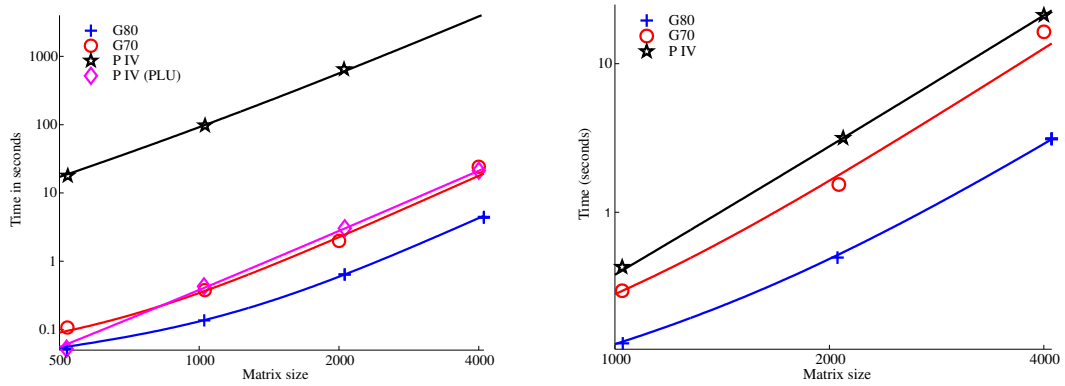
Time spent packing and transferring data between the GPU and the CPU is not included, as these are looked upon as constant startup costs. When we reuse data in multiple computations as described in Section 4 this cost will become insignificant. For single computations, however, the startup-cost will have a larger influence on the runtime, and should be included.

To measure how fast the GPU is, we have approximated a polynomial, $a + bn + cn^2 + dn^3$, to the measured runtimes using the method of weighted least squares. In our experiments, we found this to give good approximations to the data-points, even though we do not know the specific complexity of the native MATLAB implementation. By comparing the dominant factor, d, we can compute a more realistic speedup-factor than by only comparing peak measured GFLOPS. We have used this approach for all algorithms except the tridiagonal Gaussian elimination.

The algorithms have been benchmarked on two GPUs, an NVIDIA GeForce 7800 GT (G70), and an NVIDIA GeForce 8800 GTX (G80). The CPU used for the native MATLAB implementation is a 3 GHz Pentium IV system (P IV) with 2GB of RAM. The matrices being used for the benchmarks are random matrices, as neither condition number, sparsity or structure influences the runtime of the algorithm. Since the tested hardware only supports single precision, there are precision issues for poorly conditioned matrices.

Full matrix-matrix multiplication

Figure 2 shows the execution times of MATLAB and the presented toolbox. The GPU implementation is slower than the highly optimized CPU code for small matrices. This is because there is a larger overhead connected with starting computation on the GPU than on the CPU. For large matrices, however, we experience a speedup. We computed the coefficient, d_{CPU}, of our polynomial approximating the CPU runtime to be $8.36\text{e-}10$, while the single-pass coefficients for the G70 and G80 GPUs are $4.26\text{e-}10$ and $2.67\text{e-}11$. This gives us speedup factors of $1.96\times$ and $31.29\times$ for the G70 and G80 GPUs respectively. Using the same approach for the multi-pass algorithm, we get speedup factors of $3.04\times$ and $14.25\times$. These speedup factors are good approximations for matrices larger than $\sim 500 \times 500$.



(a) Gauss-Jordan **(b)** PLU

Figure 3: The weighted least squares approximation to the execution time for Gauss-Jordan elimination and PLU factorization with a subset of the measured execution times. Notice that Gauss-Jordan on the GPU is compared to PLU factorization on the CPU.

Gauss-Jordan elimination

MATLAB implements Gauss-Jordan elimination as the function `rref()`. Benchmarking the GPU version against the native MATLAB implementation, however, gave a speedup of $170\times$ and $680\times$ for the G70 and G80 GPUs respectively. This indicates that the MATLAB version is sub-optimal. To give a more appropriate speedup factor, we have chosen to compare against the PLU factorization in MATLAB. However, Gauss-Jordan elimination is a far more computationally heavy operation than PLU factorization. PLU factorization is also a far less memory heavy operation. In effect, our speedup factors are highly modest.

Figure 3a shows the execution time of the *PLU factorization in MATLAB*, and *Gauss-Jordan elimination on the GPU*. By comparing the dominant coefficient of our approximating polynomial, we can estimate the speedup factor to be $1.07\times$ and $4.27\times$ for the G70 and G80 GPUs. This speedup seems to fit the sample points well for matrices larger than $\sim 1000 \times 1000$.

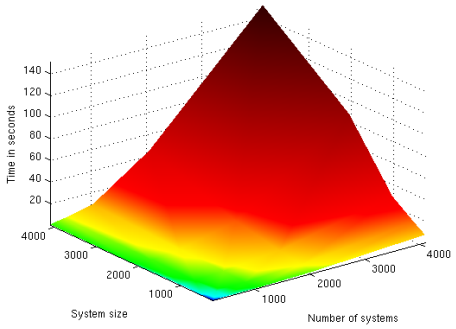
PLU factorization

Figure 3b shows both the measured times for the CPU and the GPU, as well as the least squares approximation of the sample points. The approximated speedup over MATLAB is $1.48\times$ for the G70 GPU, and $7.55\times$ for the G80. These speedup factors seem to be valid for matrices larger than $\sim 1000 \times 1000$.

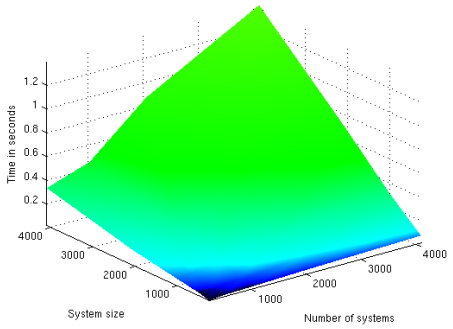
Tridiagonal Gaussian elimination

Benchmarking the tridiagonal solver is done using sparse storage on both the CPU and GPU. On the CPU, the systems have to be solved sequentially in a for-loop, while the systems are solved with one call to the GPU. On the GPU, the time includes time spent transferring data between MATLAB and the GPU, as this algorithm does not use the 2×2 packing.

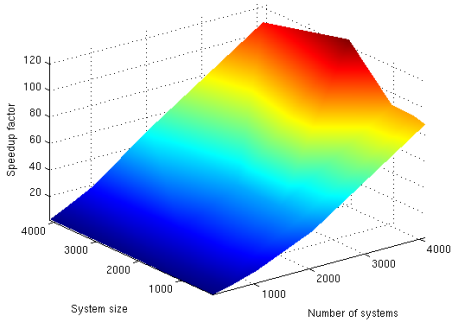
Figure 4a shows a plot of the execution time for MATLAB, and Figure 4b shows the execution time for the GPU. Figure 4c shows the computed speedup-factors. Notice that there is almost no speed-up gain by increasing the size of the systems, whilst increasing the number of systems solved in parallel yields massive speedups. The maximum observed speedup is $125\times$



(a) CPU execution time (P IV)



(b) GPU execution time (G80)



(c) Speedup factor

Figure 4: The measured execution time for tridiagonal Gaussian elimination using sparse storage. Notice that it is highly efficient to solve many systems in parallel on the GPU.

for solving 4096 systems of 2048 equations in parallel.

Background computation

In addition to using the GPU alone, the presented interface also offers background computation. This background processing enables us to utilize both the CPU and the GPU simultaneously for maximum performance. Because they operate asynchronously, we can compute a crude approximation to a good load balance by timing the CPU and the GPU.

Using the load ratio to distribute work between the CPU and the GPU, we achieve background computation on the GPU that is virtually free. The load ratio is set to solve two systems on the CPU while one system is solved simultaneously on the GPU. Figure 5 shows the average time per system for the GPU, the CPU, and the total average time when using background processing. The largest cost connected with using the GPU is copying the data into memory allocated by MATLAB. Unfortunately this step is required, as MATLAB has internal memory management, disabling thread-safe use of its memory. Because we now include the overhead imposed by packing, transferring and unpacking each matrix in our timings, we experience less speedups than previously reported. This is the worst case scenario. Typical use should try to reuse data that resides on the GPU in multiple computations to hide the overhead of data-transfer. Nevertheless, the background processing decreases the overall execution time, and we can compute the speedup-factor to be $1.68\times$ over using just the CPU.

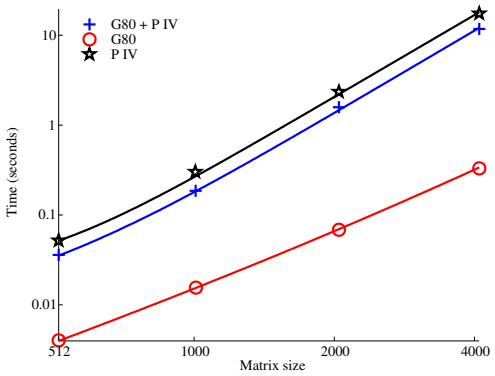


Figure 5: Average time spent computing the PLU factorization of 30 systems using both the CPU and the GPU. The graphs show time spent waiting for the result to be computed on the CPU, the GPU and total average time. Notice that utilizing the GPU is very inexpensive.

6 Conclusions and further research

We have presented an interface to the GPU from MATLAB, enabling the use of both the CPU and the GPU for maximum performance. In addition, four algorithms from numerical linear algebra have been presented for this interface, and shown to be up to $31\times$ faster than highly optimized CPU equivalents. Furthermore, a new pivoting strategy, and the use of 2×2 packing for Gauss-Jordan and PLU factorization has been presented.

The presented interface can be used to utilize both the CPU and the GPU simultaneously, and an automatically tuned load distributor will be of great interest in such a setting. The presented algorithms, matrix multiplication, Gauss-Jordan elimination and PLU factorization, are all implemented using 2×2 storage. The G80 GPU from NVIDIA implements scalar arithmetic, eliminating the need for this packing. Jiang and Snir [13] have presented an approach to automatic tuning of matrix-matrix multiplication on the GPU to the underlying hardware. Such an automatic tuning of the presented algorithms will further increase the usefulness the GPU toolbox for MATLAB. Utilizing APIs such as CUDA and CTM instead of OpenGL will probably also increase performance, as we do not need to rephrase the problem in terms of graphics. An optimized high-level mathematical interface to the GPU, such as described in this article, is not only interesting for MATLAB, but also other high-level languages, e.g., Python.

Acknowledgments

I would like to thank K.-A. Lie, T. R. Hagen, T. Dokken, J. Hjelmervik and J. Seland for their guidance and help with this document.

Bibliography

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Comp. Graph. Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[2] D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide*, 5th ed. Addison-Wesley, 2005.

[3] Microsoft Corporation. (2007) Microsoft DirectX.

[4] Advanced Micro Devices Inc. (2006) ATI CTM guide.

[5] NVIDIA Corporation. (2007) CUDA programming guide.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," in *SIGGRAPH '04*. ACM Press, 2004, pp. 777–786.

[7] M. D. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule, "Shader algebra," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 787–795, 2004.

[8] M. D. McCool and B. D'Amora, "Programming using RapidMind on the Cell BE," in *Supercomputing '06*. ACM Press, 2006, p. 222.

[9] Advanced Micro Devices Inc. (2007) AMD delivers first stream processor with double precision floating point technology.

[10] NVIDIA Corporation. (2007) Accelerating MATLAB with CUDA using MEX files.

[11] E. S. Larsen and D. McAllister, "Fast matrix multiplies using graphics hardware," in *Supercomputing '01*. ACM Press, 2001, pp. 55–55.

[12] J. D. Hall, N. A. Carr, and J. C. Hart, "Cache and bandwidth aware matrix multiplication on the GPU," 2003.

[13] C. Jiang and M. Snir, "Automatic tuning matrix multiplication performance on graphics hardware," in *Parallel Arch. and Compilation Techniques*. IEEE CS, 2005, pp. 185–196.

[14] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A memory model for scientific algorithms on graphics processors," in *Supercomputing '06*. ACM Press, 2006, p. 89.

[15] M. Peercy, M. Segal, and D. Gerstmann, "A performance-oriented data parallel virtual machine for gpus," in *SIGGRAPH '06*. ACM Press, 2006, p. 184.

[16] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha, "LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware," in *Supercomputing '05*. IEEE CS, 2005, p. 3.

[17] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Supercomputing '98*. IEEE CS, 1998, pp. 1–27.

[18] Intel Corporation. (2007) Intel math kernel library 9.1 – product brief.

[19] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 917–924, 2003.

[20] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 908–916, 2003.

[21] A. R. Brodtkorb, "A MATLAB interface to the GPU," Master's thesis, University of Oslo, May 2007.

[22] A. Källander, "Multithreading in MEX files," Personal email communication, 2007.

[23] Microsoft Corporation. (2004) PCI Express FAQ for graphics.

[24] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," in *Graph. Hardw.* ACM Press, 2004, pp. 133–137.

[25] V. Casulli, "Semi-implicit finite difference methods for the two-dimensional shallow water equations," *Journ. of Comp. Phys.*, vol. 86, pp. 56–74, 1990.

[26] H. S. Stone, "Parallel tridiagonal equation solvers," *ACM Trans. Math. Softw.*, vol. 1, no. 4, pp. 289–307, 1975.

[27] J. Dongarra and J. Wasniewski, "High performance linear algebra package - lapack90," ser. Combinatorial Optimization, Advances in Randomized Parallel Comp. Kluwer Academic Publishers, 1999, vol. 5.

[28] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, 1979.

[29] The MathWorks. (2006) MATLAB software acknowledgements.

[30] C. Moler. (2000) MATLAB news & notes - winter 2000.

[31] I. Mcleod and H. Yu. (2002) Timing comparisons of Mathematica, MATLAB, R, S-Plus, C & Fortran.

PAPER IV

AN ASYNCHRONOUS API FOR NUMERICAL LINEAR ALGEBRA

A. R. Brodtkorb

In Scalable Computing: Practice and Experience, West University of Timisoara, 9(3) (Special Issue on Recent Developments in Multi-Core Computing Systems) (2008), pp. 153–163.

Abstract: We present a task-parallel asynchronous API for numerical linear algebra that utilizes multiple CPUs, multiple GPUs, or a combination of both. Furthermore, we present a wrapper of this interface for use in MATLAB. Our API imposes only small overheads, scales perfectly to two processor cores, and shows even better performance when utilizing computational resources on the GPU.

1 Introduction

Algorithms from numerical linear algebra are important tools with a variety of uses. Common to many of these algorithms is that they require a substantial amount of computation time. Therefore, there has been a lot of research into developing optimized APIs and libraries such as BLAS [14], FLAME [16], LAPACK [2] and PLASMA [13]. However, when these libraries are used, they stall the program execution until each algorithm has completed. This prevents overlapping heavy computation with other operations, such as reading in new data from disk, without complex multithreaded programming. We present an interface that offers asynchronous and task-parallel execution of BLAS functions on different *backends*. This enables easy overlap with other operations and enables us to utilize several processor cores, multiple graphics processing units (GPUs), or both in combination.

The interface we present consists of a frontend that exposes familiar BLAS functions and several backends that implement them. The frontend schedules execution of the algorithms to the different backends run-time based on a set of simple criteria to utilize all available processing power. We further present a wrapper of this interface for use in MATLAB. This is a natural extension to our previous work [8], where we showed speed-ups using the GPU as a computational resource in MATLAB.

We have focused on utilizing multicore CPUs in conjunction with one or more GPUs. However, our ideas could also have been applied to other accelerator cores such as the Cell broadband engine (Cell BE) [17] or field programmable gate arrays (FPGAs) [10].

The rest of this article is sectioned as follows: Section 2 discusses current multi-core hardware and software trends, and we relate our API to existing programming languages and APIs. Readers familiar with these trends might opt to jump straight into Section 3, where we contrast our contribution to related work. Section 4 describes our interface, followed by Section 5 where we show performance results. We end this article with some concluding remarks and future research directions in Section 6.

2 Current Trends in Parallel Commodity Hardware

The achieved performance of computer programs has traditionally, with only minor adjustments, increased with new hardware generations. The performance gain has mainly come from an increase in four metrics: size of system memory, processor core clock frequency, speed of system memory, and number of instructions per clock cycle. The size of system memory is limited to 4 GiB on 32 bit systems, but this limitation was increased to a theoretical 256 GiB for 64 bit systems with the emergence of the x86-64 instruction set in 2003. The increase of the three other metrics, however, has come to a halt. The factors limiting further growth have been called the *power wall*, *memory wall*, and *ILP wall*, constituting a “brick wall for serial performance“ [3].

The power wall prevents further increases in clock frequency. Increasing the clock frequency requires reducing the gate size and increasing the supply voltage. Both reducing the gate size and increasing the supply voltage leads to an increase in leakage power and generated heat. This generated heat is proportional to the square of the frequency, and we seem to have reached a physical limit to what the chips can withstand without exotic cooling.

The size of on-chip caches has increased drastically in recent years. This is because the

bandwidth to off-chip memory has become relatively slower and slower, referred to as the *von Neumann bottleneck*. One of the reasons for this slow-down is that there is a physical limit to the number of pins connecting the processor to the motherboard and main memory. The speed per pin is also limited, further obstructing speed increases. This limitation is commonly known as the memory wall.

The third wall is referred to as the ILP wall. Increasing the number of instructions executed per clock cycle is a result of instruction level parallelism (ILP). Complex logic executes dependent instructions concurrently by predicting the program flow. Further development of ILP, however, is currently unfeasible because of the exponentially difficult task of predicting future instructions.

These three walls mark the end of serial performance increase for the time being, and the era of multicore computing has begun. Dual- and quad-core processors have already become the mainstream of processors, and we will see a steady increase in the number of cores in the future. This represents a great challenge to computer scientists and algorithm designers as most algorithms are designed for serial architectures, thus unable to benefit from these processor designs.

In addition to the processing power offered by the multicore CPU, most modern computers are also equipped with one or more dedicated graphics cards to accelerate rendering in games. The graphics card looks a lot like a separate computer by itself: it contains a GPU, dedicated graphics memory, and a graphics BIOS. The cards are accessed through the graphics driver that offers one or more graphics APIs.

In current games, you typically have a screen resolution of 1600×1200 pixels, with at least 60 frames drawn each second. This totals to over one hundred million pixels each second. The number of operations per pixels can be very large, as modern games often use advanced rendering techniques requiring several rendering passes per frame. To meet these demands graphics cards have developed an extreme computational capacity. Current consumer level GPUs can have a theoretical peak performance of over 1 TFLOPS, compared to CPUs that have less than 150 GFLOPS. The GPU also outperforms the CPU when it comes to memory bandwidth. Motherboards give the CPU access to main memory at speeds up to 25 GiB/s, while the typical memory speed of a commodity-level computer is around 13 GiB/s, or even less. For GPUs, however, the memory speeds reach over 140 GiB/s. And with good reason. It takes a lot of bandwidth to process over one hundred million pixels each second.

Using graphics processors to accelerate non-graphical applications is a field that has gained in popularity, see e.g., `www.gpgpu.org` and Owens et. al. [25]. One of the main bottlenecks with such use of the GPU is the relatively slow data bus between GPU and main memory. Heterogeneous processor designs such as the Cell BE and the coming AMD Fusion processors remove this bottleneck by incorporating accelerator cores on the same silicon die as the CPU. It is commonly accepted that using such accelerator cores outperform pure CPU implementations for a variety of processing tasks. See for instance Brodtkorb et. al. [9] for an example of a comparison between multi-core CPUs, the GPU, and the Cell BE.

The supercomputer community is also using accelerator cores to achieve higher performance. Bull have disclosed that they will build a supercomputer with 1080 octa-core Nehalem CPUs in conjunction with 96 NVIDIA Tesla [18] GPUs for Grand Equipement National de Calcul Intensif in France. The RoadRunner project at Los Alamos National Laboratory utilizes

6,948 AMD dual-core CPUs in conjunction with 12,960 Cell BE processors, and is the first machine to achieve over one PFLOPS sustained performance. It is currently the most powerful supercomputer in the world, and also one of the most energy efficient.

Programming languages and APIs

There are already many different programming languages and APIs for multicore computing, such as POSIX Threads (pthreads) [12], Intel Threading Building Blocks (TBB) [27] and OpenMP [24]. All these libraries use threads to utilize multiple processor cores.

POSIX Threads is a low-level API where the programmer handles each thread explicitly. This level of access offers great flexibility, but at the same time requires a lot from the programmer. Programs written using pthreads are subject to thread issues such as *race-conditions*, *deadlocks*, *starvation*, and *priority failures*, all of which require the programmer to create and maintain complex logic.

TBB is a C++ library that uses threads, but focuses on *tasks*. Given a parallel algorithm, TBB aids in partitioning and scheduling the tasks, thus reducing the risk of thread issues. It also implements task stealing that enables dynamic load balancing. The task stealing is performed runtime by splitting large tasks into smaller tasks, and then redistributing them to idle threads. Starting these tasks is 100 times faster than starting a thread on Windows XP.

OpenMP consists of a set of compiler commands. The programmer outlines the parallel sections of the code using a set of pragmas, guiding the compiler where to perform parallelization. The compiler recognizes these pragmas, and generates code that can execute in parallel on a shared memory machine. OpenMP is used in the OpenMP Multi-Threaded Template Library, which is a parallel implementation of `<numeric>` and `<algorithm>` from the standard template library (STL).

The asynchronous linear algebra API we present here uses Boost::Threads, a portable API for threading that is slightly higher level than POSIX Threads. We offer a task-parallel API, using ideas similar to TBB. However, we do not implement task subdivision and task stealing, and focus only on mathematical operations defined in the BLAS API.

Traditionally, the GPU had to be accessed through a graphics API such as OpenGL [28] or DirectX [20]. Accelerating computations using the GPU thus required that the algorithm could be rewritten in terms of operations on graphical primitives, a tedious and error prone solution for non-graphics use. An increasing demand for non-graphical APIs sparked a lot of research into creating other abstractions, and there are currently two dominant APIs for *stream computing*; NVIDIA CUDA [23] and Brook [11].

CUDA is a data-parallel programming language and paradigm that requires algorithms to be structured into blocks of independent computation. It can execute on recent NVIDIA GPUs, or in emulation mode on the CPU. The CUDA Zone at the NVIDIA website contains an updated list of academic and industrial uses of CUDA. This programming paradigm has also been implemented for multicore CPUs as MCUDA [29].

Brook is an open source API developed at Stanford. The Stanford version has not reached a final release, but has spurious updates. AMD has developed an extension to the API called Brook+ [1] for their GPUs, but the language specification is not fully implemented. The Brook API is used by Folding@home project to simulate protein folding on both GPUs and the Cell BE in the PlayStation 3 gaming console.

PyStream is a python interface to CUDA, CUBLAS [22] and CUFFT [21] that supports seamless data-transfer between CPU and GPU memory space. TechX has stopped developing this open source version, but continues developing GPULib, a library that accelerates mathematical functions using the GPU. GPULib offers bindings to Python, MATLAB and the Interactive Data Language (IDL).

RapidMind [19] is a high-level stream programming API with backends for both multi-core CPUs, GPUs, and the Cell BE. The backends are themselves responsible for low-level optimization, and RapidMind has published results where highly optimized RapidMind code outperformed hand-tuned code [26].

Our approach is similar to GPULib, as we also offer transparent data-transfers and asynchronous execution of mathematical functions on the GPU. As with RapidMind, we also support different backends for our interface. RapidMind offers a general programming framework for data-parallel execution, whereas we focus on task-parallel mathematical functions. In addition, we support using multiple backends simultaneously for task-parallel execution.

3 Related Work

For an overview of history and recent developments in linear algebra on the GPU, Owens et. al. [25], and the references therein, provide a good summary. We have previously presented an interface between MATLAB and the GPU [8], where algorithms from numerical linear algebra were accelerated by the GPU. This early work required the algorithms to be written in terms of operations on graphical primitives, as only graphics APIs were available for our NVIDIA GPU. Accelereyes are developing a product called Jacket to accelerate MATLAB using the GPU. They use ideas very similar to our original paper but also offer OpenGL visualization.

Faticia and Jeong [15] have also presented a coupling of MATLAB and the GPU. They used CUFFT to accelerate simulations run from MATLAB, and reported the GPU version to be 14 times faster for a 2D isotropic turbulence problem. GLAME@LAB [6] uses CUBLAS to accelerate FLAME@LAB [7], the MATLAB interface to the FLAME API [16]. They showed speedups over Octave for several different algorithms. The performance of CUBLAS has been explored by Barrachina et. al. [5]. They also presented a hybrid sgemm (**s**ingle-precision **ge**neral **m**atrix **m**ultiply) algorithm that utilized both CPU and GPU computational power by splitting the computation into blocks.

Our contribution offers asynchronous execution of mathematical functions that enable us to overlap computation with other operations. We also offer asynchronous data transfer, and task-parallel execution of BLAS functions, being able to benefit from both multiple CPUs, multiple GPUs, or a mixture of both.

4 Interface

We present an asynchronous interface to algorithms in numerical linear algebra. The interface is divided up into a frontend exposed to the user, and one or more backends. The backends can be implemented for different processors, allowing for a heterogeneous processing platform. This section describes the frontend, the backends, and a MATLAB wrapper of the frontend. The interface is general enough to support new backends, e.g., an FPGA backend, and new frontend wrappers, e.g., a Python wrapper.

Figure 1 shows the main layout of the interface. The frontend is responsible for creating

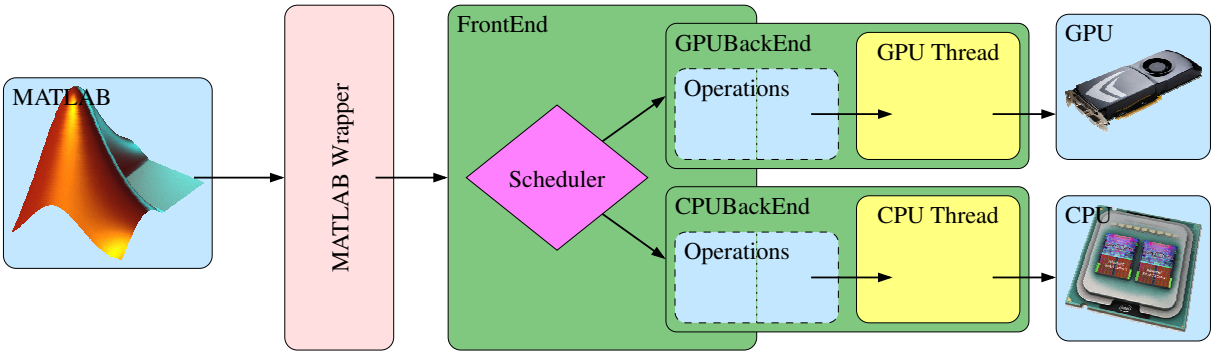


Figure 1: The interface design.

tasks, and scheduling the tasks to the backends. These tasks are lightweight data-structures that transport pointers to input and output data between the frontend and the backends. The scheduling algorithm tries to find the best backend to schedule the process to, and then adds the task to the queue of the appropriate backend. The backends run asynchronously in separate threads, continuously processing tasks from their task-queue.

Frontend

The frontend defines functions exposed by our asynchronous API. It is the entry-point for a wrapper or user, and contains a few data-handling functions and a set of functions from the BLAS API. The frontend is flexible with respect to the number, and composition, of backends, where it is up to the user to create and add backends. Once the frontend has one or more backends, it can start scheduling tasks.

Our data-handling functions transfer data between the frontend and the different backends. They create and maintain matrix objects identified by unique IDs. The IDs correspond to matrices previously created by `new_matrix`:

```
MatrixId new_matrix(int rows, int cols, void* data,
    Precision precision, Type type, StorageOrder order,
    Transpose transpose, Copy copy);
```

The first three arguments to `new_matrix` are the matrix dimensions and a pointer to the matrix data. Data allocation is left to the user. The next four arguments are BLAS specific; the data precision, the matrix type, the data storage order, and whether the matrix is transpose. The storage order is assumed to be column-major as currently required by CUBLAS. The final argument is whether the data should be copied to backend specific storage. Setting that the backend does not need to copy the matrix is used when the matrix is an output argument. This prevents the GPU backend from copying data that will be overwritten to the graphics card. The call to `new_matrix` creates a task, but does not schedule it; the task is only scheduled to a specific backend when it is a dependency of another task. The specific details for scheduling these tasks are outlined towards the end of this section.

As the tasks are executed asynchronously, and the user is unaware what backend performs the actual computation, the function `get_matrix` is supplied:

```
void get_matrix(MatrixId id, bool synchronous);
```

The arguments to `get_matrix` are the matrix ID and a *blocking* flag. The `get_matrix` function creates a task to retrieve the matrix from the backend, and the task is immediately scheduled to the backend that holds ownership of the matrix. If the blocking flag is set, the frontend waits until the task has been executed. The CPU backend simply ignores these tasks, as all its data already resides in CPU memory space. The GPU, however, must read the data back from the GPU to CPU memory if the GPU data has been altered. Using `get_matrix` without the blocking flag enables asynchronous read-back of data.

The last data-handling function offered by the frontend is `delete_matrix`, which creates a task that deallocates backend specific resources occupied by the matrix:

```
void delete_matrix(MatrixId id);
```

For the CPU backend this function does nothing, while the GPU backend frees GPU memory consumed by the matrix. The user is responsible for deallocation of the data pointer supplied to `new_matrix`.

The three functions described in the previous paragraphs enable the user to create, read and delete matrices with our asynchronous API. We will now describe how computational tasks are created, scheduled and executed using the Level 3 BLAS double precision general matrix multiply (dgemm) algorithm as an example. The CBLAS [14] function `dgemm` is defined as

```
void cblas_dgemm(const enum CBLAS_ORDER Order,
        const enum CBLAS_TRANSPOSE TransA,
        const enum CBLAS_TRANSPOSE TransB,
        const int M, const int N, const int K,
        const double alpha, const double *A, const int lda,
        const double *B, const int ldb,
        const double beta, double *C, const int ldc);
```

which computes

$$C \leftarrow \alpha \text{op}(A)\text{op}(B) + \beta C, \quad \text{op}(X) = \{X, X^T\},$$

using the three matrices A, B and C. Our frontend signature is slightly different,

```
void dgemm(int m, int n, int k,
        double alpha, MatrixId aId,
        MatrixId bId,
        double beta, MatrixId cId);
```

but performs the same computation. We have removed the `Order`, `TransA` and `TransB` arguments, as these are incorporated into the `new_matrix` function. We have also altered the way input and output matrices are sent to the function. Instead of sending raw data pointers, we send matrix IDs that correspond to matrix objects.

The implementation of `dgemm` in our frontend is quite simple. We start by looking up the three matrices identified by the matrix IDs. Then we create a `dgemm` task to hold the parameters and invoke `schedule` to schedule the task to a backend.

The `schedule` function has two tasks: first to group the incoming task with its dependencies into a cluster, and second to find the optimal backend to schedule this cluster to. The grouping is trivial, as all tasks have a dependency list. Selecting the optimal backend to schedule the cluster to, however, is nontrivial. We use two criteria to give an estimate of how good a candidate each backend is, and then schedule the cluster to the backend with the best score. The first criterion is based on the load for each backend, and the second on where dependent tasks have been scheduled.

Our load criterion tries to equalize the load of different backends. It consists of three parts: wall time spent computing, an estimate of wall time needed to process the current task queue, and an estimate of wall time needed to compute the incoming task. The time needed to process future tasks is estimated by multiplying the average time to compute a single task by the number of tasks in queue. This is a good approximation if the tasks require similar processing time, such as many matrix multiplications of equally sized matrices. There is one problem, however. When we start our frontend, we do not have enough data to calculate a valid average time needed to compute a single task. For a heterogeneous processing environment, the different backends can have orders of magnitude of difference in performance. This is where our automatic tuning comes into play. We have a pre-compilation step that benchmarks each backend to find a static average. We calculate this average by computing several matrix multiplications for a fixed matrix size for each backend. This gives a relationship between the processing power of the different backends. We incorporate this speed estimate into our source code by defining a C preprocessor macro, before recompiling the program. This constant estimate is used when the number of tasks processed by each backend is too low to give a proper run-time estimate.

The second criterion is computed by estimating the time it takes to move existing dependencies from another backend to the current backend. As you typically perform many operations on the same data, you will encounter situations where dependencies reside on different backends. Moving data between CPU backends is relatively inexpensive. Moving data between a CPU and a GPU, however, is very expensive, and should be penalized by this criterion. For each backend, we loop through the dependencies and estimate the time needed to move each dependency. If the dependency already resides on the current backend, the cost is zero. For dependencies on other backends, however, the cost is calculated as the estimated time needed to get the matrix from its current backend and transfer it to the new backend. The time needed to get the matrix requires a synchronous get, and is accordingly very expensive.

When the best backend has been selected based upon the above mentioned criteria, we synchronously get the dependencies that have to be moved. Then we reschedule them to the selected backend together with unscheduled dependencies, followed by the computational task itself. Continuing our `dgemm` example, this means that the `new_matrix` tasks for the matrices `a`, `b`, and `c` will be enqueued to the selected backend just before we enqueue the `dgemm` task.

Back-end

Each backend has two parts, as indicated by Figure 1. It consists of two threads of execution: the frontend thread and the backend thread. The frontend thread is shared by multiple backends, while each backend runs in a distinct thread of execution. Functionality for enqueuing tasks and querying load status is only called from the frontend thread, and functionality for dequeuing and executing tasks is only called from the backend thread. One can view this relationship between

the frontend thread and the different backend threads as a typical *boss-worker* relationship. The frontend acts as the boss, distributing tasks among the workers. In this section we use the term *boss* to signify the thread of execution that runs the frontend, and *worker* to signify the thread of execution that runs the backend selected by the scheduler for the current cluster.

When a task is scheduled to a specific worker, the boss simply adds the task to the task-queue of the selected worker. The boss then notifies the worker of a change, triggering the worker to check its status. While the queue is empty, the worker simply idles waiting for this notification. When notified, and a task has been added to the queue, the worker starts dequeuing and processing the tasks. For the dequeued task, it starts by identifying the type (`new_matrix`, `delete_matrix`, `dgemm`, etc.). When the task has been identified, the worker executes the corresponding function, and continues to process the rest of the queue.

In Section 4 we explained how the scheduler enqueued a cluster of tasks to a backend based on different criteria. For the dgemm example, the cluster contained the tasks to create the three matrices, `a`, `b`, and `c`, and the `dgemm` task that used these matrices. We continue the example by explaining how the backend processes and executes these tasks in its queue. Assuming that all four tasks have been added to the task-queue of the GPU backend, we start by dequeuing the first. It is the task to create matrix `a`. The GPU backend allocates GPU memory, and transfers the matrix data into the newly allocated memory. We follow the same procedure for matrix `b`, but for matrix `c` we only allocate memory, assuming we have set the copy flag to false when creating the task. We continue by dequeuing the dgemm task. This instructs us to call the CUBLAS dgemm function using the parameters and matrices defined by the task. When we have completed the dgemm task, we check the queue, and, if empty, wait for notification from the frontend.

MATLAB wrapper

We have implemented a MATLAB wrapper to offer a high-level interface to the asynchronous linear algebra API. The wrapper consists of two parts, one written in C++, and one written in the MATLAB language (M). The C++ part is very lightweight and thin, simply translating a MATLAB call into a call to one of the frontend functions. The M part defines a new class in MATLAB, and uses operator overloading for this class. Internally, this MATLAB class calls the C++ part.

MATLAB can execute user-defined MATLAB executable (MEX) files that are programmed using FORTRAN or C/C++. These MEX files can be called from MATLAB as if they were regular MATLAB functions. When a MEX file is called MATLAB executes the mexFunction, the entry point all MEX files must define:

```
void mexFunction(int nlhs, mxArray* plhs[],
                 int nrhs, const mxArray* prhs[]);
```

The arguments to the mexFunction are the number of left-hand arguments, pointers to the left-hand arguments, number of right-hand arguments, and pointers to the right-hand arguments, respectively. A general MATLAB call can be written as

```
[a, b, ..., y, z] = fun(α, β, ..., ψ, ω).
```

If `fun` is the name of a MEX file, `plhs` would contain the arguments `a` through `z` and `prhs` would contain arguments α through ω. In our wrapper, we use a single MEX file, where the mexFunction calls the correct frontend function based on the first argument. Our MATLAB function calls are on the format

```
matrix_id = async(funk_id, ...),
```

where `async` is the name of our MEX file, `matrix_id` corresponds to the internal ID used by our asynchronous API, and `func_id` determines what function the wrapper should call. The rest of the arguments are sent to the frontend function is specified by `func_id`.

When creating matrices, the C++ part of the wrapper automatically makes the memory allocated by MATLAB *persistent*. This prevents MATLAB for using automatic garbage collection, effectively giving us ownership of the data.

MATLAB has support for classes in its M script language. This enables us to call the MEX file in a more elegant way than explicitly calling `async` as indicated above. The M part of our wrapper consists of a new class, called *Matrix*, that uses operator overloading to offer an API with familiar MATLAB syntax.

Our operator overloading for the Matrix class translates operations and function calls involving a Matrix object into calls to `async`. Matrix multiplication, for example, would be defined in the file `@Matrix/mtimes.m` in the following way:

```
function c = mtimes(a, b)
c.id = async(42, a.id, b.id);
```

Here, 42 is the function id corresponding to dgemm, `a.id` is the id of matrix `a`, and likewise for `b.id`.

The following MATLAB code shows how this is used in practice to compute dgemm:

```
m = 10; n=20; k=30;
a = Matrix(rand(m, k));
b = Matrix(rand(k, n));
c = a*b;
c_data = double(c);
```

First, `a` and `b` are created as two Matrix objects that contain randomly generated data. Then the multiplication is run using operation overloading. Notice that we do not create the matrix `c`. The C++ part of the wrapper handles this by noticing that `c` has an invalid matrix ID, and thus creates a correctly shaped matrix for us. Finally, we convert `c` from a Matrix object back to double-precision data using the overloaded `double` function. This conversion actually performs a synchronous `get_matrix` that will inhibit performance. As our API is asynchronous we also offer a `read` function that corresponds to an asynchronous `get_matrix`. If there is enough time to read back the matrix before the double-precision data is required, we experience very fast data access.

5 Benchmarking and Analysis

The performance of our interface depends on two main factors; the efficiency of the underlying BLAS implementation and the time spent scheduling a task. The efficiency of several

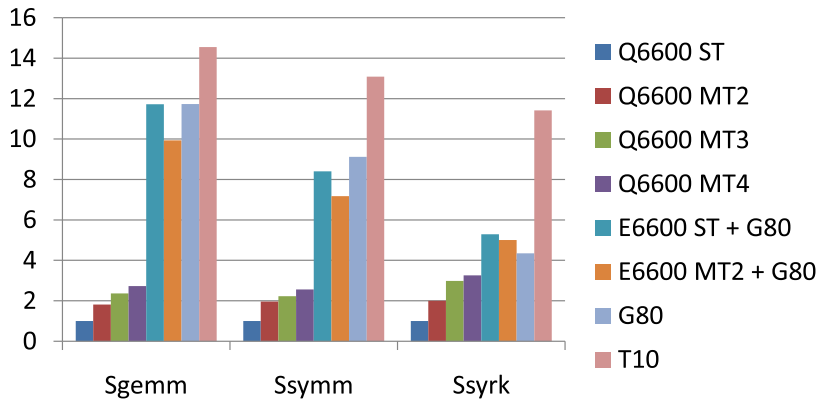


Figure 2: Speedup results from benchmarking single-precision general matrix multiplication (sgemm), symmetric matrix multiplication (ssymm), and rank-k symmetric matrix update (ssyrk). Please note that the T10 card is an early engineering sample (not full performance).

BLAS implementations has previously been analyzed and is not discussed here (see e.g., Barrachina et. al. [5] and the NCSA BLAS Performance Comparison). We show how our API behaves with different backend setups, and try to analyze the results. We further benchmark our scheduling algorithm, and the MATLAB wrapper of the API.

We utilize CUBLAS in our GPU backend. CUBLAS is part of the NVIDIA CUDA SDK, and implements single and double-precision BLAS functions using the GPU. It is steadily being developed, and provides a good interface to GPU accelerated BLAS functions. For our CPU back-ends, we use a single-threaded ATLAS implementation.

We have benchmarked our algorithms on two setups. The first setup consists of a 2.4GHz Intel Core 2 Q6600 quad-core processor with eight GiB of 12.8 GiB/s system memory, and an early engineering sample of the next generation NVIDIA Tesla T10 card (not full performance). The second system consists of a 2.4GHz Intel Core 2 E6600 dual-core processor with four GiB of 12.8 GiB/s system RAM, and an NVIDIA GeForce 8800 GTX graphics card (G80).

Our benchmark runs one thousand computations using 1000×1000 matrices, and timing results have been consistent throughout our benchmarking. The following is a pseudo code of the benchmark:

```
start timer;

for 1..10
  create input and output matrices;
  schedule computation on matrices;

for 11..990
  wait for result;
  delete matrices;
  create input and output matrices;
  schedule computation on matrices;
```

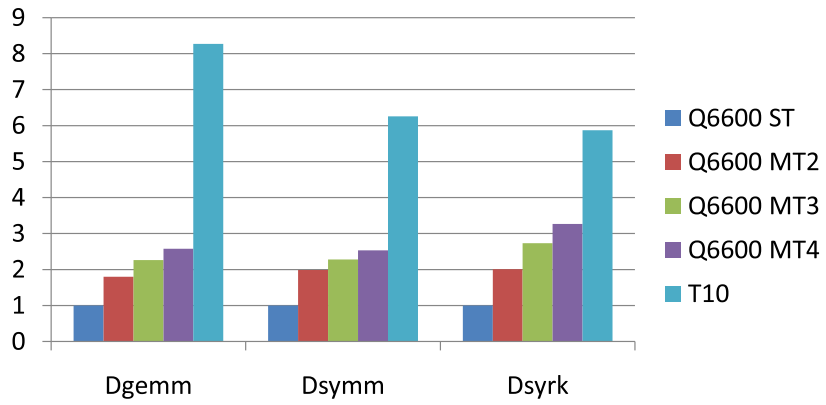


Figure 3: Speedup results from benchmarking double-precision general matrix multiplication (sgemm), symmetric matrix multiplication (ssymm), and rank-k symmetric matrix update (ssyrk). Please note that the T10 card is an early engineering sample (not full performance).

```
for 991..1000
  wait for result;
  delete matrices;

stop timer;
```

This is the worst case scenario, where we only run one computation per matrix. This severely affects the GPUs performance, as transferring data between CPU and GPU memory is expensive. In real world use, one would typically reuse data in multiple computations.

We have implemented and benchmarked three algorithms from BLAS in both single- and double-precision: general matrix multiply, symmetric matrix multiply, and rank-k symmetric matrix update. We have split the results into a single and double-precision part, as the G80 GPU does not support double-precision. Figure 2 shows the speedup of different backend setups compared to using a single CPU backend for single precision computations. Utilizing two CPU cores scales almost perfect for all algorithms. The average speedup is $1.92\times$. For three cores, however, we see a distinct difference between the algorithms. Ssyrk continues to scale almost perfect, with a speedup of 2.98, while sgemm and ssymm achieve a much smaller speedup. We believe this is because these two algorithms start trashing the cache. The Q6600 processor has a shared L2 cache of 4MiB per core pair. For two CPU backends, each backend can run on a separate core-pair with exclusive access to the 4MiB cache. For three backends, however, two of the backends must run on the same core-pair, having to share the cache. Ssyrk only uses two thirds of the memory sgemm and ssymm uses, and does not seem to be affected by having to share the processor cache.

When we have four backends running simultaneously, we do not have enough processing power for the scheduler thread to keep all back-ends occupied. Therefore, for four backends, none of the backends scale perfectly, and we do not get the expected speedup.

Figure 3 displays the gained speedups for double precision computations, showing similar results to our single-precision equivalents.

Utilizing the GPU backend outperforms using multiple CPU backends by far. This even

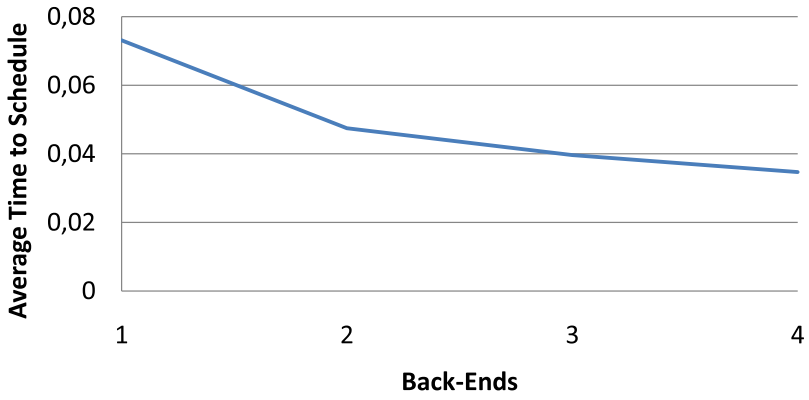


Figure 4: Average time to schedule a single task compared to the number of backends

though we transfer data to and from the GPU for each pass. Utilizing a GPU backend and CPU backend, however, does not scale as well as we could have hoped. We actually see a speed-down compared to using just the GPU for sgemm and ssymm. The reason for this is our scheduling algorithm. By examining the choices made by our scheduler, we find that it makes sub-optimal decisions when faced with a heterogeneous set of backends. We have currently not resolved this issue, but feel confident that we will remove it.

Figure 4 shows the average time spent scheduling a task compared to the number of backends. Between 0.04 and 0.08 seconds is a relatively long time. For small matrices this constant scheduling time will affect the experienced performance. Nevertheless, when the tasks we schedule take on the order of seconds to complete, this overhead is negligible. Our API is designed for computation on large matrices, making this overhead less important. However, we believe that optimizing the scheduling algorithm and front- and backend interaction will decrease this overhead.

Our MATLAB wrapper tries to be light-weight and add as little overhead as possible. Our benchmarks show that using our MATLAB wrapper only imposes a constant overhead. Calling our MEX file directly (without the M part of the wrapper) imposes an overhead of 0.3 milliseconds. Using the overloaded functions in the M wrapper is more expensive, as MATLAB must look up the correct function before the calls the C++ part of the wrapper are made. Nevertheless this only imposes an overhead of 2 milliseconds.

There has been skepticism in the scientific community towards using GPUs. It was only recently that GPUs started supporting single-precision arithmetics. These arithmetics, however, do not fully comply with the IEEE standard. Some minor parts such as denormals deviate from the standard, but the floating point implementation has been sufficient in practice for most uses. However, the lack of double-precision numbers has been criticized by the scientific community. Some algorithms require double, or even higher, precision. One example is numerical linear algebra, where even small floating point rounding errors can give large deviations in the results. Using double-precision improves the stability in this respect.

The NVIDIA Tesla T10 GPU supports double-precision in hardware. We do not give a thorough analysis of the double-precision implementation, but offer our early experiences with it. Figure 5 shows the absolute difference between CBLAS, CUBLAS, and a naïve (double for-

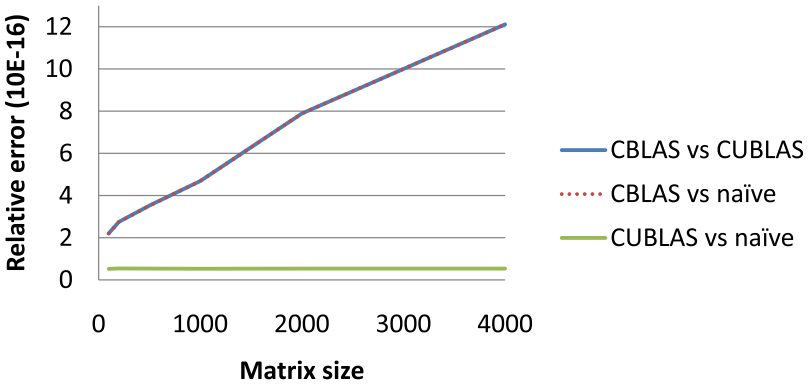


Figure 5: Measured relative difference per matrix element between CBLAS, CUBLAS and a naïve implementation of double-precision matrix multiplication.

loop) CPU implementation of double-precision general matrix multiply (dgemm). There seems to be a linear correlation between the matrix size and the difference between CUBLAS and CBLAS results. To put this into perspective, we also show the difference between CUBLAS and our naïve dgemm implementation. The difference per element between CUBLAS and our dgemm is constant at around $5 \cdot 10^{-17}$. This suggests that the CBLAS implementation reorders operations (e.g., using *Strassens algorithm*) giving different rounding errors than CUBLAS and the naïve algorithm. Reordering operations on the GPU in the same manner (if possible) should thus lessen the experienced difference between CUBLAS and CBLAS.

We will experience that results change between different runs using our API. This is because we do not know a priori where the results will be computed, and the fact that there is a small difference between CBLAS and CUBLAS results. For many uses, this is irrelevant, but for regression testing, for example, one must be aware of the possibility of changes in the result.

6 Conclusions and Future Work

We have presented a task-parallel asynchronous API for numerical linear algebra that imposes only small overheads. This API scales almost perfect to two CPU cores, and is capable of utilizing GPU processing resources for even higher performance. We have also demonstrated that the API can be used through high-level languages, such as MATLAB.

Our scheduling algorithm shows a proof-of-concept that works well for a homogeneous processing environment, but it is suboptimal for a heterogeneous composition of backends. It will be an interesting task to alter the scheduling algorithm to better utilize such heterogeneous processing platforms.

7 Acknowledgements

We would like to thank NVIDIA for the early engineering sample of the next generation Tesla T10 GPU.

Bibliography

[1] AMD CORPORATION, *Brook+ language specification, version 1.0 beta*, May 2008.

[2] E. ANDERSON, Z. BAI, C. BISCHOF, J. DEMMEL, J. DONGARRA, J. D. CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, S. OSTROUCHOV, AND D. SORENSEN, *LAPACK Users' Guide (third edition)*, Society for Industrial and Applied Mathematics, 1999.

[3] K. ASANOVIC, R. BODIK, B. C. CATANZARO, J. J. GEBIS, P. HUSBANDS, K. KEUTZER, D. A. PATTERSON, W. L. PLISHKER, J. SHALF, S. W. WILLIAMS, AND K. A. YELICK, *The landscape of parallel computing research: A view from Berkeley*, tech. report, EECS Department, University of California, Berkeley, 2006.

[4] M. BABOULIN, J. DONGARRA, AND S. TOMOV, *Some issues in dense linear algebra for multicore and special purpose architectures.*, tech. report, 2008.

[5] S. BARRACHINA, M. CASTILLO, F. D. IGUAL, R. MAYO, AND E. S. QUINTANA-ORTÍ, *Evaluation and tuning of the level 3 CUBLAS for graphics processors*, in Workshop on Parallel and Distributed Scientific and Engineering Computing, 2008.

[6] S. BARRACHINA, M. CASTILLO, F. D. IGUAL, R. MAYO, AND E. S. QUINTANA-ORTÍ, *GLAME@lab: An M-script API for linear algebra operations on graphics processors*, 2008.

[7] P. BIENTINESI, E. S. QUINTANA-ORTÍ, AND R. VAN DE GEIJN, *FLAME@lab: A farewell to indices*, tech. report, The University of Texas at Austin, Department of Computer Sciences, 2003. Draft.

[8] A. R. BRODTKORB, *The graphics processor as a mathematical coprocessor in MATLAB*, in CISIS 2008 The Second International Conference on Complex, Intelligent and Software Intensive Systems, 2008.

[9] A. R. BRODTKORB AND T. R. HAGEN, *A comparison of three commodity-level parallel architectures: Multi-core CPU, the Cell BE and the GPU*. Submitted to PARA'08 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing, May 2008.

[10] S. BROWN AND J. ROSE, *Architecture of FPGAs and CPLDs: A tutorial*, IEEE Design and Test of Computers, 13 (1996), pp. 42–57.

[11] I. BUCK, T. FOLEY, D. HORN, J. SUGERMAN, K. FATAHALIAN, M. HOUSTON, AND P. HANRAHAN, *Brook for GPUs: stream computing on graphics hardware*, in SIGGRAPH '04, ACM Press, 2004, pp. 777–786.

[12] D. R. BUTENHOF, *Programming with POSIX Threads*, Addison-Wesley, 1997.

[13] A. BUTTARI, J. LANGOU, J. KURZAK, AND J. DONGARRA, *A class of parallel tiled linear algebra algorithms for multicore architectures (lapack working note 191)*, tech. report, Innovative Computing Laboratory, 2008. http://icl.cs.utk.edu/plasma/.

[14] J. DONGARRA, *Basic Linear Algebra Subprograms Technical forum standard*, International Journal of High Performance Applications and Supercomputing, 16 (2002), pp. 1–111.

[15] M. FATICA AND W.-K. JEONG, *Accelerating MATLAB with CUDA*, 2007.

[16] J. A. GUNNELS, F. G. GUSTAVSON, G. M. HENRY, AND R. A. VAN DE GEIJN, *FLAME: Formal Linear Algebra Methods Environment*, ACM Transactions on Mathematical Software, 27 (2001), pp. 422–455.

[17] IBM, SONY, AND TOSHIBA, *Cell Broadband Engine programming handbook version 1.1*, April 2007.

[18] E. LINDHOLM, J. NICKOLLS, S. OBERMAN, AND J. MONTRYM, *NVIDIA Tesla: A unified graphics and computing architecture*, IEEE Micro, 28 (2008), pp. 39–55.

[19] M. D. MCCOOL, *Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform*, November 2006. GSPx Multicore Applications Conference.

[20] MICROSOFT CORPORATION, *MSDN DirectX developer center*.

[21] NVIDIA CORPORATION, *CUDA CUBLAS library version 1.1*, September 2007.

[22] ——, *CUDA CUFFT library version 1.1*, October 2007.

[23] ——, *NVIDIA CUDA programming guide version 1.1*, November 2007.

[24] OPENMP ARCHITECTURE REVIEW BOARD, *OpenMP application program interface version 3.0*, May 2008.

[25] J. D. OWENS, M. HOUSTON, D. LUEBKE, S. GREEN, J. E. STONE, AND J. C. PHILLIPS, *GPU computing*, Proceedings of the IEEE, 96 (2008), pp. 879–899.

[26] RAPIDMIND, *Cell BE porting and tuning with RapidMind: A case study*. Online. http://www.rapidmind.net/pdfs/RapidMindCellPorting.pdf.

[27] J. REINDERS, *Intel Threading Building Blocks:Outfitting C++ for Multi-core Processor Parallelism*, O'Reilly Media, Inc, 2007.

[28] D. SHREINER, M. WOO, J. NEIDER, AND T. DAVIS, *OpenGL Programming Guide*, Addison-Wesley, sixth ed., 2007.

[29] J. STRATTON, S. STONE, AND W. MEI HWU, *MCUDA: An efficient implementation of CUDA kernels on multi-cores*, tech. report, University of Illinois at Urbana-Champaign, 2008.

PAPER V

SIMULATION AND VISUALIZATION OF THE SAINT-VENANT SYSTEM USING GPUS

A. R. Brodtkorb, T. R. Hagen, K.-A. Lie, and J. R. Natvig

In Computing and Visualization in Science, Springer-Verlag Berlin Heidelberg, (special issue on Hot Topics in Computational Engineering), (2010). [in press]

Abstract: We consider three high-resolution schemes for computing shallow-water waves as described by the Saint-Venant system and discuss how to develop highly efficient implementations using graphical processing units (GPUs). The schemes are well-balanced for lake-at-rest problems, handle dry states, and support linear friction models. The first two schemes handle dry states by switching variables in the reconstruction step, so that bilinear reconstructions are computed using physical variables for small water depths and conserved variables elsewhere. In the third scheme, reconstructed slopes are modified in cells containing dry zones to ensure non-negative values at integration points. We discuss how single and double-precision arithmetics affect accuracy and efficiency, scalability and resource utilization for our implementations, and demonstrate that all three schemes map very well to current GPU hardware. We have also implemented direct and close-to-photo-realistic visualization of simulation results on the GPU, giving visual simulations with interactive speeds for reasonably-sized grids.

1 Introduction

Accurate simulations of shallow water waves as described by the Saint-Venant system are highly important in many application areas. Herein, our primary interest is flood simulation and interactive studies of multiple scenarios for flood prevention, for which a main challenge is simulation time: the computational power of standard CPUs cannot provide the performance needed for grids with sufficient resolution. In an effort to overcome this problem, we present efficient implementations of three high-resolution schemes for the Saint-Venant system [11, 7, 13] on graphical processing units (GPUs). Leveraging the computational power of a GPU can potentially provide close to real-time simulation and visualization, thereby significantly improving user interactivity. Our work contributes a state-of-the-art implementation of explicit finite-volume schemes on modern graphics hardware, including interactive visualization with photo-realistic effects. The particular schemes capture steady states like the lake-at-rest case, support dry states, and include simple source terms accounting for bottom friction. Verification against analytical solutions and validation against experimental data for the Malpasset dambreak are described in a separate paper [5].

Graphical processing units have in recent years developed from being hardware accelerators of computer graphics into high-performance computational engines. The use of GPUs in scientific computing has gone from early proof-of-concept studies around ten years ago (e.g., matrix operations carried out by using graphics operations [14]), to the current widespread use. Examples from a wide range of applications show how one can obtain a significant computational speedup by harnessing the computational power of a GPU [20]. A comprehensive description of current state-of-the-art GPU technology, including hardware, software, and algorithms, can be found in Brodtkorb et al. [4].

GPUs are stream processors that operate in parallel by running a single kernel on multiple instances of a data stream. This type of parallelization is particularly well suited for the stencil computations that constitute an explicit high-resolution scheme. The idea of using GPUs to accelerate (high-resolution) schemes for systems of conservation and balance laws is not new. To the best of our knowledge, it was first suggested in 2005 by Hagen et al. [7] for the Saint-Venant system and then later for the Euler equations of ideal gas dynamics [8]. Using OpenGL, the authors demonstrated how the stencil computations of several classical and high-resolution schemes could be implemented as operations in the fragment processing units, see [6]. Moreover, for systems of conservation laws, one could utilize the vector operations of four-component graphics (RGBA) to obtain acceleration beyond the number of parallel pipelines. Compared with a highly tuned CPU implementation, speedup factors in the range 15–30 were observed. Another important observation was that explicit schemes for hyperbolic conservation laws and balance laws are memory bound, and hence larger speedups were observed for high-resolution schemes that are more compute intensive than classical schemes like Lax–Friedrichs, Lax–Wendroff, etc. Since then, there have been several publications devoted to the use of GPUs for the shallow-water equations and other conservation and balance laws, see e.g., [2, 3, 22, 16, 10, 26, 15, 1].

In the current paper, we revisit the shallow-water simulations from [7], now using implementations in CUDA rather than OpenGL to give an up-to-date demonstration of the feasibility of GPU computing for the Saint-Venant system. The two main points in the paper are: (i)

a discussion of how to implement high-resolution schemes as efficiently as possible on current GPUs, and (ii) a comparison of the efficiency of GPU implementations of the Kurganov–Levy [11] and Kurganov–Petrova [13] schemes. In assessing computational efficiency, it has become quite popular to report speedup factors compared with a CPU implementation, and the literature is filled with optimistic figures that report several orders of magnitude speedups. Unfortunately, most of these findings are overly optimistic (and not examples of good science); by comparing theoretical performance numbers for GPUs and CPUs, it is easy to see that speedup factors exceeding 100 are very unlikely on current hardware. Herein, we will therefore instead consider the degree of resource utilization, which in our opinion is a better measure of how well a particular algorithm maps to the GPU.

2 Model Equations

Waves in shallow waters can be described by the following Saint-Venant system

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x - \kappa(h)u \\ -ghB_y - \kappa(h)v \end{bmatrix}. \tag{1}$$

Here h is the water depth, hu is the discharge along the x-axis, hv is the discharge along the y-axis, g is the gravitational constant, and B is the bathymetry (see Figure 1). On vector form, we can write the equation as

$$Q_t + F(Q)_x + G(Q)_y = H(Q, \nabla B), \tag{2}$$

where Q is the vector of conserved variables, F and G are flux functions, and H represents the source terms. The friction source term is assumed to be linear in velocity with a constant of proportionality that depends on h,

$$\kappa(h) = \frac{\alpha h}{1 + \beta h}. \tag{3}$$

For all synthetic test cases considered herein, α and β have been set, rather haphazardly, to 10^{-2} and 10^2, respectively.

3 Numerical Schemes

There are many aspects to consider when studying numerical methods for the Saint-Venant system. First of all, it is important for any numerical method to be conservative. Second, the method should be accurate on smooth parts of the solution and not create spurious oscillations

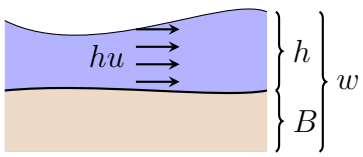


Figure 1: Variables in the shallow-water equations in one dimension: h is the water depth, B is the bathymetry, w is the total water elevation, and hu is the discharge.

near discontinuities or sharp transitions in the solution. Moreover, many simulations are perturbations of a steady state. Consider, for example, a lake at rest, in which the hydrostatic contributions to the flux in (1) perfectly balances the bathymetry gradient in the source term. An ideal method should therefore be well balanced in the sense that source terms and fluxes balance exactly also in the discretized equations for zero velocities.

Likewise, to simulate inundating (flooding), we require that the scheme does not break down in the presence of dry states ($h = 0$) and that it is well-behaved in *shoal* zones (h very small). Solving the Saint-Venant system numerically with dry states is difficult. To compute numerical fluxes, one will typically have to divide quantities by the water depth h. As h approaches zero, we get divisions by very small numbers, resulting in large errors in the fluxes. To make matters worse, if the water depth becomes negative, the whole computation breaks down since the eigenvalues of the system are $u \pm \sqrt{gh}$.

High-resolution schemes. There are many good schemes available in the literature that satisfy the criteria above. Herein, we are mainly interested in problems characterized by strong discontinuities, which typically can be satisfactorily resolved using a well-balanced second-order scheme with capabilities for resolving dry states. For other types of problems involving more smooth phenomena, e.g., the formation of eddies in shelf-slope jets [21], well-balanced schemes of higher order may be required [17]. In choosing among different second-order schemes, our previous experience is that the Kurganov–Levy scheme [11], and its slightly modified version reported in [7], offer a good compromise between simplicity of implementation and efficiency, accuracy, and robustness for the simulation scenarios considered herein. In addition, we consider an improved version developed by Kurganov and Petrova [13], which allows discontinuities in the bathymetry, contains less branching, requires less shared memory, and has been verified against both analytical and experimental data [24].

The three second-order, semi-discrete, central-difference schemes considered herein are based on the same basic discretization principles on a regular Cartesian mesh, using the generalized minmod flux limiter to obtain a high-resolution [9, 25] non-oscillatory reconstruction. We start by integrating (1) over each cell in the mesh to obtain a system of evolutionary equations for the cell averages Q_{ij} of the conserved quantities Q,

$$\begin{aligned}\frac{dQ_{ij}}{dt} = R(Q_{ij}) = H(Q_{ij}, \nabla B) - \big[F(Q_{i+1/2,j}) - F(Q_{i-1/2,j})\big] \\ - \big[G(Q_{i,j+1/2}) - G(Q_{i,j-1/2})\big].\end{aligned} \tag{4}$$

Here $F_{i,\pm 1/2,j}$ and $G_{i,j\pm 1/2}$ denote the fluxes over the cell interfaces in the x and y-directions. Then the temporal evolution of cell averages Q_{ij} in cell ij can be approximated using a second-order stability-preserving Runge–Kutta method,

$$\begin{aligned}Q^*_{ij} &= Q^n_{ij} + \Delta t R(Q^n_{ij}) \\ Q^{n+1}_{ij} &= \tfrac{1}{2} Q^n_{ij} + \tfrac{1}{2}\left[Q^*_{ij} + \Delta t R(Q^*_{ij})\right].\end{aligned} \tag{5}$$

The timestep in the Runge–Kutta solver is restricted by a CFL condition,

$$\Delta t \leq \tfrac{1}{4} \min\big\{\Delta x / \max_\Omega |u \pm \sqrt{gh}|, \Delta y / \max_\Omega |v \pm \sqrt{gh}|\big\} \tag{6}$$

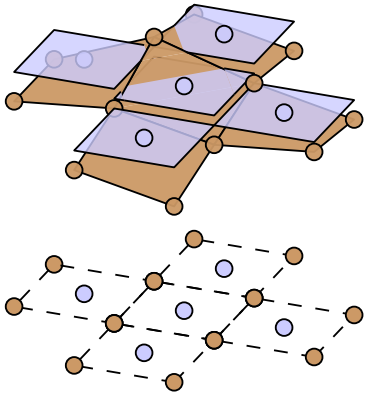


Figure 2: Reconstruction of surface elevation and bathymetry. For a bilinear reconstruction, the cell averages coincide with the values at the cell centers. The bathymetry is approximated by its values at the cell vertices.

that limits the propagation of waves to one quarter of a grid cell per timestep.

From (5), we see that we need to compute the flux and source terms twice for each timestep. To compute fluxes, we introduce a quadrature rule for the spatial integration over each cell interface and hence express each flux as a weighted sum of *point values* of the flux functions F or G. To compute these point values, we *reconstruct* a bilinear approximation of Q inside each grid cell. The slope in each spatial direction is computed as a nonlinear combination of the forward, central and backward differences using the cell averages $Q_{ij}, Q_{i\pm1,j}$ and $Q_{ij}, Q_{i,j\pm1}$, respectively. In each integration point, we hence obtain two one-sided point values, reconstructed in the two cells on opposite sides of the interface. These two values are combined through a *numerical flux function*; herein we use the central-upwind flux [12]. Finally, the source term can be computed by approximating ∇B from the bathymetry evaluated at the cell vertices, see Figure 2. The resulting stencil is obviously highly parallel, arithmetically intensive, and hence very suitable for GPUs [7].

Kurganov–Levy (KL02). To cope with the problem of dry zones, Kurganov and Levy [11] proposed to use a different reconstruction in shoal and wet zones. For the wet zones, they proposed to perform reconstruction and flux calculations based on the variables $U = [h+B, hu, hv]$ rather than on the conserved variables. By using special quadrature rules and discretizing the source term appropriately, reconstruction from these variables leads to a well-balanced and conservative scheme. However, the scheme does not guarantee a non-negative water depth h.

To guarantee non-negative values, they use another reconstruction based on the physical variables $W = [h, u, v]$ in the shoal zones given by $h < K$ for some small prescribed constant K. The resulting scheme is unfortunately not well-balanced and will cause global errors in conservation. Moreover, spurious waves can emerge initially in the shoal zones, but here the solution rapidly reaches a steady state, in which the fluxes balance the source terms. The spurious waves therefore only have a small effect on the global solution.

Modified Kurganov–Levy (KLL05). The Kurganov–Levy scheme (KL02) uses one integration point per cell interfaces to compute fluxes. This limits the scheme to second-order accuracy. Hagen et al. [7] therefore proposed a slightly modified scheme that uses a two-point interior Gaussian quadrature along each interface. This quadrature rule is accurate for recon-

structions up to fifth order and thus supports higher-order reconstructions, including the WENO reconstruction [23], as used in [6] for gas dynamics.

Kurganov–Petrova (KP07). Whilst the KL02 and KLL05 schemes avoid negative values for h by switching to physical variables, the Kurganov–Petrova scheme [13] is based on adjusting the reconstruction for cells where the value at the integration points will become negative. If the reconstructed slope creates negative values at the integration points, the steepness of the water slope is adjusted (reduced or increased) so that the negative value at the integration point becomes zero. This guarantees that all water depths used in the calculations are non-negative. However, very small water depths can still create large errors in the flux calculations. The KP07 scheme handles this by desingularizing the calculated velocity used in the flux calculation for shoal zones:

$$u = \frac{\sqrt{2}h(hu)}{\sqrt{h^4 + max(h^4, \epsilon)}}. \tag{7}$$

This slope fix will ultimately affect the fluxes in shoal zones and thus compromise the well-balanced property. However, as with the KL02 and KLL05 schemes, the fluxes rapidly balance the source term, and these spurious initial waves have small effects on the solution. One thing should be noted, though. The slope fix, and the corresponding errors introduced in the solution, depend strongly on the slope of the bathymetry. If the slope fix is triggered for a cell with a very steep bottom slope, one obtains a very steep water slope as well. Moreover, generalizations of this fix to higher-order reconstructions are difficult, if possible at all. This is because one can only alter the slope to guarantee non-negativeness at one point per cell interface, and higher-order reconstruction require more integration points.

The KL02 and KLL05 schemes assume that the bathymetry B is given as a continuous function sampled at the integration points. The KP07 scheme, on the other hand, assumes that the bathymetry is bilinear within each cell. Using this assumption, discontinuous bottom surfaces can be handled and approximated by a piecewise bilinear function.

4 Implementation

We have implemented our solver using C++ and NVIDIA CUDA [18], with heavy use of templates for both the CPU and GPU-parts of the code. We have grouped computations into a set of four kernels, as shown in Figure 3, to best suit the architecture of current GPUs and still fulfill the requirements of the algorithm.

Figure 3 illustrates the program flow of our implementation. We start by initializing the computational domain and data storage between kernels in ①. In total, we need sixteen buffers the size of the computational domain: three to hold U, three to hold Q, two to hold H, six to hold F and G, one to hold B at cell vertices, and one to hold B at the cell centers. We have precomputed B at both cell centers and vertices as a performance optimization: the flux kernel ③ needs the vertex values, whilst the Runge–Kutta kernel ⑤ requires the value at cell centers to ensure non-negative water depths. We also need a buffer to hold the eigenvalues computed in the flux kernel, ③, and used in the maximum Δt kernel, ④, to compute the timestep. This buffer, however, is much smaller than the other buffers, as will be explained later.

After allocating and initializing buffers, we enter the main simulation loop, which contains one or more Runge–Kutta substeps (③–⑥). For each substep, we start by reconstructing a piecewise planar function for each grid cell and evaluate the fluxes and source terms in ③.

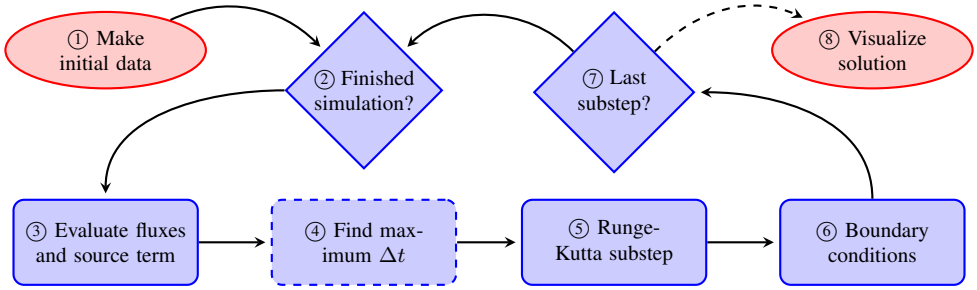


Figure 3: Program flow for the implemented schemes. Each of the four bottom boxes represents a computational kernel that executes on the GPU. The visualization is also performed directly on the GPU, without copying data over the PCI express bus.

For the first substep, we also compute the maximum eigenvalues in ③, and reduce them to the global maximum in ④. Then we can compute the maximum timestep satisfying the CFL condition (see (6)) and solve the ODEs (5) in ⑤. Finally, we can set the values of the global ghost cells to impose boundary conditions in ⑥. After the final substep, we may also opt to visualize the current solution.

The following describes in detail our implementation of the kernels used for the KP07 scheme. The other two schemes, KL02 and KLL05, use similar ideas and optimizations.

Block Decomposition

CUDA uses the concept of blocks to structure computation. Each block will execute independently of all other blocks and consists of a specified number of *threads*, the *block size*. Threads are organized in a logical 2D array, where threads belonging to the same block can communicate and cooperate using *shared memory*. Shared memory is an on-chip programmable cache on NVIDIA GPUs, accessible to threads within the same block. Its maximum size is dictated by the physically available memory on each *streaming multiprocessor*, currently 16 KB.

Our finite-volume scheme is in essence a set of complex stencil computations. The fluxes are computed using a neighborhood of four cells, as shown in Figure 4, and the source terms are similarly computed using three cells, shown in Figure 5. The Runge–Kutta substep kernel, shown in Figure 5d, uses the fluxes and source terms from the previous kernel to evolve the solution. This means that we need to use blocks with overlapping input domains for these kernels, as shown in Figures 6 and 7. Finding a good block size is vital for high performance, but determining what this block size should be is a difficult problem with many optimization parameters. We have used a block size of 16×14 for the flux kernel, and 16×16 for the Runge–Kutta kernel. For the KL02 and KLL05 schemes, our block sizes for the flux kernel are even smaller, as they use shared memory for both the physical and the conserved variables. For double precision, all block sizes are effectively cut in half. Our block sizes have been chosen through general optimization guidelines and experimentation, and the following is a rationale behind the choices.

One optimization parameter is shared-memory access. Shared memory is organized into 16 banks, where one thread can access each bank every other cycle. When multiple threads access the same bank (also called *bank conflicts*), their access is serialized. Thus, we should ensure that the block width is a multiple of 17 to avoid bank conflicts horizontally and vertically. We also

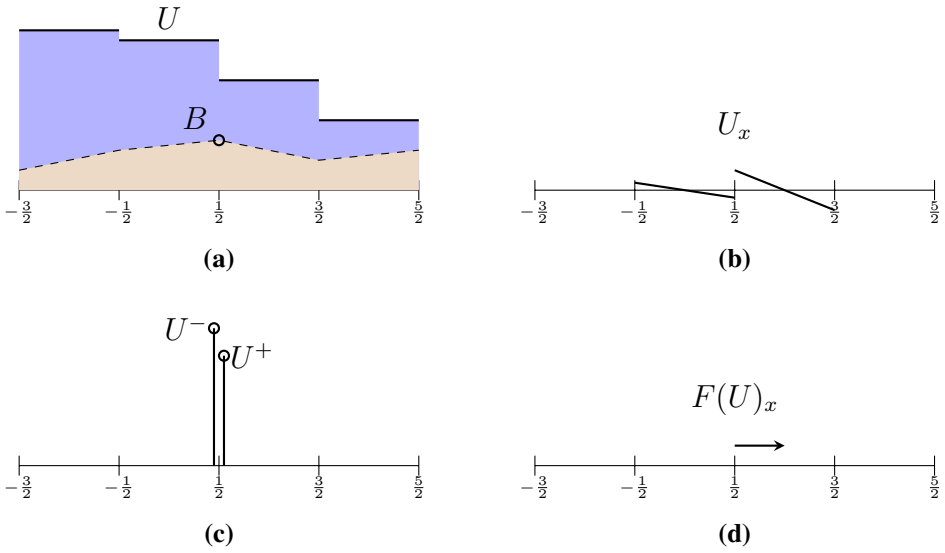


Figure 4: Data needed to compute the flux across the interface at $i+\frac{1}{2}$. In (a), we have the input water elevation and velocities, from which we reconstruct the slopes (b). We then evaluate the water elevation and velocities at the integration points from the right and left cell in (c). Finally, we compute the flux using the values at the integration points (d).

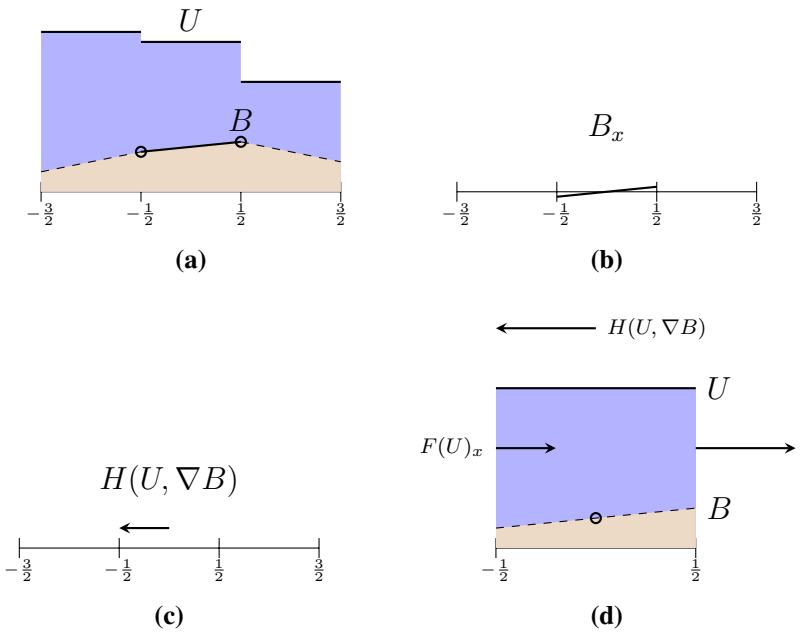
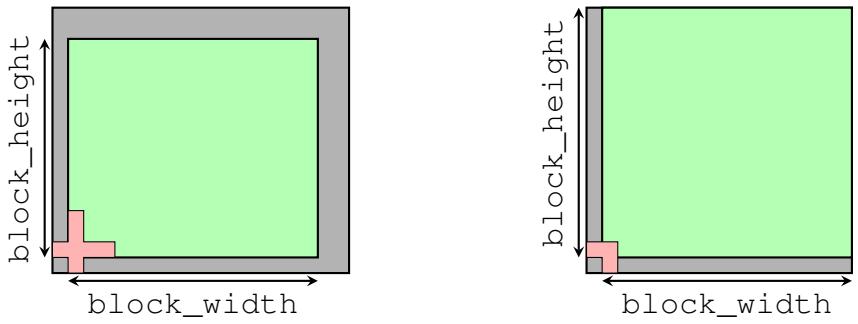


Figure 5: Data needed to compute the source term at cell i and data needed to perform time integration. In (a), we have the input water elevation from which we reconstruct the slope (b). We then evaluate the source term using the water elevation and bottom topography slope (c). The Runge–Kutta substep kernel (d) simply evolves and averages the solution to the next substep using the original water elevation, computed fluxes and source term, and the average water depth (computed by subtracting B). We also need the bottom elevation to make sure our evolved solution gives non-negative water depths.



(a) Flux and source terms.

(b) Runge–Kutta step.

Figure 6: Domain decomposition. In (a), we show a single block with ghost cells for the kernel computing flux and source terms. The seemingly asymmetric data dependency is because we compute the flux across the east and north interfaces of each cell, see Figure 4. In (b), we show a single block with ghost cells for the Runge–Kutta kernel.

want to maximize our use of shared memory, which means that we want the ratio of internal cells to ghost cells to be as high as possible for each block. We do this by aiming for a square block size, i.e., trying to equalize height and width. The block size of 16×14 gives a shared memory size of 19×17 which gives almost full use of shared memory, and the size is relatively square. This does not ensure that we have no bank conflicts, but it has been more important to ensure that the number of threads in the block is a multiple of 32, since the GPU executes *warps* of 32 threads in SIMD fashion.

To optimize memory access, we want to achieve *coalesced* reads and writes, which means that the width of data read into the kernel for each *warp* must be a multiple of 128 bytes, and that the starting address is aligned on a 128-byte boundary. Unfortunately, we cannot fulfill these two requirements at the same time for all blocks because our scheme requires overlapping blocks. To lessen the performance impact of slower global memory access, we use the texture cache to fetch data from global memory. It should be noted that the hardware used in our tests does not support double-precision texture fetches. Thus, we employ the standard technique of using an int2 representation during texture fetches, followed by reinterpreting the result as a double.

CUDA Kernels

Flux and Source Term (③). The kernel that computes flux and source terms is the main computational kernel in our solver. The kernel starts by reading data into shared memory, including the overlapping domain dictated by our stencils, shown in Figure 6a. Each thread (i, j) within each block is responsible for calculating the flux across the east $(i + \frac{1}{2}, j)$ and north $(i, j + \frac{1}{2})$ interface, in addition to the source term for cell (i, j). By examining the dependencies required by both the flux and the source term calculation, we see that we need one ghost cell to the south and west, and two ghost cells to the east and north.

The kernel begins by reading B and U into shared memory. From these, we reconstruct the slopes of U and calculate B at the integration points, totaling to twelve shared-memory variables. These variables are the ones needed by more than one thread. We compute the value

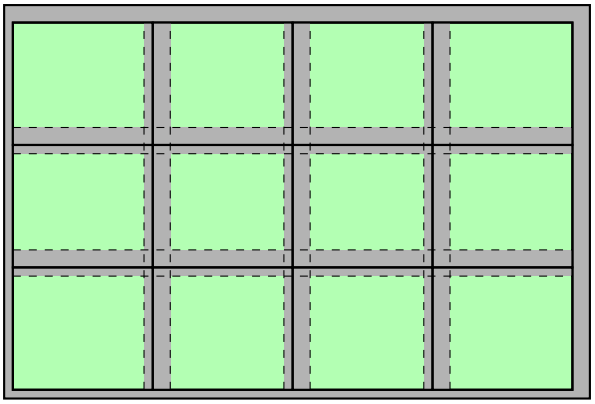


Figure 7: Grid decomposition for our flux kernel. Within each block (solid lines), we compute the source term for all cells and the flux across the north and east interfaces. The global ghost cells are used to implement boundary conditions. Notice that our computational domain covers one of the global ghost cells to bottom and left of the domain. This is because our kernel computes the flux across the north and east cell interfaces (see also Figure 6). Also notice that the blocks read overlapping data from the global domain to satisfy data dependencies dictated by our stencils.

of the bathymetry at the integration points in the kernel, as opposed to reading them from a precomputed buffer. This dramatically lessens the burden on the memory subsystem, and adds only a few extra computations. Reconstructing the slope of U is done using the branchless generalized minmod limiter [6], for which we use efficient bit operations to compute the sign function. To guarantee non-negative water depths at the integration points, we also correct the slopes for affected cells. This is done consistently in shared memory by choosing the slope that interpolates the bathymetry (at the negative integration point) and the average grid cell water elevation.

After reconstructing the slopes, we can evaluate U at the east and north integration points and compute the flux and the source term for the cell. If we are at the first Runge–Kutta substep, we also compute the eigenvalues and find the maximum within each block using reduction in shared memory. This is very efficient, as we reduce the number of elements the maximum Δt kernel ④ has to read by a factor $16 \times 14 = 224$ with our current block size. This also reduces the storage requirement, as mentioned in the beginning of this section.

Maximum Δt (④). The kernel that computes maximum Δt simply finds the maximum eigenvalue within the whole computational domain and computes the timestep Δt using (6). We use a single block, where each thread loops through a strided subset. This ensures coalescing of data reads, thus maximizing memory performance of this memory-bound kernel. Once all eigenvalues for each thread have been considered, we perform in-block reduction between threads using shared memory. Finally, one thread computes and writes the maximum timestep to global memory.

Runge–Kutta (⑤). The Runge–Kutta substep kernel computes one substep of the Runge–Kutta ODE integrator (5). It is a memory-bound kernel that performs few computations, but accesses global memory many times. First, we read the fluxes F and G into shared memory. We then read the source term H, the existing solution U (and Q^* for the second substep), the

bathymetry B, and finally the timestep Δt into registers for each thread. We then evolve the solution one substep. We also make sure all water elevations are non-negative, as floating-point round-off errors might cause negative water elevations.

Boundary Conditions (⑥). This kernel is quite similar to the maximum Δt kernel ④. The kernel is memory bound, as it performs very few computations. We only launch one block, which simply fills the ghost cells at the boundary with appropriate values. In our case, we have implemented wall conditions, i.e., we copy the cells closest to the boundary to the ghost cells and change the sign of the perpendicular velocity component. As an alternative to using a separate kernel to set boundary conditions, we could have used an extra buffer to identify boundary cells. This, however, would dramatically increase the load on the memory bus.

Other Optimizations. We need to pass a large amount of parameters to each of the kernels outlined above. For 32-bit operating systems, we can pass them in the normal fashion. However, for 64-bit systems, the size of pointers double, and exceed the maximum size allowed by CUDA. We thus use constant memory, which is auto-coalesced and cached global memory on the GPU. This enabled us to pass the parameters on 64-bit systems, and further proved to be a significant performance boost on 32-bit systems.

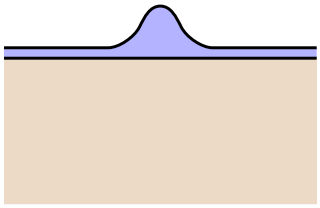
Visualization

The purpose of visualization is to improve our understanding of, and extract information from the simulation results. Hence, what variables to chose and what visualization techniques to use will strongly depend on what features of the solution on is interested in. Herein, we focus on producing a birds-eye view of the water surface and the surrounding terrain. To this end, we have implemented direct visualization of simulation results in OpenGL [19] with photo-realistic effects, as shown in Figure 8d. First, we render the terrain using a quadrilateral mesh where the nodes are displaced according to the height of the bathymetry B. The mesh is then draped with a texture and we use Phong shading (a method for calculating light reflected from surfaces by interpolating surface normals across rasterized polygons) to compute per-pixel lighting. The water surface is also rendered using a quadrilateral mesh and displaced according to the water elevation w. We use the Fresnel equations to compute the angle of reflected and refracted rays, and the amount refraction. The reflected ray is then used to look up into the environment map (the *skybox*), and our refracted ray is used to look up into the terrain texture. Environment mapping combined with reflection is a very good tool to spot discrepancies in the simulation, as our eyes rapidly detect imperfections in the mirror-like surface.

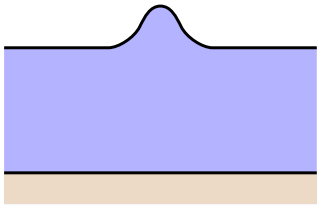
We visualize the bathymetry given at the center of each grid cell as a piecewise bilinear function, and the same is done for the cell averages of the water elevation. This means that the visualization is slightly erroneous and can give rise to visual artifacts where the water depth approaches zero (see along the shore in Figure 8d). However, we find it a reasonable approximation that gives users a good overview of the simulation results.

For each timestep to be visualized, we start by copying simulation results from CUDA memory to OpenGL texture memory. We accomplish this by copying from CUDA simulation memory to a CUDA mapped pointer of an OpenGL pixel-buffer object. Then, we synchronize

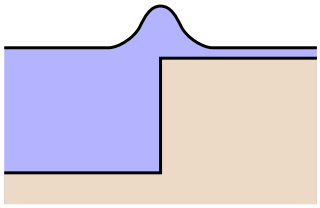
Skyboxes are used in computer graphics to create the illusion that the displayed scene is larger than it actually is. The rendered schene is embedded inside a box, and images of the sky and the distant landscapes are projected on the faces of the box to illude the unreachable 3D space surrounding the scene.



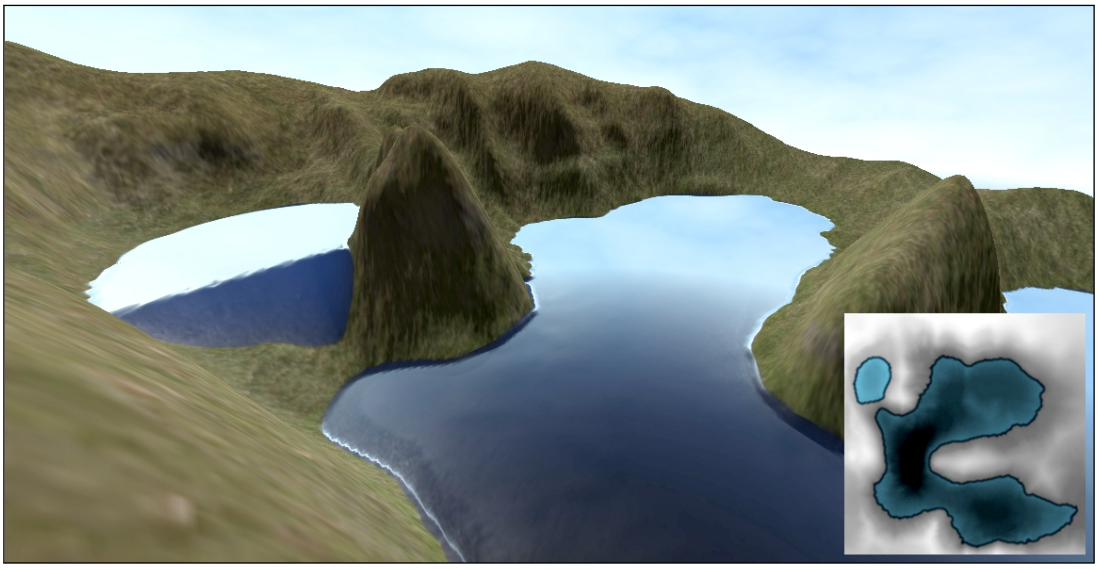
(a) Case 1a: low water elevation.



(b) Case 1b: high water elevation.



(c) Case 2: discontinuous bathymetry.



(d) Case 3: dambreak simulation.

Figure 8: The different test cases used in our benchmarks. For Cases 1a, 1b and 2, we have a 2D domain with a bell-shaped water elevation at the center of the domain. Case 3 is an synthetic terrain with a breaking dam. The height map is superimposed on the image.

the pixel-buffer object with an OpenGL texture. One optimization would be to remove the first of these copies and instead run the simulation using a CUDA mapped pointer to an OpenGL pixel-buffer object directly. However, this would increase the code complexity of the simulator and also couple the simulation code with the visualization code. The second copy might also seem superfluous, but is mandatory in current driver versions and is very efficient: it is performed entirely on the GPU, without the need to transfer data over the PCI express bus.

5 Numerical Experiments

To assess the performance of our implementation of the three schemes (KL02, KLL05, and KP07), we consider four different cases, see Figure 8. Case 1 consists of a bell-shaped water elevation over a flat bathymetry. In *Case 1a*, the water elevation is set very low so that the schemes interpret the solution to be in the shoal zone, in which dry-state reconstruction is triggered for KL02 and KLL05, and desingularized flux computation (7) is triggered for KP07. In *Case 1b*, the water elevation is so high that the entire solution is in the wet zone. *Case 2* has the same setup, but now with a discontinuous bathymetry; this to illustrate the difference between the KL02/KLL05 and KP07 schemes. Finally, *Case 3* consists of a synthetic bathymetry that defines a “dambreak” simulation in which a high-altitude dam floods an underlying valley and

lake terrain, creating a combination of wet regions, shoal regions, and dry states.

Float vs. Double Precision

Double-precision arithmetics has so far not been supported very well on GPUs, and when available, has come with a big performance penalty. Hence, it is advantageous if the high-resolution schemes can rely solely on single-precisions arithmetics. We therefore start by investigating how using single-precision influences the accuracy of our schemes. To this end, we consider the relative errors in mass conservation,

$$E^c = \frac{\int_\Omega h^0 \, dx - \int_\Omega h^n \, dx}{\int_\Omega h^0 \, dx}, \tag{8}$$

where h^0 is the initial water depth and h^n is the water depth at timestep n (using a fixed timestep). Figure 9 reports this error for single-precision (SP) and double-precision (DP) versions of the Kurganov–Petrova scheme (KP07). Likewise, we report the absolute discrepancy between the two solutions for each timestep. It is interesting to note that single precision gives round-off errors that violate conservation of water, both for low and high water elevations, even for flat bottom topographies. The absolute discrepancy between the two solutions is also growing for Cases 1a, 1b, and 2. The two Kurganov–Levy schemes exhibit the exact same behavior and plots are not included.

When dry states and varying bathymetry is included (Case 3), we see that error increases significantly and that the overall error of the scheme dominates the effects of single versus double precision. The increase in error comes from the switching in the Kurganov–Levy schemes and the slope fix in the Kurganov–Petrova scheme. Hence, we conclude that when the schemes are used for the type of problems they were designed for with shoal zones and dry states, the error induced by floating-point precision is negligible (as originally stated in [7], based on CPU simulations in single and double precision). It is interesting to see that the added integration points in the modified Kurganov–Levy seems to negatively affect the conservation. Our explanation to this is that the modified scheme performs twice the number of flux evaluations, and should thus experiences more round-off errors.

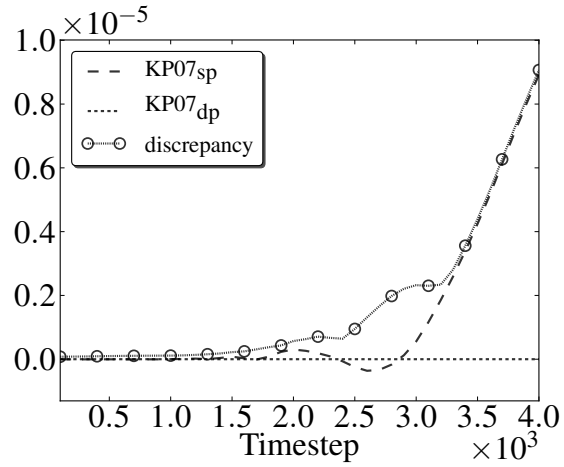
We have further verified our wall boundary conditions by checking their effect on conservation. The boundary conditions did not affect the conservation for the wet-bed test cases.

Discontinuous Bathymetry

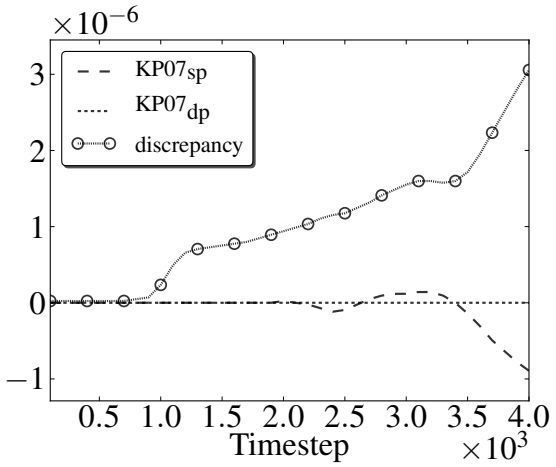
The Kurganov–Levy schemes assume a continuous bathymetry, and a straightforward sampling of a discontinuous bathymetry, as in Case 2, will result in very steep gradients in the bathymetry approximation, which in turn will effect the CFL number and drive the stable timesteps toward zero. This effect is illustrated in Figure 10: when the wave reaches the discontinuity in the bathymetry, the timestep in the Kurganov–Levy schemes decays to zero and even after 5000 timesteps, the simulation is still at time $t \approx 23$. The Kurganov–Petrova scheme, on the other hand, propagates the wave with nearly unaffected timesteps past the discontinuity.

Efficiency

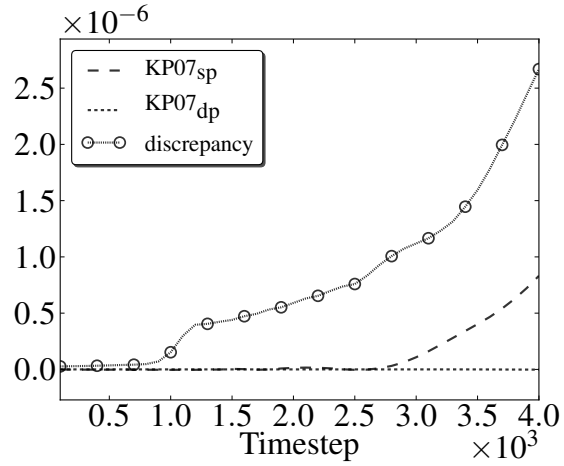
Ever since the first applications on GPUs were published, there has been a trend to report speedups over the CPU. At the time when [7, 8, 6] was written, general-purpose computation on



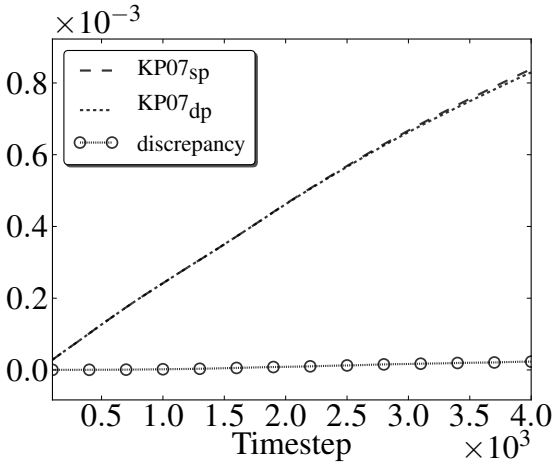
(a) Case 1a: low water elevation.



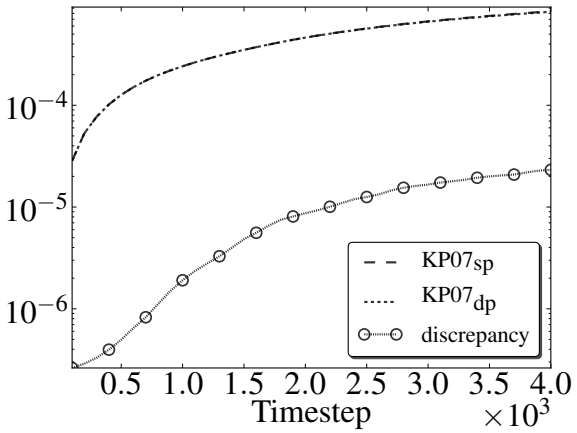
(b) Case 1b: high water elevation.



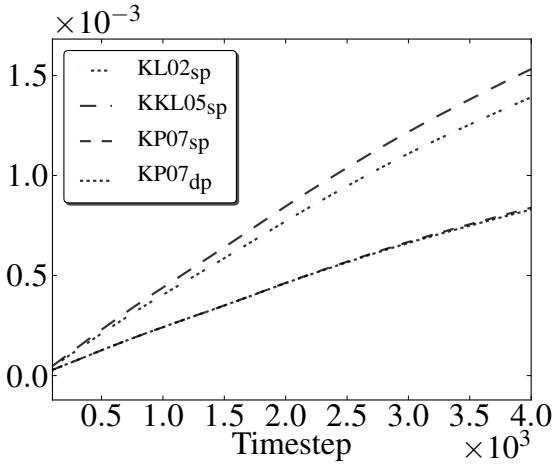
(c) Case 2: discontinuous bathymetry.



(d) Case 3: dambreak simulation.



(e) Case 3: dambreak, semilog scale.



(f) Case 3: dambreak simulation.

Figure 9: Comparison of single-precision and double-precision versions of the KP07 scheme on a 1024×1024 grid. In (a) to (e), relative errors in mass conservation (E^c in (8)) are shown together with the discrepancy between single and double-precision simulations. Subplot (f) shows the error for all three schemes.

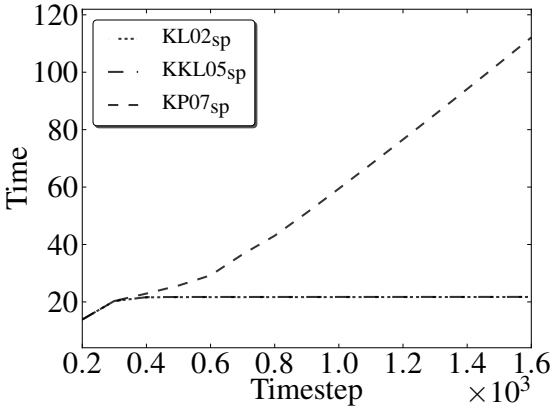


Figure 10: Plot of time versus number of timesteps for Case 2.

graphics processing units (GPGPU) was still in its infancy, and an important statement was to demonstrate that it was possible to use GPUs for general-purpose computation, and to convince the reader that a GPU code could be much faster than a corresponding CPU code. Gigaflops and execution-time metrics have been used extensively, and speedup factors between 2 to 200 are commonly found in articles, still today. However, by examining the theoretical performance numbers for GPUs and CPUs, one quickly realizes that a speedup of over 100 seems unlikely on current hardware. Typically, these figures emerge from comparing an unoptimized (or even worse, claimed to be optimized) CPU code to a highly-tuned GPU implementation. There are cases, e.g., for algorithms dominated by expensive trigonometric computations, where the use of highly efficient, albeit less accurate, hardware implementations found on the GPU can give a speedup larger than what one would expect by only comparing memory speed, clock frequencies, and number of arithmetic units. For most algorithms, however, these high speedups are not attainable. Our view is that quoting such high speedups has had its mission and is now destructive to the reputation of GPU computing.

We would like to see less of these speedup figures and more figures that show efficient hardware utilization and scalability. Reporting utilization of hardware resources will thus give the user an idea of what to expect, not only on current hardware, but also for future hardware generations. We have measured the efficiency of our implementations on the NVIDIA GeForce GTX 285 using the CUDA Visual Profiler supplied with the CUDA SDK. We have used the profiler to give us detailed statistics over runtime, memory bandwidth utilization, and instruction throughput. The profiler has also been actively used during development, to find and remove bottlenecks.

The flux and source term kernel is the most time-consuming of our kernels. For the KP07 scheme, this kernel takes up 72% of the GPU runtime. It utilizes only 17.6% of peak bandwidth for global store because we violate coalescing rules. For global load, our original version had an efficiency of 11%. However, when using texture fetches, we saw a large performance boost. The kernel has an instruction throughput of 80%, which means we idle 20% of the time. We have profiled this kernel also on other GPUs with less bandwidth relative to compute power, and the instruction throughput has remained at 80%. Our experiments and benchmarks thus indicate that the idling is caused by data dependencies and instruction latencies, and not due to waiting on data from global memory. Thus, we conclude that the flux and source term kernel is

compute bound for the benchmarked GPUs.

For the KL02 and KLL05 schemes, our flux kernel is also the most time consuming, with 82% and 84% of the total runtime, respectively. These two kernels have a memory efficiency equivalent to the KP07 kernel, as they perform the same memory operations. However, the instruction throughput is at a mere 66%. This can be explained by the number of threads we are able to schedule to each multiprocessor. Because we are limited by shared-memory usage, we have fewer threads for these two kernels, meaning we do not have as many other threads to run whilst we wait for data. This is often referred to as the occupancy, and our occupancy is 22% for the KP07 scheme, yet only 12.5% for the KL02 and KLL05 schemes.

The Runge–Kutta substep kernel is our second most time-consuming kernel, with 28%, 18%, and 16% of the GPU runtime for KP07, KL02 and KLL05, respectively. This kernel is heavily memory bound, and penalized for violating the coalescing rules. We achieve 15% global store efficiency, and 11% global load efficiency. However, as with the flux and source term kernel, we use textures for most reads, which again showed to give a substantial performance gain. This kernel has also an instruction throughput of 80%. However, when profiling on GPUs with less relative bandwidth, we see that the instruction throughput is proportional to the bandwidth. Thus, we conclude that the Runge–Kutta kernel is memory bound.

The rest of the GPU time is spent, in decreasing order, on copying data to the GPU, running the maximum Δt kernel, and downloading data to the CPU. As can be seen from the previously presented numbers, these operations are negligible for the GPU runtime. However, they do impose a software overhead. The upload of data is done for each kernel, as we need to upload the parameters to constant memory before each kernel invocation. It should be possible to make this a GPU–GPU copy instead of a CPU–GPU copy, but we have not pursued this option because of the little time it takes. The maximum Δt kernel has a memory efficiency of 20% for store, and 40% for load. The kernel obeys all coalescing rules, but is penalized because of the very few items it considers. Finally, the download to the CPU is to keep track of the global simulation time, as the maximum Δt kernel places the result in GPU memory.

Performance and Scalability.

Figure 11 shows the performance of our schemes for Case 3 on the NVIDIA GeForce GTX 285. The most important grid sizes are those larger than or equal to 512^2, as this is where we best utilize the hardware. For this grid size, the KP07 implementation is able to run at 387 iterations per second in floating-point precision, and 45 in double precision, i.e., a factor 8.6 slower. The cause for this massive speed-down is that the benchmark GPU only has one double-precision unit per streaming multiprocessor, but eight single-precision units. Further, the double-precision implementation also imposes other overheads, such as more register space and a need for handling texture fetches in a special way. Our double-precision numbers for other grid sizes are similar.

When we increase the workload by four we would expect the number of iterations per second to decrease by four as well. However, going from 512^2 to 1024^2, we get more than one fourth of the performance for single precision. This is partly caused by the need to pad our domain to fit an integer number of blocks, which has a larger impact on the smaller domain sizes. Going from 1024^2 to 2048^2, this penalty becomes negligible, and we attain almost perfect weak scaling, also reflected in the double-precision results. For larger sizes, however, we quickly run

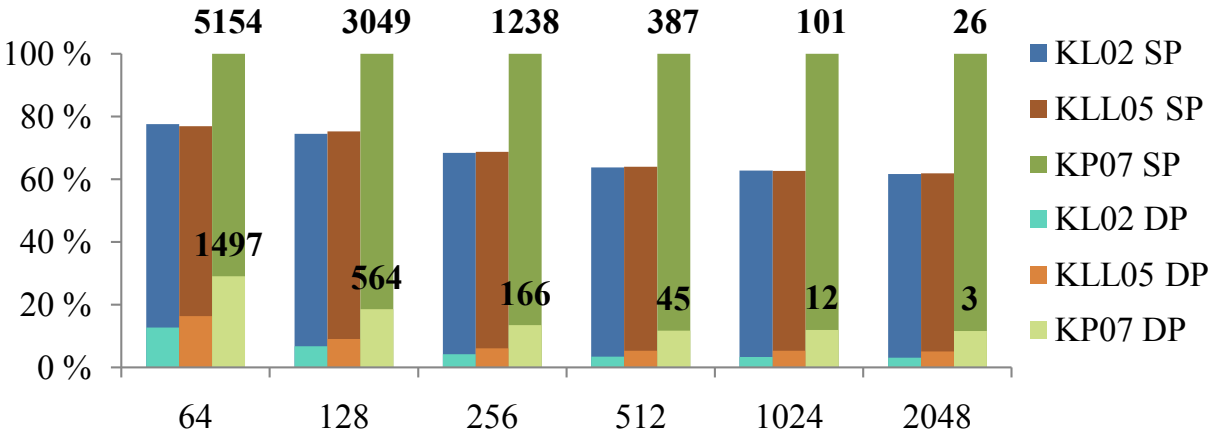


Figure 11: Performance of the different schemes for Case 3 on grids with $n \times n$ cells running on the NVIDIA GeForce GTX 285. From left to right, the three columns for each domain size display the performance of KL02, KLL05, and KP07, respectively. The graph is normalized relative to single-precision KP07. The figures in bold over each column indicates the number of timesteps for our KP07 implementation in single precision. The superimposed columns are for the same schemes in double precision, and the figures in bold indicate the number of timesteps for the KP07 implementation.

out of graphics memory, as our benchmark machine is limited to 1 GB. The maximum simulation size we have been able to run is a 3900×3900 grid, which consumes around 928 MB of graphics memory.

The main reason that the KP07 scheme is faster than KL02 and KLL05 is that the latter needs to reconstruct both from the positivity-preserving *and* the conserved variables, as we do not know, a priori, whether a cell is in the shoal zone or not. This double reconstruction increases the amount of computations dramatically. Furthermore, the KP07 scheme consumes far less shared memory in the flux and source-term kernel. The shared-memory use dictates the maximum block size, and for larger block sizes, the relative size of the overlapping ghost-cell regions goes down, lessening both the number of computations and the burden on the memory subsystem.

Finally, it is interesting to note that there seems to be no performance impact for the added integration points for the KLL05 scheme, which enables the use of higher-order reconstructions up to degree five. Our explanation to this is that the Kurganov–Levy based kernels are memory bound so that we can add more computations without severely affecting performance.

Visualization

Our simulator can run both with and without visualization. When visualization is enabled, we visualize every fifteenth timestep. This, unfortunately, has a negative impact on the simulator performance, as both the simulator and visualizer use the same hardware resources. To make matters worse, there is an overhead connected with mapping and unmapping OpenGL memory for use with CUDA, in addition to overheads related to the context switch between CUDA and OpenGL. For the 512^2 grid of Case 3, we achieve 300 timesteps per second using the KP07 scheme, which is a 22% drop from the 389 timesteps per second we get without visualization. Nevertheless, this translates directly to an interactive 20 frames per second. Several exam-

ples of the visualization can be found on YouTube: `http://www.youtube.com/user/babrodtk`.

6 Summary

In this paper, we have presented how shallow-water waves, as described by the Saint-Venant system, can be computed efficiently on graphical processing units using three different well-balanced, high-resolution schemes. By implementing direct visualization on the GPU, including various photo-realistic effects, we have developed a visual and interactive simulator.

Current GPU hardware is much more efficient when using single rather than double-precision arithmetics. For simple computational setups with no transitions between wet and shoal zones, round-off errors introduced by single-precision arithmetics cause lack of mass conservation and a significant deviation from the corresponding double-precision solution. However, for more complex cases that contain transitions between wet and shoal zones and/or between shoal and dry zones, the effect of single-precision arithmetics is masked by errors inherent in the schemes' treatment of dry zones. Hence, single-precision arithmetics can mostly likely be used for the typical complex cases the schemes were developed to handle. Preliminary experiments also indicate that use of mixed-precision arithmetics can be a way out to preserve both high accuracy *and* efficiency for single-zone cases.

Of the three schemes considered, the Kurganov–Petrova (KP07) scheme is our method of choice. This scheme has the best resource utilization of current GPU architectures and is hence more efficient, has a better treatment of dry states, and can handle discontinuities in the bathymetry. On the other hand, the treatment of dry states in KP07 only applies to bilinear reconstructions, and hence the scheme cannot be extended to higher spatial order, which may be important when studying smooth effects like eddies and other smooth phenomena.

Our implementations show relatively high utilization of computational resources and memory transfer. Still, there is room for further improvement. Increased memory throughput can for example be achieved by using Morton order for texture fetches. We also anticipate that increases in shared-memory size and a new cache, as in the new Fermi architecture from NVIDIA, will give a significant performance boost. Likewise, the performance of our visualization is likely to benefit from new functionality in CUDA 3.0 Beta for more efficient sharing of data between CUDA and OpenGL.

Our initial interest in simulating shallow-water waves on GPUs was to use high-resolution schemes as an excellent demonstrator of GPU capabilities and to provide a use case of interactive visualization. Lately, however, our interest has moved more towards full-featured shallow-water simulation: realistic dambreak scenarios, storm surges, etc. Verification, validation, and further algorithmic and implementational improvements are described in [5]. Moreover, we have developed a multi-GPU cluster implementation that shows (nearly) perfect scaling; further details will be provided in an upcoming paper.

Acknowledgements. The authors gratefully acknowledge financial support from the Research Council of Norway under grants number 180023/S10 and 186947/I30 and the Center of Mathematics for Applications, University of Oslo. The authors are also grateful for the continued support from NVIDIA.

Bibliography

[1] de la Asunción, M., Mantas, J.M., Castro, M.J.: Simulation of one-layer shallow water systems on multicore and CUDA architectures. J. Supercomput. (2010)

[2] Brandvik, T., Pullan, G.: Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware. IMechE Proc C: J. Mech. Engng. Sci. (2007). DOI 10.1243/09544062JMES813FT

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

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

[5] Brodtkorb, A.R., Sætra, M.L., Altinakar, M.: Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. In preparation (2010)

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

[7] Hagen, T., Hjelmervik, J., Lie, K.A., Natvig, J., Henriksen, M.: Visual simulation of shallow-water waves. Simul. Model. Pract. Theory **13**(8), 716–726 (2005)

[8] Hagen, T.R., Lie, K.A., Natvig, J.R.: Solving the Euler equations on graphics processing units. In: Proceedings of the 6th International Conference on Computational Science—ICCS 2006, *Lect. Notes Comp. Sci.*, vol. 3994, pp. 220–227. Springer Verlag, Berlin/Heidelberg (2006)

[9] Harten, A.: High resolution schemes for hyperbolic conservation laws. Journal of Computational Physics **49**(3), 357–393 (1983)

[10] Klöckner, A., Warburton, T., Bridge, J., Hesthaven, J.: Nodal discontinuous Galerkin methods on graphics processors. J. Comp. Phys. **228**(21), 7863–7882 (2009). DOI 10.1016/j.jcp.2009.06.041

[11] Kurganov, A., Levy, D.: Central-upwind schemes for the Saint-Venant system. Mathematical Modelling and Numerical Analysis **36**, 397–425 (2002)

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

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

[14] Larsen, E., McAllister, D.: Fast matrix multiplies using graphics hardware. In: Supercomputing, pp. 55–55. ACM, New York, NY, USA (2001). DOI http://doi.acm.org/10.1145/582034.582089

[15] Lastra, M., Mantas, J.M., na, C.U., Castro, M.J., García-Rodríguez, J.A.: Simulation of shallow-water systems using graphics processing units. Math. Comput. Simulat. **80**(3), 598 – 618 (2009). DOI 10.1016/j.matcom.2009.09.012

[16] Liang, W.Y., Hsieh, T.J., Satria, M., Chang, Y.L., Fang, J.P., Chen, C.C., Han, C.C.: A GPU-based simulation of tsunami propagation and inundation. In: Algorithms and Architectures for Parallel Processing, *Lect. Notes Comp. Sci.*, vol. 5574, pp. 593–603. Springer Verlag, Berlin/Heidelberg (2009). DOI 10.1007/978-3-642-03095-6_56

[17] Natvig, J.R., Noelle, S., Pankratz, N., Puppo, G.: Well-balanced finite volume schemes of arbitrary order of accuracy for shallow water flows. J. Comput. Phys. **13**(2), 474–499 (2006)

[18] NVIDIA: NVIDIA CUDA reference manual 2.3 (2009)

[19] OpenGL ARB, Shreiner, D., Woo, M., Neider, J., Davis, T.: OpenGL Programming Guide: The Official Guide to Learning OpenGL, 6th edition edn. Addison-Wesley (2007)

[20] Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J.: GPU computing. Proceedings of the IEEE **96**(5), 879–899 (2008). DOI 10.1109/JPROC.2008.917757

[21] Pankratz, N., Natvig, J.R., Gjevik, B., Noelle, S.: High-order well-balanced finite-volume schemes for barotropic flows. development and numerical comparisons. Ocean Modelling **18**(1), 53–79 (2007)

[22] Phillips, E.H., Zhang, Y., Davis, R.L., Owens, J.D.: Rapid aerodynamic performance prediction on a cluster of graphics processing units. In: Proceedings of the 47th AIAA Aerospace Sciences Meeting, AIAA 2009-565 (2009)

[23] Shu, C.W.: Essentially non-oscillatory and weighted essentially non-oscillatory schemes for hyperbolic conservation laws. In: Advanced numerical approximation of nonlinear hyperbolic equations (Cetraro, 1997), *Lecture Notes in Math.*, vol. 1697, pp. 325–432. Springer, Berlin (1998)

[24] Singh, J., Altinakar, M., Ding, Y.: 2D numerical model for shallow transient free surface flows over natural terrain. In: Proceedings of the International Conference on Hydroscience and Engineering. **Accepted for publication** (2010)

[25] Sweby, P.K.: High resolution schemes using flux limiters for hyperbolic conservation laws. Siam Journal of Numerical Analysis **21**(5), 995–1011 (1984)

[26] Wang, P., Abel, T., Kaehler, R.: Adaptive mesh fluid simulations on GPU. New Astron. **In press**, – (2009). DOI 10.1016/j.newast.2009.10.002

PAPER VI

EFFICIENT SHALLOW WATER SIMULATIONS ON GPUS: IMPLEMENTATION, VISUALIZATION, VERIFICATION, AND VALIDATION

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

Submitted, 2010

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 com-

plex 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

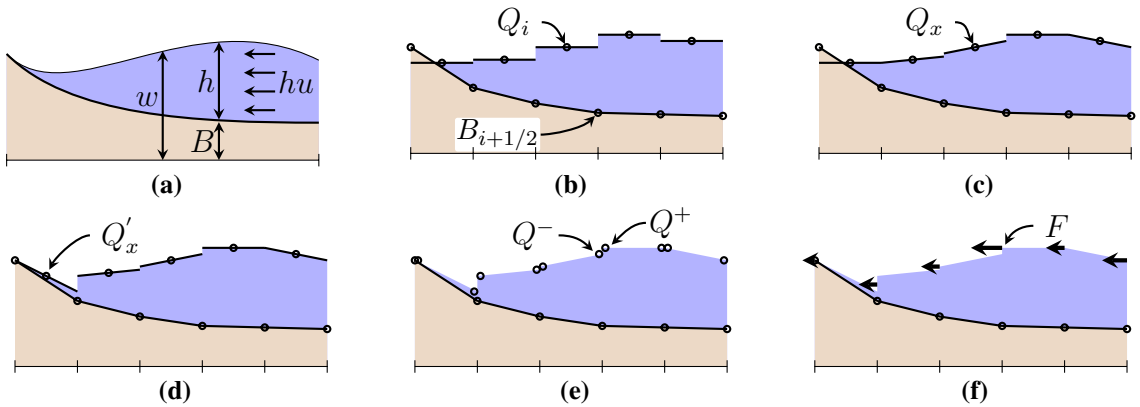


Figure 1: (a) Shallow water flow over a complex bottom topography and definition of variables; (b) conserved variables Q are discretized as cell averages and the bottom topography, B, is represented as a piecewise bilinear function in each cell based on its values at the grid cell intersections; (c) 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

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

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

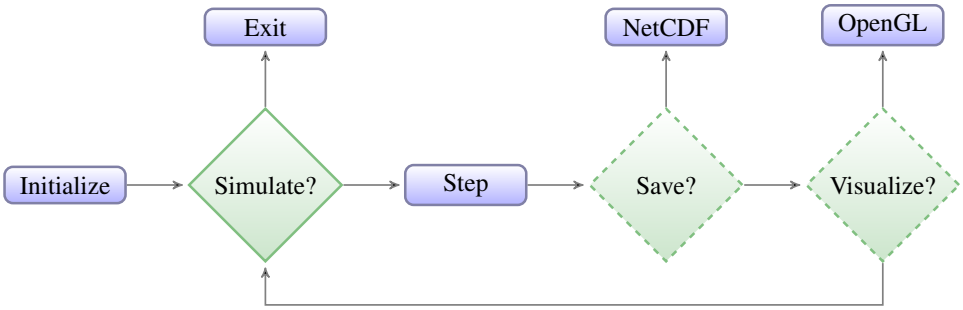


Figure 2: The program flow of our applications. Initialize sets up the simulator class with initial conditions, and step runs one timestep on the GPU. We can also save results in netCDF files and visualize them directly using OpenGL.

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,

$$\Delta t \leq \tfrac{1}{4}\min\left\{\Delta x/\max_\Omega\left|u \pm \sqrt{gh}\right|, \Delta y/\max_\Omega\left|v \pm \sqrt{gh}\right|\right\} = \frac{r}{4}. \tag{8}$$

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

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

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

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

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.

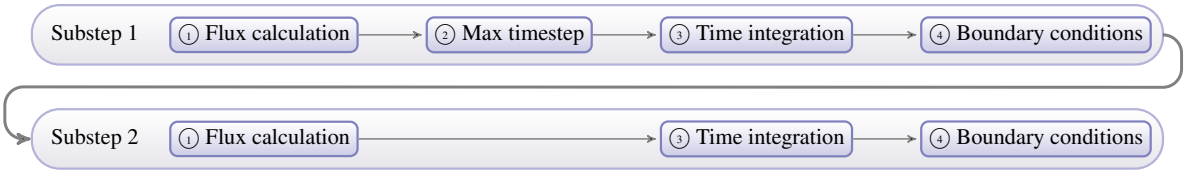


Figure 3: Zoom of the step function in Figure 2 that shows our CUDA kernels. In the first substep, ① calculates R, ② finds the maximum Δt, ③ calculates $\tilde{H}_f$ and Q^*, and ④ enforces boundary conditions on Q^*.

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

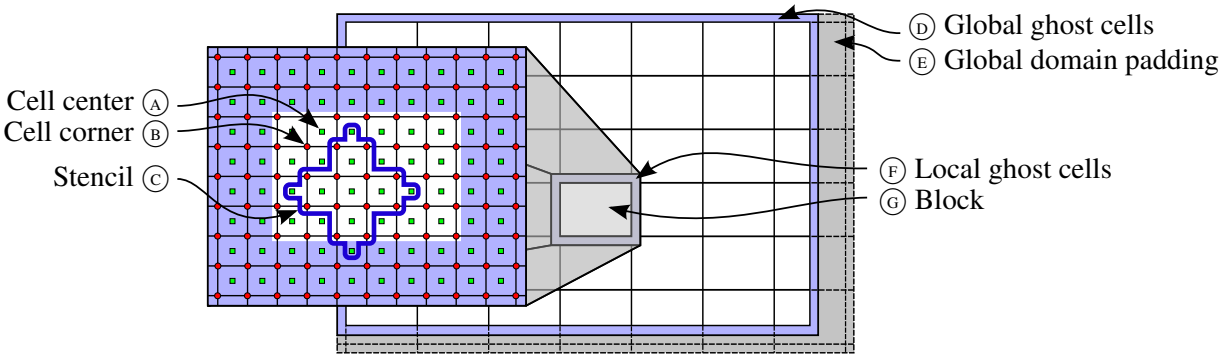


Figure 4: Domain decomposition and variable locations. The global domain is padded Ⓔ to fit an integer number of blocks. Each block Ⓖ has local ghost cells Ⓕ that overlap with other blocks to satisfy the data dependencies dictated by the stencil Ⓒ. Our data variables Q, R, H_B, and H_f are given at grid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

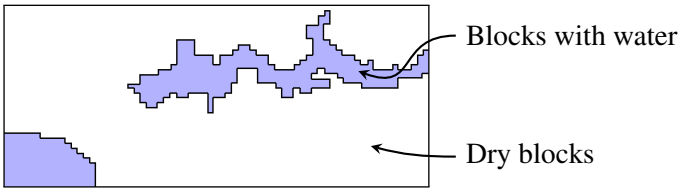


Figure 5: Auxiliary buffer used to store whether blocks contain water or not. The presence of one single wet cell in a block is sufficient to consider the whole block as wet.

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

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

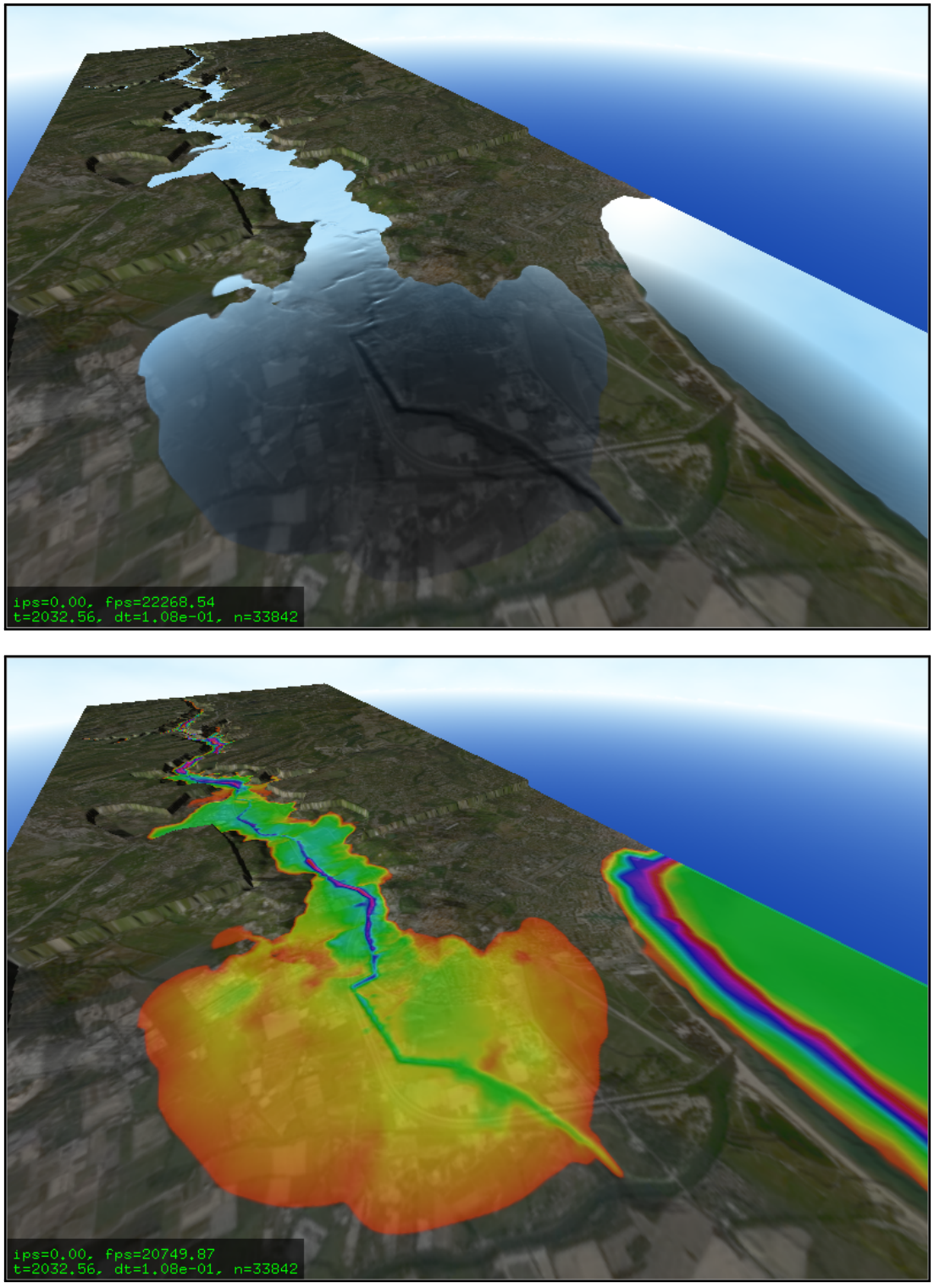
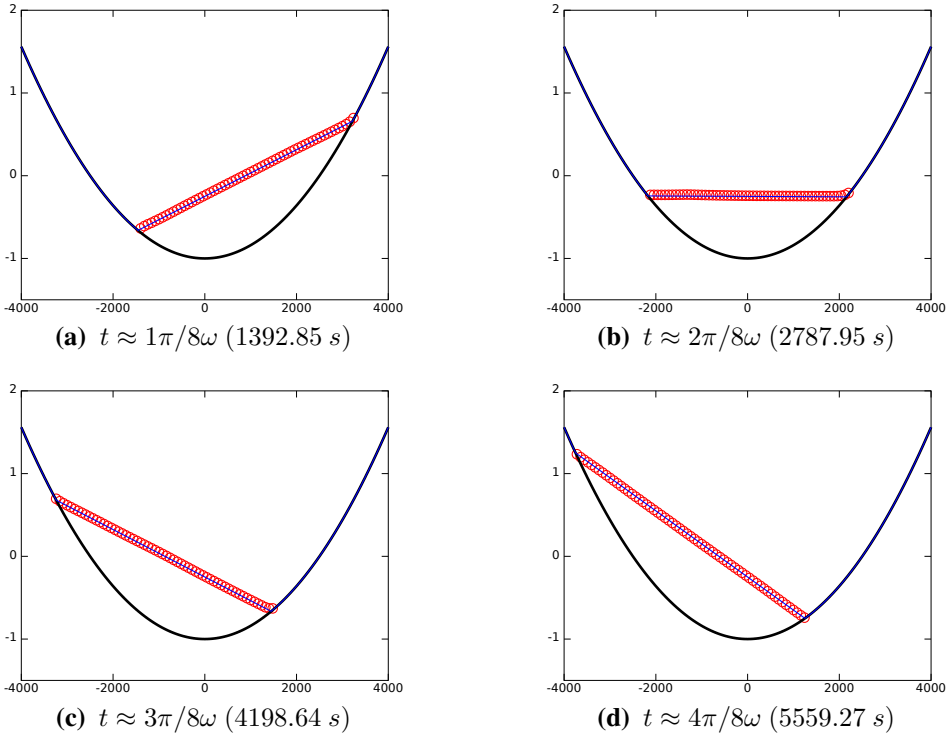


Figure 6: Two visualizations of the Malpasset dam break at $t = 2032.56$ seconds after the breach. We can visualize the results using the Fresnel equations, yielding a water surface with both reflection and refraction (left). We can also visualizes the physical variables using a color transfer function,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



(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

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

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

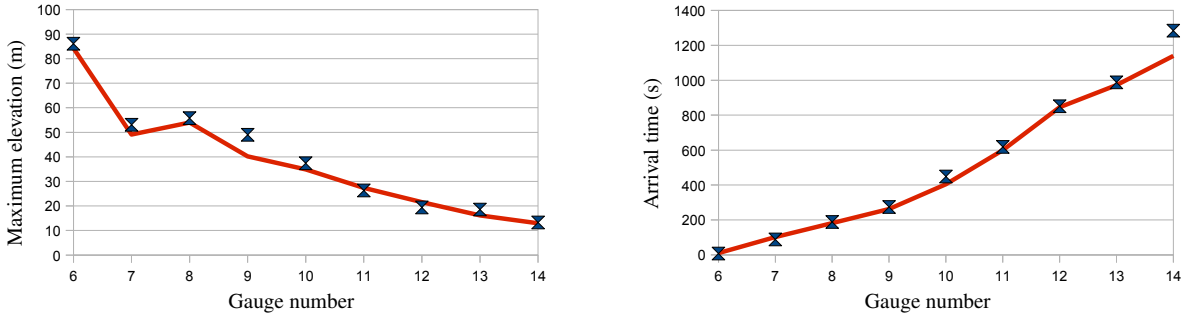
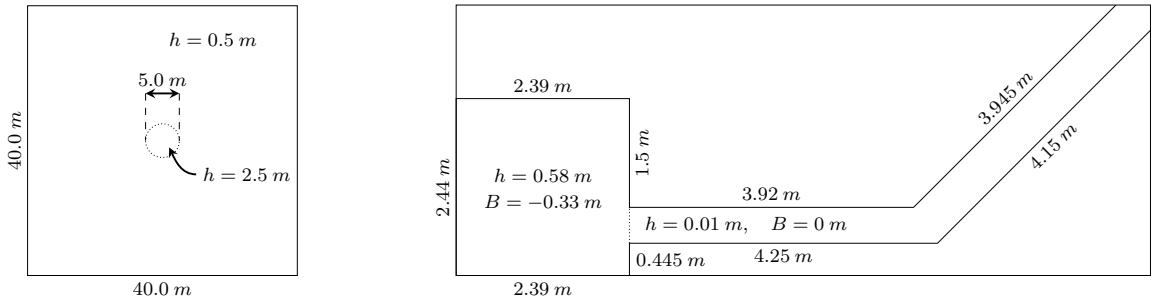


Figure 8: Validation against the maximum water elevation (left) and arrival time (right) from the Malpasset dam break case. The domain consists of 1099×439 cells with $\Delta x = \Delta y = 15$ meters. For both figures, the solid line represents the experimental data, and the symbols are the simulation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

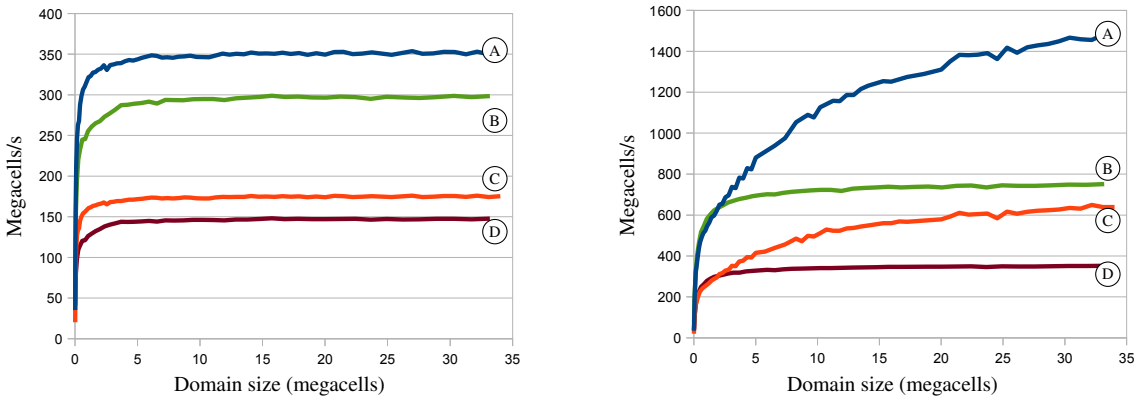


Figure 10: Absolute performance of our implementation as a function of domain size when calculating all cells (left), and when only calculating "wet" cells (right). Ⓐ and Ⓒ correspond to the dam break through 45° bend, and Ⓑ and Ⓓ correspond the to circular dam break case. Ⓐ and Ⓑ use first order accurate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

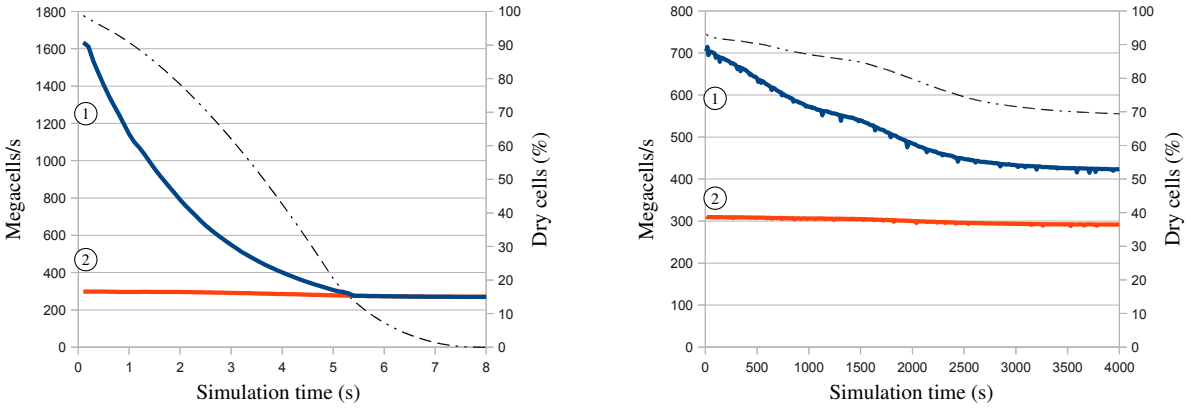


Figure 11: Absolute performance in megacells per second of our implementation for the circular dam break case (left) and the Malpasset dam break case (right). The domains contain 4000×4000 and 1099×439 cells respectively. The graphs show the estimated instantaneous performance as a function of simulation 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.