Solving Hyperbolic PDEs using Accelerator Architectures

by

Scott Rostrup

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Applied Mathematics

Waterloo, Ontario, Canada, 2009

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis investigates using highly parallel accelerator architectures to accelerate the simulation of nonlinear hyperbolic PDEs. Second-order accurate finite volume flux-limiter methods are implemented for the shallow water equations using cartesian grids and explicit time-stepping. The solver is implemented for three different architectures, a multicore x86 CPU using threading, IBM's Cell Processor, and Nvidia's Tesla GPU. The memory layout, communication patterns and optimization steps necessary to exploit the parallel architecture of the Cell processor and the GPU are described.

When comparing performance, the optimized accelerator implementations show speed-ups of between 60-75× relative to a single CPU core in single precision, with the Cell processor implementation reaching 60× speed-up and the GPU 75×. In double the Cell processor performs 18× faster than a single CPU core. Comparing with a quad-core CPU, the Cell processor accelerates computation speed by roughly 15×, while the GPU provides close to 19× speed-up. In double precision the Cell processor provides a 5× speed-up relative to the quad-core CPU.

The three implementations are extended to parallel computing clusters by making use of the Message Passing Interface (MPI). The resulting hybrid-parallel code is investigated for performance and scalability on two computing clusters. The first, a GPU cluster, shows excellent scalability and performance, while the Cell cluster is hampered by a slow interconnect speed and does not scale as well.

This study investigates the feasibility of adopting accelerator architectures for structured grid simulation of hyperbolic conservation laws, and the performance gains that can be obtained. These performance gains are quantified in a manner that can provide important information to others considering accelerator architectures. Additionally the study explains data-structures and techniques suitable for highly parallel architectures and discusses implementations for both the Cell processor and the GPU.

Acknowledgements

This work was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET:www.sharcnet.ca). I thank Hugh Merz and other members of the SHARCNET technical staff for their expert advice and technical help.
I also thank Markus Stuermer from the University of Erlangen-Nuernberg, who provided extensive and insightful information concerning the Cell processor.
Of course I thank my advisor Hans De Sterck for his advice and guidance throughout this long and winding project.
I also want to thank my family for the constant support and encouragement.
And all of the friends I have met here in Waterloo that have made my time here enjoyable and memorable.

Dedication

This is dedicated to the monster, for never being at the end of the book.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recent microprocessor advances have focused on increasing parallelism rather than frequency, resulting in the development of highly parallel architectures such as the GPU (Graphics Processing Unit) and the Cell processor [7]. Their potential for excellent performance on computation-intensive scientific applications coupled with their availability as commodity hardware has led researchers to adapt computational kernels to these parallel architectures, often referred to as accelerator architectures. This thesis investigates mapping high-resolution finite volume methods for hyperbolic equations [29] onto several classes of accelerator architecture, namely IBM's Cell processor and Nvidia GPUs [33]. These accelerator architectures are investigated as both stand-alone computational accelerators and as components of parallel clusters. This is accomplished by implementing a high-resolution explicit scheme for the shallow water equations on structured grids for three architectures (x86 CPU, GPU, and Cell), and in parallel using MPI (Message Passing Interface).
This thesis is organized as follows:

- Chapter 1 provides motivation and a review of related work.
- Chapter 2 introduces high-resolution finite volume methods.
- Chapters 3, 4, and 5 introduce architecture specific implementations of the shallow water solver for x86 CPU, GPU, and Cell.
- Chapter 6 compares the relative performance of the implementations.
- Chapter 7 discusses our findings and summarizes results.

1.1 Space Weather

The broader public is typically unaware of the possible dangers posed by space weather, but they are well documented [27]. They include danger of radiation exposure to astronauts and airline passengers, damage to satellite infrastructure, and as detailed in a recent NASA report [42], the potential for catastrophic damage to power and telecommunication infrastructure. Timely prediction of space weather events would ensure that proper precautions could be taken such as limiting critical infrastructure operations to only essential functions, overriding automatic control procedures or delaying operations until the event has subsided.

Modeling the interaction between the Earth's magnetic field and the solar wind can be done using the hyperbolic MHD (Magnetohydrodynamics) system of equations. Modeling MHD is the most computationally expensive component in a simulation of the Earth's space environment [18]. The class of high-resolution numerical schemes investigated in this thesis is suitable for modeling MHD flows [29]. Future acceleration of these types of calculations is the underlying motivation for this thesis.

1.2 Research Methodology

Wave-based high-resolution finite volume methods [29, 30] are described in Chapter 2. They are implemented in two dimensions (2D) on structured grids and applied to the shallow water equations (see Chapter 2). After verifying the accuracy of the implementation against analytic test cases, the implementation is extended to parallel and accelerator architectures (for distinction see section 1.3.6). First, coarse-grained parallelism is exploited to implement the algorithm at a cluster level using MPI. Second, thread-level parallelism is exploited on shared memory architectures using POSIX threads (pthreads). Thereafter, the implementation diverges into three separate architecture-specific implementations focused on optimizing the computational throughput by exploiting fine-grained parallelism on the three architectures of interest:

- Intel Xeon x86 CPU (Chapter 3).
- Nvidia Tesla GPU (Chapter 4).
- IBM Cell Broadband Engine Architecture (CBEA or Cell BE) (Chapter 5).

All three implementations are highly optimized for performance and multiple metrics are used in order to gauge their performance characteristics. This thesis will measure performance by comparing:

- Runtime speed-up against a single-core x86 CPU implementation.
- GFLOPS performance with the peak theoretical throughput.
- Runtime scaling speed-up on MPI clusters.

Investigating these measures helps give an accurate picture of the performance characteristics for the hyperbolic PDE computational kernel on different architectures.

A separate and often very important aspect of adopting a new computer architecture is the amount of time it takes to learn to program it efficiently. This question is not addressed directly in this study, but some anecdotal experience relating to this is given in Chapter 7.

1.3 Parallel Architectures

1.3.1 The End of the Uniprocessor

No longer can researchers expect today's codes to run faster on tomorrow's computers. Manufacturers are facing physical limitations to increasing processor frequency and are turning to increased on-chip parallelism as a solution. The factors limiting further increases in processor clock speed are summarized as the three walls [8]:

1. **Power Wall**:
    - As devices get smaller, leakage power dissipation is increasing and is now approaching the amount used to actively power transistors.
2. **ILP Wall**:
    - All easy instruction level parallelism (ILP) has been exploited.
    - There are diminishing returns in trying to exploit more ILP at the hardware level.
3. **Memory Wall**:

- Memory performance lags behind compute performance.
- Latency to main memory can be orders of magnitude slower than the length of time to perform floating point operations.

Chip manufacturers are responding to these challenges by developing chip multiprocessors (CMP) and other novel processor designs. In what follows the current major architecture designs are briefly discussed, along with how they address the issues associated with developing an efficient parallel microprocessor architecture.

1.3.2 Chip Multiprocessors

CMPs or multi-core processors are the first and most logical extension from uniprocessor to multiprocessor. The successful technologies used in single processor design are reused and replicated multiple times onto a single chip. They are backwards compatible, flexible, and easy to market to the general public. Though designed for general purpose computing they come with computational accelerators in the form of on-chip SIMD (Single Instruction Multiple Data) vector floating point units. On x86 chips, these units are accessible through the use of SSE (Streaming SIMD Extensions) assembly instructions. Latency to main memory is a major issue, as memory caching systems often must be shared among cores. This unresolved memory bottleneck is a major issue that has prompted researchers at Berkeley [8] to question whether CMPs can scale well past four or eight cores.

In the marketplace, dual and quad-core CPUs are already the standard for desktop, server, and laptop computing. Regardless of whether they can effectively scale up to many more cores, the multi-core CPU will be a reliable computing resource for many years to come.

1.3.3 Cell BE

The Cell Broadband Engine Architecture (Cell BE) developed jointly by Sony, Toshiba, and IBM is a heterogeneous multi-core chip design. A high performance to power ratio is obtained by employing eight SIMD vector processors called Synergistic Processing Elements (SPE) in tandem with a standard PPC 970 CPU (Central Processing Unit) called the Power Processing Element (PPE). The PPE runs the operating system but computations should be offloaded to the more powerful vector units. The Cell also has high bandwidth access to main memory (25.6 GB/s) and deals with memory latency by giving the developer low-level control over the flow of data [40, 7].

1.3.4 Nvidia Tesla GPU

The Nvidia Tesla architecture is a streaming multiprocessor design with hundreds of identical scalar processors. These processors are grouped into blocks that work together to implement a Single Instruction Multiple Thread (SIMT) stream processing model, implemented in the CUDA (Compute Unified Device Architecture) programming framework. The high latency to memory problem is addressed by having many more threads than processors and zero-penalty context switching between threads. Memory-stalled threads are simply swapped out of context for active threads, hiding the latency. The GPU is attached to the host CPU across the PCI-Express Bus, which has relatively low bandwidth and high latency. In order to reduce the amount of traffic across the PCI-Express Bus the GPU has its own DRAM [33].

1.3.5 ATI/AMD GPU

In this work only Nvidia GPUs are used; however ATI GPUs are functionally similar in design to the Nvidia streaming multiprocessor design. They are also streaming multiprocessors with hundreds of scalar processors built on top of ATI's graphics processor technology. The programming model used by ATI GPUs is the Brook GPU programming language developed at Stanford [13]. The reason for the exclusion of ATI GPUs from this study is the current lack of ATI GPU resources within SHARCNET and the University of Waterloo, and a less well supported programming environment than Nvidia's CUDA.

1.3.6 Accelerator Architectures

Since the term parallel architecture is often used to refer to parallel clusters of computers, in the interest of clarity the term **accelerator architecture** is adopted in this thesis to define a heterogeneous extension to a base processor that is designed to maximize computational throughput through the use of fine-grained data parallel hardware. This definition can include on-chip SIMD floating point units, GPUs, and the SPEs of a Cell Processor.

1.4 Application Programming Interfaces

At present, most work is done using vendor-specific APIs for each platform and, while powerful, each presents its own learning curve. This can present a significant time investment

and a barrier to adoption of the hardware. Work on a unified framework for programming accelerator architectures is an active area of research. Some approaches to meeting this challenge are discussed below. OpenCL aims to be a low level API, while RapidMind and HONEI [47] are meant as higher level abstractions for programming accelerator architectures.

OpenCL

OpenCL is an open source standard for stream processing developed by the Khronos group [19]. It aims to unite hardware manufacturers behind a common stream processing model. As more manufacturers develop drivers for their hardware, the OpenCL standard should expand to have support for multi-core, GPU, and Cell architectures.

RapidMind

RapidMind is a company that offers a development platform for multi-core, GPU, and Cell architectures. The platform was first conceived of as *sh*, a shader metaprogramming language for GPUs [31] but has grown to include support for other architectures. It is an attractive development option as it allows developers to ignore the lower level workings of the hardware and focus on developing their application using RapidMind's C++-integrated API. Once a program is written using the API it may then be compiled for any of the different architectures.

HONEI

HONEI [47] is similar in concept to RapidMind. It is a collection of libraries that allow developers to use a hardware-independent programming API to compile their program for Cell, GPU or SSE-optimized multi-core CPU. It is aimed at a higher level of abstraction than OpenCL by providing built-in linear algebra routines, but it also has lower level container functions and support for architecture-specific implementations.

1.5 GPU and Cell for Finite Volume Methods

The use of parallel clusters for astrophysics simulations is standard practice (for instance using FLASH [17], CLAWPACK [30], Zeus-MP [23], OpenGGCM [37]) but the use of

accelerators is not yet as widespread. This is an area of active research and since the software development tools are not yet as sophisticated as for x86 CPUs, many are waiting for the technology to mature. Some examples of work using finite volume methods or programming ideas important for this thesis are discussed below.

GPU

Several researchers have looked at using the GPU, but much of this work was done prior to the development of the CUDA framework for general purpose calculations on Nvidia GPUs. They were forced to perform computations via shading languages and other graphics APIs. Additionally the hardware design of the GPU has changed considerably and has become far more flexible and programmable.

Fatica [15] develops an Euler equation solver for a generalized stream processing architecture using the Brook programming language. Brandvik [11, 12] develops a solver for the Euler equations in both 2D and 3D using ATI GPUs. Hagen [21] also develops an Euler equation solver in 2D using Nvidia GPUs. They use OpenGL and the cg shader language in order to implement their methods on the GPU. Later Hagen [20] further develops the methods in a more general conservation law framework using the same GPU programming techniques. They apply their results to both the shallow water equations and the Euler equations in 2D. Their results demonstrate that graphics processors can be successfully employed to accelerate finite volume methods. In order to increase arithmetic intensity they used second-order high-resolution methods similar to those used in this work. One difference is that they deal with uneven bottom topography and small perturbation flows in the case of the shallow water equations which are not treated in this work. Saetra [39] extends the work done by Hagen to a system of four Nvidia GPUs connected via a gigabit ethernet switch. The shallow water equations and the 2D Euler equations are simulated using the same numerical methods and graphics APIs as in [20]. He demonstrates that small clusters of GPUs are a viable strategy for accelerating calculations and he develops some methods for alleviating the slow interconnect bottleneck.

Cell Processor

The Cell processor has not been explored as fully as the GPU for finite volume hyperbolic equation simulation. In two works by Shalf et al [49, 48] the potential of the Cell processor for scientific computing is explored by implementing several representative computational kernels drawn from scientific applications. They implement matrix multiply, sparse matrix multiply, stencil calculations, and 1D/2D FFTs using the Cell system simulator. The

stencil calculations are finite difference schemes for the 3D wave equation and the 3D heat equation on regular grids. Another CFD scheme to be successfully implemented on the Cell processor is the Lattice Boltzmann method for simulating biological fluid flow [45]. The Lattice Boltzmann method is highly parallel and maps well to different types of accelerator architectures. Performance on the Cell processor is compared to other accelerator architectures and the Cell processor is found to compare favourably. Van Dyk et al [47] are developing the HONEI framework discussed earlier. As an example use of their programming framework they implement a finite difference relaxation scheme to solve the shallow water equations on the PlayStation 3 Cell Processor. They compare performance using their hardware-independent libraries against a hardware-specific implementation.

1.6 Computational Motivation

Much of the motivation to pursue this work comes from successful use of accelerators outside of the domain of finite volume methods. The GPU and the Cell processor have been used in a wide variety of contexts to do general purpose computation, many examples of which are maintained at the website *gpgpu.org* [1].

One of the more striking demonstrations of the computational potential of accelerator architectures is the large distributed computing project *Folding@Home* developed at Stanford University. The project takes advantage of idle computer cycles around the world to run simulations on a massively parallel scale. At present there are 400 000 active CPUs running a combined total of 8.3 PFLOPS. Molecular simulations are particularly well suited to the GPU hardware [16] and of the total 8.3 PFLOPS, close to 8.0 are entirely contributed by GPUs and PlayStation 3s, despite then only accounting for 12 percent of the total number of CPUs [35].

Some other applications involving GPUs are medical imaging applications such as magnetic resonance imaging [44] as well as computed tomography reconstruction [50]. Commercial interests are pursuing the same ideas: two examples are *Acceleware* [5] and *North Star Imaging* [25]. Molecular dynamics simulations and visualization can both be accelerated by GPUs [43]. There have also been applications of the GPU to Discontinuous Galerkin methods [26], n-body simulations [22, 34], and many other astrophysics applications [4].

The Cell processor has been used more extensively in the context of high-performance computing than the GPU to date. The Cell processor is a very power-efficient architecture and currently installations using the Cell processor occupy the top seven slots on the Green 500 List [2] of the most power-efficient supercomputers, not to mention the top spot on

the Top 500 List [3] of the most powerful supercomputers. That particular distinction currently belongs to RoadRunner, a hybrid Cell and AMD Opteron cluster developed at LANL (Laurence Livermore National Laboratory). Some sample applications are Ray Tracing [10], Seismic Imaging [6], and Molecular Dynamics [14].

Both GPU and Cell are capable of excellent performance on a wide variety of applications; this is clear from the many examples in the literature. In the next chapter the high-resolution finite volume methods on structured grids are introduced. They are highly suitable for acceleration via GPU and Cell since they have a very high computational intensity and their structured data layout allows them to be readily parallelized.

Chapter 2

Numerical Method

In this chapter a high-resolution finite volume shock capturing Godunov-type solver for hyperbolic conservation laws is described [29]. This method is applied to problems in two spatial dimensions on regular cartesian grids using the method of dimensional splitting (see section 2.8). Solution accuracy is investigated by comparing with two 1D dam break test cases for the shallow water equations.

High-resolution methods are used in this study since their high number of calculations per grid cell makes them particularly well suited to high-throughput computational architectures. These methods are also readily extendible to more complex geometries and to other hyperbolic systems such as the Euler or MHD equations [29].

2.1 1D Hyperbolic System

Hyperbolic systems of PDEs are characterized by a finite speed of information propagation. Below a general first order hyperbolic PDE is introduced in 1D, however this definition may be readily generalized to more dimensions or to account for non-conservative effects through the addition of source terms to the right hand side of (2.1).

1D Hyperbolic System

Given a domain $[a, b] \in \mathbb{R}$, a state vector function $q(x, t) : \mathbb{R} \times \mathbb{R} \to \mathbb{R}^d$, and a C^1 flux function $f(q) : \mathbb{R}^d \to \mathbb{R}^d$, with $d \in \mathbb{N}$. A general form for a 1D hyperbolic conservation law

is given by,

$$\frac{\partial q}{\partial t} + \frac{\partial}{\partial x} f(q) = 0. \tag{2.1}$$

This PDE is hyperbolic provided that the Jacobian matrix $A = f'(q)$ has d real eigenvalues and is diagonalizable.

1D Conservation Law

The hyperbolic system (2.1) can be integrated over any subinterval $[x_1, x_2]$ of the 1D domain and written as,

$$\frac{d}{dt} \int_{x_1}^{x_2} q(x,t)dx = f(q(x_1,t)) - f(q(x_2,t)). \tag{2.2}$$

In this form it is clear that the conserved quantity $\int_{x_1}^{x_2} q(x,y)dx$ can only change due to a flux difference across the boundaries.

2.2 Finite Volume Formulation

Let the domain be discretized into equal-sized grid cells of width $h = x_{i+\frac{1}{2}} - x_{i-\frac{1}{2}}$, where x_i lies at the center of grid cell i and $x_{i\pm\frac{1}{2}} = x_i \pm \frac{h}{2}$ are the left and right boundaries. The discrete cell averaged values are then

$$Q_i(t) = \frac{1}{h} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} q(x,t)dx. \tag{2.3}$$

The conservation law (2.2) is then interpreted over one cell volume,

$$h\frac{d}{dt} Q_i(t) = f(q(x_{i-\frac{1}{2}},t)) - f(q(x_{i+\frac{1}{2}},t)), \tag{2.4}$$

giving an exact relationship between the fluxes across the cell boundaries and the time derivative of the cell average.

2.3 Time Evolution

Equation (2.4) is now integrated over a time step of length $k = t_{n+1} - t_n$ giving,

$$h\left(Q_i(t^{n+1}) - Q_i(t^n)\right) = -\int_{t_n}^{t_{n+1}} f(q(x_{i+\frac{1}{2}}, t))dt + \int_{t_n}^{t_{n+1}} f(q(x_{i-\frac{1}{2}}, t))dt. \tag{2.5}$$

The average flux through the boundary over time step n is defined as

$$F^n_{i\pm\frac{1}{2}} = \frac{1}{k}\int_{t_n}^{t_{n+1}} f(q(x_{i\pm\frac{1}{2}}, t))dt. \tag{2.6}$$

This can be used to recast (2.5) as an equation for the evolution of the solution forwards in time,

$$Q_i^{n+1} = Q_i^n - \frac{k}{h}\left(F^n_{i+\frac{1}{2}} - F^n_{i-\frac{1}{2}}\right). \tag{2.7}$$

To evaluate (2.6) the value of q at the boundary is needed over the entire time step. Godunov's method assumes a piecewise constant approximation to $q(x, t_n)$:

$$\tilde{q}(x, t_n) = Q_i^n \quad \text{for} \quad x \in [x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}). \tag{2.8}$$

This allows $\tilde{q}(x_{i-\frac{1}{2}}, t)$ for all $t \in [t_n, t_{n+1}]$ to be obtained from the solution of the Riemann problem between neighbouring states $q_l = Q_{i-1}$ and $q_r = Q_i$. Solving the Riemann problem exactly is very computationally expensive so typically approximate Riemann solvers are used. In this work Roe's linearized Riemann solver is used [38].

2.4 Roe Solver

The Roe solver computes an all-shock approximation ($\hat{q}(x, t)$) to the exact Riemann problem solution. The nonlinear system (2.1) is replaced with an approximate linear system,

$$\frac{\partial \hat{q}}{\partial t} + \hat{A}(q_l, q_r)\frac{\partial \hat{q}}{\partial x} = 0, \tag{2.9}$$

where $\hat{A}$ is taken to be the Jacobian of the flux function f evaluated at a specific state called the Roe averaged state [30]. This state is chosen such that the linear system accurately approximates the nonlinear problem at the interface between states q_l and q_r. The full Roe averaged state is derived for the shallow water equations in [30]. (In what follows, when it is clear that all variables are taken at the same time step, the superscript n is omitted.)

Once the Jacobian is determined, the Riemann problem at the cell interface can be solved in terms of the eigenvectors r^p and eigenvalues λ^p of $\hat{A}$. The jump in the cell-averaged conserved variables across a cell interface is given by the jump vector,

$$\Delta Q_{i-\frac{1}{2}} = Q_i - Q_{i-1}. \tag{2.10}$$

This is then written in terms of the eigenvectors, with coefficients denoted by α^p:

$$\Delta Q_{i-\frac{1}{2}} = \sum_{p=1}^{d} \alpha^p_{i-\frac{1}{2}} r^p_{i-\frac{1}{2}}. \tag{2.11}$$

Each component $\alpha^p_{i-\frac{1}{2}} r^p_{i-\frac{1}{2}}$ may be interpreted as the contribution of one wave and is commonly written as the wave vector at an interface,

$$W^p_{i-\frac{1}{2}} = \alpha^p_{i-\frac{1}{2}} r^p_{i-\frac{1}{2}}. \tag{2.12}$$

To get the flux at the interface, each component of the jump vector is upwinded based on the value of the eigenvalue. This can be done from either direction, giving two equivalent update formulas,

$$\begin{gathered} F_{i-\frac{1}{2}} = f(Q_i) - \sum_{p=1}^{d} (\lambda^p)^+ W^p_{i-\frac{1}{2}} \\ \text{or} \\ F_{i-\frac{1}{2}} = f(Q_{i-1}) + \sum_{p=1}^{d} (\lambda^p)^- W^p_{i-\frac{1}{2}}, \end{gathered} \tag{2.13}$$

where at an interface $(\lambda^p)^+ = \max(\lambda^p, 0)$ and $(\lambda^p)^- = \min(\lambda^p, 0)$. By substituting equations (2.13) into (2.7) such that the $f(Q_i)$ terms cancel, a first-order accurate update scheme is obtained,

$$\begin{aligned} Q_i^{n+1} &= Q_i^n - \frac{k}{h} \left(\sum_{p=1}^{d} (\lambda^p_{i+\frac{1}{2}})^- W^p_{i+\frac{1}{2}} + \sum_{p=1}^{d} (\lambda^p_{i-\frac{1}{2}})^+ W^p_{i-\frac{1}{2}} \right) \\ &= Q_i^n - \frac{k}{h} \left(A^- \Delta Q_{i+\frac{1}{2}} + A^+ \Delta Q_{i-\frac{1}{2}} \right). \end{aligned} \tag{2.14}$$

where $A^\pm \Delta Q = \sum_{p=1}^{d} (\lambda^p)^\pm \alpha^p r^p$.

2.5 Second Order Corrections

In order to reduce the amount of numerical diffusion introduced into the solution, a better approximation to the initial data is used. Rather than representing the data as piecewise constant, a piecewise linear approximation is used,

$$\tilde{q}(x, t_n) = Q_i^n + \sigma_i^n (x - x_i) \quad \text{for} \quad x \in [x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}), \tag{2.15}$$

where $\sigma_i^n \in \mathbb{R}^d$ contains approximations to the slopes in cell i at time t_n. The improved approximations to q_l and q_r may be used to define the Riemann problem at a cell interface more accurately and extend the method to second order. Another option (see [30] pp. 100-105) is to leave the first order solution unchanged and include the linear contributions separately as a flux correction $\tilde{F}$ which is left undefined for the moment:

$$Q_i^{n+1} = Q_i^n - \frac{k}{h}\left(A^- \Delta Q_{i+\frac{1}{2}} + A^+ \Delta Q_{i-\frac{1}{2}}\right) - \frac{k}{h}\left(\tilde{F}^n_{i+\frac{1}{2}} - \tilde{F}^n_{i-\frac{1}{2}}\right). \tag{2.16}$$

One common problem with adding second order corrections is that near discontinuities, the second order correction will introduce spurious oscillations into the approximated solution. A remedy is to use a flux-limiter to limit the impact of the correction term whenever the solution is near a discontinuity.

To compose the high-resolution scheme, the second-order scheme (2.16) is used, with flux corrections

$$\tilde{F}_{i-\frac{1}{2}} = \sum_{p=1}^{d} \frac{|\lambda^p|}{2}\left(1 - |\lambda^p|\frac{k}{h}\right)\tilde{W}^p_{i-\frac{1}{2}}, \tag{2.17}$$

where $\tilde{W}^p_{i-\frac{1}{2}}$ is a limited version of the wave vector $W^p_{i-\frac{1}{2}}$. This is a generalization of the Lax-Wendroff flux function for linear systems (see [30] pp. 100-101, 118-121). Given a limiter ϕ and $\theta^p_{i-\frac{1}{2}}$ a measure of the local smoothness of the solution, the limited wave vectors are

$$\tilde{W}^p_{i-\frac{1}{2}} = \phi(\theta^p_{i-\frac{1}{2}}) W^p_{i-\frac{1}{2}}, \tag{2.18}$$

where the smoothness measure θ is defined as

$$\theta^p_{i-\frac{1}{2}} = \frac{W^p_{i-\frac{1}{2}} \cdot W^p_{I-\frac{1}{2}}}{W^p_{i-\frac{1}{2}} \cdot W^p_{i-\frac{1}{2}}}, \tag{2.19}$$

with,

$$I = \begin{cases} i-1 & \text{if} \quad \lambda^p > 0 \\ i+1 & \text{if} \quad \lambda^p < 0 \end{cases}. \tag{2.20}$$

There is a significant amount of theory involved in choosing and applying limiters that is beyond the scope of this brief descriptive treatment. The interested reader is recommended to read pp. 106-121 of [30]. The above described method of limiting fluxes along the different wave vector directions separately is called the **wave-limiter** approach. In this work the limiter used is the superbee limiter:

$$\phi(\theta) = \max(0, \min(1, 2\theta), \min(2, \theta)). \tag{2.21}$$

This concludes this brief treatment of the wave-limiter high-resolution Godunov-type finite-volume methods. The interested reader is encouraged to consult the works of Leveque [29, 30] for a more thorough treatment of the topic. The rest of this chapter introduces the shallow water equations and the method of dimensional splitting for extending the 1D methods to 2D Cartesian meshes.

2.6 Shallow Water Equations

The shallow water equations in the absence of viscosity or source terms form a hyperbolic conservation law system of three equations:

$$\frac{\partial}{\partial t}\begin{bmatrix} h \\ hu \\ hv \end{bmatrix} + \frac{\partial}{\partial x}\begin{bmatrix} hu \\ hu^2 + \frac{gh^2}{2} \\ huv \end{bmatrix} + \frac{\partial}{\partial y}\begin{bmatrix} hv \\ huv \\ hv^2 + \frac{gh^2}{2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \tag{2.22}$$

Here h is the height of the fluid, u and v are the fluid velocities and g is the gravitational constant (for a derivation, see Kundu [28] or Leveque [30]). The equations are most commonly interpreted as a height-averaged simplification to the Euler equations suitable for situations where the height scale is small compared to the length scale. In the rest of this thesis the value of the gravitational constant is taken to be one ($g = 1$).

2.7 2D Finite Volume Discretization

A rectangular 2D grid with grid spacings Δx and Δy is used to discretize the system. The finite volume method is now formulated in terms of average values over grid cells (i, j) with areas Ω_{ij},

$$Q_{ij}^n = \frac{1}{\Delta x \Delta y} \iint_{\Omega_{ij}} q(x, y, t_n) dx dy. \tag{2.23}$$

2.8 Dimensional Splitting

The 1D high-resolution method may be used to solve the 2D problem by using dimensional splitting. The 2D problem is split into two 1D problems which are solved in an alternating fashion. Consider the 2D hyperbolic system,

$$\frac{\partial}{\partial t}q + \frac{\partial}{\partial x}f(q) + \frac{\partial}{\partial y}g(q) = 0, \tag{2.24}$$

where f and g are the fluxes in the x and y directions. Instead of solving this problem directly, the equation may be approximated by instead discretizing two 1D subproblems:

$$\frac{\partial}{\partial t}q + \frac{\partial}{\partial x}f(q) = 0, \tag{2.25}$$

$$\frac{\partial}{\partial t}q + \frac{\partial}{\partial y}g(q) = 0. \tag{2.26}$$

The approximate solution is then evolved forwards in time by alternately applying the 1D update formula (2.7) to (2.25) and (2.26),

$$\begin{aligned} Q^*_{ij} &= Q^n_{ij} - \frac{k}{\Delta x}\left(F^n_{i+\frac{1}{2},j} - F^n_{i-\frac{1}{2},j}\right), \\ Q^{n+1}_{ij} &= Q^*_{ij} - \frac{k}{\Delta y}\left(G^*_{i,j+\frac{1}{2}} - G^*_{i,j-\frac{1}{2}}\right). \end{aligned} \tag{2.27}$$

There is an associated splitting error that is formally first order but in practice is normally second order (see [30] pp. 388). The main benefit to this approach is its simplicity, it allows the 1D second-order methods to be applied directly to obtain second-order results in 2D. No derivatives transverse to the grid cell alignment need to be explicitly calculated, nor does a two-stage time integration scheme need to be used.

2.9 Boundary Conditions

Boundary conditions are implemented by using two imaginary layers of ghost cells outside of the physical domain. This allows boundary cells to be treated the same as other grid cells and allows for a number of different types of boundary conditions to be implemented. More details about implementing different types of boundary conditions may be found in [29]. In the numerical tests below, ideal wall boundary conditions are employed.

2.10 Accuracy Tests

The high-resolution methods are applied to two test problems to demonstrate the accuracy of the method and implementation. Both problems are dam break problems involving an initial discontinuity in the height of the fluid. The first is a 1D dam break problem and the second is a radially symmetric dam break problem.

2.10.1 1D Dam Break Problem

The test case is the Riemann problem with initial data

$$h(x,0) = \begin{cases} 1.0 & \text{if} \quad x < 0 \\ 0.5 & \text{if} \quad x > 0 \end{cases} \quad , \quad u(x,0) = 0. \tag{2.28}$$

This results in a shock wave travelling to the right and a rarefaction wave travelling to the left which may be computed in closed form [46]. Figure 2.1 shows a 1D slice of the computed results on a 50×50 grid. Both the first-order accurate update method and the high-resolution second-order scheme are plotted against the analytic solution. The grid is aligned to the x and y axes but the initial discontinuity is placed along the line from $(1,-1)$ to $(-1,1)$. The results are thus computed at a diagonal to the Cartesian mesh, demonstrating that the dimensional splitting does not introduce excessive diffusion.

Figure 2.2 contains close-ups of the shock and the rarefaction wave at time 0.25s. The plots show that as the grid is refined the numerical results approach the exact solution.

2.10.2 Radial Dam Break Problem

A cylindrical dam break flow with radial symmetry is produced by the initial data,

$$\begin{aligned} h(x,y,0) &= \begin{cases} 2.0 & \text{if} \quad x^2 + y^2 < 0.25 \\ 1.0 & \text{if} \quad x^2 + y^2 > 0.25 \end{cases} \quad , \\ u(x,y,0) &= v(x,y,0) = 0. \end{aligned} \tag{2.29}$$

Figure 2.3 shows a time series of scatter plots of the second-order accurate numerical solution on a 100×100 grid compared to a high-resolution 200000 point 1D simulation with radial symmetry. Even with relatively few points the method closely follows the higher resolution simulation and no grid alignment effects are visible.

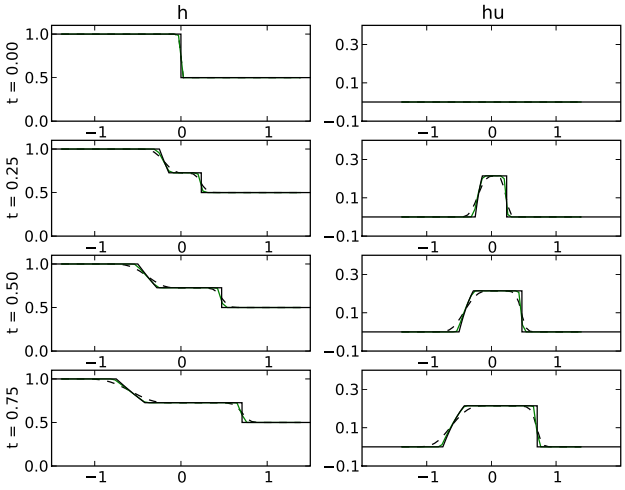


Figure 2.1: **1D Dam Break Problem:** The initial discontinuity separates into a right moving shock wave and a leftward travelling rarefaction wave. The black line is the analytic solution, light gray solid line is the second order method, and the black dashed line is the first order method. The additional smearing of the first order method is clearly visible near the shock discontinuity.

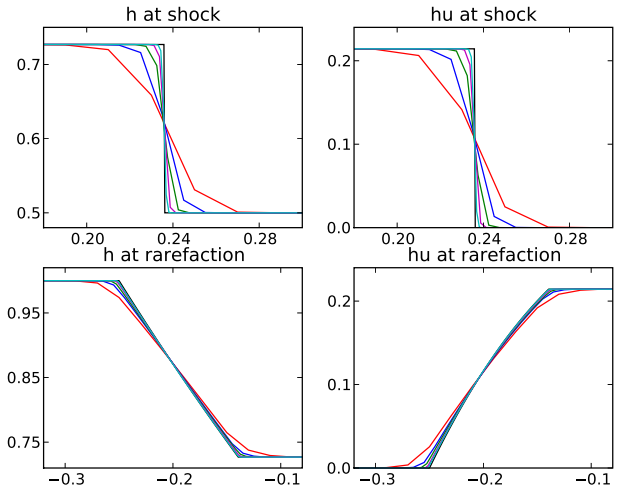


Figure 2.2: **1D Dam Break Problem:** These plots demonstrate the grid convergence properties of the second-order method on a discontinuous shock solution. They are close-ups at $t = 0.25$ for the flow of Figure 2.1. The analytic solution is the sharpest discontinuity, and the increasingly accurate numerical approximations are from simulations with 100, 200, 400, 800, and 1600 grid cells respectively.

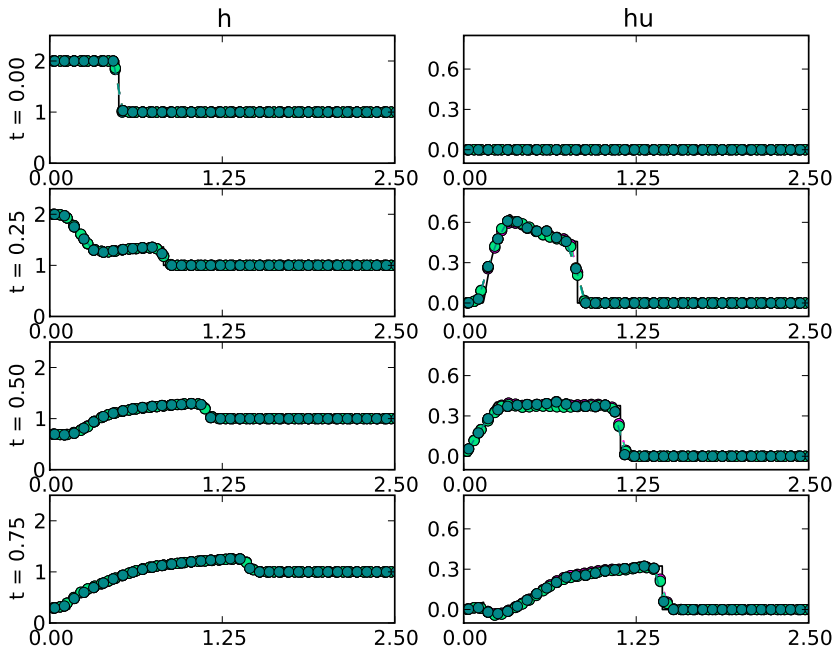


Figure 2.3: **Radial Dam Break Problem:** Scatter plot of eight slices compared with a high-resolution 1D radially symmetric solution. Slices of the 2D simulation are taken from $(0, 0)$ out to a radius of 2.5 along angular slices with increments of $\pi/4$.

2.11 Benchmark Tests

2.11.1 Single Processor Test Case

For performance comparisons the initial condition used is a linearly increasing height across the entire 2D domain with zero initial velocity,

$$\begin{aligned} h(x,y,0) &= \frac{1}{2}\left(\frac{x-x_-}{L_x} + \frac{y-y_-}{L_y}\right) + 1, \\ u(x,y,0) &= v(x,y,0) = 0, \end{aligned} \tag{2.30}$$

where x_- is the western boundary, y_- is the southern boundary, and L_x and L_y are the lengths of the x and y sides respectively.

The length of the domain will vary somewhat but is specified in the caption accompanying each set of results. The standard grid size used is a 1000×1000 grid on the domain $[-10, 10] \times [-10, 10]$ with a fixed time step $\Delta t = 3e - 4$.

2.11.2 MPI Cluster Test Case

In the case of MPI cluster results, the test case used is the radial dam break problem of section 2.10.2 with a grid size of 10000×10000 and the domain is $[-1, 1] \times [-1, 1]$ with a time step size of $\Delta t = 1e - 5$.

2.12 Summary

In this chapter a Godunov-type shock preserving high-resolution finite volume method due to Leveque [30] was introduced and extended to 2D by using the method of dimensional splitting [30]. Numerical results were verified against analytic and high-resolution 1D results for the shallow water equations. In the next chapters the implementation of this method for x86 CPU, GPU, and Cell are discussed.

Chapter 3

x86 CPU Implementation

In this chapter the high-resolution finite volume method described in chapter 2 is implemented for multi-core CPUs. Modern CPUs offer a significant amount of parallelism to the developer. Getting good performance out of a CPU involves using SIMD vector floating point units, enabling multi-threading and applying data access patterns that use cache memory efficiently.

Two layers of parallel optimizations are described in this chapter. MPI process level parallelism for distributed memory systems and thread level parallelism for shared memory architectures.

3.1 Multi-Core CPU

Microprocessor design has in the past focused on increasing clock frequency and exploiting instruction level parallelism as a means of improving processor performance. Further improvements in these areas are difficult due to the *Power wall* and *ILP wall* [41, 24]. These difficulties have led manufacturers to pursue thread and vector level parallelism through multi-core designs and increased use of on-chip floating point vector processing units (Figure 3.1), respectively.

A thread, or serial thread of execution is a set of tasks running sequentially. Typically a computer program will have several different threads executing tasks in parallel. In the context of scientific computing threads are an efficient way to divide workloads among different cores on a shared memory machine.

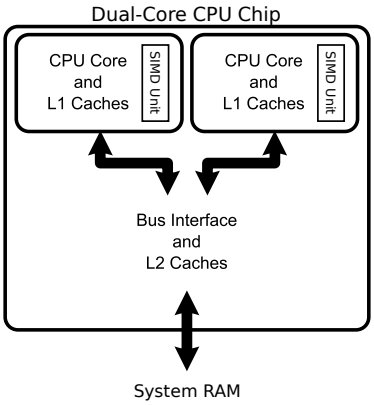


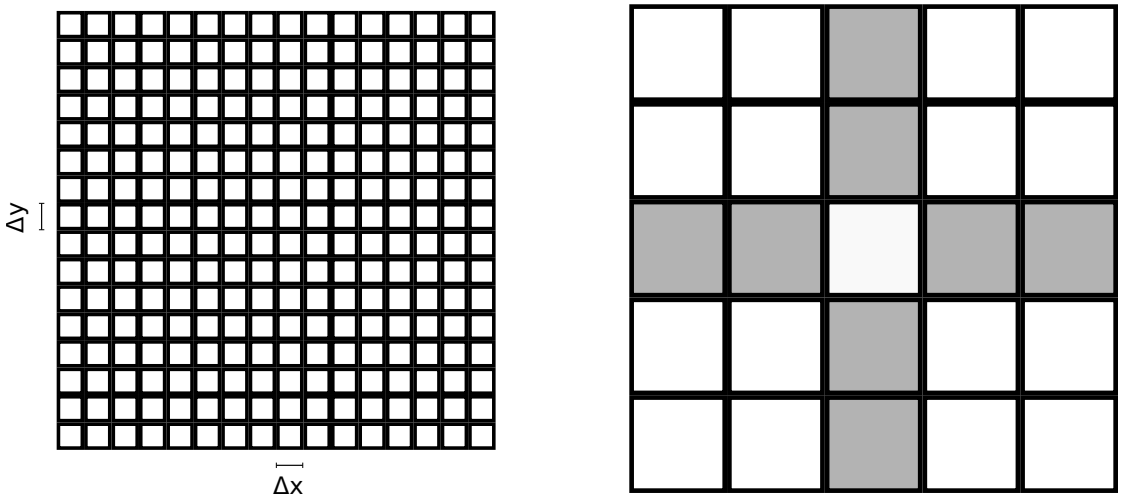
Figure 3.1: A schematic diagram of a generic dual-core microprocessor. Visible in the diagram are two levels of cache and the SIMD floating point units.

CPUs also have SIMD vector processing units to accelerate calculations. A CPU typically has a 128 bit wide SIMD vector unit capable of performing four single precision calculations or two double precision calculations at the same time. These units may be accessed manually through the use of the SSE (Streaming SIMD Extensions) assembly instruction set. More typically one simply compiles with a high optimization setting and allows the compiler to vectorize the code.

As the available on-chip parallelism increases, memory bandwidth and latency issues can easily become a limiting factor. Memory bandwidth and latency improvements lag behind those made in floating point calculation performance [24]. Since latency to main memory is orders of magnitude slower than the cost of doing calculations, it is important to be able to hide this latency behind other operations. In a CPU this is done through the use of one or more large on-chip memory caches. Automatic pre-fetching algorithms are used to bring data into the cache before it is needed for calculation. Performance penalties are experienced whenever pre-fetching fails and calculations must wait for accesses to main memory.

3.1.1 SHARCNET CPU Cluster

All of the CPU code performance results to be presented in this thesis are performed on SHARCNET's 'angel' cluster. It is SHARCNET's GPU development cluster, but for performance testing one can just as easily run CPU codes without enabling the GPUs, (the GPU aspect is discussed later in Chapter 4). The cluster has the following specifications:



(a) Computational domain (b) 9-point stencil

Figure 3.2: The computational domain is a regular 2D grid of data with grid spacings Δx and Δy. When updating a grid cell, information is needed from two neighbouring cells in each direction.

- 22 dual-socket servers, with 2 quad-core Xeon, E5430 @ 2.66GHz on each server.
- 16 GB Memory per server.
- DDR Infiniband (16 GB/s) interconnect between CPU nodes.

3.2 Serial Implementation

The computational domain consists of a structured 2D grid (Figure 3.2a). As the simulation evolves forwards in time, each grid cell must be updated to the next time step based on the values in neighbouring cells. The second order numerical scheme requires data from the two nearest neighbours in each direction, as illustrated in the data stencil in Figure 3.2b. In the dimensional splitting approach (see section 2.8), the grid cell updates are done in two passes. First the updates based on intercell fluxes in the x direction are performed, and then these updated values are used to perform the y updates. This results in a basic program structure of two inner loops over all of the grid cells. Since cell updates may be executed in any order this is where all parallel optimizations are applied.

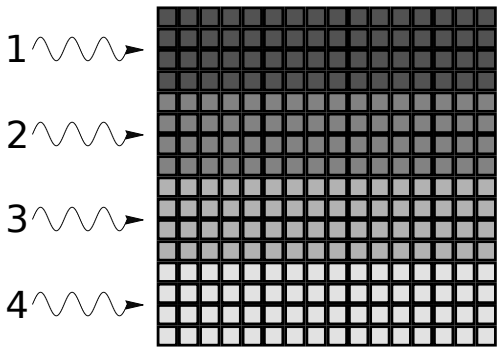


Figure 3.3: In order to minimize cache conflicts, threads are assigned grid cells in a striped tiling pattern. In the diagram each of the 4 threads is assigned one tile of the array.

3.3 Single Server Optimization

3.3.1 Threaded Implementation

The implementation is parallelized for shared memory architectures by making use of pthreads. The workload is divided evenly between threads by splitting up the computational domain. An alternative approach to parallelizing for shared memory machines is to use OpenMP directives. In this work pthreads are used instead for portability reasons, since in both the Cell Processor and GPU implementations pthreads are necessary to control separate processes.

Memory issues can arise when multiple cores need to read and write data to the same cache-line. Behaviour in this case depends on the architecture but will always result in poorer performance. In order to minimize cache conflicts, threads are given their own copy of all constants to be used in the calculation and the domain is divided among the threads by tile (Figure 3.3). This decomposition creates a thread dependency when updating in the y direction but not in the x direction. This is because the neighbouring x values are not overwritten by different threads but y neighbours are. The consequence of this is that when updating in the x direction the updated value may be saved directly back to the array, whereas in the y direction the residual is saved to another array and then added back to the result after all threads are finished computing the y residuals.

3.3.2 SSE SIMD Implementation

SSE instructions and other lower level optimizations are enabled through automatic compiler optimizations (including -O3). A hand-tuned application could improve cache and vectorization performance, but this has not been pursued. The effects of the compiler optimization as well as multi-threading on performance are summarized in table 3.1. For this application the SSE vectorization is not enabled automatically due to a non-standard choice of loop structure, however in the future this will be improved.

	Single Precision		Double Precision	
Kernel	time (s)	Speed-Up (x)	time (s)	Speed-Up (x)
icc O0	814.60	0.43	837.77	0.51
icc O3	353.93	1.00	425.26	1.00
2 Cores	175.44	2.02	208.32	2.04
4 Cores	85.80	4.12	106.87	3.98
8 Cores	43.52	8.13	58.15	7.31

Table 3.1: Performance benefit of multi-threading and compiler optimizations on one Xeon processor for up to 4 cores. The 8 core version is running on one dual-socket server with two Xeon processors. The base speed (1.0x) is taken to be the compiler optimized version of the code, this is used throughout this work as a base reference point to which everything else is compared. The multi-threading scales almost linearly with a fixed problem size.

3.4 MPI Implementation

The serial code is parallelized at a coarse level by dividing the 2D grid of data into subgrids as shown in Figure 3.4a. Data is assigned to the MPI nodes in a grid layout with each node getting a rectangular subdomain of data. Since the subdomains are equally sized, each node has four neighbouring nodes with which it must communicate. MPI communications are done after each time step, sending updated grid cell data to the neighbouring nodes. The amount of data that is sent depends on the numerical scheme being used. In the case of the nine point stencil each grid cell requires information from the two nearest cells in each direction. When the neighbouring cells lie across inter-node boundaries, the two nodes must exchange two rows or columns of updated grid cell data (Figure 3.4b).

In order to make the MPI implementation portable across different architectures, the MPI exchanges are done entirely in separate exchange buffers. Each MPI exchange consists

of three distinct stages: packing, sending/receiving, and unpacking (Figure 3.5). In the packing stage neighbour data is copied from the computational grid into MPI send buffers. The sending/receiving stage is when the MPI data transfer calls take place, sending data across the network between nodes. Then in the final stage the data is copied back from the buffers to the computational grid. By copying data to buffers, rather than applying MPI calls directly to the grid data, the data structure of the grid is decoupled from the MPI implementation. This allows the underlying data structure to change without affecting the communication protocols.

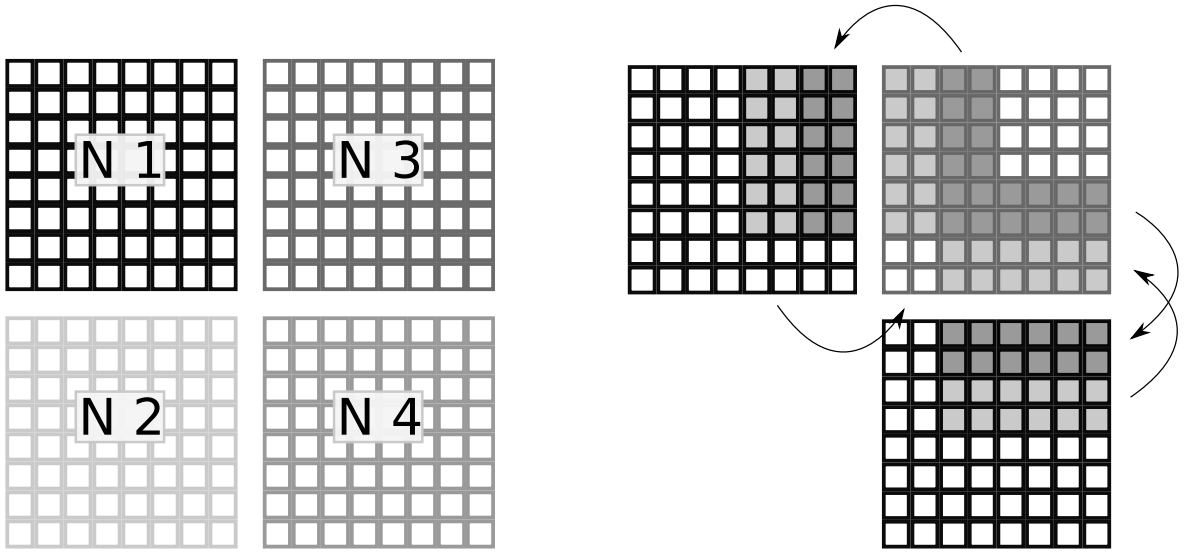
In table 3.2 the scaling of the CPU code across multiple MPI nodes is investigated.

	Single Precision		Double Precision	
N	time (s)	scaling (x)	time(s)	scaling (x)
1	4096	1.00	4609	1.00
2	1995	2.05	2314	1.99
4	991.8	4.12	1139	4.04
8	483.9	8.46	566.1	8.14
16	302.8	13.53	302.8	13.46

Table 3.2: MPI performance results on the benchmark system. The benchmark system has dual socket Xeon processors, so each MPI node consists of eight cores. The test case is the radial dam break problem, executed for 1000 time steps at 10000×10000 grid resolution (section 2.11.2). This is a constant total problem size and as the problem size shrinks its clear that performance suffers and gives less than linear scaling.

3.5 Summary

In this chapter the CPU implementation to be used as a reference implementation was introduced and it was parallelized to take advantage of multi-core parallelism and parallel cluster level parallelism. In the next chapter the solver is implemented for the GPU accelerator architecture.



(a) MPI domain decomposition

(b) Internode exchanges

Figure 3.4: The global domain is partitioned into equally sized subdomains. Each subdomain is stored at a separate node. The updated grid cell values are passed along to each grid neighbour, as shown by the arrows in the diagram. The corner values are not used in the cell update scheme.

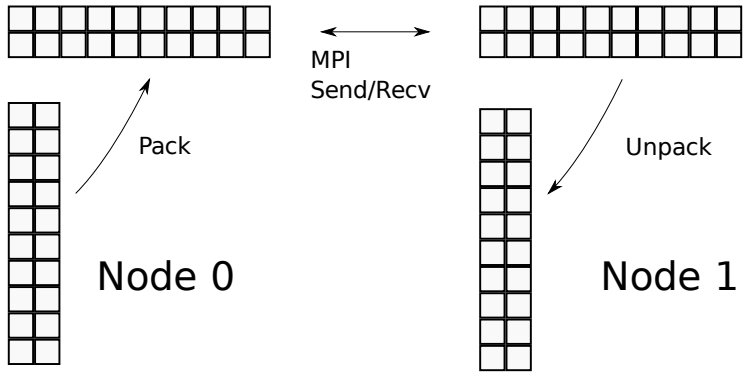


Figure 3.5: To enable portability across architectures, MPI exchanges are encapsulated in their own data structure. Each architecture-specific implementation has a packing scheme unique to its data layout but the communication is uniform.

Chapter 4

GPU Implementation

The GPU (Graphics Processing Unit) is a specialized microprocessor, optimized for high floating point throughput and high memory bandwidth applications. Though originally specialized for graphics applications, researchers have attempted to adapt this computational power to scientific applications since the beginning of GPU development. Historically the inflexible hardware, low floating point precision and confusing graphics-specific APIs have been major limitations. Nvidia's Tesla GPU architecture (section 4.1) and CUDA (Compute Unified Device Architecture) programming model (section 4.2) attempt to address these issues and bring GPGPU (General Purpose GPU) computing to a broader audience. The CUDA model is used to map the finite volume hyperbolic solver to the Nvidia GPU hardware (section 4.3). The solver is then placed into the MPI framework described in Chapter 3 and performance is investigated on a Tesla GPU Cluster.

4.1 Nvidia Tesla GPU

The Nvidia Tesla architecture is a streaming multiprocessor design that uses hundreds of cores and thousands of threads at a time. The GPU is attached to a host CPU system via the PCI Express Bridge as an add-on component. The massive parallelism comes from hundreds of identical scalar processors (SP) that work together in a SIMT (Single Instruction Multiple Thread) format. The efficiency of the system is due to the organization of the SPs in a hierarchical structure. The SPs are grouped into blocks of eight, called **streaming multiprocessors** (SMP). The SMPs are organized into pairs and threads are assigned and scheduled in batches of 32 at a time, called a **warp**.

In practice many more threads are allocated than there are scalar processors. Thus there is always a pool of threads waiting, so that when a warp must wait on memory or instruction latency, a new warp is selected from the pool of waiting threads. This context switching is performed in hardware without penalty and provides an effective mechanism for hiding memory latency.

Another strength of the Tesla architecture for scientific computing is that each SMP also has two special function units for processing transcendental functions (e.g. cos, sin, exp, sqrt, rsqrt). The different components are illustrated schematically in Figure 4.1. Additional details of the hardware will be further elaborated in the context of the CUDA programming model (section 4.2).

4.1.1 SHARCNET GPU Systems

Two SHARCNET GPU systems are used to profile performance. The first is SHARCNET's 'tope' GPU development system:

- Two quad-core Xeons, @ 2.0GHz.
- 32 GB system RAM.
- C1060 tesla GPU Computing System, with 4 Tesla C870 GPUs with 1.6 GB device memory per GPU.

This system has a desktop/workstation type of set-up and uses the older Tesla C870 GPU hardware.

The second GPU system used is SHARCNET's 'angel' GPU cluster (already introduced in section 3.1.1).

- 22 dual-socket servers, with 2 quad-core Xeons, E5430 @ 2.66GHz, on each server.
- 16 GB Memory per server.
- 11 Nvidia S1070 GPU computing systems, each with 4 Tesla T10 GPUs with 4 GB device memory per GPU.
- DDR Infiniband (16 GB/s) interconnect between CPU nodes.
- GPUs attached via PCI Express 2.0 Bridge, 1 GPU per Xeon processor.

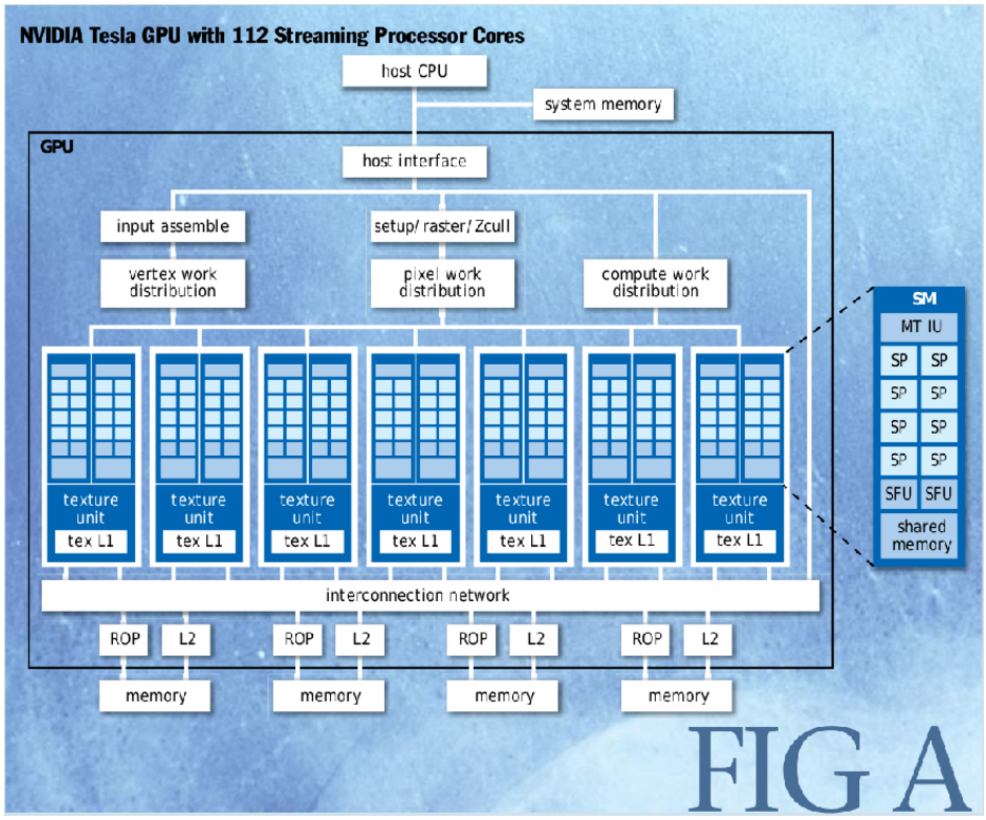


Figure 4.1: Overview of the Tesla Architecture. Illustrated are 14 SMPs and a close-up of an SMP showing the scalar processors (SP) and the special function units (SFU), this is the older generation Tesla C870 GPU without the double precsion units. Image reproduced from [32].

In this parallel set-up, the MPI level communication takes place between CPU nodes and the communication between the GPU and CPU takes place via a threaded local implementation.

4.2 CUDA Programming Model

CUDA provides extensions to the C language which allow developers to map data-parallel computations onto Nvidia GPUs. It accomplishes this through a system of layered hierarchies of thread-level parallelism. This system allows developers to write a single serially

executing compute kernel and execute it on the GPU in parallel on hundreds or thousands of threads.

In the CUDA programming model, code and memory are divided between a **host** CPU and the GPU **device**. The host CPU runs the main application and offloads certain tasks to the device by calling a **kernel** function.

4.2.1 Kernel

A kernel is defined almost identically to a typical C function. The difference occurs at runtime, when the kernel is executed: rather than being executed once on the host, it is instead executed concurrently on many threads on the GPU device. Listing 1 shows a simple kernel definition and how it would be called from the host code.

Listing 1 Definition of a simple vector multiplication kernel and the correct calling syntax. The keyword `__global__` labels a function as a kernel function, kernels must always return `void`. The syntax $<<<1, N>>>$ is used to define the number of thread blocks, 1, and the number of threads within a thread block, N.

```
__global__ void mult_kernel(float * a, float * b, float * c)
{
  int i = threadIdx.x;
  c[i] = b[i]*a[i];
}

//In Host code, calling mult_kernel with N threads in 1 thread block
  mult_kernel<<<1,N>>>(a, b, c)
```

4.2.2 Thread Blocks

Threads are grouped together into thread blocks consisting of up to 512 threads. During kernel execution each thread block is assigned to one pair of SMPs. Each thread within a thread block gets a unique three-dimensional id number that can be accessed inside of the kernel function, `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`. These variables are automatically passed to every kernel as an argument and can be directly accessed inside of the kernel definition as in Listing 1.

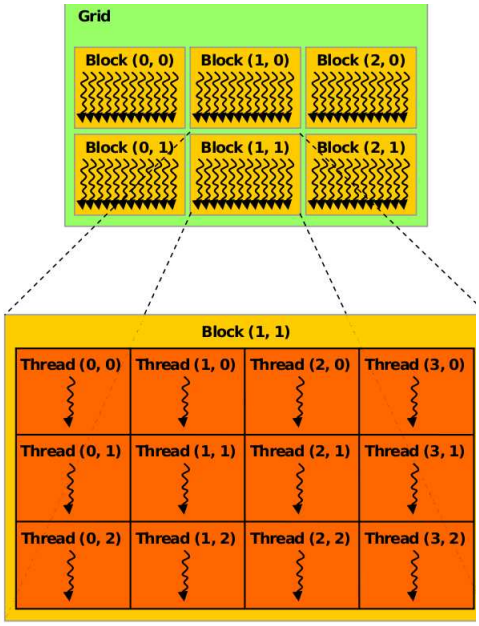


Figure 4.2: A 3×2 grid of 4×3 thread blocks. This starts a total of 72 concurrent threads. Note that 12 is a fictional thread block size: the thread block size should be chosen to be a multiple of the warpsize, 32. Reproduced from the Nvidia CUDA Programming Manual [33].

4.2.3 Thread Block Organization

In Figure 4.2 a grid of thread blocks is illustrated, thread blocks are organized into grids because it provides a simple way of identifying each thread uniquely via its thread ID and its block ID number: `blockIdx.x` and `blockIdx.y` (Figure 4.2). This is suitable for 1D or 2D grid type data which is a very common GPU workload.

4.2.4 Memory

The CUDA model contains several different memory spaces. The two most frequently used are the host memory (system RAM) on the CPU and the device memory (**global memory**) which resides on the GPU device as its own separate discrete store of DRAM. The GPU has its own memory in order to limit the amount of data that must travel between device and host.

Device memory is accessible to threads executing on the device, but host memory is not accessible to them. Data must be transferred explicitly from host memory to GPU memory prior to kernel execution by using memory copy instructions called in the host application (see Listing 2).

Memory Hierarchy

In addition to the global memory space on the device, there are several other smaller but lower latency memory spaces which are important to getting good performance (Figure 4.3).

- **Texture Memory:** Read-only memory space visible to all threads and optimized for texture fetches (memory operations optimized for bitmap textures). It uses a different API than normal memory accesses, but if used effectively may yield performance improvements.
- **Constant Memory:** Read-only memory space that is visible to all threads.
- **Shared Memory:** Read/Write memory space that is visible to all threads in a thread block. Allows threads in a thread block to share data and synchronize.
- **Local Memory:** Memory space used if there are not enough registers, allocated out of global memory so it is very high latency.

In addition to using these memory spaces to reduce latency, a developer should also ensure that any accesses to device memory are **coalesced**.

Memory Coalescing

Memory operations are coalesced on a **half-warp** (16 threads) basis. Since the exact coalescing rules are updated continuously with the hardware [33], only the most inflexible prescription for memory coalescing is introduced. When accessing global memory, if all 16 threads access memory at sequential addresses separated by 32, 64, or 128 bits and the initial address is aligned to a 32, 64 or 128 byte boundary, then only 1 or 2 memory operations will be performed for all of the data (Figure 4.4). If the memory operations are not coalesced, then 16 separate individual memory operations are performed. This is of particular importance since coalesced memory operations are between 2 and 10 times faster than uncoalesced ones [33]. The more recent the hardware, the more flexible these rules

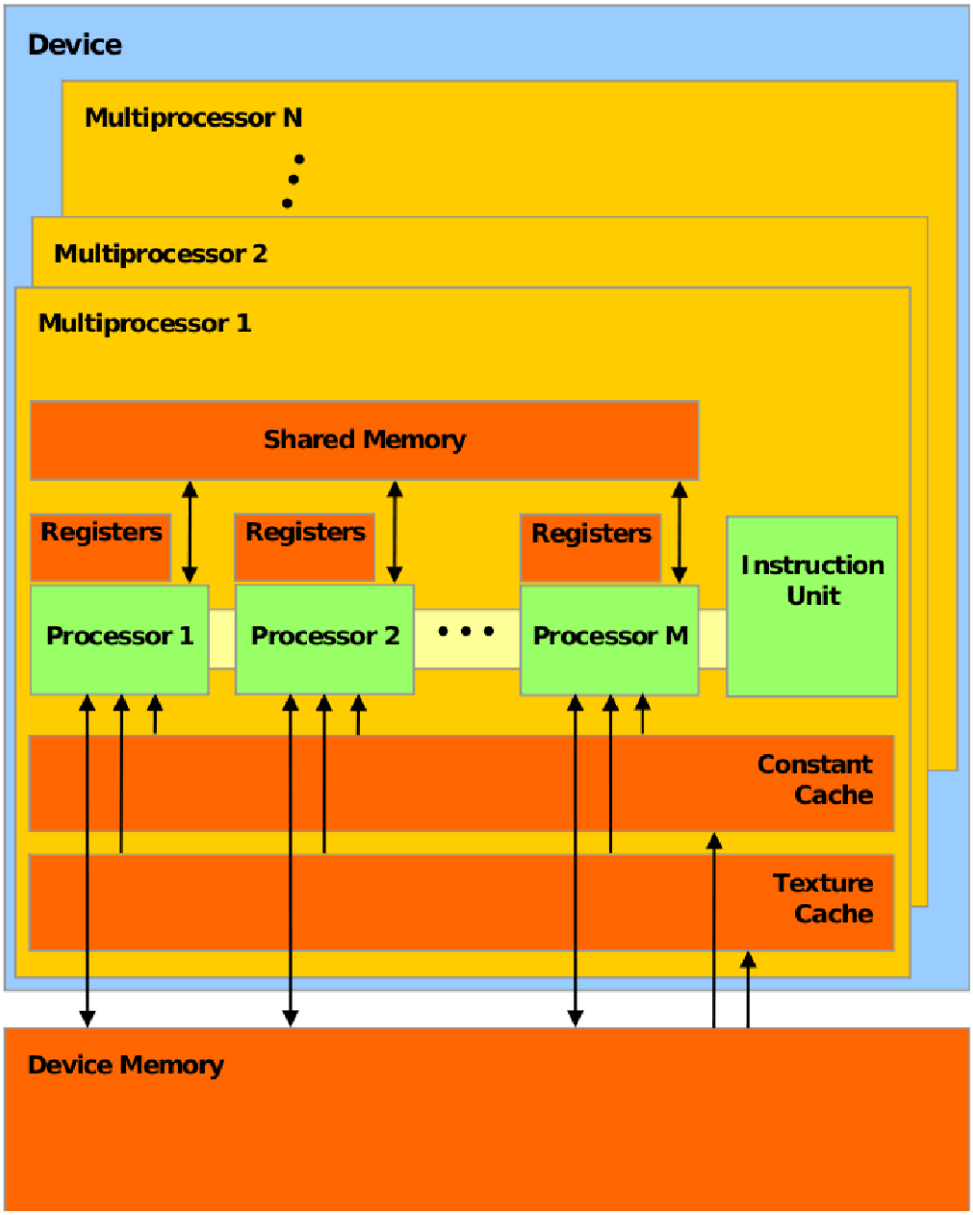


Figure 4.3: A view of a streaming multiprocessor with M streaming processors. Visible in the diagram are the shared, constant, texture, and device memory spaces. The former three are on-chip and low-latency, while device memory is off-chip and high-latency. Image reproduced from the Nvidia CUDA programming guide [33]

are but in general, nearby threads should access nearby, aligned data when possible (the full listing of current coalescing requirements is given in the CUDA programming guide [33]).

4.2.5 Synchronization

Synchronization may be achieved efficiently within a thread block through the use of barriers. Thread blocks may synchronize through the use of atomic memory operations or through multiple kernel calls (for more information consult the programming guide [33]).

Listing 2 This code shows the data pointed to by `A` in host memory being transferred to the address `Ad` in device memory before being used as input to the kernel function. After kernel execution the data is transferred back to the host memory. The `cudaMalloc`, `cudaFree`, and `cudaMemcpy` operations are the same as their C cousins except that they deal with addresses in device memory.

```
int main(int argc, char * argv[])
{
  int N = 128;
  float A_host[N];
  init_array_vals(A);
  float * Ad;
  cudaMalloc((void**)&Ad, sizeof(float)*N);
  cudaMemcpy(Ad, A_host, sizeof(float)*N,
             cudaMemcpyHostToDevice);
  kernel_A<<<1,N>>>(A);
  cudaMemcpy(A_host, Ad, sizeof(float)*N,
             cudaMemcpyDeviceToHost);
  print_result(A);
  cudaFree(Ad);
}
```

4.2.6 CUDA Programming Model Summary

This section introduced the CUDA programming model and how its data-parallel programming framework maps onto the hardware. The hierarchy of memory spaces, in particular

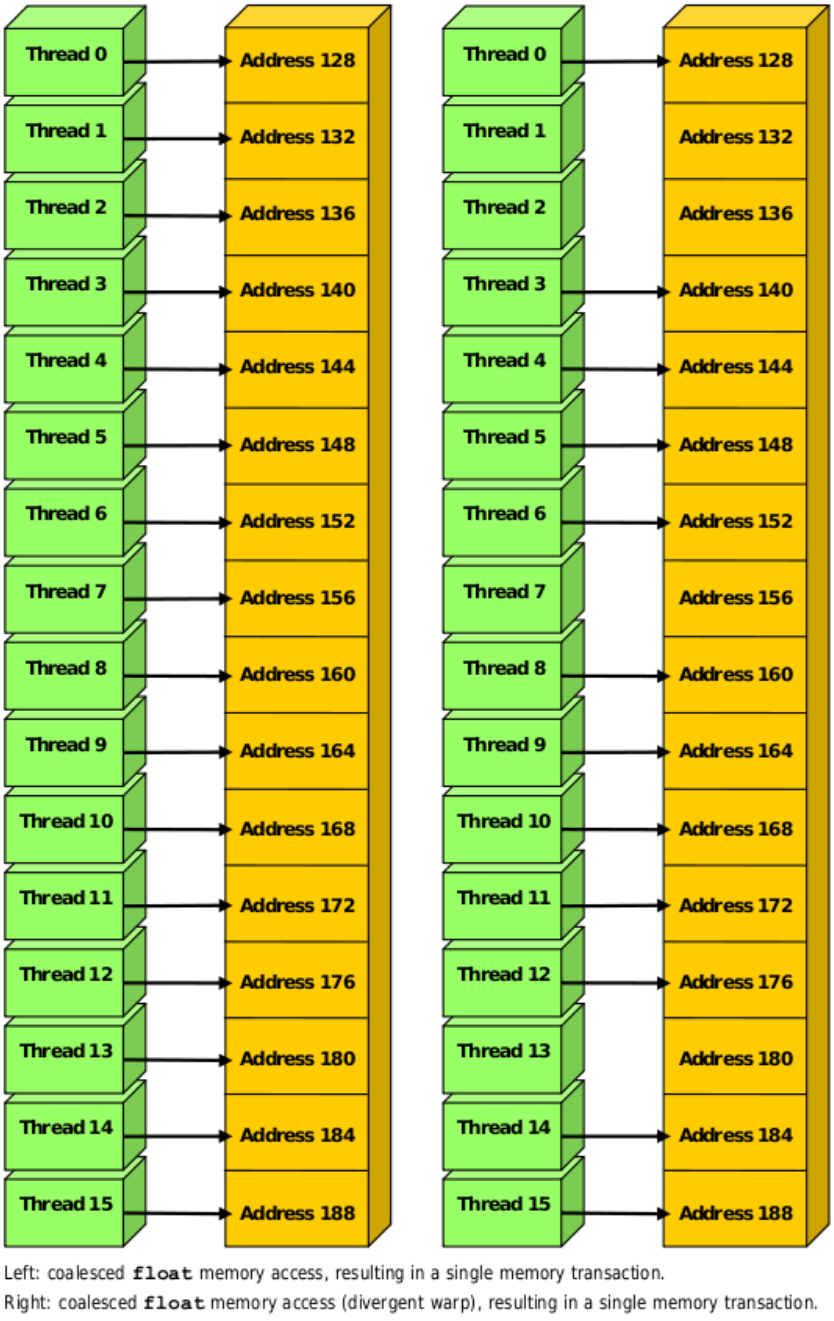


Figure 4.4: In this diagram from the CUDA programming guide [33], memory operations will be coalesced since each thread accesses a neighbouring 32-bit value and the first thread accesses an address aligned to a 128-byte boundary.

shared, constant, and device memory are all used in the next section to write efficient CUDA code. While the CUDA model is successful as an abstraction for writing highly parallel algorithms, writing high performance CUDA code depends on optimizing many hardware specific factors.

4.3 Mapping the Hyperbolic Equation Solver to the CUDA Programming Model

Three different aspects of the implementation are investigated: the layout of data in memory, the implementation of the kernel, and extensions to using multiple GPUs.

4.3.1 Memory Layout

Kernels operate only on data in device memory so it is important to implement a data stucture such that the time spent communicating between host and device is minimized. Two memory layouts are implemented, the first is generic and suitable for all GPU devices, while the second may only be used when the entire problem fits into device memory.

Host-Centric

Listing 3 illustrates the host centric implementation. Based on the size of the available device memory, the computational grid is partitioned into tiles. Computations are performed by first sending a tile to the GPU, executing the kernel, and then sending the tile back to host memory. This kernel is suitable for all CUDA devices but suffers poor performance since no attempt is made to hide the communication time between host and device. This is the least invasive method of making use of the GPU as the underlying data structure remains unchanged.

One potential improvement that has not been implemented is that the data transfers between host and device may be pipelined with computation in a double buffered approach. On newer CUDA enabled GPUs memory transfer may take place at the same time as kernel execution, allowing data-transfer time to be hidden behind computation. For very large problem sizes for which the GPU centric approach is unsuitable this is likely the best approach.

GPU-Centric

The GPU often has large amounts of device memory and provided the entire dataset fits into device memory there is no reason to transfer data back to the host at every time step (Figure 4.5). Data is sent back to the host only when saving to disk, or in the multi-GPU and MPI versions, when performing ghost cell communications (Section 4.3.3). This implementation eliminates almost all communication time (Table 4.1) of the host-centric implementation. In practice the memory restriction of fitting the dataset into GPU memory is not a major issue as typically the amount of host memory is roughly equalized with the amount of GPU memory.

Listing 3 Host-Centric Kernel Call

```
cudaMemcpy(Q_device, Q_host, Q_size, cudaMemcpyHostToDevice);
update_kernel<<<dim_Grid, dim_Block>>>(call_params, Q_device, Q_dev_out);
cudaMemcpy(Q_host, Q_dev_out, Q_size, cudaMemcpyDeviceToHost);
```

Host/GPU Centric	Comm Time (s)	Compute Time (s)
Host-Centric	21.08	24.91
GPU-Centric	0.11	24.83

Table 4.1: These results are obtained on a 1000×1000 grid using the initial conditions of section 2.11.1, on an Nvidia Tesla C870 (Section 4.1.1) using the host-centric, and GPU-centric memory layouts. Since the entire problem fits into device memory, the communication overhead time is eliminated in the GPU-centric version. The Kernel used in this case is the kernel with constant memory enabled (discussed in section 4.3.2)

4.3.2 Kernel Implementations

The kernel implementation must take into account hardware constraints in order to obtain good performance. The current GPU implementations perform double the work of the CPU or Cell version, since every grid cell computes the fluxes through all four of its boundaries, which means that interior fluxes are computed twice. In future versions the additional computations can likely be eliminated or at least reduced. The performance of the different kernels is compared in table 4.2.

Naive

The naive version of the kernel is a copy of the CPU compute kernel with minimal alterations. The loop over all of the grid cells is replaced by a kernel call with one thread spawned for every grid cell. The compute kernel performs the x update, y-update, and then saves the updated cell value into an output grid. This means that the flux calculations are duplicated at every grid cell interface.

Eliminate Local Memory Usage

The first optimization is to eliminate any use of the local memory space. In the case of the naive kernel, some excess variables that should be stored in registers are spilling into the slow local memory space. By eliminating unnecessary variables, the memory requirements may be reduced and the kernel variables fit into the registers.

Constant Memory

Constants that are used frequently are stored in the faster constant memory space rather than in the device memory space.

Split Kernel

The single kernel that updates an entire grid cell is broken into two separate kernels, one for x updates and one for y updates. This reduces the number of registers needed per thread and since each SMP has a fixed number of registers, it allows more threads to be scheduled per SMP. Having more threads per SMP means a larger pool of waiting threads and that more memory latency can be concealed through context-switching.

Coalescing Loads and Stores

All device memory loads and stores are aligned to appropriate byte boundaries and shared memory space is used to shift data if unaligned data is needed.

Increase Data Reuse

Each thread is given two grid cells to update instead of one, decreasing the number of threads launched but reducing the total number of memory transactions performed. Additionally data is transposed prior to updating so that all grid cells are always aligned to the correct byte boundaries. See table 4.2 for performance results using these optimizations.

Reduce Work

As mentioned earlier, the GPU version is performing double the work of the CPU or Cell versions. By changing the thread IDs to reference cell interfaces instead of grid cells, all of the duplicate work could be eliminated. This would increase the memory footprint of the algorithm as interface fluxes would need to be saved, and it would increase the number of kernel calls, but would likely provide a considerable performance boost.

	Tesla C870	
Kernel	time (s)	Speed-Up (x)
One Core Xeon	353.93	1.00
Naive	209.86	1.68
No Local Memory	27.13	13.05
Constant Memory	24.83	14.25
Split Kernel	12.11	29.23
Coalesce	11.31	31.29
Reuse Data	7.52	47.06

Table 4.2: Single precision performance improvements achieved on the Tesla C870 GPU (Section 4.1.1). The results are compared against performance on one Xeon core (system specifications given in Section 3.1.1). The most highly-optimized version gives a speed-up of just over 47× relative to the Xeon execution time and almost 30× relative to a naive GPU implementation.

4.3.3 Multiple GPUs

Multiple GPUs attached to the same host CPU may be used by starting separate pthreads and associating GPU contexts to them. A GPU may be assigned to a single thread or can be shared among several threads. In the host-centric memory layout there are no changes

necessary when moving to multiple GPUs. In the GPU-centric implementation updated cell information must be communicated between the device memories of the GPUs. In current GPUs there is no support for data transfers directly between devices so the host is used as an intermediary. Updated ghost cell data is passed to buffers in host memory and then to ghost cells on the neighbouring GPU. Performance when using one or two GPUs on the same machine is investigated in table 4.3.

	Tesla C870		Tesla T10	
Num GPUs	time (s)	Speed-Up (x)	time (s)	Speed-Up (x)
1	47.90	1.00	15.18	3.15
2	25.00	1.96	8.29	5.78

Table 4.3: Two Tesla T10 GPUs (system specs in Section 4.1.1) are compared to two Tesla C870 GPUs (system specs in Section 4.1.1). The test problem domain is a 1000×2000 grid for 1000 time steps with initial conditions from Section 2.11.1. The kernel used is the constant memory kernel from Section 4.3.2. The T10 is clearly much faster than the C870 and both of them show approximately linear scaling when moving from one GPU to two.

4.4 MPI Implementation

The CUDA-enabled code may be fit into the MPI communication framework described in Chapter 3. If the host-centric implementation is used then no changes are necessary since the memory layout remains unchanged. In the-GPU centric scheme it is necessary to ship data from the device memory into buffers in host memory and back, before and after performing the MPI communication phase.

4.4.1 MPI Packing (GPU-Centric Scheme)

Depending on whether the boundary is a row or a column, two different packing schemes are employed. This is because all data is stored in row-major format, transferring a row may be done by directly pulling each row from the GPU into the MPI send buffers. For a column, since CUDA does not support strided data transfers, the columns must be first extracted from the grid and placed into contiguous blocks in device memory prior to being transferred to the MPI send buffers in host memory. When receiving data, the same procedure operates in reverse, unpacking the MPI receive buffers and then putting them into position in device memory.

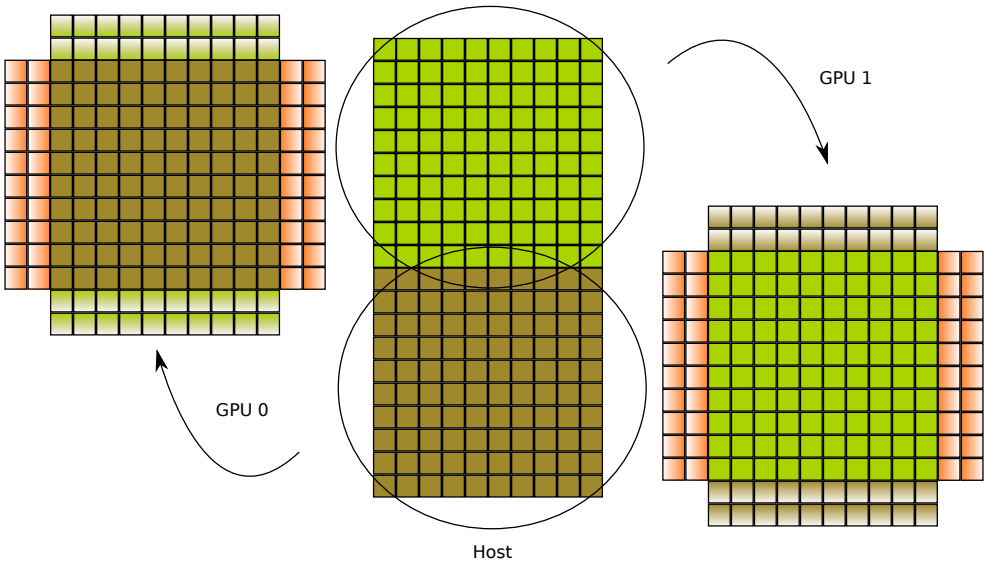


Figure 4.5: Visualization of the GPU centric data layout, the entire array is divided into equally sized blocks divided along rows. The blocks are stored on separate GPUs, only the boundary ghost cells are communicated back to the host and the other GPU at each time step.

4.4.2 MPI Scaling Results

The results of running the code on increasingly many Tesla GPUs is shown in Table 4.4 compared with performance on the same cluster without making use of the GPUs. The GPUs used in the scaling are the more powerful Tesla T10 GPUs and not the Tesla C870s used in the optimization table 4.2. Double precision results are not yet available, but the Tesla T10 GPUs support double precision at roughly an eighth the performance level of single precision.

	2x Xeon		2x Tesla T10	
N	time (s)	scaling (x)	time(s)	speed-up (x)
1	4096	1.00	197.2	20.79
2	1995	2.05	103.2	19.33
4	991.8	4.12	53.29	18.61
8	483.9	8.46	29.19	16.58
16	302.8	13.53	17.22	17.58

Table 4.4: MPI Scaling results for the GPU and Xeon systems in single precision. The problem is simulated on a 10000×10000 grid for 1000 time steps, with initial conditions of Section 2.11.2. This table compares the Nvidia Tesla T10 GPU to the Intel Xeon E5430 on a chip-by-chip basis. Each MPI node represents 2 quad-core Xeon processors and 2 Tesla GPUs, giving a comparison of 4 Xeon cores to 1 GPU. Speed-up is obtained by dividing the Xeon time for N nodes by the Tesla time for N nodes. Both the Tesla and the Xeon lose performance in the $N = 16$ case due to the small problem size, as seen by the poor scaling. Since the Xeon nodes are the host CPUs, these results are a direct representation of the improvement attainable by attaching GPU acceleration to nodes in a cluster.

4.5 Summary

In this Chapter an overview of Nvidia's Tesla GPU hardware and CUDA programming model was given. This data-parallel programming model was used to map the cell update kernel to the GPU hardware. In doing so it was demonstrated that knowledge of the hardware is important when trying to optimize for better performance. In the next section the Cell processor will be introduced and the solver will be mapped to its heterogeneous multi-core architecture.

Chapter 5

Cell Implementation

The Cell Broadband Engine Architecture (CBEA) developed jointly by IBM, Sony, and Toshiba is a microprocessor design focussed on maximizing computational throughput and memory bandwidth while minimizing power consumption. The first implementation of the CBEA is the Cell processor and it has already been used successfully in several large scale scientific clusters [2, 3], notably Los Alamos National Laboratory's Petaflop scale system Roadrunner [9]. Encouraged by these successes the Cell processor is explored as an accelerator for finite volume simulation of hyperbolic equations.

The strengths and challenges of the Cell architecture derive from its unconventional heterogeneous multi-core architecture which is introduced in more detail in Section 5.1. Developing software that can meet the challenges and exploit the strength of the Cell for floating point throughput is addressed in general in Section 5.2 and for the shallow water solver in particular in section 5.3.

The last stage of development is to incorporate a level of MPI process level parallelism on top of the optimized solver. This is discussed in Section 5.4 and allows the implementation to be run on SHARCNET's Cell development cluster (see Section 5.1.3). Scaling and performance at the MPI level is compared with performance of the reference CPU code on the Xeon cluster (see Section 3.1.1) used as a benchmark system.

5.1 Hardware Overview

The Cell processor achieves a high floating-point-throughput-to-power ratio by using a heterogeneous multi-core architecture. The Cell design may be thought of a as a network on

a chip with different cores specialized for different computational tasks (Figure 5.1). Since the Cell is designed for high computational throughput applications, eight of its nine processor cores are vector processors, called Synergistic Processing Elements (SPE). The other core is a more conventional PPC 970 CPU, called the Power Processing Element (PPE). It runs the operating system and is suitable for general purpose computing. However, in practice its main task is to coordinate the activities of the SPEs.

Communication on the chip is carried out through the Element Interconnect Bus (EIB) (Figure 5.1). It has a high bandwidth (204.8 GB/s) and connects the PPE, SPEs, and main memory through a four channel ring topology, with two channels going in each direction. For main memory the Cell uses Rambus XDR DRAM memory which delivers 25.6 GB/s maximum bandwidth on two 32 bit channels of 12.8 GB/s each.

5.1.1 Power Processing Element (PPE)

The PPE is a 64-bit processor with a 512KB L2 cache that uses IBM's POWER PC (PPC) instruction set. It may run any typical computer application using the PPC instruction set and operate as a general-purpose CPU. However, this is not a particularly effective use of the Cell's computational resources and in most Cell-specific applications the role of the PPE is limited to the coordination of the SPEs. For more in-depth detail about the PPE, the interested reader may refer to Scarpino's textbook [40] or IBM's whitepaper [36].

5.1.2 Synergistic Processing Element (SPE)

The SPE has a Synergistic Processing Unit (SPU), which is a 3.2GHz SIMD processor that operates on 128-bit wide vectors which it stores in its 128 128-bit registers. The SPU is a dual issuing processor, and, provided the 2 instructions satisfy certain constraints, it can issue 2 instructions per cycle. For instance it can issue a load operation at the same time as a multiplication instruction.

Each SPE has 256KB of on-chip memory called the Local Store (LS). The SPU draws on the LS for both its instructions and data. If data is not in the LS, it has no automatic mechanism to look for it in main memory. All data transfers between the LS and main memory are controlled via software-controlled direct memory access (DMA) commands. Each SPE has a memory flow controller that takes care of DMAs and operates indepently of the SPU. DMAs may also transfer data directly between the local stores of different SPEs.

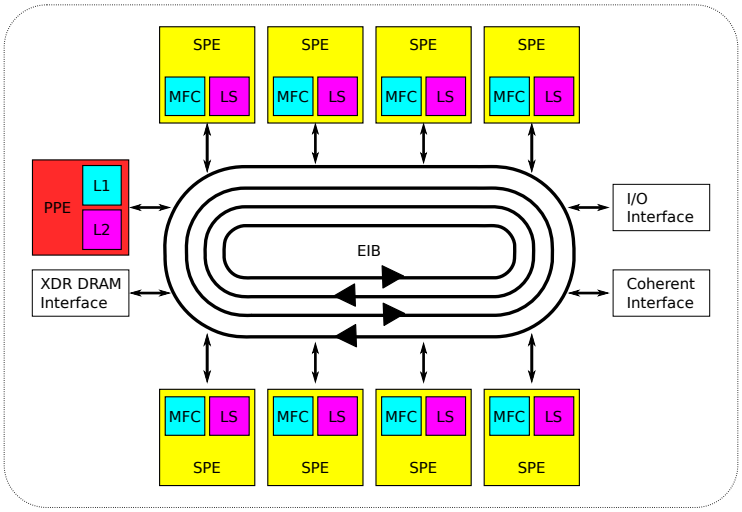


Figure 5.1: This schematic diagram shows the different elements of the cell hardware and how they interact with each other.

The SPU has only static branch predicting capabilities and has no other registers besides the 128 bit registers. It supports both single and double precision floating point instructions. However, hardware support for transcendental functions is only available for single precision reciprocal and reciprocal squareroot estimates. Single and double precision results require several Newton-Raphson method iterations in order to obtain the desired accuracy.

5.1.3 Cell Cluster

The SHARCNET Cell Cluster system 'prickly' that we used for code development and testing has the following specifications:

- eight QS22 blade servers
- each QS22 blade has two PowerXCell 8i processors @ 3.2GHz in a dual-socket configuration that gives access to 16GB of shared memory
- the eight QS22 blades have a GigE Interconnect

In addition, the system has four more traditional dual-Xeon server blades. Note that this setup is markedly different from Roadrunner nodes, since a relatively slow GigE link

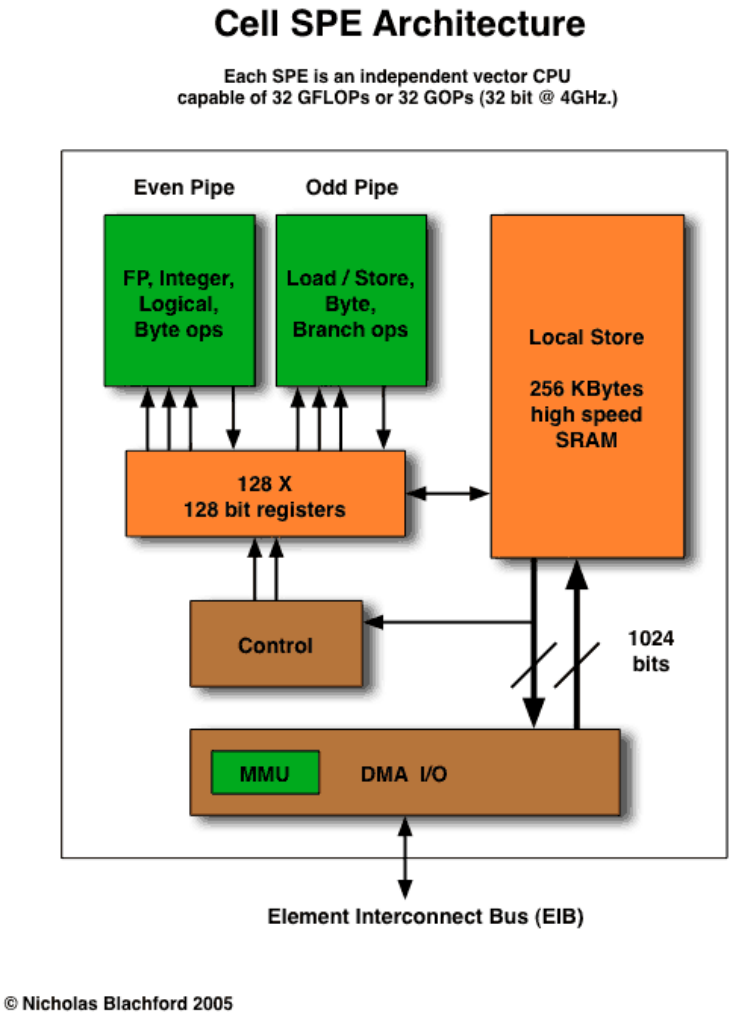


Figure 5.2: This diagram shows the components of the SPE: the LS, the DMA engine, and the SPU composed of the Control, the registers, and the two pipelines.

interconnects the Cell and Xeon blades in prickly, while Roadrunner has PCIe x8 links between QS22 and Opteron blades and Infiniband between nodes. Due to the relatively slow interconnect in prickly, we do not utilize the Xeon blades for MPI message passing, but use instead the Cell PPEs for this purpose (see Section 5.4).

5.2 Programming the Cell Processor

Programming complexity is one of the challenges of using the Cell Processor, as its design sacrifices programming ease for increased computational throughput. This section is not an introduction to programming the Cell processor, but merely gives a general idea of the complexities of Cell programming. For more in-depth information on programming Cell the interested reader is referred to Scarpino's textbook [40] or IBM's programming guide [7].

The Cell processor may be programmed in either C/C++ or fortran using a common API and set of libraries. A program consists of one PPE executable and potentially many SPE executables. The SPE executables are included into the PPE executable as libraries at compile time and may then be executed on an SPE using pthreads and library function calls.

Data is moved between main memory and the LS on each SPE by using library-provided memory get and put functions, as shown in Listing 4. Each takes an address in main memory, an address in the LS, and a size in bytes. A get sends data to the LS, while a put sends data to main memory. Both of these functions may only be called from an SPU executable, have a maximum transfer size of 16KB, and the addresses and sizes must be aligned to 16 bytes.

Listing 4 Direct Memory Access (DMA) functions that transfer data from main memory to the LS (`mfc_get`) or from the LS to main memory (`mfc_put`).

```
//volatile void * lsa : LS address
//uint64_t ea : effective address in main memory
//uint32_t size : size of the transfer in bytes
//uint32_t tag : transfer label
mfc_get(lsa, ea, size, tag,0,0);
mfc_put(lsa, ea, size, tag,0,0);
```

Calculations on the SPE are vectorized using vector intrinsic types and functions. The types introduced are a variety of vector types similar to the altivec vector types in the PPC architecture. The functions that operate on these vector types are calls to inline assembly that are inserted into the program code. This is similar to programming using altivec vector intrinsics or SSE assembly instructions. An example of multiplying and adding three floating point vectors using two separate instructions and then using the combined multiply-add instruction are shown in Listing 5.

Alignment is important on the SPE as it can only operate on inputs that are aligned to 16 byte boundaries. Any values not properly aligned must first be shuffled into the correct alignment before they may be used. This is done using a `spu_shuffle` vector intrinsic that accepts a bit mask and two vectors as input and shuffles them into the desired position (Listing 6). Additionally the SPU operates only on vectors so any scalar operations are performed as vector operations.

Listing 5 Operations on vector intrinsics are carried out by using SPU vector intrinsic functions, similar to the altivec vector instruction set.

```
//Performing d = a*b+c
vector float a,b,c,d;
//Assume a,b,c intialized (not shown)
//Using 1 multiply and 1 add
d = spu_mul(a,b);
d = spu_add(c,d);
//Or equivalently using 1 madd instruction
d = spu_madd(a,b,c);
```

Listing 6 The shuffle operation is very useful as it allows components from two 16 byte vectors to be combined into one new 16 byte vector. The two vectors are concatenated together and then bytes are selected from them by the values in the third input, a pattern vector. The values in the pattern vector correspond to the byte positions in `a` and `b` concatenated together. In the example 12 corresponds to the twelfth byte of `a` but 16 refers to the zeroth byte of `b`.

```
//perform c = spu_shuffle(a,b,pattern)
//Pattern to select a[3],b[0],b[1],b[2]
vector unsigned char pattern = {12,13,14,15,16,17,18,19,20,21,22
                                23,24,25,26,27 };
vector float a = {1.0f,2.0f,3.0f,4.0f};
vector float b = {-1.0f, -2.0f, -3.0f, -4.0f};
vector float c = {0.0f,0.0f,0.0f,0.0f};
c = spu_shuffle(a,b,pattern);
// c = [4.0, -1.0, -2.0, -3.0 ]
```

5.3 Mapping the Hyperbolic Solver to the Cell Processor

The Cell processor by itself offers two levels of parallelism: multi-core thread-level parallelism (the multiple SPEs) and SIMD vector-level parallelism (on the SPEs). Thread-level parallelism is exploited by partitioning the computational task of evolving each grid cell forwards in time, into data-parallel sub-tasks. This is done by dividing the computational grid into blocks of grid cells whose size is determined by the size of the LS. Workloads are assigned to each SPE by giving each SPE a collection of grid blocks to be updated. In order to efficiently send blocks into and out of the local stores a distributed memory layout is adopted, almost identical to the MPI distributed layout described in Chapter 3.

SIMD vector parallelism is used to accelerate the individual cell update computational kernels using the SPU vector intrinsics. In addition to SIMD parallelism, several other optimizations are performed on the compute kernel, the effects of which are discussed in Section 5.3.3.

5.3.1 Memory Layout

Two optimizations are done to minimize the effect of memory latency on performance. The first optimization is motivated by the fact that few large DMA transfers may be executed more efficiently than many small DMA transfers (Figure 5.3). Thus, when transferring a subblock out of a larger array it is inefficient to start a new DMA transfer for every row of the matrix, as the rows of the block are not contiguous in memory. A better approach is to store the entire array as an array of blocks, exactly as in an MPI distributed memory implementation. By doing this each block may be transferred into and out of a LS in far fewer DMA operations (Figure 5.4). The cost of doing this is that updated grid cell values must be communicated between blocks through the use of a halo of ghost cells surrounding each block. These halo communications may be efficiently implemented through the use of exchange buffers (see Section 5.3.2). The net result of the distributed layout is far fewer DMA transfers than in a single array layout.

The second optimization exploits the ability of the SPU to operate independently of its memory flow controller. This means that the SPUs calculations may be overlapped with memory transactions through the use of two buffers to hold data. Given two buffers, A and B, in the local store, while calculations are being performed on buffer A, data is being transferred into and out of buffer B. Once both computation and data transfer are complete, their roles reverse and buffer B is used for calculation and buffer A for

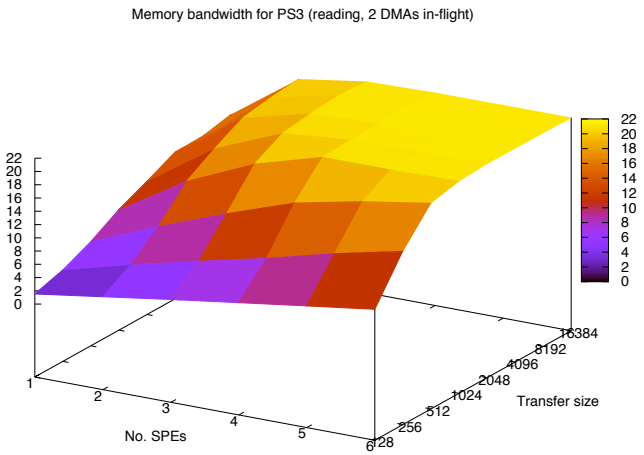


Figure 5.3: Bandwidth statistics from a PlayStation 3, demonstrating that higher bandwidth to main memory is achieved with larger DMA transfer sizes. The height of the plot is given in GB/s (Plot courtesy of Markus Stuermer, University Erlangen-Nuernberg).

data transfer. This double-buffering combined with the previous optimization virtually eliminates memory latency from the overall computation time, as shown in Table 5.1.

5.3.2 Neighbour Communication

The halo ghost cell communications introduced by the distributed memory layout are implemented in two separate ways and their performance is compared. The first is a naive implementation that utilizes the PPE to transfer all of the data and the second uses the SPEs and exchange buffers.

PPE Neighbour Exchange

In this implementation, at each time step the PPE waits until all SPEs have completed updating all of their grid blocks. The PPE then moves updated data into the ghost cells of each grid block while the SPEs stall. This implementation is inefficient for a number of reasons:

1. It fails to use the thread-level parallelism by not using the SPEs.

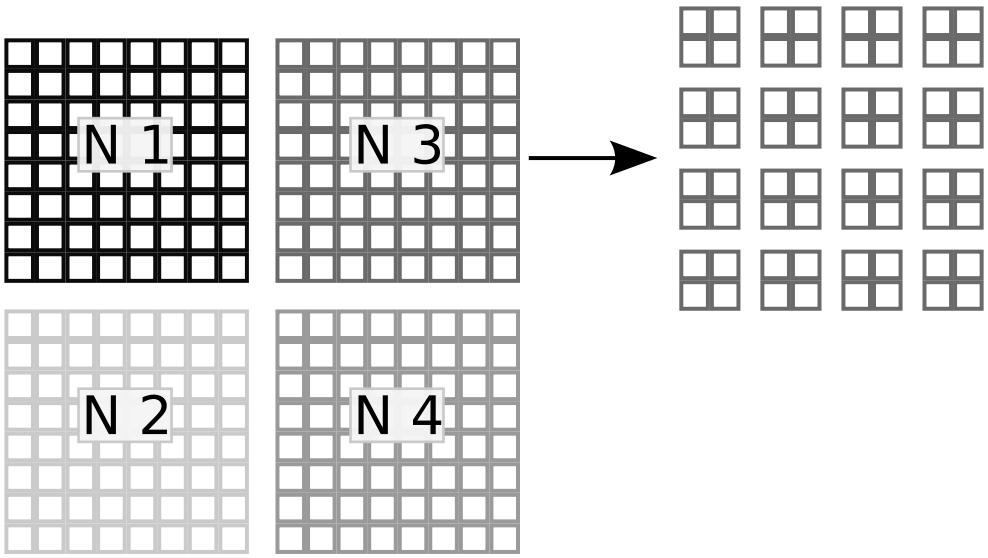


Figure 5.4: The blocks on the left represent an MPI distributed memory layout, the blocks on the right are the internal representation of Node 3's grid in local store-sized blocks.

2. The PPE has a lower bandwidth to main memory than an individual SPE.

3. It causes the SPEs to stall unnecessarily.

SPE Neighbour Exchange

The solution to these problems is to use the SPEs to directly send updated data into exchange buffers that are stored with the neighbouring block in main memory (Figure 5.5). When the block is next loaded into a local store, the exchange buffer is brought in as well. The SPE then shuffles the buffer values into the ghost cells and computation proceeds as normal. By doing this, the time taken to do neighbour communications is hidden behind calculations and no longer affects performance (Table 5.1).

5.3.3 Kernel Optimizations

Several different kernel implementations are provided in order of increasing complexity along with the corresponding performance data in Table 5.2.

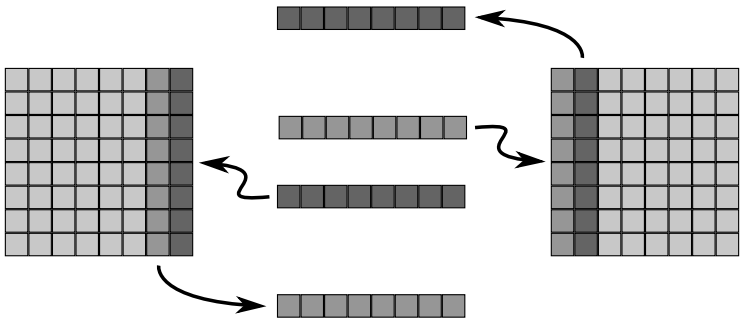


Figure 5.5: In this diagram two neighbouring blocks are shown, with one layer of ghost cells. The shading of the buffers indicates ownership, the darker buffers belong to the left block and the lighter buffers to the right block. While one buffer is receiving updated cell information from the neighbouring block, the other is loading data for the current time step. At the next time step the receiving and loading buffers alternate function.

		PPE Exchange	SPE Exchange	
single precision	Exchange Compute	19.3 5.88	Combined	6.02
double precision	Exchange Compute	51.55 22.53	Combined	22.82

Table 5.1: This table compares the performance of the PPE and SPE neighbour exchange schemes. The numbers quoted are execution time in seconds to do neighbour exchanges and computations respectively. Notice that in the SPE scheme the exchange time is combined with compute time but the time to do the exchanges is almost entirely hidden behind calculations.

Naive

In the naive kernel, the grid cell update compute kernel is directly copied from the CPU version with minimal alterations. This does not give very good performance as it ignores the SIMD nature of the SPEs and significant amounts of branching are present inside of nested loops (the SPE has very poor performance on branch prediction).

SIMD, Shuffle

The changes made in this version involve making use of the SPU vector intrinsics to SIMDize all floating point operations. Two difficulties arise while doing this. The first is that when updating in the x direction, neighbouring cell data is not aligned to 16 byte boundaries. (All data in the SPE local store is accessed in 16-byte blocks that are located at 16-byte boundaries.) In this implementation, data in the SPE local store is shuffled into the correct alignment directly before it is needed for calculation, using an efficient SPU intrinsic shuffling instruction. The second difficulty is that the branching statements used in the flux limiter calculations must be eliminated by using vector comparisons and selections (they work on four floating point numbers for single precision, and two for double precision). The code then calculates all branches for all entries and selects the correct branch at the end, which is much faster than the SPU's native static branch prediction. SPU intrinsics used for vector comparisons and selections include `spu_cmpgt` and `spu_sel`.

SIMD, Transpose

This implementation is similar to the previous one except that, instead of shuffling data in the interior loops, the entire grid block is transposed prior to calculation (using shuffling).

5.3.4 SPU Scaling

Here the parallel scaling of the optimized Cell code (with Shuffle optimization) is investigated on a single QS22 blade. On a QS22 blade, up to all 16 SPEs of the two Cell processors may be steered by one of the two PPEs. (The two Cell processors have access to the shared blade memory via a dual-socket configuration.) This allows the user to utilize the full processing power of the two Cell processors on the QS22 blade just by using the single-Cell code with a larger number of SPEs, and without the need for an additional layer of threading or an MPI implementation. Table 5.3 shows that excellent, nearly linear scaling is obtained for a speed-up test on a single QS22 blade (total problem size is kept constant throughout the test).

5.4 MPI Level Parallelism

The MPI framework of chapter 3 is adapted to the Cell case by redefining the packing schemes to account for the locally distributed memory layout. Prior to neighbour com-

	Single Precision		Double Precision	
Kernel	time (s)	Speed-Up (x)	time (s)	Speed-Up (x)
One Xeon Core	353.93	1.00	425.26	1.00
Naive	115.82	3.06	150.01	2.83
SIMD, Trans	6.59	53.71	25.28	16.82
SIMD, Shuf	5.95	59.48	22.81	18.64

Table 5.2: The kernel optimizations are compared against each other and a reference single-threaded CPU implementation running on one Xeon core @ 2.67GHz (Section 3.1.1). All Cell runs are done using one PowerXCell 8i with all 8 SPEs active. The grid size is 1000×1000 and the simulation runs for 1000 time steps, with initial conditions given in section 2.11.1. The naive implementation is 3× faster than the CPU in double and single precision. The shuffling kernel is seen to be slightly faster than the transpose kernel, giving a 60× speed-up in single precision and almost 20× in double precision. Based purely on the 16 byte vector width which holds two double precision variables, compared with four in single precision, one would expect double precision to be 2 times slower. However, in addition to this there is no hardware support for square roots and division in double precision. These operations thus take many more cycles in double precision than single precision. These two factors contribute to make the double precision version 3 times slower than the single precision.

	Single Precision		Double Precision	
SPEs	time (s)	speed-up (x)	time (s)	speed-up (x)
1	47.62	1.00	181.24	1.00
2	23.85	2.00	91.23	1.99
4	11.96	3.98	45.63	3.97
8	6.02	7.91	22.82	7.94
16	3.00	15.87	11.41	15.88

Table 5.3: Parallel scaling (constant total problem size) of optimized Cell code (with Shuffle optimization) on a single QS22 blade (runtime in seconds) for the test problem of Section 2.11.1. The scaling is almost perfectly linear. Grid size is 1000×1000.

munication the PPE gathers the updated neighbour cell values from the grid blocks and stores them in a buffer. These values are then traded with the neighbouring MPI node and then the PPE distributes the ghost cell data back into the grid blocks.

In Table 5.4 scaling results for the MPI code on the prickly cluster are shown. The prickly cluster has eight QS22 blades with two Cell processors each, and a GigE interconnect. The computation time (comp) scales well, but the inter-node communication time (comm) does not scale well, and since it makes up a significant part of the total time, the total scaling is also sub-optimal. Scaling for double precision calculations is somewhat better since communication takes up relatively less time there. The limitation of having a slow GigE interconnect becomes apparent here. Due to the slow interconnect, the PPEs are used to handle all MPI traffic, rather than letting faster processors on separate blades perform the MPI communication (like on Roadrunner nodes). The PPEs are slow, and the network is slow as well. Both these factors contribute to the large communication time, and the sub-optimal scaling results that follow from it. A faster interconnect (like Infiniband) would alleviate this, and would allow faster processors on separate blades to be used to perform the MPI communication. It has been demonstrated successfully on Roadrunner and similar systems that good scalability can be obtained in this way, but unfortunately the hardware currently available (prickly) does not allow these improvements to be explored.

	Single Precision				Double precision			
nodes	comm	comp	total	speed-up	comm	comp	total	speed-up
1	21.45	299.76	321.21	1.00	25.18	1132.91	1158.09	1.00
2	22.63	150.36	172.99	1.73	29.24	566.56	595.80	1.90
4	36.57	75.48	112.05	2.94	43.29	285.46	328.75	3.45
8	30.40	42.68	73.08	4.01	36.01	163.36	199.37	5.68

Table 5.4: Scaling results for the MPI Cell code on the prickly cluster. All times are in seconds. Each of the two PPEs on a parallel node steer eight SPEs. Grid size is 10,000×10,000 on the test problem of Section 2.11.2.

5.5 Summary

In this Chapter the Cell Processor was introduced, as well as some of the lower level memory and vector operations necessary for writing SPU code. These operations were used to implement the hyperbolic equation solver in a manner that almost completely eliminates memory latency through a double buffered transfer scheme into and out of main memory. The SPU code is optimized by making use of the SIMD vector intrinsics and using aggressive compile options.

Chapter 6

Comparisons

In this section the performance results on the three different architectures are analyzed by comparing:

- Runtime speed-up against a single-core x86 CPU implementation.
- GFLOPS performance with the peak theoretical throughput.
- Runtime scaling speed-up on MPI clusters.

All of the test results and numbers shown below omit the amount of time taken for initialization and saving data to disk.

6.1 Single Processor Runtime Comparison

The results in this section compare performance on the Cell and GPU to the runtime of both a single Xeon core and a four Xeon cores on a quad-core chip. All tests for a single chip are performed on a 1000×1000 grid, for 1000 time steps with the initial conditions of Section 2.11.1.

6.1.1 Single Precision

These results are interesting in that they demonstrate the efficacy of accelerator architectures in exploiting the data-parallel nature of the problem in comparison with a purely

sequential execution. The inclusion of the quad-core Xeon results allows for performance across architectures to be compared on a chip-by-chip basis.

The first thing to notice in the results of Table 6.1 is that adding thread level parallelism to a CPU code adds a significant performance boost and is critical to getting reasonable performance on a multi-core CPU. Of the accelerator architectures, the Tesla T10 GPU displays the fastest single precision performance, 4.75s. This is because it is the newest card being compared, and its peak performance numbers are 5× those of its closest rival, the Cell processor. The Cell processor with 8 SPEs activated comes in second place at 5.95s. Compared with a sequential implementation the two implementations run 75 and 60 times faster respectively. Compared with a quad-core implementation this is reduced by a factor of 4 to between 15 and 18 times faster.

The PlayStation 3 is included to provide a comparison with a commodity hardware product. In single precision it compares favourably with the others. However, it has poor double precision support. The Tesla C870 GPU is the generation previous to the Tesla T10 and it has roughly half the number of cores of the T10 and consequently its performance is also roughly half.

System	Time (s)	Speed Up
Xeon (1 Core)	353.93	1x
Xeon (4 Cores)	85.80	4.12x
PlayStation 3 (6 SPEs)	7.98	44.4x
Tesla C870 GPU	7.52	47.1x
QS22 Blade (8 SPEs)	5.95	59.5x
Tesla T10 GPU	4.75	74.5x

Table 6.1: Single precision comparison across architectures.

These performance improvements underline the fact that single precision performance is currently the major strength of accelerator architectures.

6.1.2 Double Precision

Double precision performance is compared between the Xeon and the Cell Processor. The Tesla T10 GPU also supports double precision however this has not yet been explored, however the peak performance of a T10 in double precision is an eighth of its single precision performance. If one simply estimates expected double precision performance as eight times

slower than single precision, it would be roughly 38s, about twice as slow as the Cell implementation. The Cell processor displays a roughly 18× speed-up over a single Xeon core and close to 5× over a quad-core Xeon. The PlayStation 3 is omitted here since its double precision performance is roughly eight times worse than the QS22, making it slower than the quad-core CPU.

System	Time (s)	Speed Up
Xeon (1 Core)	425.26	1x
Xeon (4 Cores)	106.87	3.98x
QS22 Blade (8 SPEs)	22.82	18.64x

Table 6.2: Double precision comparison across architectures.

6.2 Performance Efficiency

While GFLOPS are the most commonly quoted figure when discussing processor performance, it is not a particularly effective measure when comparing different architectures. The performance efficiency figures here should be interpreted as specific to each architecture and simply a rough estimate of how efficiently the implementation is making use of the hardware. In this work the aim was to achieve at least 20% of peak efficiency on all architectures. This was achieved for the Cell and GPU implementations, however the CPU version does not make use of SIMD vectors and so exhibits poorer performance. (For other peak performance comparisons see pp 51-52 in [24])

The calculation of GFLOPS is not easy across architectures as for instance the GPU performs square roots as one operation whereas they account for 15 operations on an x86 CPU. The GFLOPS are computed here by counting by hand the number of operations in the compute kernel, multiplying by the number of time steps and grid cells and then dividing by the amount of time it takes to perform the simulation. Other ways of computing GFLOPS are to use a standard reference, such as a Matlab implementation that simply counts the number of x86 instructions performed. This approach allows GFLOPS to be compared across architectures, but it does not help provide an accurate picture of how efficiently the hardware is being used. Another way is to use hardware counters when available to count the number of operations performed, which would be an effective alternative in this case. Peak performance numbers are obtained by multiplying the clock speed, the number of cores, the SIMD vector width, and the number of operations that may be completed per cycle.

The most successful implementation with regards to achieving peak performance was the Cell processor code with roughly 36% peak in single precision and 48% in double precision. These represent ultimately much higher efficiencies since the peak numbers are based on performing one multiply and one add every cycle and there are very few places where this is required in the actual compute kernel.

The GPU numbers quoted below are based on counting all calculations twice, since the GPU implementation computes inter-cell fluxes twice. Eliminating the additional calculations may improve GPU GFLOPS performance numbers considerably, if an efficient way can be found to do this.

System	Time (s)	GFLOPS	Peak	% Peak
Xeon (1 Core)	353.93	2.71	21.36	12.9%
Xeon (4 Cores)	85.80	11.17	85.44	13.0%
PlayStation 3 Cell (6 SPEs)	7.98	54.66	153.6	35.6%
Tesla C870 GPU	7.52	126.32	512	24.7%
QS22 Blade Cell (8 SPEs)	5.95	73.31	204.8	35.8%
Tesla T10 GPU	4.75	199.98	933	21.4%

Table 6.3: Table of performance efficiencies in single precision.

System	Time (s)	GFLOPS	Peak	% Peak
Xeon (1 Core)	425.26	2.25	10.68	21.0%
Xeon (4 Cores)	106.87	8.97	42.72	21.0%
QS22 Blade Cell (8 SPEs)	22.82	48.67	102.4	47.5%

Table 6.4: Table of performance efficiencies in double precision.

6.3 MPI Scaling

6.3.1 Single Precision

Comparison of single precision performance between an unaccelerated Xeon cluster (Section 3.1.1), the same cluster with GPU acceleration (Section 4.1.1), and the Cell QS22 Cluster (Section 5.1.3) is displayed in table 6.5. The test case is a radial dambreak solved on the

	2x Xeon		2x Tesla T10 GPU		2x PowerXCell 8i	
N	time (s)	scaling (x)	time(s)	speed-up (x)	time (s)	speed-up (x)
1	4096	1.00	197.2	20.79	321.21	12.75
2	1995	2.05	103.2	19.33	172.99	11.53
4	991.8	4.12	53.29	18.61	102.05	9.72
8	483.9	8.46	29.19	16.58	73.08	6.62
16	302.8	13.53	17.22	17.58	-	-

Table 6.5: Comparison of performance in single precision using the Xeon, GPU, and Cell. The clear winner here is the GPU as the Cell implementation is hampered by slow performance of the PPU as well as a higher latency interconnect. The speed-up is performance relative to the same number of Xeon Nodes.

physical domain $[-1,1] \times [-1,1]$ on a grid of $10^4 \times 10^4$ grid cells for 1000 timesteps of fixed size $\Delta t = 1e-5$ (Section 2.11.2).

Comparison of the results in Table 6.5 shows that both the Cell and GPU implementations provide performance better than the Xeon cluster with 16 nodes with only 1 node. The GPU cluster scales reasonably well, yet still sub-linearly but this is largely due to the problem size remaining constant across all tests. As more nodes are added, the local problem size becomes too small and during computation the GPU does not become fully saturated. A larger test problem This is similar to how the CPU performance drops at 16 nodes due to the smaller problem size. A larger test problem would likely remedy this problem or a comparison with a constant problem size per node would make a better demonstration of the scaling factors. In the case of the Cell Cluster, relatively poor scaling performance is achieved. The main factors contributing to this are the slow interconnect speed coupled with the slow PPU performance. For more discussion about this please refer to the earlier Cell specific results in Section 5.4.

6.3.2 Double Precision

Double precision performance is compared between the reference Xeon cluster and the prickly Cell cluster in Table 6.6. The double precision Cell Cluster still outperforms the Xeon cluster, however, the poor scaling quickly drags down performance. An algorithmic improvement to combat high interconnect latency is to reduce the number of communication steps performed. This may be accomplished by performing multiple time steps at each node on overlapping domains, before performing inter-node communications (as im-

	2x Xeon (8 Cores)		Cell (16 SPEs)	
nodes	time (s)	Scaling (x)	time (s)	Speed-Up (x)
1	4609	1.00	1158.09	3.97
2	2314	1.99	596.8	3.88
4	1139	4.04	328.6	3.47
8	566.1	8.14	199.4	2.84

Table 6.6: Performance results comparing double precision timing results of the Cell cluster (section 5.1.3) against the Xeon cluster (section 3.1.1). The same problem as in table 6.5 is used.

plemented for a cluster of GPUs in [39]). This comes at a cost of a somewhat increased total number of calculations and a larger amount of memory dedicated to MPI communications.

Chapter 7

Conclusions and Future Work

This thesis is not so much a final statement as a first preliminary exercise in using accelerator architectures for simulating hyperbolic PDEs on regular grids using explicit time stepping. This was done both using single accelerators and using them together to accelerate cluster calculations. The compute kernel used, while simplified somewhat, is quite representative of a typical compute kernel that might be used to do MHD or Euler equation simulations in 3D.

In all cases a clear performance advantage is demonstrated for accelerators. In single precision, comparing the accelerator results to one Xeon core, a speed-up of anywhere from 40× using the commodity PlayStation 3 up to 75× using the new Nvidia Tesla T10 GPUs is shown. Comparing against the quad-core multi-threaded CPU implementation, speed-ups of between 10× and close to 20× are obtained. In double precision only Cell results were pursued, and speed-ups of 18× compared with one core or 5× compared with a quad-core chip are obtained.

These results carry over into the parallel cluster domain, particularly in the case of the GPU system which displayed nice scaling properties. The Cell system did not scale so well, however, this is due to the combination of high latency GigE interconnect and slow PPU performance for MPI packing and calls.

In addition to the performance results, the programming techniques, communication patterns and hardware-specific optimizations were examined in detail, which provides a basis for further work in this area.

In general the experience of learning to program the Cell and GPU architectures came with a definite learning curve, however for someone with a background in optimizing x86 code the extensions to the new hardware are likely quite natural. The Nvidia CUDA model seemed quite successful at providing an intuitive abstraction layer for programming the GPU hardware. It is relatively easy to get an initial poor-performing code up and running and subsequently to apply increasingly complex hardware-specific optimizations. The Cell is far less friendly to the beginner and has a much steeper learning curve. As the reader could likely guess from the code examples in the Cell Chapter, bit-masks, bit-shifts, are commonly used tools and many Cell developers eventually simply move to assembly-level programming. This is a lower level of programming than most researchers would find desirable, however, once the initial hurdles are passed the Cell does provide fine low level control over both memory and compute operations, and as shown in Chapter 5, it scales perfectly as more SPEs are enabled because of the software controlled memory operations. To summarize the Cell offers very low level control over the hardware, while the CUDA model abstracts some of the complexities away. However in both cases knowledge of the hardware and its performance characteristics is essential to obtaining good performance.

Some interesting avenues for future research lie in taking the computational kernel used here and expanding it to the Euler or MHD equations in 3D. While 3D presents new challenges with regards to data structure, some of the lessons learned here should prove highly applicable. Another area of interest is in mixed precision methods, as accelerator architectures show significant benefits on single precision calculations. Knowing when one can use single precision without penalty has the potential to accelerate many types of applications.

The results support indications that clusters with heterogeneous multi-core architectures may become increasingly important for scientific computing applications in the near future, especially when one considers their typically low power requirements [2] and the fact that heterogeneous multi-core architectures may scale more easily to large numbers of on-chip cores than homogeneous chips with full x86 cores.

References

[1] http://gpgpu.org/. 8

[2] http://www.green500.org/index.php. 8, 45, 65

[3] http://www.top500.org. 9, 45

[4] *Astrogpu 2007*, 2007. 8

[5] Acceleware, www.acceleware.com. 8

[6] Mauricio Araya-Polo, Félix Rubio, Raúl de la Cruz, Mauricio Hanzich, José María Cela, and Daniele Paolo Scarpazza, *3D seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors*, Sci. Program. **17** (2009), no. 1-2, 185–198. 9

[7] Abraham Arevalo, Ricardo M. Matinata, Maharaja (Raj) Pandian, Eitan Peri, Kurtis Ruby, Francois Thomas, and Chris Almond, *Programming the cell broadband engine architecture: Examples and best practices*, IBM, 2008. 1, 4, 49

[8] Krste Asanovic, Ras Bodik, Jim Demmel, John Kubiatowicz, Kurt Keutzer, Edward Lee, George Necula, Dave Patterson, Koushik Sen, John Shalf, John Wawrzynek, and Kathy Yelick, *The landscape of parallel computing research: A view from berkeley*, Tech. Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. 3, 4

[9] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho, *Entering the petaflop era: The architecture and performance of Roadrunner*, SC08, 2008. 45

[10] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich, *Ray tracing on the Cell processor*, Sept. 2006, pp. 15–23. 9

[11] Tobias Brandvik and Graham Pullan, *Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware*, Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science **221** (2007), 1745–1748. 7

[12] ———, *Acceleration of a 3D Euler solver using commodity graphics hardware*, 46th AIAA Aerospace Sciences Meeting and Exhibit, 2008. 7

[13] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan, *Brook for GPUs: Stream computing on graphics hardware*, ACM TRANSACTIONS ON GRAPHICS, vol. 23, 2004, pp. 777–786. 5

[14] G. De Fabritiis, *Performance of the cell processor for biomolecular simulations*, Computer Physics Communications **176** (2007), no. 11-12, 660 – 664. 9

[15] Massimiliano Fatica, Antony Jameson, and Juan J. Alonso, *AIAA 2004–1090*, IAA Paper 2004-1090, 42nd Aerospace Sciences Meeting and Exhibit Conference, 2004. 7

[16] Mark S. Friedrichs, Peter Eastman, Vishal Vaidyanathan, Mike Houston, Scott Legrand, Adam L. Beberg, Daniel L. Ensign, Christopher M. Bruns, and Vijay S. Pande, *Accelerating molecular dynamic simulation on graphics processing units*, Journal of Computational Chemistry **30** (2009), no. 6, 864–872. 8

[17] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo, *Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes*, The Astrophysical Journal Supplement Series **131** (2000), 273–334. 6

[18] K. Germaschewski, J. Raeder, and D. Larson, *Geospace simulations on the Cell BE processor*, AGU Fall Meeting Abstracts (2008), C3+. 2

[19] Khronos Group, *Opencl*, www.khronos.org/opencl. 6

[20] Trond Runar Hagen, Martin O. Henriksen, Jon M. Hjelmervik, and Knut-Andreas Lie, *Geometric modelling, numerical simulation, and optimization*, ch. How to Solve Systems of Conservation Laws Numerically Using the Graphics Processor as a High-Performance Computational Engine, pp. 211–264, Springer Berlin Heidelberg, 2007. 7

[21] Trond Runar Hagen, Knut-Andreas Lie, and Jostein R. Natvig, *Solving the Euler equations on Graphics Processing Units*, Lecture Notes in Computer Science **3994** (2006), 220–227. 7

[22] T. Hamada and T. Iitaka, *The chamomile scheme: An optimized algorithm for n -body simulations on programmable graphics processing units*, arXiv:astro-ph/0703100v1 (2007). 8

[23] J. C. Hayes, M. L. Norman, R. A. Fiedler, J. O. Bordner, P. S. Li, S. E. Clark, A. ud Doula, and M.-M. Mac Low, *Simulating radiating and magnetized flows in multi-dimensions with zeus-mp*, arXiv:astro-ph/0511545v2 (2006). 6

[24] John L. Hennessy, David A. Patterson, and Andrea C. Arpaci-Dusseau, *Computer architecture: a quantitative approach*, Morgan Kauffman, 2007. 22, 23, 60

[25] North Star Imaging, www.4nsi.com. 8

[26] A. Kloeckner, T. Warburton, J. Bridge, and J.S.Hesthaven, *High-order discontinuous galerkin methods on graphics processors*, Journal of Computational Physics **submitted** (2009). 8

[27] H. Koskinen, E. Tanskanen, R. Pirjola1, A. Pulkkinen, C. Dyer, D. Rodgers, P. Cannon, J.-C. Mandeville, and D.Boscher, *Space weather effects catalogue*, Tech. report, ESA, 2002. 2

[28] Pijush K. Kundu and Ira M. Cohen, *Fluid mechanics 3rd edition:*, Elsevier, 2004. 15

[29] Randall J. Leveque, *Nonlinear conservation laws and finite volume methods for astrophysical fluid flow*, Computational Methods for Astrophysical Fluid Flow, 27th Saas-Fee Advanced Course Lecture Notes, Springer– Verlag. Available, 1998, pp. 1–160. 1, 2, 10, 15, 16

[30] Randall J. LeVeque, *Finite volume methods for hyperbolic problems*, Cambridge University Press, 2002. 2, 6, 12, 14, 15, 16, 21

[31] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa, *Shader metaprogramming*, HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (Aire-la-Ville, Switzerland, Switzerland), Eurographics Association, 2002, pp. 57–68. 6

[32] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron, *Scalable parallel programming with cuda*, Queue **6** (2008), no. 2, 40–53. xiv, 31

[33] Nvidia, *Nvidia cuda programming guide*. xiv, 1, 5, 33, 34, 35, 36, 37

[34] Lars Nyland, Mark Harris, and Jan Prins, *Gpu gems 3*, ch. Chapter 31. Fast N-Body Simulation with CUDA, pp. 677–696, Addison-Wesley, 2008. 8

[35] Stanford University Pande Lab, *Folding @ home*, http://folding.stanford.edu/, 05 2009, Statistics used were collected on Tues. May 26th, 2009. 8

[36] D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D.L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa, *Overview of the architecture, circuit design, and physical implementation of a first-generation Cell processor*, Solid-State Circuits, IEEE Journal of **41** (2006), no. 1, 179–196. 46

[37] J. Raeder, J. Dorelli, D. Larson, and B. Loring, *Physical, numerical, and computational challenges in modeling the geospace environment*, Numerical Modeling of Space Plasma Flows, ASP Conference Series, vol. 359, 2006. 6

[38] P. L. Roe, *Approximate riemann solvers, parameter vectors, and difference schemes*, J. Comput. Phys. **135** (1997), no. 2, 250–258. 12

[39] Martin Lilleeng Saetra, *Solving systems of hyperbolic PDEs using multiple GPUs*, Master's thesis, University of Oslo, 2007. 7, 63

[40] Matthew Scarpino, *Programming the cell processor: For games, graphics, and computation*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008. 4, 46, 49

[41] John Paul Shen and Mikko H. Lipasti, *Modern processor design: fundamentals of superscalar processors*, McGraw-Hill Professional, 2004. 22

[42] Engineering Space Studies Board and Physical Sciences, *Severe space weather events–understanding societal and economic impacts: A workshop report*, Tech. report, SSB, DEPS, 2008. 2

[43] John E. Stone, Jan Saam, David J. Hardy, Kirby L. Vandivort, Wen mei W. Hwu, and Klaus Schulten., *High performance computation and interactive display of molecular orbitals on GPUs and multi-core CPUs.*, Proceedings of the 2nd Workshop on General-Purpose Processing on Graphics Processing Units, ACM International Conference Proceeding Series, vol. 383, 2009, pp. 9–18. 8

[44] S. S. Stone, J. P. Haldar, S. C. Tsao, W. m. W. Hwu, B. P. Sutton, and Z. P. Liang, *Accelerating advanced MRI reconstructions on GPUs*, J. Parallel Distrib. Comput. **68** (2008), no. 10, 1307–1318. 8

[45] N. Thuerey, Goetz J., Stuermer M., S. Donath, Christian Feichtinger, Iglberger K., T. Preclik, and U. Ruede, *Parallel simulation and animation of free surface flows with the lattice Boltzmann method*, Lehrstuhl fuer Informatik 10 (SystemSimulation), 2008. 8

[46] Markus Uhlmann, *Shallow water equations tutorial*, `http://www.ciemat.es/sweb/comfos/personal/uhlmann/reports_comp/shallow/report.html`, 2007. 17

[47] Danny van Dyk, Markus Geveler, Sven Mallach, Dirk Ribbrock, Dominik Goeddeke, and Carsten Gutwenger, *HONEI: A collection of libraries for numerical computations targeting multiple processor architectures*, Computer Physics Communications **In Press, Corrected Proof** (2009), –. 6, 8

[48] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, *Scientific computing kernels on the Cell processor*, Int. J. Parallel Program **35** (2007), no. 3, 263–298. 7

[49] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick, *The potential of the cell processor for scientific computing*, CF '06: Proceedings of the 3rd conference on Computing frontiers (New York, NY, USA), ACM Press, 2006, pp. 9–20. 7

[50] F. Xu and K. Mueller, *Real-time 3D computed tomographic reconstruction using commodity graphics hardware*, Physics in Medicine and Biology **52** (2007), 3405–3419. 8