# Parallel Hyperbolic PDE Simulation on Clusters: Cell versus GPU

Scott Rostrup and Hans De Sterck

*Department of Applied Mathematics, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada*

**Abstract**

Increasingly, high-performance computing is looking towards data-parallel computational devices to enhance computational performance. Two technologies that have received significant attention are IBM's Cell Processor and NVIDIA's CUDA programming model for graphics processing unit (GPU) computing. In this paper we investigate the acceleration of parallel hyperbolic partial differential equation simulation on structured grids with explicit time integration on clusters with Cell and GPU backends. The message passing interface (MPI) is used for communication between nodes at the coarsest level of parallelism. Optimizations of the simulation code at the several finer levels of parallelism that the data-parallel devices provide are described in terms of data layout, data flow and data-parallel instructions. Optimized Cell and GPU performance are compared with reference code performance on a single x86 central processing unit (CPU) core in single and double precision. We further compare the CPU, Cell and GPU platforms on a chip-to-chip basis, and compare performance on single cluster nodes with two CPUs, two Cell processors or two GPUs in a shared memory configuration (without MPI). We finally compare performance on clusters with 32 CPUs, 32 Cell processors, and 32 GPUs using MPI. Our GPU cluster results use NVIDIA Tesla GPUs with GT200 architecture, but some preliminary results on recently introduced NVIDIA GPUs with the next-generation Fermi architecture are also included. This paper provides computational scientists and engineers who are considering porting their codes to accelerator environments with insight into how structured grid based explicit algorithms can be optimized for clusters with Cell and GPU accelerators. It also provides insight into the speed-up that may be gained on current and future accelerator architectures for this class of applications.

*Keywords:* parallel performance, Cell processor, GPU, hyperbolic system, code optimization

**PROGRAM SUMMARY**

*Program Title:* SWsolver
*Journal Reference:*
*Catalogue identifier:*
*Licensing provisions:* GPL v3
*Programming language:* C, CUDA
*Computer:* Parallel Computing Clusters. Individual compute nodes may consist of x86 CPU, Cell processor, or x86 CPU with attached NVIDIA GPU accelerator.
*Operating system:* Linux
*RAM:* Tested on Problems requiring up to 4 GB per compute node.
*Number of processors used:* Tested on 1-128 x86 CPU cores, 1-32 Cell Processors, and 1-32 NVIDIA GPUs.
*Keywords:* Parallel Computing, Cell Processor, GPU, Hyberbolic PDEs
*Classification:* 12
*External routines/libraries:* MPI, CUDA, IBM Cell SDK
*Subprograms used:* numdiff (for test run)
*Nature of problem:*
MPI-parallel simulation of Shallow Water equations using high-resolution 2D hyperbolic equation solver on regular Cartesian grids for x86 CPU, Cell Processor, and NVIDIA GPU using CUDA.
*Solution method:*
SWsolver provides 3 implementations of a high-resolution 2D Shallow Water equation solver on regular Cartesian grids, for CPU, Cell Processor, and NVIDIA GPU. Each implementation uses MPI to divide work across a parallel computing cluster.
*Running time:*
The test run provided should run in a few seconds on all architectures. In the results section of the manuscript a comprehensive analysis of performance for different problem sizes and architectures is given.

## 1. Introduction

Recent microprocessor advances have focused on increasing parallelism rather than frequency, resulting in the development of highly parallel architectures such as graphics processing units (GPUs) [1, 2] and IBM's Cell processor [3, 4]. 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, which are often referred to as accelerator architectures.

This paper investigates mapping high-resolution finite volume methods for nonlinear hyperbolic partial differential equation (PDE) systems [5] onto two different types of accelerator architecture, namely, IBM's Cell processor and NVIDIA GPUs. Performance on these architectures is then compared with performance on Intel x86 central processing units (CPUs). The accelerator architectures are investigated as both stand-alone computational accelerators and as components of parallel clusters. A high-resolution explicit numerical scheme is implemented for a relatively simple but representative model problem

in this class, namely, the shallow water equations. The numerical method is implemented on two-dimensional (2D) structured grids, for three architectures (x86 CPU, GPU, and Cell), and in parallel using the message passing interface (MPI).

A major goal of this paper is to compare the computational performance that can be obtained on clusters with these three types of architectures, for a 2D model problem that is representative of a large class of structured grid based simulation algorithms. Simulations of this type are widely used in many areas of computational science and engineering. Another important goal is to provide computational scientists and engineers who are considering porting their codes to accelerator environments with insight into techniques for optimizing structured grid based explicit algorithms on clusters with Cell and GPU accelerators, and into the learning curve and programming effort involved. It was also our aim to write this paper in a way that is accessible to computational scientists who may not have specific background in Cell or GPU computing.

There is extensive related work in the literature on the use of Cell processors and GPUs for scientific computing applications. Many of the papers in the literature deal with optimized implementations for either Cell processors [6, 7, 8, 9] or GPUs [10, 11, 12, 13, 14, 15]. Most of these papers deal with standalone or shared-memory hardware configurations, and do not involve distributed memory communication and MPI. Related work in the computational fluid dynamics area can be found in [16, 17, 18, 19]. Work that directly compares Cell with GPU performance is not widespread [20], and applications on parallel clusters with Cell and GPU accelerators have only more recently started to come to the forefront [21, 22]. Our paper goes further than existing work in comparing Cell with GPU performance on clusters with MPI, and these are relevant extensions of existing work since large clusters with accelerators are already being deployed and appear to be a promising direction for the future.

In our approach we have developed a unified code framework for our model problem, for hardware platforms that include distributed memory clusters with x86 CPU, Cell and GPU components. Several levels of parallelism are exploited (see Fig. 1). At the coarsest level of parallelism, we partition the computational domain over the distributed memory nodes of the cluster and use MPI for communication. We carry out performance tests on clusters provided by Ontario's Shared Hierarchical Academic Research Computing Network (SHARCNET, [23]) and the Juelich Supercomputing Centre (JSC, [24]). These clusters have two CPUs, Cell processors or GPUs per cluster node. At finer levels of parallelism, we exploit the parallel acceleration features provided by x86 CPUs, and Cell and GPU devices. The x86 CPUs we use feature four cores per CPU, and the cores provide single instruction, multiple data (SIMD) vector parallelism through streaming SIMD extensions (SSE). The Cell processors feature eight SIMD vector processor cores. The GPUs feature dozens of streaming multiprocessors with single instruction multiple thread (SIMT) parallelism. We exploit these different levels of parallelism through optimization of data layout, data flow and data-parallel instructions. Our development code is available on our website [25] and via the Computer Programs in Physics (CPiP) program library. We report runtime performance results for the various levels of optimization performed, and first compare Cell and GPU performance to performance on a single CPU core, as is customary in the literature. We also compare CPU, Cell and GPU performance on a chip-by-chip basis, on a node-by-node basis (i.e., on single cluster nodes without MPI), and on clusters (with MPI). Our GPU cluster results use NVIDIA Tesla GPUs with GT200 architecture, but we also include some results on recently introduced NVIDIA GPUs with the next-generation Fermi architecture. Our Fermi results are preliminary: we did not further optimize our code for the Fermi platform, but found it interesting to include results that show how a code developed on the GT200 architecture performs on Fermi. We conclude on the suitability of the accelerator architectures studied for the application class considered, and discuss the speed-up that may be gained on current and future accelerator architectures for this class of applications.

The rest of this paper is organized as follows. In Section 2 we briefly describe the class of scientific computing problems we target in this study, and the specific model problem we have implemented. Section 3 gives a brief overview of the aspects of the CPU, Cell and GPU architectures that are important for code optimization. Section 4 describes how our simulation code implementation was optimized for the architectures under consideration. Section 5 describes the clusters we use and compares performance of the optimized simulation code on the CPU, Cell and GPU platforms, and Section 6 formulates conclusions.

## 2. Hyperbolic PDE Simulation Problem

In this paper we target acceleration of a class of structured grid simulations in which grid quantities are evolved from step to step using information from nearby grid cells. One application area where this type of successive short-range updates are used is fluid and plasma simulation with explicit time integration, but there are many other use cases with this pattern in the computational science and engineering field. The particular problems we study are nonlinear hyperbolic PDE systems, which require storage of multiple unknowns in each grid cell, and which involve a relatively large number of floating point operations (FLOPS) per grid cell in each time step. (Note that, in this paper, we will write FLOPS/s when we mean floating point operations per second.) For ease of implementation and experimentation, we chose a relatively simple fluid simulation problem and a relatively simple but commonly used algorithmic approach. However, these choices are representative of a large class of existing simulation codes, and our approach can easily be generalized. Therefore, many of our findings carry over to this general class of simulation problems. In particular, we chose to investigate shallow water flow on 2D Cartesian grids, using a high-resolution finite volume method with explicit time integration [5].

Our code computes numerical solutions of the shallow water

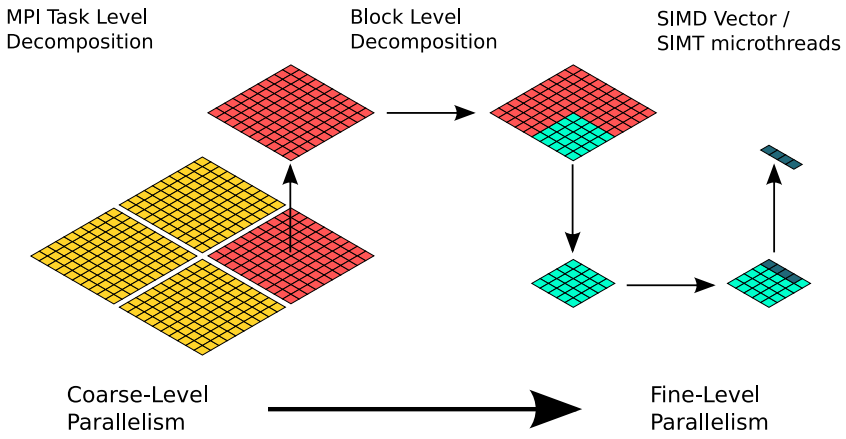


Figure 1: General overview of the different levels of parallelism exploited. At the coarsest level of parallelism (left) we partition the computational domain over the distributed memory nodes of the cluster and use MPI for communication between neighboring partitions. At the finest level of parallelism (right), we utilize SIMD vectors (CPU and Cell) or SIMT thread parallelism (GPU). At intermediate levels, we use Local Store-sized blocks of data (Cell) or thread blocks (GPU). The actual details of the different levels of parallelism depend on the platform and are represented more explicitly in Figs. 4 (CPU), 5 (Cell), and 7 (GPU).

equations, which are given by

$$\frac{\partial}{\partial t}\begin{bmatrix} h \\ hu \\ hv \end{bmatrix} + \frac{\partial}{\partial x}\begin{bmatrix} hu \\ hu^2 + \frac{gh^2}{2} \\ huv \end{bmatrix} + \frac{\partial}{\partial y}\begin{bmatrix} hv \\ huv \\ hv^2 + \frac{gh^2}{2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \quad (1)$$

where $h$ is the height of the water, $g$ is gravity, and $u$ and $v$ represent the fluid velocities. The gravitational constant $g$ is taken to be one in the test simulations reported in this paper. The shallow water system is a nonlinear system of hyperbolic conservation laws [5], and given an initial condition, a 2D domain and appropriate boundary conditions, it describes the evolution in time of the unknown functions $h(x, y, t)$, $u(x, y, t)$ and $v(x, y, t)$. We discretize the equations on a rectangular domain with a structured Cartesian grid, and evolve the solution numerically in time using a finite volume numerical method with explicit time integration [5]. In what follows we write $U = [h \quad hu \quad hv]^T$. We update the solution in each grid cell $(i, j)$ using an explicit difference method. One approach to this problem is to use so-called unsplit methods of the form

$$U^{n+1}_{i,j} = U^n_{i,j} - \frac{\Delta t}{\Delta x}(F^n_{i+\frac{1}{2},j} - F^n_{i-\frac{1}{2},j}) - \frac{\Delta t}{\Delta y}(G^n_{i,j+\frac{1}{2}} - G^n_{i,j-\frac{1}{2}}). \quad (2)$$

Here, $i$, $j$ are the spatial grid indices and $n$ is the temporal index, and $F$ and $G$ stand for numerical approximations to the fluxes of Eq. (1) in the $x$ and $y$ directions, respectively. The vector $U^n_{i,j}$ is the vector of three unknown function values in cell $(i, j)$ at time level $n$. Alternatively, one can consider a dimensional splitting approach

$$\begin{aligned} U^*_{i,j} &= U^n_{i,j} - \frac{k}{\Delta x}\left(F^n_{i+\frac{1}{2},j} - F^n_{i-\frac{1}{2},j}\right), \\ U^{n+1}_{i,j} &= U^*_{i,j} - \frac{k}{\Delta y}\left(G^*_{i,j+\frac{1}{2}} - G^*_{i,j-\frac{1}{2}}\right), \end{aligned} \quad (3)$$

and this is the method we chose to implement. An advantage of the dimensional splitting approach is that Eq. (3) leads to accuracy that is in practice close to second-order time accuracy (see [5], pp. 386, 388, 444) without the need for a two-stage time integration. We use an expression for the numerical fluxes $F$ and $G$ ([5], p. 121, Eqs. (6.59)-(6.60)) that is second-order accurate away from discontinuities, utilizing a Roe Riemann solver ([5], p. 481) with flux limiter. The update formula for any point $(i, j)$ on the grid involves values from two neighboring grid points in each of the up, down, left and right directions, leading to a nine-point stencil for grid cell updates. For parallel implementations, this means that two layers of ghost cells need to be communicated between blocks after each iteration [5]. For numerical stability, the timestep size is limited by the well-known Courant-Friedrichs-Lewy condition, which implies that the timestep size must decrease proportional to the spatial grid size as the grid is refined. Grid cell updates may be computed in parallel and the arithmetic density per grid point is high (see Table 1), which, along with the structured nature of the grid data, makes this algorithm a good candidate for acceleration on Cell or GPU. The arithmetic density is computed by calculating the minimum number of floating point operations necessary to update all grid cells. That is, flux calculations are counted once per cell interface and the calculation of intermediate results that may be reused is not counted multiple times in the number of operations. This is a flat operation count: no special consideration is given to square root or division operations. It is useful to point out that, among the 360 FLOPS per grid cell, there are 2 square roots and 16 divisions. This is important since square roots and divisions may be evaluated in software or on a restricted number of processor sub-components on Cell and GPU devices (depending on the precision, see below), so actual arithmetic density on those platforms may effectively be higher than what is reported in Table 1. Note that our algorithm has such a high effective arithmetic density for several reasons: we have a coupled system of three PDEs (3×9=27 values enter into the formula to update each grid value, instead of just 9 for uncoupled equations solved with the same accuracy), the

system is highly nonlinear and requires sophisticated numerical flux formulas based on Riemann solvers ([5], p. 481), and the flux formulas involve square roots and divisions. Since our algorithm is implemented in two passes, the minimum number of memory operations is each grid cell being read twice, and then stored twice, in each timestep.

| FLOPS per grid cell | 360 | |
|---|---|---|
| Precision | SP | DP |
| Memory per grid cell | 48 Bytes | 96 Bytes |
| FLOPS/Byte | 7.5 | 3.75 |

Table 1: The compute kernel requires a minimum of 7.5 and 3.75 FLOPS per Byte of data loaded or stored in single precision (SP) and double precision (DP), respectively.

The test problem used for the simulations in this paper has initial conditions

$$h(x, y, 0) = \frac{1}{4}\left(\frac{x}{L} + \frac{y}{W}\right) + 1,$$
$$u(x, y, 0) = v(x, y, 0) = 0,$$

on a square domain $\Omega = [-L, L] \times [-W, W]$. Boundary conditions are perfect walls [5].

As noted above, we have chosen a relatively simple set of hyperbolic equations for this optimization and performance study paper. However, more complicated hyperbolic systems, including the compressible Euler and Magnetohydrodynamics equations which are widely used for fluid and plasma simulations, can be approximated numerically by the same or similar methods, and extension of our approach from 2D to 3D body-fitted structured grids or to unsplit explicit methods is also not difficult. We have deliberately chosen this relatively simple model problem for this paper because its simplicity allows us to explain the essential aspects of optimizing structured grid problems for Cell and GPU architectures, without being distracted by non-essential details of a more complicated application. Similarly, readers can easily investigate and comprehend the details of our implementation in the simulation code that we provide, without being overwhelmed by complications of the application. However, the approach and conclusions of our paper carry over directly to a broad class of important fluid and plasma simulation problems and algorithms.

## 3. Hardware Description

In this section we give a brief overview of the aspects of the x86 CPU, IBM Cell and NVIDIA GPU architectures that are important for optimization of our algorithmic approach.

### *3.1. Intel Xeon CPU*

The Intel Xeon E5430 processors have four cores, and the particular features that are important in the context of this paper are the cache-based architecture and the SIMD vector parallelism provided through the streaming SIMD extensions (SSE) mechanism. Each core has SIMD vector units that are 128 bits wide and are capable of performing four single precision calculations or two double precision calculations at the same time. While compiler features are being developed that can automatically exploit this functionality, we found that for good performance it is at present still necessary to explicitly call intrinsic library functions that access these SIMD capabilities efficiently (see Section 4.1). The Intel Xeon E5430 quad-core processors used in this study have a clockrate of 2.66GHz, a 12MB L2 cache, and each core has a 16KB L1 cache.

### *3.2. Cell Processor*

The Cell Broadband Engine Architecture (CBEA), developed jointly by IBM, Sony, and Toshiba is a microprocessor design focused on maximizing computational throughput and memory bandwidth while minimizing power consumption [3, 4]. The first implementation of the CBEA is the Cell processor and it has been used successfully in several large-scale scientific computing clusters [26, 27], notably Los Alamos National Laboratory's petaflop-scale system Roadrunner [28].

The heterogeneous multi-core design of the Cell processor may be thought of as a network on a chip, with different cores specialized for different computational tasks (Fig. 2). Since the Cell processor is designed for high computational throughput applications, eight of its nine processor cores are vector processors, called synergistic processing elements (SPEs). The other core is a more conventional (and relatively slow) CPU, called the PowerPC processing element (PPE). The PPE has a 64-bit processor (called the PowerPC processing unit (PPU)) as well as a memory subsystem containing a 512KB L2 cache. The PPU 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. 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 (Fig. 2). 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.

The SPE is the main computational workhorse of the Cell Processor. It has a 3.2GHz SIMD processor (called the synergistic processing unit (SPU)) that operates on 128-bit wide vectors which it stores in its 128 128-bit registers.

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 independently of the SPU. DMAs may also transfer data directly between the local stores of different SPEs.

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

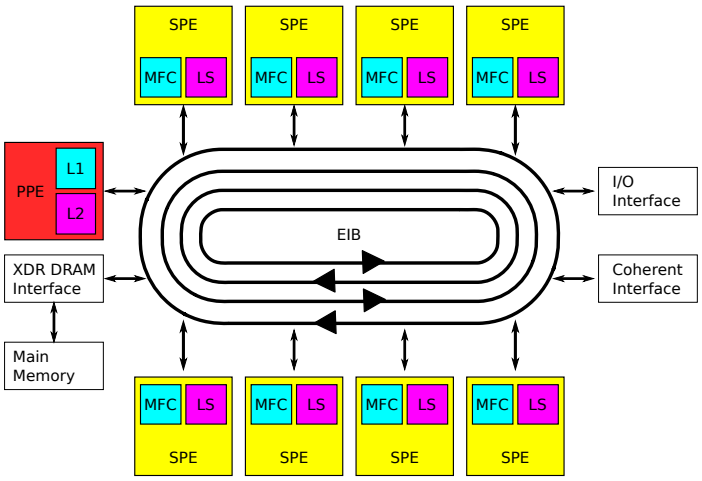


Figure 2: Hardware diagram of the Cell processor. The 8 SPEs are the SIMD vector processors, the PPE is the PowerPC CPU, and the rings illustrate the four-channel ring topology of the Element Interconnect Bus. Also shown is the XDR DRAM memory interface to the Cell blade main memory, and the I/O interfaces which allow two Cell processors on one blade to share SPEs.

is only available in the form of reduced precision approximations of reciprocals and reciprocal square roots. Full single and double precision transcendentals must be evaluated in software.

Most Cell tests in this paper are performed on the cluster described in Section 5.1.2 with PowerXCell 8i processors, but we also include some tests on Cell processors in Sony's PlayStation 3, which are an earlier generation of the Cell processor with less hardware support for double precision calculations, and which have two of their SPEs disabled.

### *3.3. NVIDIA GPUs and CUDA Programming Model*

GPUs are not, as their name would suggest, solely used for graphics applications: NVIDIA Tesla GPUs have evolved to be general purpose high-throughput data-parallel computing devices [1]. The GPU attaches to a host CPU system via the PCI Express bridge as an add-on computational accelerator with its own separate DRAM (up to 4GB), which we call GPU global memory, and some specialized on-chip memory. Programs may be developed to make use of the GPU by using NVIDIA's CUDA programming model which provides extensions to the C programming language [29, 30]. (CUDA stands for compute unified device architecture.) The GPU is incorporated into a program's execution by calling what is known as a kernel function from within the CPU host code. A kernel is defined similarly to a normal C function but when called, a user-specified number of threads are spawned, each of which executes the kernel function on the GPU in parallel. The threads are mapped into groups of up to 512 called thread blocks, and the threads within a thread block are grouped into smaller groups of 32 threads called warps.

The NVIDIA GT200 architecture uses a hierarchal organization of thread processors and memory to implement a single instruction multiple thread (SIMT) streaming multiprocessor design, shown schematically in Fig. 3. Threads are farmed out to the hundreds of identical scalar processors (SPs) on the GPU. (The Tesla T10 GPU we use has 240 SPs.) Ideally, many more threads are spawned than the number of SPs. The SPs are organized into blocks of eight, called streaming multiprocessors (SMs). Each SM in addition to the eight SPs has a special function unit (SFU) for computing transcendental functions and a double precision unit (DP) which can also act as an SFU. Each SM also has a block of local memory called shared memory visible to all threads within a thread block, and a scheduling unit used to schedule warps. The GPU is capable of swapping warps into and out of context without any performance overhead. This functionality provides an important method of hiding memory and instruction latency on the GPU hardware.

When a kernel function is called, it is initiated on the GPU by mapping multiple thread blocks onto the SMs. Thread blocks are divided on the SMs into groups of 32 threads called warps and execution proceeds in a SIMT fashion within each warp. Threads within a thread block may be synchronized if necessary. However, there is no generally efficient mechanism for synchronization across the thread blocks within a kernel function.

#### *3.3.1. Fermi GPU*

The Fermi architecture, released in the spring of 2010, is NVIDIA's next-generation GPU. It is the successor of the GT200 architecture described above, and is the first in which NVIDIA focussed on general-purpose computation performance. The main improvements to note for this paper are the full IEEE floating point compliance, the improved double precision performance, and the addition of a cache hierarchy. The double precision performance on Fermi is half the speed of single precision, bringing it in line with most CPUs. The addition of a cache hierarchy, consisting of a global L2 cache, as well as a per-SM L1 cache gives more flexibility in non-uniform memory accesses. The Fermi C2050 features 448 SPs organized in 14 SMs. Each SM has 32 SPs, 16 DPs, and 4 SFUs. The Fermi C2050 features a 1.15 GHz clock speed which is slower than the Tesla T10's 1.30 GHz. For the rest of the de-

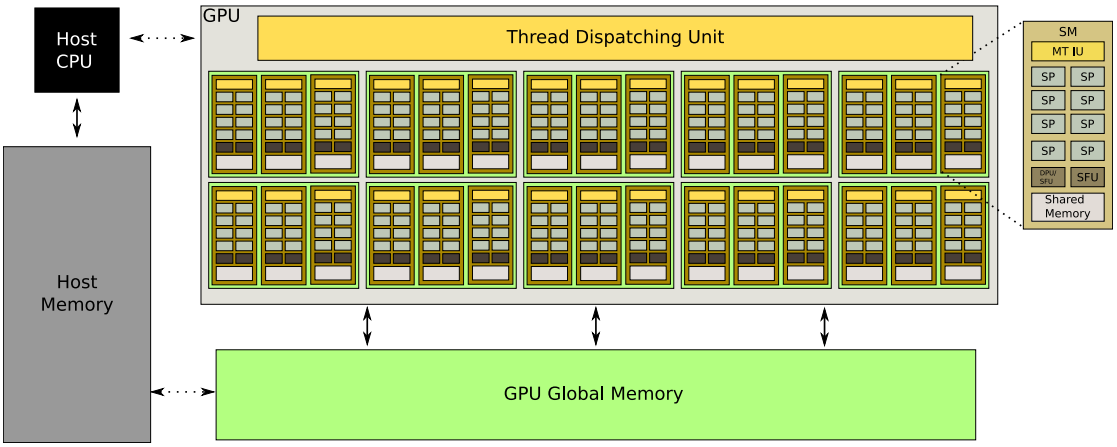


Figure 3: Hardware diagram of a Tesla T10 GPU. The GPU consists of 30 Streaming Multiprocessors (SMs), each of which has 8 Scalar Processors (SPs). The GPU has 4 GB of GPU global memory. The GPU board is connected to the CPU board (with the node host memory) using a PCI Express bridge.

tails of the new architecture we refer to NVIDIA's white paper [31]. Though the underlying architecture has gone through significant change, the CUDA programming interface remains the same, allowing code to simply be recompiled with a different architecture specification.

Most GPU tests in this paper are performed on the 'angel' S1070 GPU cluster described in Section 5.1.2 with Tesla T10 GPUs of GT200 architecture, but we also include some tests on Tesla GPUs with the earlier generation G80 architecture, which have no double precision unit and 128 SPs. Additionally we have performed preliminary tests on a Fermi C2050 card (GTX480 with 3GB GPU global memory), allowing performance to be compared across three generations of GPU hardware (G80, GT200 and Fermi).

## 4. Code Development and Optimization

We have developed a code framework with three hardware-specific back-end implementations for x86 CPU, Cell, and GPU. The framework makes use of MPI to distribute computations across cluster nodes. In this section we explain the main code optimizations used to accelerate the CPU, Cell and GPU implementations and also describe the MPI parallel implementation. Some additional details on our approach can be found in [32]. See also [33] for a preliminary discussion of our Cell processor results. The interested reader may retrieve our research code on our website [25] or via the CPiP program library to study the complete details of our implementation.

### *4.1. CPU Implementation*

For the x86 CPU implementation, we optimized our CPU code to incorporate cache optimization and SIMD vector optimization via SSE intrinsics. Cache optimization is obtained via a simple cache-blocking technique [34] for updates in the $x$ and $y$ directions. The global loops over the $x$ and $y$ directions are replaced with an outer loop over cache-sized blocks of the computational domain. By doing this we minimize the impact of L2 cache misses on performance. We choose cache block size based on the size of the L2 cache and the number of cores which will be sharing it, and then allocate a fixed percentage of the L2 cache to each core. SIMD optimization is implemented using hand-coded SSE-vector intrinsic funtions that access the SIMD capabilities efficiently. This is similar to the Cell code optimizations described below in the subsection on 'SIMD, shuffle' (Section 4.2.2). Unaligned data must be shuffled into 16-byte aligned vectors prior to computation. Also, we eliminate branch statements for the flux limiters. Since this is an important aspect of our Cell optimization approach, we prefer to discuss it in detail in Section 4.2 on Cell optimization below. The SSE-optimized and cache-optimized CPU code on one CPU core is the baseline with which we compare Cell and GPU code performance below. For comparison, we will also include results for our initial unoptimized CPU implementation in the numerical results of Section 5. All implementations are compiled using the Intel C Compiler version 11.0 at optimization level -O3. See Fig. 4 for an illustration of the different levels of parallelism in our CPU code.

### *4.2. Cell Implementation*

Each Cell processor has eight SPEs for data processing. To divide the calculations among the eight SPEs, the computational domain assigned to a particular Cell processor is further decomposed into blocks of grid cells (see Fig. 5) which we call LS blocks because their size is determined by the size of the SPE's local store (256KB). Note also that we have, in fact, separate arrays for $h$, $u$ and $v$ (not shown in Fig. 5). The PPE assigns workloads to SPEs by giving each SPE a collection of these LS blocks to update. The PPE steers the SPEs through pthreads and calls to Cell-specific library functions. SIMD vector parallelism is used to accelerate the computational kernel that updates the cells via SPU vector intrinsics.

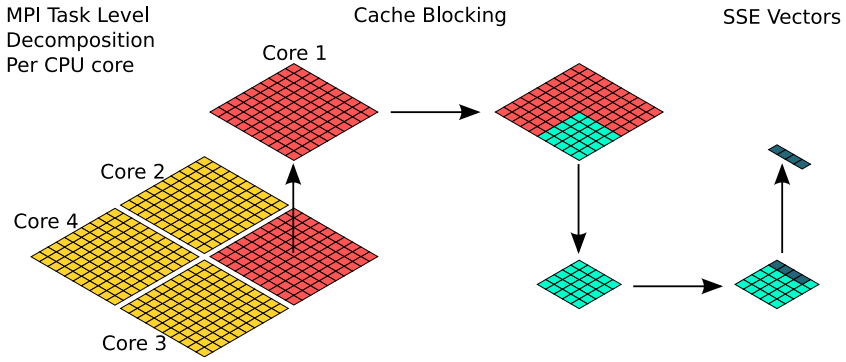


Figure 4: Different levels of parallelism and actual data layout in the CPU implementation of our code. We employ one MPI process per CPU core. On a single core, computation is done per cache block-sized block of data. At the lowest level, SSE SIMD vector intrinsic functions are used.

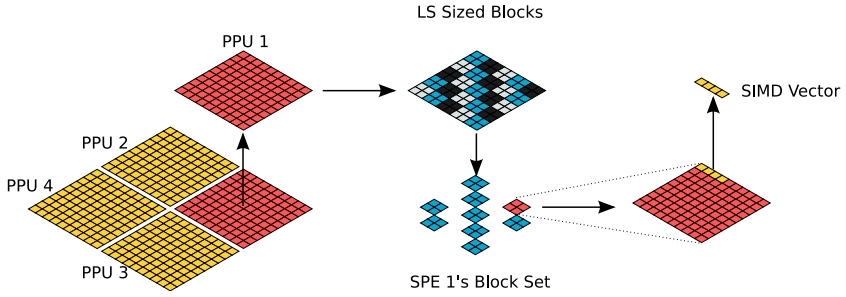


Figure 5: Different levels of parallelism and actual data layout in the Cell implementation of our code. The blocks on the left-hand side represent a logical decomposition of the domain into 4 tasks at the MPI level. Since each Cell blade server node has 2 PPUs, we have 2 MPI processes per blade server. The blocks in the middle of the diagram display the actual data layout in the Cell blade main memory for the part of the domain assigned to one MPI task (i.e., one Cell processor or one PPU). These blocks are sized so that exactly two of them fit into the local store of an SPE, and we call them LS blocks. This Cell blade main memory layout allows us to hide latency between the Cell blade main memory and the eight SPE local stores, by transferring one block of data, while at the same time doing calculations on another block. Also illustrated on the right is the 128 bit wide SIMD vector level parallelism of the SPEs.

| | PowerXCell 8i Cell processor | | | |
|---|---|---|---|---|
| | single precision | | double precision | |
| | time (s) | speed-up (×) | time (s) | speed-up (×) |
| one Xeon core | 162.50 | 1.00 | 192.44 | 1.00 |
| Cell naive | 131.88 | 1.23 | 154.12 | 1.25 |
| Cell SIMD, transpose | 7.30 | 22.26 | 26.66 | 7.22 |
| Cell SIMD, shuffle | 6.72 | 24.18 | 24.41 | 7.88 |

Table 2: The runtimes (in seconds) of the three kernel optimizations running on a PowerXCell 8i using all eight SPEs are compared to the SSE-optimized x86 CPU implementation running on one Xeon core. This test uses a $1000 \times 1000$ grid with $L = 10$, $W = 10$ and 1000 timesteps.

#### *4.2.1. Memory Layout*

In Cell processors data transfers may be decoupled from computations. A standard double-buffering approach is used to keep the SPE working on one LS block of data, while the DMA engine is transferring another LS block of data. We adopt a distributed memory layout (Fig. 5) for the data assigned to a given Cell processor, which stores each LS block in a continuous section of Cell blade main memory. This allows the total time spent transferring data into and out of the SPE's local store to be reduced, since bandwidth between the Cell blade's main memory and the SPE's LS is maximized when data transfers are large contiguous blocks. Updated ghost cell values must be communicated in between grid blocks after each $x$ or $y$ sweep, exactly as in the MPI implementation. This may be implemented efficiently through the use of exchange buffers in the Cell blade main memory associated with each LS grid block. Each SPE performs the $x$ or $y$ update calculation on all of the grid cells in the LS block, before sending updated values to the neighboring LS grid blocks' exchange buffers in the Cell blade main memory. This is more efficient than having the PPE update the ghost cells for the LS blocks in the Cell blade main memory. By using the distributed memory layout and a double-buffering approach to transfer data into and out of the local stores, we were able to almost completely eliminate the effect of memory latency and bandwidth from the overall computation runtime.

One other factor in designing the Cell blade main memory layout is that all data in the SPE local store is accessed by the SPE in 16-byte vectors that are located at 16-byte boundaries. In order to ensure that all matrix rows begin on 16-byte vector boundaries, we pad grid block rows in the Cell blade main memory with enough extra data to ensure that proper alignment is maintained, and transfer the padded arrays between Cell blade main memory and SPE local stores.

#### *4.2.2. Kernel Optimization*

Timing results for three different SPE kernel implementations are presented in Table 2. The first kernel is simply a naive porting of the unoptimized x86 CPU grid cell update kernel code to the Cell architecture. The two other kernels represent two different ways of dealing with unaligned data in the SPE LS, optimized for the Cell hardware.

*Naive*

In the naive kernel, the grid cell update compute kernel is directly copied from the unoptimized x86 CPU version, with minimal alterations. This does not give good performance since 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 using the SPU vector intrinsics to SIMDize all floating point operations. Two difficulties arise while doing this. The first is that neighbouring cell data needs to be aligned in groups at 16-byte boundaries in LS memory. Consider update equations (3). In single precision, for example, four adjacent cells can be updated simultaneously by the SIMD units. For efficient execution, all values for $h$ (and $u$ and $v$) for the horizontal and vertical neighboring cells need to be aligned in groups of four at 16-byte boundaries before invoking the SIMD calculations. If the data is stored by $x$-rows in LS memory, then it is properly aligned for the updates in the $y$ direction, but not in the $x$ direction (see Fig. 6). In this kernel implementation, the four-vectors of aligned data required for $x$-updates are created by shuffling, directly before they are needed for calculation, using efficient SPU intrinsic shuffling instructions.

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 executes all branches for all entries and selects the correct result for each entry separately at the end by a masking intrinsic. This is much faster than the SPU's native static branch prediction.

*SIMD, Transpose*

This kernel implementation is similar to the previous one except that the 16-byte alignment problem is dealt with by transposing the entire grid block in the LS before and after doing the $x$ direction sweeps (by using the efficient intrinsic shuffling instructions).

Table 2 shows timing results for the various kernel optimizations, compared to the serial CPU reference code running on

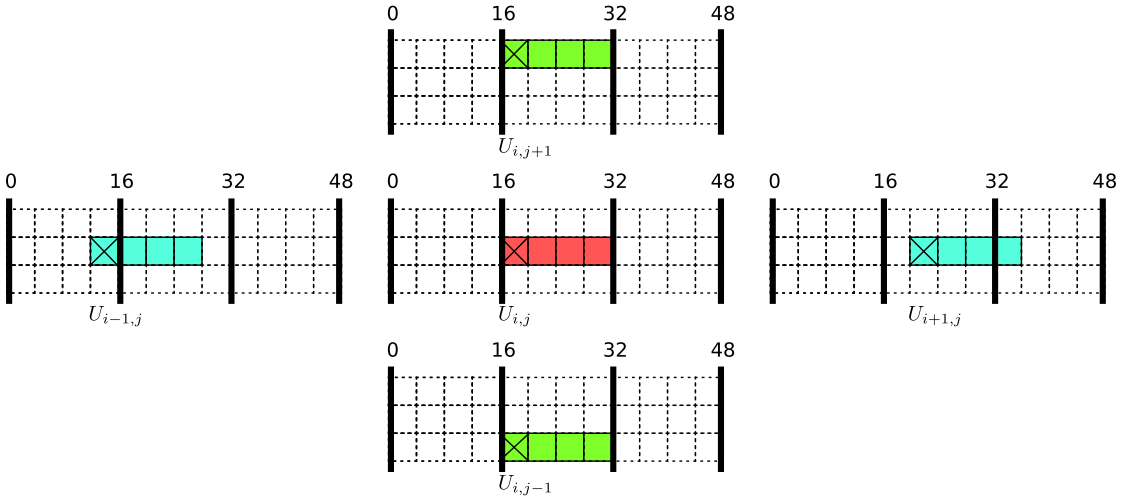


Figure 6: If data is stored by $x$-rows in LS memory, proper alignment in groups of four (for single precision) is automatic for updates in the $y$ direction: the four values in the middle diagram are updated using the aligned groups of four values in the top and bottom diagrams. However, for updates in the $x$ direction, the four values in the middle diagram need to be updated using the unaligned groups of four values in the left and right diagrams. For efficiency, aligned four-vectors of data need to be created for the left and the right four-vectors by intrinsic shuffling instructions. An alternative is to transpose all the data in the LS memory before doing all updates in the $x$ direction. (The cells with crosses correspond to unknowns in the vectors $U_{i,j}$, $U_{i-1,j}$ etc. that are needed to evaluate the updates of Eq. (3), for a certain choice of $i$ and $j$.)

one Xeon core. The Cell versions are compiled using IBM's ppuxlc and spuxlc compilers, with optimization settings -O3 or -O5, whichever gives better performance. The higher setting performs more aggressive interprocedural function call optimization and high level loop analysis which in our experience did not necessarily relate to improved performance.

Comparing performance, we see that the naive Cell implementation is already faster than the reference x86 CPU implementation. In both single and double precision we see that the fastest Cell kernel implementation is the SIMD, shuffle version. The transpose version also performs well but in the shuffle version the shuffle instructions are able to be interleaved with computations whereas in the transpose version they are separated from the computation so no overlap is possible. In the remainder of this paper only the shuffle implementation is used. Excellent speedups of approximately 22× and 7.5× are obtained for the optimized Cell code, for single and double precision, respectively. For the Cell, single precision is at least twice as fast as double precision, because the SIMD vectors can hold four single precision numbers or two double precision numbers. In addition, hardware support for transcendental functions is limited to reduced precision approximations of reciprocals and reciprocal square roots. Full single and double precision transcendentals must be evaluated in software, with the double precision ones requiring more operations. It is therefore no surprise that the timing results in Table 2 show that our double precision Cell kernel is roughly four times slower than the single precision kernel. Note that the penalty for using double precision is much smaller on the Xeon for our application.

### *4.3. GPU Implementation*

Our GPU cluster nodes have two x86 CPUs connected to two GPUs. Each GPU card has a large amount of GPU global memory (4GB) on board. We choose to hold all problem data in the GPU global memory and to perform all calculations on the GPUs, while the CPUs are used for input/output and for MPI communication (the GPUs cannot communicate via MPI directly). This approach minimizes the bandwidth use over the PCI Express link between GPU global memory and server host memory (only ghost cells need to be transfered). It also simplifies code development since all calculations are done on the GPU, but the computing power of the CPUs remains underutilized. For applications in which larger arrays are to be used, if the server host's memory is larger than the memory of the attached GPUs, or if the application algorithm features a natural division in work that can be done partially on the CPU and partially on the GPU, one may consider farming out work from the CPU to the GPU in a double-buffered data transfer scheme as in the Cell processor case. However, we found it simpler and quite efficient to perform all calculations on the GPUs for the problem we consider. Moreover, it is also not clear how much could be gained for our application type by utilizing the CPUs for calculations, since the calculations to be done for our application turn out to be performed between 3 and 8 times faster on the GPUs than on the CPUs (see the timing results in Table 5 below).

We adapt the CPU code to the GPU platform by mapping the unoptimized CPU computational kernel performing the grid cell updates to a CUDA kernel function. Each thread in the CUDA kernel launch is then responsible for one or several grid cell updates, depending on the implementation as described be-

| | Tesla T10 GPU (GT200 architecture) | | | |
|---|---|---|---|---|
| | single precision | | double precision | |
| | time (s) | speed-up (×) | time (s) | speed-up (×) |
| one Xeon core | 162.50 | 1.00 | 192.44 | 1.00 |
| GPU naive | 12.87 | 12.63 | 40.01 | 4.81 |
| GPU rewrite kernel | 8.32 | 19.53 | 28.84 | 6.67 |
| GPU single flux calculation | 6.17 | 26.34 | 15.84 | 12.15 |
| GPU thread coarsening | 4.97 | 32.70 | 15.03 | 12.80 |

Table 3: Single and double precision performance improvements achieved on the Tesla T10 GPU. The results are compared against the SSE-optimized CPU implementation on one Xeon core. The most highly-optimized version gives a speed-up of just over 32× relative to the Xeon execution in single precision and over 12× in double precision. This test uses a $1000 \times 1000$ grid with $L = 10$, $W = 10$ and 1000 timesteps.

| | C2050 GPU (Fermi architecture) | | | |
|---|---|---|---|---|
| | single precision | | double precision | |
| | time (s) | speed-up (×) | time (s) | speed-up (×) |
| one Xeon core | 162.50 | 1.00 | 192.44 | 1.00 |
| GPU naive | 6.62 | 24.55 | 14.09 | 13.66 |
| GPU rewrite kernel | 6.57 | 24.73 | 14.06 | 13.69 |
| GPU single flux calculation | 3.44 | 47.24 | 8.10 | 23.76 |
| GPU thread coarsening | 3.32 | 48.95 | 10.23 | 18.81 |

Table 4: Single and double precision performance improvements achieved on the Fermi C2050 GPU. The results are compared against the SSE-optimized CPU implementation on one Xeon core. The most highly-optimized version gives a speed-up of almost 50× relative to the Xeon execution in single precision and 23× in double precision (single flux implementation). This test uses a $1000 \times 1000$ grid with $L = 10$, $W = 10$ and 1000 timesteps. The GPU runs all use 128×1 thread blocks, the thread coarsening kernel updates 6 grid cells per thread in single precision and 1 in double precision. The single flux kernel updates 6 grid cells per thread in both single and double precision.

low. The host CPUs steer the GPUs through pthreads, with one thread for each GPU device. Data is sent to and from the GPU by calling CUDA memory transfer functions from the host thread on the CPU. Similarly, calculations are executed on the GPU by calling CUDA kernel functions from the host thread on the CPU.

#### *4.3.1. Memory Layout*

GPU threads can efficiently load data from the GPU global memory when the data is organized such that a load statement in all threads in a warp accesses data in the same aligned 128-byte block in the GPU global memory, and the same holds for memory stores. To enhance this so-called 'optimal coalescing' of loads and stores, it is beneficial to align grid rows on 128-byte boundaries by padding, even though coalescing of loads and stores is taken care of automatically on the newer NVIDIA hardware also for memory accesses that are not aligned perfectly to 128-byte boundaries [29]. This data layout adjustment along with the data-parallel nature of the algorithm and its high arithmetic intensity ensure that even our naive CUDA version (see below) exhibits relatively good performance on the GPU.

#### *4.3.2. Kernel Optimization*

The effect of several different CUDA kernel optimizations is described below, and their performance is compared in Table 3.

*Naive*

The naive version of the kernel is a copy of the unoptimized CPU compute kernel with minimal alterations. The $x$ and $y$ update loops over the grid cells are replaced by kernel calls with one thread spawned for every grid cell. Note that, in this naive version, we calculate four boundary fluxes in every grid cell thread, and thus perform the flux calculations at every grid cell interface twice (since fluxes between adjacent grid cells are equal).

*Rewrite Kernel*

The kernel was rewritten so that divisions and square roots involving the same numbers were never repeated within a kernel calculation. This was taken care of by the compiler in the CPU implementation but needs to be explicitly handled for the moment when writing CUDA kernels. As expected, the performance boosts on this kernel are significant: rewriting removed 3 divisions and performance increased markedly. This is due to the fact that division is performed in the special function unit (SFU) and this unit can become a computational bottleneck on the GPU since there are only two SFUs for eight scalar processors (SPs) (in single precision).

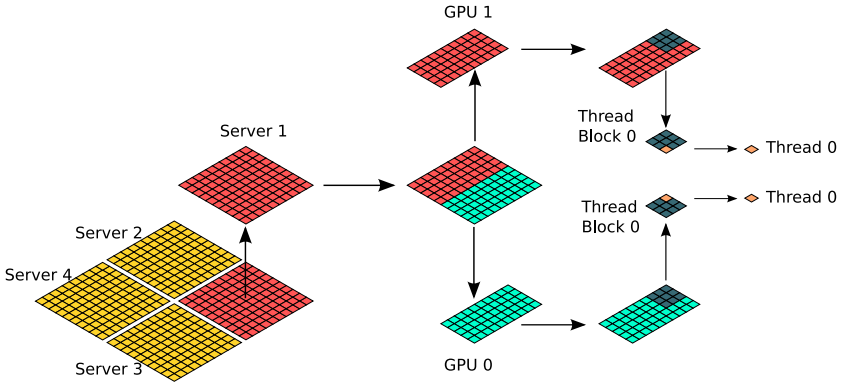


Figure 7: Different levels of parallelism and actual data layout in the GPU version of our code. The GPU implementation uses one MPI task per server node to divide work at the coarsest level. On each server two threads are launched, one to drive each of the GPUs, and the domain is further divided in two, with half being stored in each GPUs device memory. On the GPU device itself work is further divided into thread blocks, warps (omitted in the diagram), and eventually down to the individual thread level.

*Thread Coarsening*

In this kernel implementation, each thread updates multiple grid cells instead of just one. Empirical tests determined that it is optimal to update four grid cells in the $x$ direction and six grid cells in the $y$ direction. Thread coarsening is similar to the well known CPU loop optimization technique called unroll and jam [35, 36]. A similar type of 'kernel unrolling' has also been used extensively in other GPU codes although typically in memory bound applications [37, 2, 38]. The effect of this is twofold. First, the total number of floating point operations is reduced since flux calculations may be reused within one thread, rather than being recomputed for each grid cell. Second, the number of memory operations is reduced since data may be reused, rather than reloaded by each thread. Note that, in our naive GPU implementation, every thread loads four neighbor cells for the cell it updates, resulting in each cell being loaded five times instead of once, as assumed in the ideal memory transfer count in Table 1. The thread coarsening implementation improves on this since data for each flux interface is reused for multiple grid cell updates.

*Single Flux Calculation*

As mentioned earlier, the naive GPU kernel version is performing double the work of the CPU or Cell versions since fluxes are calculated twice. By changing the thread IDs to reference cell interfaces instead of grid cells, almost all of the duplicate work can be eliminated. To further reduce the duplicate work, the thread coarsening technique is again employed when computing the fluxes. However, we found that this implementation increased the memory footprint due to the storing of interface fluxes. The increased number of memory accesses gave this implementation worse performance despite reducing the total number of floating point operations performed. The thread coarsening approach described above provided a better compromise by reducing work while also reducing the number of memory accesses. In the remainder of this paper only the thread coarsening implementation is used unless otherwise stated.

See Fig. 7 for an illustration of the different levels of parallelism in our GPU code. Finally, Table 4 contains preliminary results for our GPU code on the recently introduced Fermi GPU platform. It can be seen that our code, without further optimization specific to Fermi, runs between 1.5 and 2 times faster than on the previous-generation Tesla GPU, in single and double precision, respectively. It is also interesting to note that the 'rewrite kernel' optimization is not necessary on Fermi (likely due to improvements in the nvcc compiler), and that the 'single flux calculation' method becomes the fastest in double precision due to Fermi's improved memory access mechanisms.

### *4.4. MPI Implementation*

As illustrated in Figs. 1, 4, 5, and 7, the problem is parallelized by doing a standard domain decomposition that divides the 2D grid of data into equally-sized rectangular sub-domains of grid cells. Each MPI task is assigned one sub-domain of the 2D grid. The MPI tasks are thus connected in a 2D Cartesian topology, giving each MPI task four neighboring tasks with which it must communicate. MPI communications are done after each $x$ and $y$ sweep, sending updated ghost cell data to the neighbouring tasks. The amount of data that is sent depends on the numerical scheme being used. In this paper a nine-point stencil is used, which requires information from the two nearest cells in each direction. When the neighbouring cells lie across inter-task boundaries, the two tasks must exchange two rows or columns of grid cell data which is done using a ghost cell approach [5]. The top-level data layout employs different arrays for each of the three grid variables, $h$, $u$, and $v$. This is beneficial for the use of SIMD vector units, since this tends to align groups of variables in memory that can be fed to the SIMD units together, and it also makes alignment easier for transfer of array parts between system components. For these reasons, separate arrays for $h$, $u$, and $v$ are, in fact, used on all levels of parallelism in our implementation. Since each hardware-specific back-end implementation makes use of a different data-storage scheme, the MPI exchanges are done in separate exchange buffers in order to maintain portability across different architectures. Each

| System | Time (s) | Speed-Up (×) | Speed-Up vs quad-core (×) |
|---|---|---|---|
| CPU Xeon (1 Core, naive) | 778.46 | 0.48 | 0.12 |
| CPU Xeon (1 Core+SSE) | 372.69 | 1.00 | 0.25 |
| CPU Xeon (4 Cores+SSE) | 95.43 | 3.91 | 1.00 |
| Cell PlayStation 3 (6 SPEs) | 22.6 | 16.49 | 4.22 |
| GPU Tesla (G80 architecture) | 17.68 | 21.08 | 5.40 |
| Cell PowerXCell 8i (8 SPEs) | 16.96 | 21.97 | 5.63 |
| GPU Tesla T10 (GT200 architecture) | 11.42 | 32.63 | 8.36 |
| GPU C2050 (Fermi architecture) | 6.62 | 56.30 | 14.42 |

(a) single precision

| System | Time (s) | Speed-Up (×) | Speed-Up vs quad-core (×) |
|---|---|---|---|
| CPU Xeon (1 Core, naive) | 975.06 | 0.48 | 0.12 |
| CPU Xeon (1 Core+SSE) | 464.31 | 1.00 | 0.26 |
| CPU Xeon (4 Cores+SSE) | 119.43 | 3.89 | 1.00 |
| Cell PowerXCell 8i (8 SPEs) | 61.4 | 7.56 | 1.95 |
| GPU Tesla T10 (GT200 architecture) | 36.15 | 12.84 | 3.30 |
| GPU C2050 (Fermi architecture) | 16.66 | 27.88 | 7.17 |

(b) double precision

Table 5: Single-chip performance comparisons between architectures in single and double precision. The runtime and speed-up versus 1 and 4 Xeon CPU Cores is indicated. The Cell kernel used is the shuffle implementation of section 4.2.2 and the GPU kernel used is the thread coarsening kernel of section 4.3.2. This test uses a $5000 \times 5000$ grid with $L = 50$, $W = 50$ and 100 timesteps. Fermi results in double precision use the single flux kernel. Due to memory constraints on our Fermi card the code is executed on a 4000×4000 grid, and the numbers are scaled up for comparison.

MPI exchange consists of three distinct stages: packing, sending/receiving, and unpacking, where the packing and unpacking stages are implemented differently for each architecture but the sending and receiving remains the same.

## 5. Performance Comparison

In this Section, we compare the x86 CPU, Cell and GPU performance at various levels of parallelism. We first compare single Cell and GPU chips (including commodity versions) with the reference SSE-optimized CPU implementation on one core (as is customary in the literature), and then with the SSE-optimized CPU code running on all four cores on the Xeon CPU chip (Table 5). After this performance on a chip-by-chip basis, we compare the Cell and GPU performance on a node-by-node basis, i.e., on single cluster nodes without MPI. Finally, we compare performance and scaling of the CPU, Cell and GPU codes on multi-node clusters with MPI parallelism.

### *5.1. Comparison Setup and Cluster Information*

We summarize the versions of the code used in the performance comparisons in this section, as well as the details of the clusters used.

#### *5.1.1. Comparison Setup and MPI layout*

For each of the three platforms, we list below the optimization version of the code used in the comparison tests. We also list the MPI configuration used in the MPI scaling tests.

- CPU: SSE-optimized, 1 MPI process per Xeon Core (8 per Xeon Server).
- Cell: SIMD Shuffle, 1 MPI process per PPU (i.e., 2 per blade server), 8 SPEs per PPU.
- GPU: Thread Coarsening (4 $x$ unrolls, 6 $y$ unrolls), 1 MPI process per Xeon Server, 2 threads and GPUs per MPI process.

#### *5.1.2. Cluster Details*

We use the following clusters provided by SHARCNET [23] and JSC [24].

*JSC Cell Cluster*

The JSC Cell cluster system 'JUICEnext' has the following specifications:

- The cluster has 35 QS22 blade Cell servers mounted in three BladeCenter H chassis.
- Each QS22 blade has two PowerXCell 8i processors @ 3.2GHz in a dual-socket configuration that gives access to eight GB of shared memory. (We call this memory shared by the two Cell processors the Cell blade main memory.)
- The QS22 blades have a 4x DDR Infiniband (16 GB/s) interconnect between the server nodes.

Note that this configuration is different from the configuration of nodes in Los Alamos National Laboratory's Roadrunner supercomputer, which was the first computer to break the

petaflop performance barrier [28]. Roadrunner nodes or 'triblades' contain two QS22 blades coupled with a conventional LS21 opteron blade. The opteron blades can be used for part of the computations and for fast communication. The JSC Cell cluster is composed of QS22 blades only, and all computations and the communication between Cell blades is performed by the Cell processor. In this paper, we thus report on developing an efficient simulation code for this more affordable but also more challenging QS22-only environment.

*SHARCNET Xeon/GPU Cluster*

The SHARCNET GPU cluster 'angel' that was used for both CPU and GPU code development and testing has the following specifications:

- The cluster has 22 server nodes, each consisting of two quad-core Xeon E5430 processors @ 2.66GHz with eight GB of shared memory. (We call this memory that is shared between the two Xeon processors the node host memory.)
- The cluster has 11 NVIDIA S1070 GPU computing systems, each featuring four Tesla T10 GPUs with four GB device memory per GPU. (We call this the GPU global memory.)
- The cluster has a 4x DDR Infiniband (16 GB/s) interconnect between the server nodes.
- The GPUs are attached to the server nodes via a PCI Express 2.0 bridge, with two GPUs attached to each server node.

### 5.2. Single-chip Comparison

Table 5 compares our optimized Cell and GPU codes with CPU implementations on one and four CPU cores. Note that the problem sizes for the tests in Table 5 are chosen larger than for the optimization comparisons described in Section 4. In single precision, the CUDA implementation running on an NVIDIA Tesla T10 GPU chip gives a speed-up of 32× over the reference SSE-optimized implementation executing on a single Xeon Core, with the Fermi results between 1.5 and 2 times faster than that. It is worth noting that even the slowest GPU/-Cell version, the Cell implementation executing on the PlayStation 3, still gives 16× speed-up over the reference CPU implementation. The results in Table 5 show that for single precision both the GPU and Cell provide excellent acceleration of roughly 5-8× over a multithreaded SSE implementation running on all four cores of the Xeon processor. The Fermi GPU improves this by a factor between 1.5 and 2.

In double precision the Cell and Tesla implementations are 2 and 3× faster than the SSE-optimized Xeon version on 4 cores, respectively, as shown in Table 5. The Fermi GPU improves this by a factor of more than 2. Comparing single and double precision numbers from Table 5 one can see that the Cell version slows down by a factor of roughly four when going from single to double precision. This is partially due to the fact that that SIMD vector width is four for single precision and two for double precision. Another reason is that division, comparison and square root operations require more cycles in double precision. The GPU version slows down by a factor of about 3, which is better than the 8 times slowdown one might expect (there is only 1 double precision unit for 8 scalar thread processors). The reason for this is that the single precision version never comes close to the peak theoretical throughput because some operations, namely division and square root, are performed in the special function unit (SFU) and so are done only two at a time and not eight at a time.

### 5.3. Comparison with Theoretical Peak Performance Specifications

In this section we compare the compute and memory performance measured for our CPU, Cell and GPU implementations with the theoretical peak performance of the devices, and discuss whether our codes are compute bound or memory bound [39, 40].

#### 5.3.1. Algorithmic and Device FLOPS/Byte Ratios

In Table 6, we first make a comparison between the ratio of FLOPS per Byte of data transfered for an ideal implementation of our algorithm (with optimal data reuse and minimal number of FLOPS per grid cell), and the peak theoretical FLOPS/Byte ratio of the three platforms. The data in Table 6(a) are repeated from Table 1 for convenience. The theoretical peak FLOPS/Byte ratios for the three devices are obtained by taking the ratio of their theoretical peak GFLOPS/s and GB/s specifications (Table 6(b)). The peak FLOPS/Byte rates give an indication, in an idealized setting, of the maximum number of FLOPS that can be performed per Byte of data transfered before an algorithm becomes compute bound, on each of the architectures. From Table 6, we don't expect our codes to be memory bound for most cases, since the algorithmic FLOPS/Byte ratio is normally larger or approximately equal to the peak FLOPS/Byte ratio of the devices. The only exception may be the SP GPU case, where our algorithm's FLOPS/Byte ratio is slightly smaller than the device's peak theoretical FLOPS/Byte ratio. We might thus expect our GPU code to be slightly memory bound in the SP case. However, we want to stress that the numbers in Table 6 are theoretical. In reality, the number of FLOPS executed per grid cell is larger than 360 in our GPU code, since we do some flux calculations twice. Also, the SP peak GFLOPS/s rate assumes multiply-add operations, but many FLOPS in our algorithm, such as square roots and divisions, can only be done on the special function unit, so only two at a time in SP, compared to eight at a time for additions and multiplications. For these reasons, we expect the GPU algorithm to be close to compute bound also in the SP case. Note that, in DP, the peak GFLOPS/s value for the Tesla GPU assumes that all FLOPS use the single available double precision unit (and not the eight scalar processors), and it is thus more easily reachable than the SP peak GFLOPS/s value for an application like ours, since it assumes that all FLOPS are multiply-adds that can be executed together.

| | |
|---|---|
| FLOPS per grid cell | 360 |
| Memory Ops per grid cell | 48 Bytes SP, 96 Bytes DP |
| SP FLOPS/Byte | 7.5 |
| DP FLOPS/Byte | 3.75 |

(a) For our algorithm, a minimum of 7.5 and 3.75 floating point operations are computed per Byte of data loaded or stored, in SP and DP, respectively.

| Architecture | peak GFLOPS/s | | peak GB/s | peak FLOPS/Byte | |
|---|---|---|---|---|---|
| | SP | DP | | SP | DP |
| CPU Xeon (4 Cores) | 43 | 21 | 11 | 4 | 2 |
| Cell PowerXCell 8i | 205 | 102 | 26 | 8 | 4 |
| GPU Tesla T10 (GT200 architecture) | 1037 | 86 | 102 | 10 | 0.8 |
| GPU C2050 (Fermi architecture) | 1288 | 515 | 144 | 9 | 3.6 |

(b) Peak Theoretical Peformance for the three architectures. An estimate is also given for the maximum number of FLOPS that can be performed per Byte of data transfered before an algorithm becomes compute bound (peak FLOPS/Byte).

Table 6: By comparing the algorithmic FLOPS/Byte ratio with the peak theoretical FLOPS/Byte ratio of the hardware we can estimate if we expect the algorithm to be compute or memory bound. Even though values in these tables are ideal lower and upper bounds and real algorithms and implementations will behave in non-ideal ways, a code is likely to be memory bound if the algorithmic FLOPS/Byte rate is significantly smaller than the peak theoretical FLOPS/Byte rate, and compute bound otherwise.

#### *5.3.2. Estimated Compute and Memory Performance of our Implementations*

In Table 7, we investigate how close we are getting to the theoretical peak performance on the different architectures. For the simulations of Table 5, we compute the minimum total number of FLOPS over all cells and iterations using the FLOPS data from Table 6(a), and divide this by the execution times from Table 5, to obtain the estimated actual GFLOPS/s rates of our implementations. We then divide by the theoretical peak GFLOPS/s rates from Table 6(b) to obtain the %Peak values. Similarly, the estimated attained GB/s rates are obtained by dividing the minimum total number of memory transfers by the actual execution time. Note that the GFLOPS/s %Peak values may actually be underestimates on some platforms. For example, on the Cell square roots and divisions, which are only counted once in the 360 FLOPS, may take significantly more than one FLOP each. For the Tesla T10 and the C2050 (Fermi), we report additional numbers ('actual'), in which we also count the FLOPS and memory transfers that are duplicated on different threads during neighbor flux computations (that load neighbor cells). Note, however, that we still do not account for square roots and divisions being executed only two at a time, instead of multiply-adds being executed eight at a time for the peak performance measure. While it is clear that further optimization is possible, the results in Table 7 show respectable performance on all platforms. As expected, the CPU results appear compute bound, the Cell results are quite balanced, and the GPU results appear somewhat memory bound in SP, and compute bound in DP. In terms of attained GFLOPS/s, the Cell %Peak performance drops when going from SP to DP. A major cause for this is that the DP Cell operations of square root and division are not supported in hardware. The GFLOPS/s %Peak increases for the GPU when going from SP to DP, which may be attributed to the SP code being slightly memory bound, while the DP code is compute bound, and to the %Peak value being more easily attainable in DP than in SP for our application type. The Fermi results are about twice as fast as the Tesla (GT200 architecture) results.

### *5.4. Problem Size Analysis*

The effect of increasing the problem size is investigated in Table 8. On the CPU we see some variation but overall the performance level remains rather stationary. On the GPU the dominant effect in single precision is that for smaller problem sizes, not enough threads are launched to fully saturate the GPU device. This also occurs in DP. However, the overall performance penalty for the problem sizes we tested is smaller in DP, because there are eight times fewer double precision units than single precision units, so less threads are required for saturation. For the Cell processor in single precision there is also degradation for small problem sizes. This is because the local store block size is large compared with the actual grid size ($52 \times 52$-sized blocks compared to $100 \times 100$ or $200 \times 200$ grid sizes) and not enough local store blocks are generated to use all SPEs and fully saturate the device.

### *5.5. Single blade/single node Comparison and Scaling*

On a Cell blade or cluster node with shared memory, multiple Cell processors or GPUs can be used in parallel without the need to communicate via MPI. This is attractive for some applications since code complexity can be reduced by eliminating the domain decomposition and communication layer at the top MPI-level of parallelism. To illustrate scaling in a shared-memory environment, we show single blade/single node scaling performance in Tables 9 and 10.

Table 9 shows scaling results on one QS22 Cell blade with 2 Cell processors and 8GB of shared Cell blade main memory. Here the optimized Cell code with 'shuffle' optimization is used. The two Cell processors have a total of 16 SPEs. In these tests, all SPEs are steered by a single PPE (on one of the two Cell processors) using pthreads. 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. Table 9 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).

Multiple GPUs attached to the same host node may be used by starting separate pthreads on the host node and associating GPU contexts to them. In our GPU-centric implementation, updated cell information must be communicated between the GPU global memories of the GPUs. In current GPUs there is no support for data transfers directly between devices so the

| | Time (s) | GFLOPS/s | %Peak | GB/s | %Peak |
|---|---|---|---|---|---|
| CPU Xeon (4 Cores+SSE) | 92.18 | 9.76 | 23% | 1.30 | 12% |
| Cell PowerXCell 8i (8 SPEs) | 16.96 | 53.07 | 26% | 7.08 | 28% |
| GPU Tesla T10 (GT200) | 11.42 | 78.81 | 7.6% | 10.51 | 10% |
| GPU Tesla T10 (GT200) (actual) | 11.42 | 140.85 | 13.6% | 36.78 | 36% |
| GPU C2050 (Fermi) | 6.62 | 135.95 | 11% | 18.12 | 13% |
| GPU C2050 (Fermi) (actual) | 6.62 | 242.97 | 19% | 63.41 | 44% |

(a) single precision

| System | Time (s) | GFLOPS/s | %Peak | GB/s | %Peak |
|---|---|---|---|---|---|
| CPU Xeon (4 Cores+SSE) | 119.43 | 7.54 | 35% | 1.00 | 9% |
| Cell PowerXCell 8i (8 SPEs) | 61.40 | 14.66 | 14% | 1.95 | 8% |
| GPU Tesla T10 (GT200) | 36.15 | 24.90 | 29% | 3.32 | 3% |
| GPU Tesla T10 (GT200) (actual) | 36.15 | 44.49 | 52% | 11.62 | 11% |
| GPU C2050 (Fermi) | 16.66 | 54.02 | 10% | 14.41 | 10% |
| GPU C2050 (Fermi) (actual) | 16.66 | 54.03 | 10% | 52.83 | 37% |

(b) double precision

Table 7: Comparison of estimated real performance with peak theoretical performance for the three platforms. The values for the 'GPU Tesla T10 (GT200) (actual)' row take into account how many flux calculations and memory transfers the implementation actually does, as opposed to the minimum number that is necessary for an ideal implementation of the algorithm without flux calculation duplication and additional loads of neighbor cells, which is listed in the 'GPU Tesla T10 (GT200)' row for comparison. In the Fermi double precision runs, we are using the (faster) single flux kernel instead of the thread coarsening kernel.

| n | Size (MB) | CPU (4 cores) | Cell | GPU |
|---|---|---|---|---|
| 100 | 0.11 | 20.00 | 100.00 | 50.00 |
| 200 | 0.46 | 25.00 | 100.00 | 100.00 |
| 400 | 1.83 | 26.23 | 145.45 | 145.45 |
| 800 | 7.32 | 26.78 | 145.45 | 188.24 |
| 1600 | 29.3 | 25.15 | 144.63 | 209.84 |
| 3200 | 117 | 24.79 | 147.98 | 215.58 |
| 6400 | 469 | 25.36 | 147.39 | 219.15 |

(a) single precision

| n | Size (MB) | CPU (4 cores) | Cell | GPU |
|---|---|---|---|---|
| 100 | 0.23 | 16.67 | 33.33 | 25.00 |
| 200 | 0.92 | 19.05 | 36.36 | 40.00 |
| 400 | 3.66 | 19.51 | 39.02 | 47.06 |
| 800 | 14.6 | 19.88 | 40.25 | 58.18 |
| 1600 | 58.6 | 19.56 | 39.88 | 67.90 |
| 3200 | 234 | 20.56 | 40.63 | 64.85 |
| 6400 | 938 | 20.39 | 40.71 | 67.87 |

(b) double precision

Table 8: Effect of problem size on performance for Xeon CPU (4 cores), PowerXCell 8i Cell, and Tesla T10 GPU. This test uses a square problem domain of size $n \times n$, with $L = W = \frac{2n}{100}$ and 100 timesteps. Results are listed in units of $10^6$ grid cell updates per second. The CPU performance is not very sensitive to problem size, except for relative slowdown for the smallest problem sizes, which is due to MPI communication overhead. For the GPU case, as the problem size increases, the overall performance rapidly improves until thread saturation is reached. In double precision the same effect is seen but the GPU becomes saturated much earlier. The Cell processor also shows a jump in single precision performance once the problem size becomes large enough to use all eight SPEs.

| | QS22 blade with two PowerXCell 8i Cell processors | | | |
|---|---|---|---|---|
| | single precision | | double precision | |
| SPEs | time (s) | speed-up (×) | time (s) | speed-up (×) |
| 1 | 53.24 | 1.00 | 194.79 | 1.00 |
| 2 | 26.67 | 2.00 | 97.56 | 2.00 |
| 4 | 13.37 | 3.98 | 48.81 | 3.99 |
| 8 | 6.72 | 7.92 | 24.41 | 7.98 |
| 16 | 3.33 | 15.99 | 12.21 | 15.95 |

Table 9: Scaling on one QS22 Cell blade with 2 Cell processors and 8GB of shared memory (no MPI). The two Cell processors have a total of 16 SPEs. In these tests, all SPEs are steered by a single PPE (on one of the two Cell processors) using pthreads. This test uses a $1000 \times 1000$ grid with $L = 10$, $W = 10$ and 1000 timesteps.

| | S1070 computing server with two Tesla T10 GPUs | | | |
|---|---|---|---|---|
| | single precision | | double precision | |
| GPUs | time (s) | speed-up (×) | time (s) | speed-up (×) |
| 1 | 286.11 | 1.00 | 905.78 | 1.00 |
| 2 | 145.15 | 1.97 | 456.45 | 1.98 |

Table 10: Scaling test for one and two Tesla T10 GPUs on a single S1070 computing server steered from one Xeon server node (no MPI). Linear scaling is demonstrated. This test uses a $8000 \times 8000$ grid with $L = 80$, $W = 80$ and 1000 timesteps.

server host is used as an intermediary. Updated ghost cell data is passed to buffers in the server host memory and then to ghost cells on the neighboring GPU (but MPI is not required). Performance when using one or two GPUs on the same server node is investigated in Table 10.

Note that, in order to get nearly linear scaling, the problem size has to be chosen larger in the GPU case than in the Cell case. This is because the two GPUs do not share memory, and communication between them has to proceed via the host server CPU and memory. The two Cell processors, on the other hand, share memory, and the 16 SPEs can be used as if they were located on one single Cell processor.

### *5.6. Cluster Comparison and Scaling*

In this section, we discuss two types of parallel cluster scaling tests for the three platforms. In strong scaling tests the total problem size is kept constant as the number of nodes is increased, and the speedup is measured. In weak scaling tests, the problem size increases proportional to the number of nodes (the problem size per node is kept constant), and ideally the execution time should not change as the number of nodes increases.

In Table 11 we show MPI strong scaling results for the CPU, Cell and GPU implementations on parallel clusters. We use as many quad-core Xeon chips as Cell or GPU chips in this parallel scaling tests, and the CPU code is SSE-optimized. The cluster results largely mirror the single-chip results. The Cell processor cluster results do not scale as well as the CPU and GPU cluster results. This is because we use the PPE cores to perform the MPI communication. The advantage of this is that we have direct MPI communication between Cell processors, but, unfortunately, the PPE cores are slow and affect the parallel scaling negatively. As we said in Section 5.1.2, a QS22-only system is a challenging environment for developing MPI-parallel codes that scale well. In our single precision test case, for instance, 42% of the time (4.83 s) is spent in MPI calls on eight nodes,and 47% (3.36 s) on 16 nodes. On the contrary, in the GPU implementation, the amount of time spent in communication is 0.36s for eight nodes, and 0.53 for 16 nodes. This can be remedied by employing more expensive Roadrunner-like 'tri-blades' with Cell processors, in which MPI communication can be performed by faster Opteron processors, or one can reduce the number of times that MPI communication must occur by adding more ghost layers and doing multiple iterations between MPI communication calls. The GPU results also do not scale very well in this strong scaling test. This is because the communication time begins to take a significant fraction of the total run time as the number of nodes increases. Note that this slowdown is not due to the problem sizes per node dropping below the threshold at which the GPUs' threads are fully saturated. Indeed, Table 8 shows that this drop happens around problem size $1600^2$, but the problem sizes per cluster node in Table 11 are greater than twice $1600^2$ grid cells, even for the runs with 16 nodes. See also [21] for a discussion on strong scaling on a GPU cluster.

We also carried out weak scaling tests using the same configurations, and the results are shown in Table 12. In the weak scaling table the GPU performs better, since the computation time remains relatively large compared with the communication time. The CPU MPI runs display some variability, which may be due to system load variability, over which we did not have explicit control. Also, since the CPU implementation starts eight MPI processes per Xeon server, there is increased messaging and synchronization overhead between server nodes, whereas in the GPU case there is only a single MPI process per server node. For the Cell processor implementation, the SPE compute time is remarkably uniform, likely owing to the simplicity of the individual SPE cores themselves, but the overall

| | 2 x Xeon CPU | | | | 2 x Tesla T10 GPU | | | | 2 x PowerXCell 8i | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | Ex (s) | Wk (s) | T (s) | **S (×)** | Ex (s) | Wk (s) | T (s) | **S (×)** | Ex (s) | Wk (s) | T (s) | **S (×)** |
| 1 | 2.46 | 315.8 | 318.26 | **1.00** | 0.02 | 35.79 | 35.81 | **1.00** | 4.29 | 53.40 | 57.69 | **1.00** |
| 2 | 1.94 | 155.53 | 157.47 | **2.02** | 0.26 | 17.97 | 18.23 | **1.96** | 4.48 | 26.72 | 31.20 | **1.85** |
| 4 | 3.00 | 77.49 | 80.49 | **3.95** | 0.37 | 9.08 | 9.45 | **3.79** | 6.13 | 13.44 | 19.57 | **2.95** |
| 8 | 1.34 | 38.77 | 40.11 | **7.93** | 0.36 | 4.66 | 5.02 | **7.13** | 4.83 | 6.74 | 11.57 | **4.99** |
| 16 | 0.75 | 19.38 | 20.13 | **15.81** | 0.53 | 2.39 | 2.92 | **12.26** | 3.36 | 3.79 | 7.15 | **8.07** |

(a) Single Precision

| | 2 x Xeon CPU | | | | 2 x Tesla T10 GPU | | | | 2 x PowerXCell 8i | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | Ex (s) | Wk (s) | T (s) | **S (×)** | Ex (s) | Wk (s) | T (s) | **S (×)** | Ex (s) | Wk (s) | T (s) | **S (×)** |
| 1 | 3.12 | 380.76 | 383.88 | **1.00** | 0.07 | 113.52 | 113.59 | **1.00** | 6.10 | 195.69 | 201.79 | **1.00** |
| 2 | 3.81 | 185.88 | 189.69 | **2.02** | 0.57 | 56.96 | 57.53 | **1.97** | 6.53 | 98.10 | 104.63 | **1.93** |
| 4 | 4.15 | 92.79 | 96.94 | **3.96** | 0.66 | 28.66 | 29.32 | **3.87** | 8.40 | 48.84 | 57.24 | **3.53** |
| 8 | 2.19 | 46.43 | 48.62 | **7.90** | 0.71 | 14.70 | 15.41 | **7.37** | 6.39 | 24.44 | 30.83 | **6.55** |
| 16 | 1.40 | 23.28 | 24.68 | **15.55** | 0.83 | 7.59 | 8.42 | **13.49** | 7.21 | 12.20 | 19.41 | **10.40** |

(b) Double Precision

Table 11: Strong scaling performance comparison between architectures in single and double precision using from N=1 to N=16 cluster nodes with MPI. This test uses a $16000 \times 10000$ grid with $L = 160$, $W = 100$ and 100 timesteps. We give the total runtime (T), the mpi-exchange time (Ex), the time to do the actual calculations (Wk), and the scaling of the total runtime (S). For all platforms we use our most performant code, as listed in Section 5.1.1.

| | 2 x Xeon CPU | | | | 2 x Tesla T10 GPU | | | | 2 x PowerXCell 8i | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | Ex (s) | Wk (s) | T (s) | **R** | Ex (s) | Wk (s) | T (s) | **R** | Ex (s) | Wk (s) | T (s) | **R** |
| 1 | 1.51 | 97.19 | 98.70 | **1.00** | 0.01 | 11.25 | 11.26 | **1.00** | 2.14 | 16.70 | 18.84 | **1.00** |
| 2 | 1.96 | 97.46 | 99.42 | **0.99** | 0.19 | 11.31 | 11.50 | **0.98** | 3.70 | 16.70 | 20.40 | **0.92** |
| 4 | 2.11 | 97.10 | 99.21 | **0.99** | 0.29 | 11.26 | 11.55 | **0.97** | 3.94 | 16.70 | 20.64 | **0.91** |
| 8 | 4.07 | 97.22 | 101.29 | **0.97** | 0.43 | 11.30 | 11.73 | **0.96** | 7.33 | 16.70 | 24.03 | **0.78** |
| 16 | 3.65 | 97.42 | 101.07 | **0.98** | 0.54 | 11.26 | 11.80 | **0.95** | 7.73 | 16.70 | 24.43 | **0.77** |

(a) Single Precision

| | 2 x Xeon CPU | | | | 2 x Tesla T10 GPU | | | | 2 x PowerXCell 8i | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | Ex (s) | Wk (s) | T (s) | **R** | Ex (s) | Wk (s) | T (s) | **R** | Ex (s) | Wk (s) | T (s) | **R** |
| 1 | 1.92 | 121.47 | 123.39 | **1.00** | 0.01 | 35.30 | 35.31 | **1.00** | 3.27 | 61.05 | 64.32 | **1.00** |
| 2 | 5.75 | 122.35 | 128.10 | **0.96** | 0.41 | 35.55 | 35.96 | **0.98** | 5.52 | 61.10 | 66.62 | **0.97** |
| 4 | 7.15 | 121.56 | 128.71 | **0.96** | 0.58 | 35.30 | 35.88 | **0.98** | 8.77 | 61.13 | 69.90 | **0.92** |
| 8 | 7.17 | 121.40 | 128.57 | **0.96** | 0.92 | 35.52 | 36.44 | **0.97** | 10.93 | 61.12 | 72.05 | **0.89** |
| 16 | 12.77 | 123.41 | 136.18 | **0.91** | 1.16 | 35.31 | 36.47 | **0.97** | 11.02 | 61.14 | 72.16 | **0.89** |

(b) Double Precision

Table 12: Weak scaling performance comparison between architectures in single and double precision using from N=1 to N=16 cluster nodes with MPI. This test uses grid sizes of $10000 \times 5000$,$10000 \times 10000$, $20000 \times 10000$, $20000 \times 20000$, and $40000 \times 20000$ with grid $L = W = 800$ and 100 timesteps. We give the total runtime (T), the mpi-exchange time (Ex), the time to do the actual calculations (Wk), and the ratio of the total runtime for N=1 divided by the total runtime on N nodes (R). For all platforms we use our most performant code, as listed in Section 5.1.1.

performance does not scale well because of the poor PPU performance on the inter-node MPI calls.

## 6. Conclusions and Future Directions

We have shown how a numerical simulation method for nonlinear hyperbolic PDE systems can be implemented efficiently for the Cell processor and NVIDIA GPUs using CUDA. We have described memory layout, communication patterns and optimization steps that were performed to exploit the low-level parallelism of these two platforms. A coarse-level layer of MPI parallelism was added to obtain a hybrid parallel code that can be executed efficiently on GPU or Cell clusters. Performance tests were conducted on JSC's Cell cluster system 'JUICEnext' [24] and SHARCNET's GPU cluster 'angel' [23].

Compared to a reference CPU implementation with cache and SSE optimization executed on one Xeon core, significant speed-ups were obtained by both the Cell processor and GPU implementations. In single precision the Tesla T10 GPU implementation (GT200 architecture) gave almost 32× speed-up, and the Cell provided a 22× speed-up. In double precision the Tesla GPU gave a 13× speed-up over a single Xeon core, while the Cell gave just under 7.5× better performance (Table 5).

In a chip-to-chip comparison of single precision results, the Cell processor was 5× faster than a quad-core Xeon implementation, and the Tesla GPU was 8× faster. In double precision the Cell achieves a 2× speed-up, and the Tesla GPU roughly 3× (Table 5). For our CUDA code, recently released GPUs with next-generation Fermi architecture improve on the Tesla results (GT200 architecture) by a factor of about two (in preliminary results without further Fermi-specific optimization).

In a cluster-to-cluster comparison, the speed-up ratios remain roughly the same. However, the Cell implementation scales poorly on our QS22-cluster due to slow performance of the PPE. There are work-arounds for the slow PPE, such as the hybrid AMD-Cell approach used in Roadrunner, or one can reduce the number of times that MPI communication must occur by adding more ghost layers and doing multiple iterations between MPI communication calls.

The overall results of our study demonstrate that Cell and GPU clusters can be used for efficiently simulating nonlinear hyperbolic PDE systems on structured grids with explicit timestepping. Our model problem is representative of a large class of structured grid based local-interaction simulation algorithms with relatively large FLOP counts per Byte of data transfered, and our conclusions carry over to many simulation codes used in broad areas of computational science and engineering.

Our discussion of Cell and GPU optimization techniques shows that there is still a significant learning curve and optimization effort involved in porting codes to Cell and GPU environments. We have found the effort required to obtain significant speed-ups smaller on GPU platforms than on Cell platforms, and find the GPU platform generally simpler and more intuitive to program. Extensive efforts are underway to improve automatic compiler optimization for both Cell and GPU platforms: the RapidMind platform, the HONEI libraries [20] and the new OpenCL framework [41] are examples. However, for the time being, automatic approaches cannot provide yet the level of speed optimization that we were seeking in this paper.

In future work we will extend our approach to bodyfitted adaptive multi-block codes in three dimensions (3D), which allow for simulation of real 3D problems with more complex geometry. Extending our work to unstructured grids or implicit time integration would require different optimization approaches, since those problems require unstructured data and/or nonlocal connectivity. Mixed-precision calculations are also an interesting option, due to the excellent performance of Cell and GPU on single precision calculations.

Our preliminary results on the recently introduced Fermi GPU show that GPU calculations can be up to 14 times faster than quad-core Xeon E5430 CPU calculations in single precision, and up to 7 times faster in double precision, for our application. Improvements in the newest GPUs that are important for HPC include full IEEE floating point compliance, error correcting memory, and L1 and L2 caches. Direct communication between GPUs over Infiniband networks [43] is another exciting prospect.

Our results support indications that clusters with heterogeneous multi-core architectures may become increasingly important for scientific computing applications in the near future, especially also if one considers their typically low power requirements [26, 22] 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 [42].

## Acknowledgements

This work was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET, [23]) and the Juelich Supercomputer Centre (JSC, [24]), and by an NVIDIA Professor Partnership grant. We thank Hugh Merz, John Morton and other members of the SHARCNET technical staff for their expert advice and technical help. We also thank Markus Stuermer, Lucian Ivan and Matthias Bolten for their advice, and Willy Homberg for JSC support.

## References

[1] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, NVIDIA Tesla: A Unified Graphics and Computing Architecture, IEEE Micro **28** (2008) 39–55.

[2] J. D. Owens , M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, GPU Computing, Proceedings of the IEEE 96 (2008) 879–899.

[3] 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, IEEE Journal of Solid-State Circuits **41** (2006) 179–196.

[4] A. Arevalo, R. M. Matinata, M. Pandian, E. Peri, K. Ruby, F. Thomas, and C. Almond, Programming the Cell Broadband Engine Architecture: Examples and Best Practices (IBM Redbooks, 2008).

[5] R. J. LeVeque, Finite volume methods for hyperbolic problems (Cambridge University Press, 2002).

[6] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich, Ray tracing on the Cell processor, in: Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006, 15–23.

[7] G. De Fabritiis, Performance of the Cell processor for biomolecular simulations, Computer Physics Communications **176** (2007) 660–664.

[8] M. Stuermer, J. Goetz, G. Richter, A. Doerfler, and U. Ruede, Fluid flow simulation on the Cell Broadband Engine using the lattice Boltzmann method, Computers and Mathematics with Applications **58** (2009) 1062–1070.

[9] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, Scientific computing kernels on the Cell processor, Int. J. Parallel Programming **35** (2007) 263–298.

[10] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, and V. S. Pande, Accelerating molecular dynamic simulation on graphics processing units, Journal of Computational Chemistry **30** (2009) 864–872.

[11] 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.

[12] Lars Nyland, Mark Harris, and Jan Prins, Fast N-Body Simulation with CUDA, in: GPU gems 3, Addison-Wesley, 2008, Chapter 31, 677–696.

[13] S. S. Stone, J. P. Haldar, S. C. Tsao, W. Hwu, B. P. Sutton, and Z. P. Liang, Accelerating advanced MRI reconstructions on GPUs, J. Parallel Distrib. Comput. **68** (2008) 1307–1318.

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

[15] J. E. Stone, J. Saam, D. J. Hardy, K. L. Vandivort, W.-m. W. Hwu, and K. Schulten, High performance computation and interactive display of molecular orbitals on GPUs and multi-core CPUs, in: Proceedings of the 2nd Workshop on General-Purpose Processing on Graphics Processing Units, ACM International Conference Proceeding Series, vol. 383, 9–18, 2009.

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

[17] T. R. Hagen, K.-A. Lie, and J. R. Natvig, Solving the Euler equations on Graphics Processing Units, Lecture Notes in Computer Science **3994** (2006) 220–227.

[18] A. Kloeckner, T. Warburton, J. Bridge, and J.S. Hesthaven, High-order discontinuous Galerkin methods on graphics processors, Journal of Computational Physics, submitted, 2009.

[19] M. L. Saetra, Solving systems of hyperbolic PDEs using multiple GPUs, Master's thesis, University of Oslo, 2007.

[20] D. van Dyk, M. Geveler, S. Mallach, D. Ribbrock, D. Goeddeke, and C. Gutwenger, HONEI: A collection of libraries for numerical computations targeting multiple processor architectures, Computer Physics Communications, in press, 2009.

[21] J. C. Phillips, J. E. Stone, and K. Schulten, Adapting a message-driven parallel application to GPU-accelerated clusters, in: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 2008, 1–9.

[22] V. V. Kindratenko, J. J. Enos, M. T. Guochun Shi Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, W.-m. Hwu, GPU clusters for high-performance computing, in: IEEE International Conference on Cluster Computing, 2009.

[23] SHARCNET website, www.sharcnet.ca .

[24] Juelich Supercomputing Centre website, www.fz-juelich.de .

[25] Scalable Scientific Computing group website, University of Waterloo, http://www.math.uwaterloo.ca/groups/SSC/software.

[26] Green 500 website, www.green500.org .

[27] Top 500 website, www.top500.org .

[28] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, Entering the petaflop era: The architecture and performance of Roadrunner, in: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 2008, 1–11.

[29] NVIDIA CUDA Programming Guide. Version 2.3.1.

[30] J. Nickolls, I. Buck, M. Garland, and K. Skadron, Scalable parallel programming with CUDA, Queue **6** (2008) 40–53.

[31] NVIDIA's Next Generation CUDA Compute Architecture Fermi, white paper, 2009.

[32] S. Rostrup, Solving Hyperbolic PDEs using Accelerator Architectures, Master's thesis, Department of Applied Mathematics, University of Waterloo, 2009.

[33] S. Rostrup and H. De Sterck, Hybrid MPI-Cell Parallelism for Hyperbolic PDE Simulation on a Cell Processor Cluster, in: Proceedings of the High Performance Computing Symposium Symposium, Kingston, Ontario, 2009.

[34] Intel C++ Compiler User and Reference Guides, Ch. 26, pp. 1385-1387. Document number: 304968-023US.

[35] D. Callahan, S. Carr, and K. Kennedy, Improving Register Allocation for Subscripted Variables, SIGPLAN Not. 39, 4 (Apr. 2004), 328-342.

[36] C. Ding and P. Sweany, Improving Software Pipelining with Unroll-and-Jam and Memory Reuse Analysis, Master's thesis, Department of Computer Science, Michigan Technological University, 1996.

[37] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, Accelerating molecular modeling applications with graphics processors, Journal of Computational Chemistry 28 (2007) 2618–2640.

[38] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, 2008, 73–82.

[39] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf and K. Yelick, Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures, in: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 2008, 1–12.

[40] S.W. Williams, A. Waterman, and D.A. Patterson, Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures (Tech. Report UCB/EECS-2008-134, EECS Department, University of California, Berkeley, 2008).

[41] Khronos Group, OpenCL, www.khronos.org/opencl .

[42] K. Asanovic, R. Bodik, J. Demmel, J. Kubiatowicz, K. Keutzer, E. Lee, G. Necula, D. Patterson, K. Sen, J. Shalf, J. Wawrzynek, and K. Yelick, The landscape of parallel computing research: A view from Berkeley (Tech. Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006).

[43] NVIDIA Tesla GPUs To Communicate Faster Over Mellanox InfiniBand Networks, www.nvidia.com/object/io_1258539409179.html, retrieved December 2, 2009.