Technical Report TR-08-21
Computational Science Laboratory
Department of Computer Science
Virginia Tech

Multilayered Heterogeneous Parallelism Applied to Atmospheric Constituent Transport Simulation

John C. Linford Adrian Sandu

October 2008

Technical Report TR-08-21

Multilayered Heterogeneous Parallelism Applied to Atmospheric Constituent Transport Simulation

Chemical Transport Modeling and the Cell Broadband Engine Architecture

John C. Linford · Adrian Sandu

Preprint submitted to Journal of Supercomputing

Abstract Heterogeneous multicore chipsets with many levels of parallelism are becoming increasingly common in high-performance computing systems. Effective use of parallelism in these new chipsets constitutes the challenge facing a new generation of large scale scientific computing applications. This study examines methods for improving the performance of two-dimensional and three-dimensional atmospheric constituent transport simulation on the Cell Broadband Engine Architecture (CBEA). A function offloading approach is used in a 2D transport module, and a vector stream processing approach is used in a 3D transport module. Two methods for transferring incontiguous data between main memory and accelerator local storage are compared. By leveraging the heterogeneous parallelism of the CBEA, the 3D transport module achieves performance comparable to two nodes of an IBM BlueGene/P, or eight Intel Xeon cores, on a single PowerXCell 8i chip. Module performance on two CBEA systems, an IBM BlueGene/P, and an eight-core shared-memory Intel Xeon workstation are given.

Keywords Chemical Transport Modeling · Cell Broadband Engine · Vector Stream Processing · Fixedgrid

John C. Linford
Department of Computer Science
Virginia Polytechnic Institute and State University
2202 Kraft Dr., Blacksburg, VA 24061
Tel.: +001-540-231-6186, Fax: +001-540-231-6075, E-mail: jlinford@vt.edu

Adrian Sandu
Department of Computer Science
Virginia Polytechnic Institute and State University
2202 Kraft Dr., Blacksburg, VA 24061
Tel.: +001-540-231-2193, Fax: +001-540-231-6075, E-mail: sandu@cs.vt.edu

1 Introduction

Increasing power consumption, heat dissipation, and other issues have lead to the rapid prevalence of multicore microprocessor chip architectures in recent years. Exploiting the potential parallelism of multiple cores on a single chip is an unprecedented challenge to application developers. A growing body of knowledge suggests heterogeneous multicore chipsets will be integral to the next generation of supercomputers. Roadrunner, a heterogeneous distributed memory supercomputer based on the Cell Broadband Engine Architecture (CBEA), is the first system to achieve a sustained petaFLOPS.[1] This is prominent evidence of the potential of heterogeneous multicore chipsets. Multilayered heterogeneous parallelism offers an impressive number of FLOPS, but at the cost of staggering hardware and software complexity. In order to build the next generation of scientific software for heterogeneous systems, multiple levels of heterogeneous parallelism must be applied effectively.

Geophysical models are comprehensive multiphysics simulations common in the environmental and life sciences. These models consider hundreds of variables such as temperature, pressure, terrain features, diffusion rates, chemical concentrations, reaction rates, and wind vector fields for every point in the domain. These models solve systems of equations involving between 10^6 and 10^9 double-precision floating point values, require hours or days of runtime on small compute clusters, and tend to be highly parallel. Mass flux may be approximated though a finite difference scheme on a fixed domain grid, and chemical reactions, photosynthetic processes, and particulate processes are embarrassingly parallel on a fixed domain grid. Subdividing the problem through domain decomposition, appropriate application of data-parallel BLAS operations, and numerical integration schemes with weak data dependencies have been used successfully to parallelize geophysical models on shared-memory and distributed-memory clusters.

Multicore chipsets push the physical constraints of computation by allowing many processes to be assigned to the same node while reducing the machine's physical footprint. Many instances of the same core on a single chip may be economically or physically infeasible, particularly if caches are shared between cores. In practice, memory I/O bottlenecks scalability long before every node achieves maximum load. Heterogeneous, or *accelerated*, multicore chipsets specify subsets of cores for particular applications to optimize on-chip space. For example, cores may be specialized for memory I/O or compute-intensive liner algebra. The specialized cores are called *accelerators* since they rely a traditional CPU for general computation. An accelerator's high FLOP count and additional memory I/O efficiency hides memory latency as the chip is oversubscribed. Therefore, accelerated multicore chipsets exhibit much stronger scalability than homogeneous multicore chipsets while facilitating good physical scalability.

[1] http://www.ibm.com/news/us/en/2008/06/2008_06_09.html

This study demonstrates the potential of heterogeneous multicore architectures to speed up geophysical simulations. Several methods for improving the performance of the chemical constituent transport module in FIXEDGRID, a prototype comprehensive atmospheric model, on the CBEA are explored. The major contributions of this study are:

1. An examination of function offloading, a popular multicore programming paradigm, as a method for porting a two-dimensional chemical constituent transport module to the CBEA.
2. A scalable method for transferring incontiguous data to and from CBEA accelerator cores. Virtually every scientific code involves multidimensional data structures with incontiguous data. Scalability can be severely hampered if careful consideration is not given to data layout and transfer.
3. A method for using every layer of polymorphic parallelism in the CBEA called *vector stream processing*. This method combines asynchronous explicit memory transfers with vectorized kernel functions to maximize accelerator core throughput while maintaining code readability. Our implementation refactors statically for single- or double-precision data types.
4. A detailed performance analysis of two- and three-dimensional finite difference fixed grid chemical constituent transport on three CBEA systems, an IBM BlueGene/P distributed-memory system, and a Intel Xeon shared memory multiprocessor.

The experimental results show that function offloading is a good method for porting existing scientific codes, but it may not achieve maximum performance. Vector stream processing produces better scalability for computationally-intense constituent transport simulation. By using vector stream processing to leverage the polymorphic parallelism of the CBEA, the 3D transport module achieves the same performance as two nodes of an IBM BlueGene/P supercomputer, or eight cores of an Intel Xeon homogeneous multicore system, with one CBEA chip.

2 Related Work

2.1 Homogeneous Multicore Chipsets

There is a growing body of work documenting the performance of specialized algorithms on multicore processors. Chen et al. [10] optimized the Fast Fourier Transform on the IBM Cyclops-64 chip architecture, a large-scale homogeneous multicore chip architecture with 160 thread units. Their FFT implementation, using only one Cyclops-64 chip, achieved over four times the performance of an Intel Xeon Pentium 4 processor, though a thorough understanding of the problem and of the underlying hardware were critical to achieving good performance in their application.

Douillet and Gao [15] developed a method for automatically generating a multi-threaded software-pipelined schedule from parallel or non-parallel imperfect loop nests written in a standard sequential language such as C or

FORTRAN. Their method exhibits good scalability on the IBM Cyclops-64 multicore architecture up to 100 cores, showing that enhanced compilers can produce efficient multicore code, even if the programmer lacks domain-specific knowledge. However, these methods are not easily transferable to heterogeneous architectures, as they assume every core is equally applicable to every task.

Many high-performance applications show significant performance improvement on multicore processors. Alam and Agarwal [2] characterized computation, communication and memory efficiencies of bio-molecular simulations on a Cray XT3 system with dual-core Opteron processors. They found that the communication overhead of using both cores in the processor simultaneously can be as low as 50% as compared to the single-core execution times.

2.2 Heterogeneous Multicore Chipsets

The Cell Broadband Engine (Cell BE) was developed as a multimedia processor, so it's no surprise that it achieves excellent performance as an H.264 decoder [4], JPEG 2000 encoding server [28], speech codec processor [34], ray tracing engine [26, 5], and high performance MPEG-2 decoder [3]. The Cell BE is also a promising platform for high-performance scientific computing. Williams et al. achieved a maximum speedup of 12.7x and power efficiency of 28.3x with double-precision general matrix multiplication, sparse matrix vector multiplication, stencil computation, and Fast Fourier Transform kernels on the Cell BE, compared with AMD Opteron and Itanium2 processors [33]. Their single precision kernels achieved a peak speedup of 37.0x with 82.4x power efficiency when compared with the AMD Opteron.

Leveraging the parallelism of heterogeneous systems can be particularly fruitful. Blagojevic et al. [6] developed a runtime system and scheduling policies to exploit polymorphic and layered parallelism on the CBEA. By applying their methods to a Maximum Likelihood method for phylogenetic trees, they realized a 4x performance increase on the Cell BE in comparison with a dual-processor Hyperthreaded Xeon system. Hieu et al. [21] combined the heterogeneous SIMD features of the SPEs and PPE to achieve an 8.8x speedup for the Striped Smith-Waterman algorithm as compared to an Intel multicore chipset with SSE2 and a GPGPU implementation executed on NVIDIA GeForce 7800 GTX. Our work applies multiple layers of heterogeneous parallelism to finite difference codes on a fixed grid.

Applying heterogeneous SIMD to scientific codes is an active topic of research. Ibrahim and Bodin [24] introduced *runtime data fusion* for the CBEA to dynamically reorganize finite element data to facilitate SIMD-ization while minimizing shuffle operations. By combining this method with hand-optimized DMA requests and buffer repairs, they achieved a sustained 31.2 gigaFLOPS for an implementation of the Wilson-Dirac Operator. Furthermore, by using the heterogeneous features of the PPE to analyze data frames, they were able to reduce memory bandwidth by 26%. Our work takes advantage of the prob-

lem structure of finite diference atmospheric simulation to minimize runtime data preparation. Data is padded and organized statically such that only vector starting addresses must be calculated at runtime.

2.3 Stream Processing Architectures

Stream processing is a programming paradigm for exploiting parallelism. Given a stream of data, a set of computational kernels (functions or operations) are applied to each element in the stream. Often the kernels are pipelined. This method is particularly useful when programming several separate computational units without explicitly managing allocation, synchronization, and communication.

Sequoia is a popular stream processing programming language with implementations for CBEA, GPGPU, distributed memory clusters, and others [17]. Sequoia simplifies stream processing, but often imposes a performance penalty. The current version of Sequoia for the CBEA makes extensive use of the mailbox registers, resulting in significant SPE idle times. Improving Sequoia performance is an active research topic. SPADE is a declarative stream processing engine for the System S stream processing middleware [19]. It targets versatility and scalability to be adequate for future-generation stream processing platforms. A prototype implementation for the CBEA is under development. Erez and Ahn [16] mapped unstructured mesh and graph algorithms to the Merrimac [13] stream processing infrastructure and achieved between 2.0x and 4.0x speedup as compared to a baseline architecture that stresses sequential access. We use triple-buffered explicit memory transfers with instruction barrier synchronization to create multiple SIMD data streams on the CBEA while avoiding the mailbox registers. We call this approach *vector stream processing*.

3 The Cell Broadband Engine Architecture

The IBM Cell Broadband Engine (Cell BE) is a heterogeneous multicore processor which has drawn considerable attention in both industry and academia. The Cell BE was originally designed for the game box market, and therefore it has a low cost and low power requirements. Nevertheless, it has archived unprecedented peak single-precision floating point performance, making it suitable for high-performance computing.

The Cell BE is the first implementation of the Cell Broadband Engine Architecture (CBEA). The main components of the CBEA are a multithreaded Power Processing element (PPE) and eight Synergistic Processing elements (SPEs) [18]. These elements are connected with an on-chip Element Interconnect Bus (EIB) with a peak bandwidth of 204.8 Gigabytes/second. The PPE is a 64-bit dual-thread PowerPC processor with Vector/SIMD Multimedia extensions [12] and two levels of on-chip cache. Each SPE is a 128-bit processor with two major components: a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC). All SPE instructions are executed on the SPU.

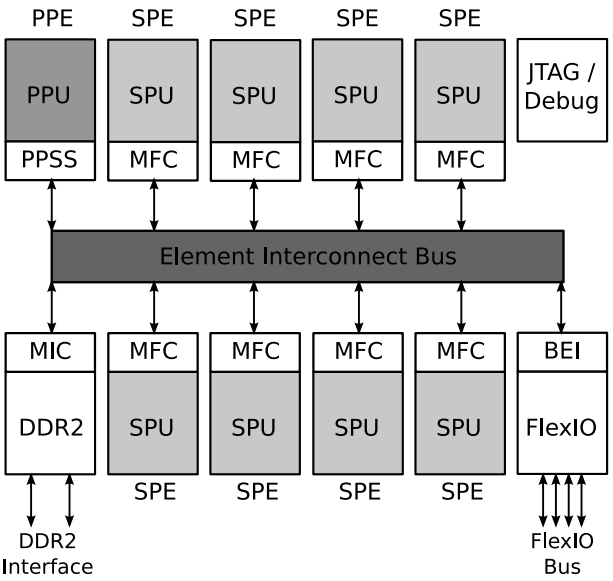


Fig. 1 A high-level description of the Cell Broadband Engine based on the PowerXCell 8i product brief. Available components vary by platform. JTAG and debug interfaces are only available in IBM BladeCenter installations, and the Sony PlayStation 3 has only six programmable SPEs

The SPE includes 128 registers of 128 bits and 256 KB of software-controlled local storage.

The only memory directly available to an SPE is its own local storage. An SPE must use direct memory access (DMA) requests to access RAM. All DMA transfers are handled by the MFC. Data transferred between local storage and main memory must be 8-byte aligned and no more than 16 KB in size. The MFC supports only DMA transfer of blocks are are 1, 2, 4, 8, or multiples of 16 bytes long. Data blocks that are a multiple of 128 bytes in size and 128-byte aligned are transfered most efficiently.

The Cell BE is primarily a single-precision floating point processor. Its peak single-precision FP performance is 230.4 Gflops, but peak performance for double-precision is only 21.03 [11]. Mixed-precision methods should be considered when solving double-precision problems on this chipset [7]. The PowerXCell 8i processor is the second implementation of the CBEA and is intended for high-performance, double-precision floating-point intensive workloads that benefit from large-capacity main memory. According to the processor technology brief, double-precision codes may see up to an 8x performance boost with this chipset. Roadrunner at Los Alamos, the first computer to achieve a sustained petaFLOPS, uses the PowerXCell 8i processor.

The CBEA's heterogeneous design and non-blocking data access capabilities allow for a variety of programming methodologies. A *traditional approach*, which views each SPE as an independent processor, is certainly possible. This

is the common approach for codes which are predominately composed of homogeneous parallel routines and codes ported directly from homogeneous multicore architectures. The traditional approach is often easier to program and may reduce debugging and optimization times, but it sacrifices much of the CBEA's processing power. The CBEA also excels as a *stream processor*. When used as a stream processor, the SPUs apply kernel functions to data provided by the MFC. SPEs can be pipelined to rapidly apply multiple kernels to the same stream of data. Another popular model is the *function offload model*. In this case, functions with SIMD operations or stream-like properties are offloaded to the SPEs, while the PPE performs the bulk of general processing. The CBEA's polymorphic parallelism allows combinations of all these methods to be dynamically applied as needed.

4 Atmospheric Chemistry and Transport Modeling

Earth's atmospheric state is the result of a complex and dynamic combination of hundreds of interacting processes. Weather prediction, air quality policy, and climate change prediction are just a few of the many practical applications that require accurate atmospheric models. In order to accurately describe the chemical state of the atmosphere, a model must consider the chemical interactions of air-born chemical species together with their sources, long range transport, and deposition. A chemical kinetics module is used to describe chemical interactions, and a chemical constituent transport module is used to describe the motion of chemicals in the atmosphere. These processes are referred to generally as *chemistry* and *transport*, respectively. This study focuses on transport.

4.1 Chemical Transport Modeling

Chemical transport modules (CTMs) solve mass balance equations for concentrations of trace species in order to determine the fate of pollutants in the atmosphere [30]. These equations are described in detail in [31] and [22]. The basic mathematical equations for transport and reaction can be derived by considering these mass balance equations. Let $\mathbf{c}_x^t = (c_1(x,t)...c_s(x,t))^T$ be concentrations of s chemical species, with spatial variable $x \in \Omega \subset \mathbb{R}^d$ (d = 2 or 3), and time $t \geq 0$. Transport and reaction are given by

$$\frac{\partial}{\partial t}\mathbf{c}_x^t = -\sum_{k=1}^{d} \frac{\partial}{\partial x_k}\left(a_k(x,t)\mathbf{c}_x^t\right) + \sum_{k=1}^{d} \frac{\partial}{\partial x_k}\left(d_k(x,t)\frac{\partial}{\partial x_k}\mathbf{c}_x^t\right) + f_j(\mathbf{c}_x^t, x, t) \quad (1)$$

where a_k are the atmospheric wind velocities, d_k are the turbulent diffusion coefficients, and $f_j(\mathbf{c}_x^t, x, t)$ describes the nonlinear chemistry of chemical species s together with emissions and deposition.

By considering a simple space discretization on a uniform grid ($x_i = ih$) with a mesh width $h = \frac{1}{m}$, solutions to Equation 1 can be approximated by

finite difference methods. This produces the third order upwind-biased advection discretization (Equation 2) and the second order diffusion discretization (Equation 3) for $t' > t$ [23].

$$\frac{\partial}{\partial t}\mathbf{c}_x^t = \begin{cases} \frac{a}{h}\left(-\frac{1}{6}\mathbf{c}_{x-2}^t + \mathbf{c}_{x-1}^t\right) - \frac{a}{h}\left(\frac{1}{2}\mathbf{c}_x^t - \frac{1}{3}\mathbf{c}_{x+1}^t\right) & \text{if } a \geq 0 \\ \frac{a}{h}\left(\frac{1}{3}\mathbf{c}_{x-1}^t + \frac{1}{2}\mathbf{c}_x^t\right) - \frac{a}{h}\left(\mathbf{c}_{x+1}^t + \frac{1}{6}\mathbf{c}_{x+2}^t\right) & \text{if } a < 0 \end{cases} \tag{2}$$

$$\frac{\partial}{\partial t}\mathbf{c}_x^t = \frac{(\mathbf{d}_{x-1}^t + \mathbf{d}_x^t)(\mathbf{c}_{x-1}^t + \mathbf{c}_x^t)^T}{2h^2} - \frac{(\mathbf{d}_x^t + \mathbf{d}_{x+1}^t)(\mathbf{c}_x^t + \mathbf{c}_{x+1}^t)^T}{2h^2}. \tag{3}$$

The 3D stencil for upwind-biased discretization is shown in Figure 2. *Dimension splitting* can be used to to apply Equations 2 and 3 to each dimension of a d-dimensional model independently. Parallelism is introduced by reducing a d-dimensional problem to a set of independent one-dimensional problems. Equations 2 and 3 can be implemented as a single computational routine and applied to each dimension of the concentration matrix individually and in parallel. This forms the *kernel* functions for use in streaming architectures. Dimension splitting is common in many transport models and other scientific applications, thus the techniques given in this paper are applicable to existing codes.

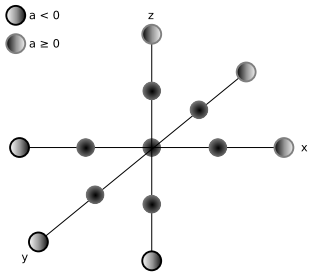


Fig. 2 3D discretization stencil for explicit upwind-biased advection/diffusion

Dimension splitting introduces a local truncation error at each time step. This error can be reduced by a symmetric time splitting method [23]. Time splitting methods are frequently used when applying different time stepping methods to different parts of an equation. For example, chemical processes are stiff, which calls for an implicit ODE method, but explicit methods are more suitable for space-discretized advection. By interleaving time steps taken in each dimension, the truncation error is reduced. In brief, a linear ODE describing time-stepped chemical transport, $w'(t) = Aw(t)$ where $A = A_1 +$

$A_2 + A_3$, can be approximated by

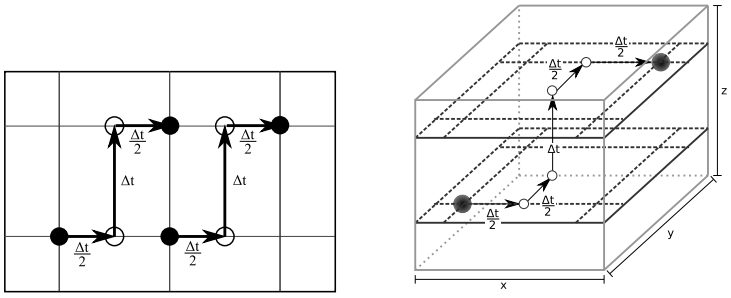
$$w_{n+1} = e^{\Delta t A} w_n \approx e^{\frac{\Delta t}{2} A_3} e^{\frac{\Delta t}{2} A_2} e^{\Delta t A_1} e^{\frac{\Delta t}{2} A_2} e^{\frac{\Delta t}{2} A_3} w_n, \tag{4}$$

where $e^B = I + B + \frac{1}{2}B^2 + \ldots + \frac{1}{k!}B^k + \ldots$ is the exponential function of a matrix, and Δt is the timestep. This is a multi-component second order Strang splitting method [32]. For three-dimensional discretization, take A_1, A_2, A_3 to be representative of discretization along the z, y, and x axes, respectively, or for two-dimensional discretization, take $A_3 = [0]$ and A_2, A_3 to be representative of discretization along the x and y axes, respectively. Figure 3 illustrates time splitting applied to 2D and 3D discretization. A half time step ($\frac{\Delta t}{2}$) is used when calculating mass flux along the minor axes, and a whole time step is used when calculating mass flux along the major axis. This process is equal to calculating the mass flux in one step, but reduces truncation error to $\mathbf{O}(\Delta t^3)$ [23].

5 FIXEDGRID

FIXEDGRID is a comprehensive prototypical atmospheric model based the state-of-the-art Sulfur Transport and dEposition Model version 3 (STEM-III) [8]. FIXEDGRID describes chemical transport according to Equations 2 and 3 and uses an implicit Rosenbrock method to integrate a 79-species SAPRC'99 [9] atmospheric chemical mechanism for VOCs and NOx. Chemistry and transport processes can be selectively disabled to observe their effect on monitored concentrations, and FIXEDGRID's modular design makes it easy to introduce new methods. 2D and 3D transport kernels allow FIXEDGRID to address a range of mass-balance problems.

FIXEDGRID is an experimental platform for geophysical modeling. The domains simulated by FIXEDGRID are simplistic in comparison to the real world, but the processes are those used in production models. Properly-formatted real-world data may be used without issue. Mechanisms can be optimized and



(a) Two-dimensions (b) Three-dimensions

Fig. 3 Second order time splitting methods for 2D and 3D transport discretization

profiled in FIXEDGRID without confounding performance variables, and various programming methodologies can be reliably compared without rewriting a comprehensive model.

Most emerging multicore technologies support the C programming language or some derivative of C, so FIXEDGRID is written in 100% C. STEM-III is a mix of FORTRAN 77, Fortran 90, and C, so the relevant parts of STEM-III were translated from Fortran to C for FIXEDGRID. Although there are several tools for automatically translating Fortran to C, such tools often produce code that is difficult to read and maintain, so the Fortran source was translated by hand. All Fortran parameter statements were changed to C preprocessor directives and modules were encapsulated by C header files. This produced the basic serial version, suitable for any C compiler, which is the baseline for experimental measurement. This version does not support multiple cores, SIMD, or any accelerator features, but it has been tuned to minimize array copies and exploit locality. The source code is approximately 62% larger than the original Fortran, mostly due to C's lack of array syntax.

5.1 Data Storage and Layout

Atmospheric simulation requires a considerable amount of data. For each point in a d-dimensional domain, a state vector consisting of wind vectors, diffusion rates, temperature, sunlight intensity, and chemical concentrations for every species of interest are required. On an $x \times y \times z$ domain of $600 \times 600 \times 12$ points considering $n = 79$ chemical species, approximately 367.2×10^6 double-precision values (2.8GB) are calculated at every iteration. FIXEDGRID stores this data in a dynamically allocated 4D array: `CONC[`s`][`z`][`y`][`x`]`.

Data organization in memory is critically important. Accelerator architectures derive much of their strength from SIMD operations and/or asynchronous explicit memory transfers. For these operations to succeed, the data must conform to the architecture requirements. When FIXEDGRID executes on the CBEA, data transferred between SPE local storage and main memory must be contiguous, 8-byte aligned, no larger than 16 KB, and in blocks of 1, 2, 4, 8, or multiples of 16 bytes. Transfers of less than 16 bytes are highly inefficient. To meet these requirements in three dimensions, each row of the data matrices is statically padded to a 16-byte boundary, effectively adding buffer y-z planes (see Figure 7) and aligned with `__attribute__((aligned(128)))`. Two-dimensional FIXEDGRID simulations are padded in the same way, but with a vector instead of a plane ($z = 1$).

Aligning and padding permits access to all data, however not every element can be accessed efficiently. One single-/double-precision floating-point array element uses 4/8 bytes of memory. Since transfers of 16 bytes or more are most efficient on the CBEA, an array element with row index i where $address(i)|16 \neq 0$ should be fetched by reading at least 16 bytes starting at row index j where $0 \leq j < i$ and $address(j)|16 == 0$ into a buffer and then extracting the desired element from the buffer. Preprocessor macros can be used

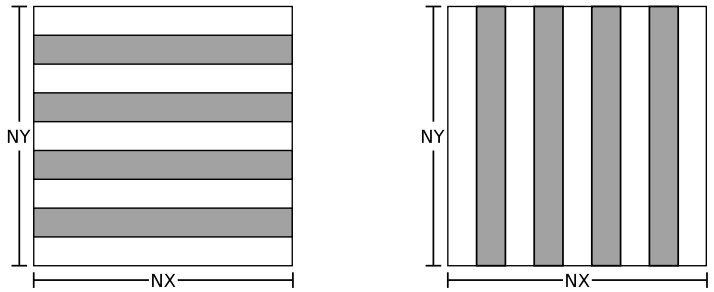
to calculate basic vector metrics (i.e. number of elements per 16-byte vector) and statically support both 4-byte and 8-byte floating point types with a clear, simple syntax.

FIXEDGRID uses second order time splitting to reduce truncation error in 2D/3D transport (see Figure 3). A half time step ($\frac{\Delta t}{2}$) is used when calculating mass flux along the domain minor axes, and a whole time step is used when calculating mass flux along the domain major axis. Note that this reduces the truncation error but doubles the work required to calculate mass flux for the minor axes. It is therefore desirable for 2D row and 3D x/y-discretization to be highly efficient. Concentrations of the same chemical species are stored contiguously to maximize locality and avoid buffering in transport calculations.

The block nature of FIXEDGRID's domain makes it ideally parallelizable. Distributing a large matrix across many compute nodes is a thoroughly-explored problem. This study considers execution on one node.

6 Two-dimensional Transport

This section discusses the implementation, analysis, and optimization of FIXEDGRID's 2D transport module for the CBEA using a *function offload* approach. Transport in the concentration matrix is calculated in rows (discretization along the x-axis) and in columns (discretization along the y-axis), as shown in Figure 4. The 2D transport module was ported from the original Fortran, rather than written it from scratch, as an exercise in porting an existing scientific code to the CBEA. This was a nontrivial effort and resulted in three new versions of the transport module. Each new version was carefully profiled and analyzed to gauge the benefit of the changes introduced. Performance is reported in Section 6.4.



(a) Row-order decomposition.

(b) Column-order decomposition.

Fig. 4 Domain decomposition for two-dimensional transport

6.1 Naive Function Offload

A *function offloading* approach was used to accelerate the 2D transport mechanism. Function offloading identifies computationally-intense functions and moves these functions to the SPEs. The main computational core for 2D transport is the `discretize()` function, which calculates mass flux in a single row or column. A naive offload was applied to move the unmodified function code to the SPEs while ignoring the architecture details of the SPE, such as the SPE's SIMD ISA and support for overlapping computation and communication. When `discretize()` is invoked, each SPE receives the main-memory address of a `fixedgrid_spe_argv_t` structure (Listing 1) containing the parameters to the `discretize()` function. The SPE fetches the parameters via DMA. `fixedgrid_spe_argv_t` is exactly 128 bytes long and aligned on a 128-byte boundary to maximize transfer speed. The PPU signals the SPE to invoke the offloaded function via mailbox registers.

```
1  typedef union
2  {
3    double   dbl;
4    uint64_t u64;
5    uint32_t u32[2];
6  } arg_t;

8  typedef struct
9  {
10   arg_t arg[16];
11 } fixedgrid_spe_argv_t
12   __attribute__((aligned(128)));
```

Listing 1 User-defined data types for passing arguments to the SPE program

The only memory available to the SPE directly is its own 256KB of local storage, which is shared between code and data. Considering code size and neglecting overlays, approximately 2000 double-precision variables can be held in SPE local storage. Any data exceeding this limit is processed piecewise. Data in main memory must be contiguous and aligned on a 16-byte boundary to be eligible for DMA transfer. Column data in the concentration matrix is problematic for the CBEA, since it is incontiguous and odd-indexed columns (for double-precision data) are never on a 16-byte boundary.

The PPU was used to buffer column data contiguously in main memory, permitting DMA copy to and from SPE local storage. This is a straight-forward method for solving data discontinuity, but it triples the number of copies required for column-wise updates and consumes $\mathbf{O}(y \times N)$ additional main-memory elements, where y is the number of double-precision elements in a column and N is the number of SPEs. However, no additional local storage is consumed.

6.2 Improved Function Offload

Naive function offload neglects the SPE's SIMD ISA and asynchronous memory I/O capabilities. Offloaded function performance can be improved through

triple-buffered asynchronous DMA, loop unrolling, and SIMD intrinsics. Tripple-buffered DMA enables each SPE to simultaneously load the next row or column into local storage, process the current row or column, and write previously-processed data back to main memory. Six buffers are required to triple-buffer incontiguous data: three in local storage and three for data reorganization in main memory by the PPE. Only three local storage buffers are required for contiguous data. Asynchronous DMA was used in the 2D transport function to reduce runtime by approximately 23%.

The vector data types and vector arithmetic functions provided in the CBE SDK library of SIMD intrinsics simplify vector programming on the SPE. These intrinsics signal the SPE compiler to vectorize certain operations, although the compiler can recognize and auto-vectorize some code. Both manual vectorization through SIMD intrinsics and compiler auto-vectorization were used to reduce runtime by a further 18%.

The SPU hardware assumes linear instruction flow and produces no stall penalties from sequential instruction execution. A branch instruction may disrupt the assumed sequential flow. Correctly predicted branches execute in one cycle, but a mispredicted branch incurs a penalty of approximately 18-19 cycles. The typical SPU instruction latency is between two and seven cycles, so mispredicted branches can seriously degrade program performance. Function inlining and loop unrolling were used to avoid branches in the offloaded function.

The byte-size of a data element must be carefully considered when optimizing the offloaded function. If single-precision floating point data is used, a vector on the SPU contains four elements, as opposed to just two for double-precision floating point data. Since data must be aligned on a 16-byte boundary to be eligible for DMA transfer, retrieving column elements of the concentration matrix which are not multiples of the vector size will result in a bus error. Thus, two/four columns of data must be retrieved when using single/double precision vector intrinsics. This reduces runtime by increasing SPE bandwidth, but puts additional strain on the SPE local storage.

6.3 Scalable Incontiguous DMA

Incontiguous data can be transferred directly to SPE local storage via *DMA lists*. A DMA list is an array of main memory addresses and transfer sizes, stored as pairs of unsigned integers, which resides in SPE local storage. Each element describes a DMA transfer, so the addresses must be aligned on a 16-byte boundary and the transfer size may not exceed 16KB.

To transfer a column of y elements in the concentration matrix requires a DMA list of y entries. The same DMA list can be used for both writing to and reading from main memory. Single-/double-precision variables are four/eight bytes long, so at least four/two columns should be fetched simultaneously, even if only one column is desired. When the DMA list is processed, the two columns are copied to the same buffer in local storage with their elements

interleaved. The desired column is then extracted by the SPE from the local storage and fed into the offloaded function. Scalability is greatly improved by moving the array copy loops to the SPEs, but $\mathbf{O}(y \times 2 \times 3)$ additional local storage elements are required. However, no additional buffers are required in main-memory.

Figure 5 shows how DMA lists improve 2D transport performance. "Row discret" and "Col discret" give the inclusive function time for transport calculations in the rows/columns of the concentration matrix, respectively. In Figure 5(b), "Row discret" is clearly decreasing as more SPEs are used, yet "Col discret" remains relatively constant. When DMA lists are used (Figure 5(c)) both "Col discret" and "Row discret" decrease as more SPEs are used.

6.4 Analysis and Comparison

This section discusses the performance of FIXEDGRID's 2D transport module on three CBEA systems, a distributed memory multicore system, and a shared memory multicore system. The experiments calculate ozone (O_3) concentrations on a domain of $600 \times 600 \times 12$ grid cells for 12 hours with a $\Delta t = 50$ seconds time step. The CBEA systems presented are:

JUICEnext: a 35-node IBM BladeCenter QS22 at Forschungszentrum Jülich. Each node has 8GB XDRAM and four 3.2GHz PowerXCell 8i CPUs with eight SPEs each.

CellBuzz: a 14-node IBM BladeCenter QS20 at Georgia Tech. Each node has 1GB XDRAM and two 3.2 GHz Cell BE CPUs with eight SPEs each.

Monza: a 10-node PlayStation 3 cluster at Virginia Tech. Each Monza node has 256 MB XDRAM and one 3.2GHz Cell BE CPU with six SPEs.

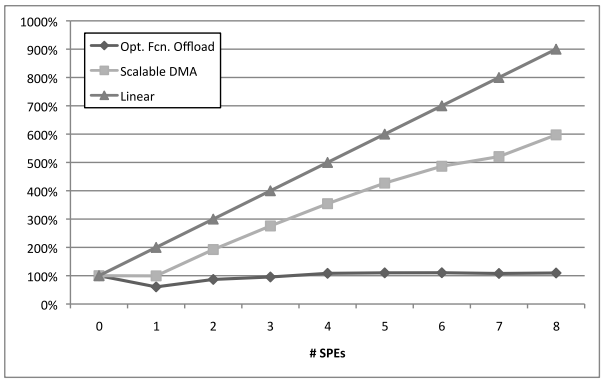
The performance of the CBEA systems is compared with the performance of two homogeneous multicore systems:

Jugene: a 16-rack IBM BlueGene/P at Forschungszentrum Jülich.[2] Each rack contains 32 nodecards, each with 32 nodes. Each node contains an 850MHz 4-way SMP 32-bit PowerPC 450 CPU and 2GB DDR RAM (512MB RAM per core). Jugene premiered as the second fastest computer on the Top 500 list and presently ranks in 6th position.

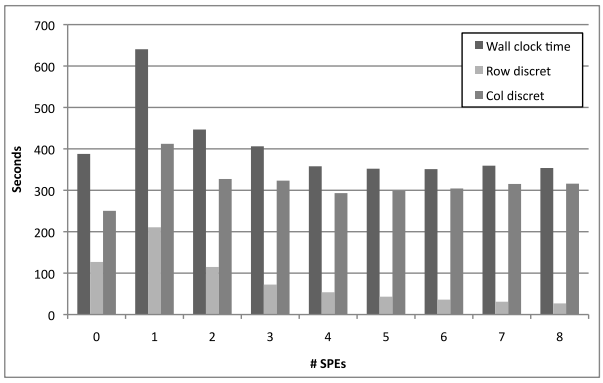
Deva: a Dell Precision T5400 workstation with two 2.33GHz Intel Quad-core Xeon ES5410 CPUs and 16GB DDR RAM.

The experiments were compiled with the GCC on all CBEA platforms. IBM XLC was used on Jugene, and the Intel C Compiler (ICC) was used on Deva. All experiments were compiled with maximum optimization (-O5

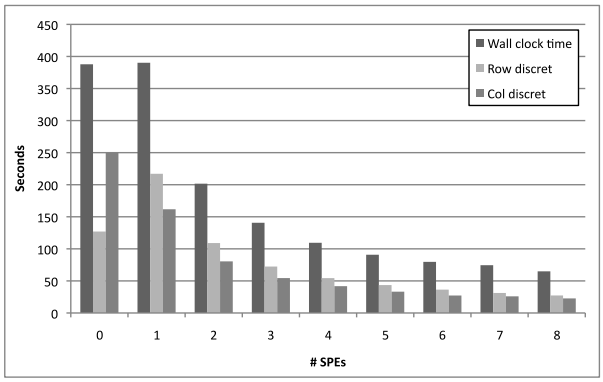
[2] We classify the BlueGene/P as a homogeneous multicore system, even though it may be considered a heterogeneous multicore system. The head nodes and "Double Hummer" dual FPUs introduce a degree of heterogeneity, however the BlueGene/P is homogeneous on the node-level.



(a) Speedup for optimized function offload with and without DMA lists



(b) Inclusive function times for optimized function offload.



(c) Inclusive function times for optimized function offload with DMA lists

Fig. 5 Performance of the 2D transport module executing on JUICEnext (IBM BladeCenter Q22) when matrix columns are buffered by the PPU (Opt. Fcn. Offload) and when DMA lists are used (Scalable DMA)

or -O3 where appropriate) and double-precision floating point operations.[3] The presented results are the average of three runs on each platform. The measurement error was insignificant, so error bars are not shown.

Tables 1 and 2 list the wall clock runtime performance on the CBEA systems and homogeneous multicore systems, respectively. These experimental results show the importance of leveraging all levels of heterogeneous parallelism in the CBEA. Initial versions of the 2D transport module, which make little or no use of heterogeneous parallelism, perform poorly in comparison to the final version, which uses asynchronous memory I/O and SIMD. It is clear that the PPU can be a severe scalability bottleneck if it preprocesses for the SPEs. It is more efficient to to use methods which can be offloaded to the SPEs, even though this introduces additional copy operations and consumes SPE local storage. Figure 6 shows the wall clock runtime performance and speedup of the final version of FIXEDGRID's 2D transport module on all five systems.

The CBEA is intended for compute-intensive workloads with many fine-grained and course-grained parallel operations. By porting the original Fortran code, we were unable to fully exploit every level of heterogeneous parallelism in the CBEA. Furthermore, 2D transport in the chosen domain size does not challenge the capabilities of the test systems. We believe this explains the low scalability of the 2D transport module, compared to the homogeneous systems. The analysis of the 3D transport module in Section 7.4 corroborates this theory.

The low runtimes on the shared memory machine are attributable to hardware support for double-precision arithmetic and more mature compiler technology. Kurzak and Dongarra found that the LINPACK benchmark is fourteen times slower on the Cell BE when double-precision arithmetic is used compared to single-precision arithmetic [25]. Their results and these experimental results suggest that a CBEA implementation with hardware support for double-precision arithmetic would out-perform a homogeneous processor of similar clock rate.

CellBuzz and Monza do not support double-precision arithmetic in hardware, yet they occasionally out-perform JUICEnext, which advertises full hardware pipelining for double-precision arithmetic. By examining the memory I/O performance of JUICEnext, DMA writes to main memory were found to be significantly slower on JUICEnext than on all other platforms. Technical support at Forschungszentrum Jülich is seeking a solution. Without access to another IBM BladeCenter QS22, it is difficult to determine if the PowerXCell 8i chipset can achieve its advertised potential.

[3] At the time of this writing, compiler technologies are significantly less mature for heterogeneous multicore chipsets. A more fair comparison of platforms should use -O0 for all platforms, however we wish to present the state-of-the-art performance of these platforms, as it is what the user will expect from his application.

Monza (PlayStation 3 Cluster)						
	Naïve Fcn. Offload		Opt. Fcn. Offload		Scalable DMA	
# SPEs	Wall clock	Speedup	Wall clock	Speedup	Wall clock	Speedup
1	**2324.197**	13.74	677.282	47.16	394.100	81.04
2	1165.798	27.40	369.438	86.45	201.853	158.23
3	797.546	40.05	291.917	109.41	138.606	230.43
4	627.608	50.89	269.018	118.72	107.201	297.93
5	553.026	57.75	284.881	112.11	90.589	352.57
6	654.322	48.81	275.304	116.01	79.043	404.07

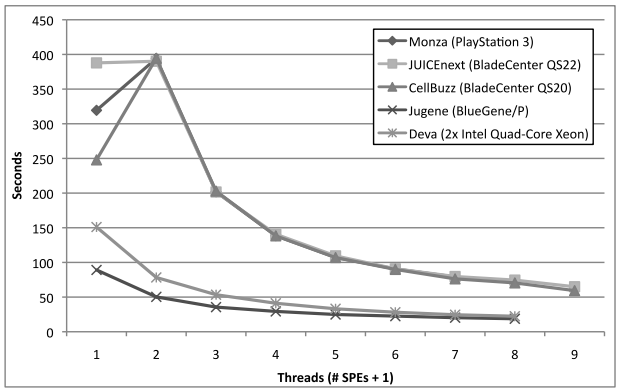
CellBuzz (IBM BladeCenter QS20)						
	Naïve Fcn. Offload		Opt. Fcn. Offload		Scalable DMA	
# SPEs	Wall clock	Speedup	Wall clock	Speedup	Wall clock	Speedup
1	2066.130	**11.61**	545.152	45.48	394.487	62.84
2	1052.147	23.56	295.112	84.01	202.896	122.19
3	715.939	34.63	190.461	130.16	138.319	179.23
4	570.133	43.48	156.472	158.44	107.043	231.60
5	435.496	56.93	139.371	177.88	89.903	275.76
6	381.251	65.03	128.880	192.36	76.296	324.94
7	334.724	74.06	122.472	202.42	70.469	351.81
8	307.779	80.55	115.022	215.53	**59.325**	417.89

JUICEnext (IBM BladeCenter QS22)						
	Naïve Fcn. Offload		Opt. Fcn. Offload		Scalable DMA	
# SPEs	Wall clock	Speedup	Wall clock	Speedup	Wall clock	Speedup
1	1617.694	23.97	640.491	60.55	390.172	99.39
2	858.288	45.18	446.678	86.82	201.607	199.29
3	654.199	59.28	406.151	95.48	140.669	296.89
4	648.914	59.76	357.931	108.35	109.456	391.73
5	642.692	60.34	352.095	110.14	90.817	486.46
6	642.466	62.10	351.066	110.46	79.706	587.38
7	644.576	60.16	359.384	107.91	74.491	634.91
8	639.315	60.66	353.773	109.62	64.920	**744.14**

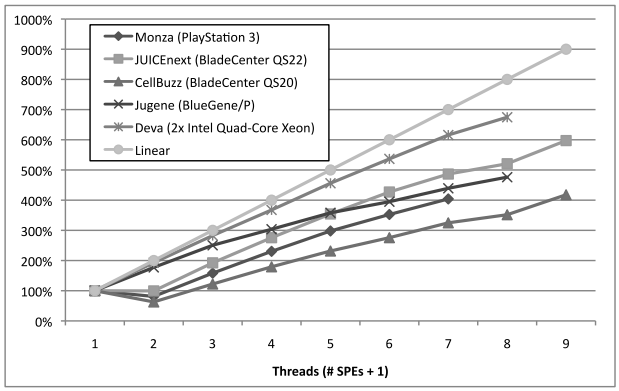
Table 1 Wall clock runtime in seconds and percentage speedup for the 2D transport module on three CBEA systems. The domain is double-precision ozone (O3) concentrations on a 60km^2 area mapped to a grid of 600×600 points and integrated for 12 hours with a $\Delta t = 50$ seconds timestep. Global extremes are emphasized

	Jugene (IBM BlueGene/P)		Deva (2x Quad-Core Xeon)	
# Threads	Wall clock	Speedup	Wall clock	Speedup
1	89.134	100.00	**151.050**	100.00
2	50.268	**177.32**	78.320	192.86
3	35.565	250.62	53.427	282.72
4	29.348	303.72	41.094	367.57
5	24.897	358.01	33.119	456.09
6	22.574	394.84	28.151	536.58
7	20.287	439.37	24.527	615.84
8	**18.717**	476.22	22.386	**674.75**

Table 2 Wall clock runtime in seconds and percentage speedup for the 2D transport module on two homogeneous multicore systems. The domain is double-precision ozone (O3) concentrations on a 60km^2 area mapped to a grid of 600×600 points and integrated for 12 hours with a $\Delta t = 50$ seconds timestep. Global extremes are emphasized



(a) Wall clock runtime



(b) Speedup

Fig. 6 Wall clock runtime in seconds and speedup for the 2D transport module on three CBEA systems and two homogeneous multicore systems

7 Three-dimensional Transport

This section discusses the implementation, analysis, and optimization of FIXED-GRID's 3D transport module on the CBEA using a *vector stream processing* approach. Transport is calculated according to Equations 2 and 3 along the x-, y-, and z-axes using the stencil shown in Figure 2. The 3D module was written from scratch to achieve maximum performance. Performance is given in Section 7.4.

7.1 Vector Stream Processing

Stream processing is a programming paradigm for exploiting parallelism. Given a stream of data, a set of computational kernels (functions or operations) are applied to each element in the stream. Often the kernels are pipelined.

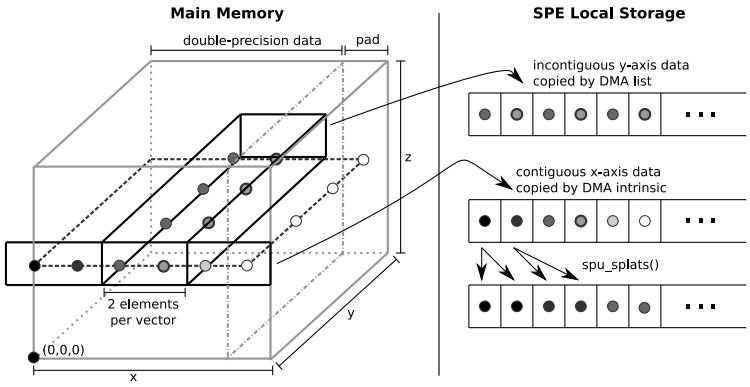


Fig. 7 FIXEDGRID data storage scheme for the 3D transport module showing padding and vectorization with double-precision data types

This method is particularly useful when programming several separate computational units without explicitly managing allocation, synchronization, and communication. Stream processing is appropriate for programs using dimension splitting to reduce multidimensional operations to sets of 1D operations. The 1D operations can be implemented as kernel functions and the domain data streamed in parallel through the kernels.

Vector stream processing extends stream processing by making every element in the data stream a vector and using vectorized kernel functions. This approach uses every level of heterogeneous parallelism in the CBEA by overlapping memory I/O with SIMD stream kernels on multiple cores. In order to achieve this, data reorganization in the stream must be minimal. Interleaving caused by transferring incontiguous data via DMA lists (Section 6.3) implicitly arranges stream data into vectors, so DMA lists are an integral part of this approach. A SIMD kernel can be applied directly to this unorganized data stream. Contiguous data must either be vectorized or processed with a scalar kernel.

FIXEDGRID uses dimension splitting to calculate 3D transport separately along the x-, y-, and z-axis with the same kernel function. Similar to the 2D module in Section 6.3, DMA lists are used to transfer incontiguous y- and z-axis data to SPE local storage in the 3D module. FIXEDGRID uses second order time splitting to reduce truncation error in 3D transport (see Figure 3). This doubles the work required to calculate mass flux along the x- and y-axis, so discretization in these directions should be highly efficient.

7.2 Streaming Vectorized Data with the CBEA

Streams of vector data are formed in the 3D module by combining DMA intrinsics with a triple-buffering scheme. Triple-buffering permits a single SPE to simultaneously fetch data from main memory, apply the kernel function,

and write processed data back to main memory. A separate DMA tag is kept for each buffer, and MFC instruction barriers keyed to these tags prevent overwriting an active buffer while allowing simultaneous read/write from different buffers by the same MFC. A stream of data from main memory, through the SPU, and back to main memory is maintained. Redirecting the outward-flowing data stream to the inward-flowing stream of another SPE allows kernel function pipelining. The concentration matrix and associated data structures in main memory are divided into blocks, which are streamed simultaneously through the SPEs according to block starting address.

Scalar elements transfered via DMA lists are interleaved into SPE local storage as vectors, so data organization is not required for y- or z-axis data (see Figure 7). X-axis data is contiguous, so the data must either be vectorized manually or processed with a scalar kernel. There are two ways to vectorize scalar data: manually interleave multiple scalar data streams, or copy the scalar value to every vector element. Manually interleaving multiple streams requires copying four/two buffers of single-/double-precision data to SPE local storage and interlacing the data to form vectors before processing. This approach requires many scalar operations and an additional DMA transfer. The SPU is optimized for vector operations, so it is approximately 8% faster to copy the scalar value to every vector element via the `spu_splats` intrinsic, process with the vectorized kernel, and return to scalar data by discarding all vector elements except the first. By carefully manipulating the DMA tags associated with local storage buffers, the `spu_splats` operations for one buffer (i.e. chemical concentrations) can overlap with the next buffer fetch (i.e. wind field data). This improves performance by a further 4%.

Synchronization in the data stream is required at the start and end of a mass flux computation. Mailbox registers and busy-wait loops are two common methods of synchronization, but they may incur long SPE idle times and waste PPE cycles. MFC intrinsics can synchronize streaming SPEs and avoid these problems. The PPE signals an SPE by placing a DMA call on the proxy command queue of an SPE context, causing the MFC to fetch the data to local storage. The SPE waits on the PPE by setting an MFC instruction barrier and stalling the SPU until the barrier is removed by a proxy command from the PPE. This allows the MFC to continue data transfers asynchronously while the SPU waits and results in virtually zero SPE idle time.

The streaming operations were encapsulated in a small library of static inline functions and user-defined data types. Local storage buffers are described by a data type with fields for local storage address, main memory address, and buffer size. Functions for fetching, processing, and flushing buffers maximize the source code readability and ease debugging. The unabridged source code for x-axis mass flux calculation is shown in Listing 2.

```
1  /* Start buffer 0 transfer */
2  fetch_x_buffer(0, 0);

4  /* Start buffer 1 transfer */
5  fetch_x_buffer(1, NX_ALIGNED_SIZE);

7  /* Process buffer 0 */
8  transport_buffer(0, size, dt);

10 /* Loop over rows in this block */
11 for(i=0; i<block-2; i++)
12 {
13   w = i % 3;
14   p = (i+1) % 3;
15   f = (i+2) % 3;

17   /* Write buffer w back to main memory (nonblocking) */
18   write_x_buffer(w, i*NX_ALIGNED_SIZE);

20   /* Start buffer f transfer (nonblocking) */
21   fetch_x_buffer(f, (i+2)*NX_ALIGNED_SIZE);

23   /* Process buffer p */
24   transport_buffer(p, size, dt);
25 }

27 /* Discretize final row */
28 w = i % 3;
29 p = (i+1) % 3;

31 /* Write buffer w back to main memory (nonblocking) */
32 write_x_buffer(w, i*NX_ALIGNED_SIZE);

34 /* Process buffer p */
35 transport_buffer(p, size, dt);

37 /* Write buffer p back to main memory (nonblocking) */
38 write_x_buffer(p, (i+1)*NX_ALIGNED_SIZE);

40 /* Make sure DMA is complete before we exit */
41 mfc_write_tag_mask( (1<<w) | (1<<p) );
42 spu_mfcstat(MFC_TAG_UPDATE_ALL);
```

Listing 2 X-axis mass flux calculation for blocks with more than one row where a whole row fits in local storage

7.3 The Vectorized Transport Kernel Function

The 3D advection/diffusion kernel function was vectorized to use the SPE's SIMD ISA. Because the advection discretization is upwind-biased, different vector elements may apply different parts of Equation 2, based on the wind vector's sign. Disassembling the wind vector to test each element's sign introduces expensive branching conditionals and scalar operations. It is more efficient to calculate both parts of Equation 2 preemptively, resulting in two vectors of possible values, and then bitwise mask the correct values from these vectors into the solution vector. The `spu_cmpgt`, `spu_and`, and `spu_add` intrinsics, each mapping to a single assembly instruction, are the only commands required to form the solution vector. `spu_cmpgt` forms a bit-mask identifying

elements with a certain sign, and `spu_and` and `spu_add` assemble the solution vector. Although half the calculated values are discarded, all branching conditionals in the kernel function are replaced with only a few assembly instructions and scalar operations are eliminated entirely. Performance of the vectorized kernel is approximately an order of magnitude above a scalar kernel with branches. The original C kernel is shown in Listing 3 and the vectorized kernel is shown in Listing 4.

```
1  inline real_t
2  advec_diff(real_t cell_size,
3    real_t c2l, real_t w2l, real_t d2l,
4    real_t c1l, real_t w1l, real_t d1l,
5    real_t   c, real_t   w, real_t   d,
6    real_t c1r, real_t w1r, real_t d1r,
7    real_t c2r, real_t w2r, real_t d2r)
8  {
9    real_t wind, diff_term;
10   real_t advec_term, advec_termL, advec_termR;

12   wind = (w1l + w) / 2.0;
13   if(wind >= 0.0) advec_termL = (1.0/6.0) * ( -c2l + 5.0*c1l +
         2.0*c );
14   else advec_termL = (1.0/6.0) * ( 2.0*c1l + 5.0*c - c1r );
15   advec_termL *= wind;
16   wind = (w1r + w) / 2.0;
17   if(wind >= 0.0) advec_termR = (1.0/6.0) * ( -c1l + 5.0*c + 2.0*
         c1r );
18   else advec_termR = (1.0/6.0) * ( 2.0*c + 5.0*c1r - c2r );
19   advec_termR *= wind;
20   advec_term = (advec_termL - advec_termR) / cell_size;
21   diff_term = ( ((d1l+d)/2)*(c1l-c) - ((d+d1r)/2)*(c-c1r) ) / (
         cell_size * cell_size);
22   return advec_term + diff_term;
23 }
```

Listing 3 Stream processing kernel

7.4 Analysis and Comparison

This section discusses the performance of FIXEDGRID's 3D transport module on two CBEA systems, a distributed memory multicore system, and a shared memory multicore system. The experiments calculate ozone (O_3) concentrations on a 60km^2 $\times$ 1.2km area mapped to a grid of $600 \times 600 \times 12$ points and integrated for 12 hours with a $\Delta t = 50$ seconds timestep. With the exception of Monza, the 3D experiments were performed on the same systems and with the same compiler options as in Section 6.4. Monza results are not shown because the PlayStation 3 has insufficient RAM to calculate a domain of this size without paging. A single experiment took approximately 35 hours to complete on Monza, making the results incomparable to the other CBEA systems. Again, the presented results are the average of three runs on each platform.

```
1  inline vector real_t
2  advec_diff_v(vector real_t cell_size,
3    vector real_t c2l, vector real_t w2l, vector real_t d2l,
4    vector real_t c1l, vector real_t w1l, vector real_t d1l,
5    vector real_t   c, vector real_t   w, vector real_t   d,
6    vector real_t c1r, vector real_t w1r, vector real_t d1r,
7    vector real_t c2r, vector real_t w2r, vector real_t d2r)
8  {
9    vector real_t acc1, acc2, acc3;
10   vector real_t wind, diff_term, advec_term;
11   vector real_t advec_term_pos, advec_term_neg;
12   vector real_t advec_termR, advec_termL;

14   acc1 = spu_add(w1l, w);
15   wind = spu_mul(acc1, HALF);
16   acc1 = spu_mul(c1l, FIVE);
17   acc2 = spu_mul(c, TWO);
18   advec_term_pos = spu_add(acc1, acc2);
19   advec_term_pos = spu_sub(advec_term_pos, c2l);
20   acc1 = spu_mul(c1l, TWO);
21   acc2 = spu_mul(c, FIVE);
22   advec_term_neg = spu_add(acc1, acc2);
23   advec_term_neg = spu_sub(advec_term_neg, c1r);
24   acc1 = (vector real_t)spu_cmpgt(wind, ZERO);
25   acc1 = spu_and(acc1, advec_term_pos);
26   acc2 = (vector real_t)spu_cmpgt(ZERO, wind);
27   acc2 = spu_and(acc2, advec_term_neg);
28   advec_termL = spu_add(acc1, acc2);
29   advec_termL = spu_mul(advec_termL, SIXTH);
30   advec_termL = spu_mul(advec_termL, wind);
31   acc1 = spu_add(w1r, w);
32   wind = spu_mul(acc1, HALF);
33   acc1 = spu_mul(c, FIVE);
34   acc2 = spu_mul(c1r, TWO);
35   advec_term_pos = spu_add(acc1, acc2);
36   advec_term_pos = spu_sub(advec_term_pos, c1l);
37   acc1 = spu_mul(c, TWO);
38   acc2 = spu_mul(c1r, FIVE);
39   advec_term_neg = spu_add(acc1, acc2);
40   advec_term_neg = spu_sub(advec_term_neg, c2r);
41   acc1 = (vector real_t)spu_cmpgt(wind, ZERO);
42   acc1 = spu_and(acc1, advec_term_pos);
43   acc2 = (vector real_t)spu_cmpgt(ZERO, wind);
44   acc2 = spu_and(acc2, advec_term_neg);
45   advec_termR = spu_add(acc1, acc2);
46   advec_termR = spu_mul(advec_termR, SIXTH);
47   advec_termR = spu_mul(advec_termR, wind);
48   acc1 = spu_sub(advec_termL, advec_termR);
49   advec_term = VEC_DIVIDE(acc1, cell_size);
50   acc1 = spu_add(d1l, d);
51   acc1 = spu_mul(acc1, HALF);
52   acc3 = spu_sub(c1l, c);
53   acc1 = spu_mul(acc1, acc3);
54   acc2 = spu_add(d, d1r);
55   acc2 = spu_mul(acc2, HALF);
56   acc3 = spu_sub(c, c1r);
57   acc2 = spu_mul(acc2, acc3);
58   acc1 = spu_sub(acc1, acc2);
59   acc2 = spu_mul(cell_size, cell_size);
60   diff_term = VEC_DIVIDE(acc1, acc2);
61   return spu_add(advec_term, diff_term);
62 }
```

Listing 4 Vectorized stream processing kernel

The measurement error was insignificant, so error bars are not shown. Tables 3 and 4 list the wall clock runtime performance on the CBEA systems and homogeneous multicore systems, respectively.

Figure 8 shows the wall clock runtime performance of FIXEDGRID on two CBEA systems and two homogeneous multicore systems. JUICEnext, with hardware support for fully-pipelined double precision arithmetic, achieves the best performance. Both CBEA systems achieve performance comparable to two nodes of the IBM BlueGene/P and eight Intel Xeon cores in a single chip.

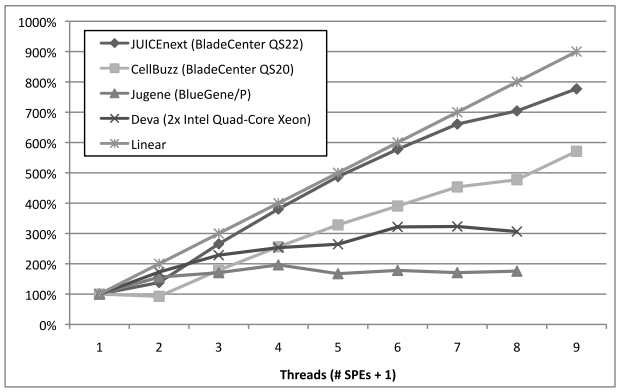
There is a performance asymptote beginning around the 5 SPE mark. We believe this is due to the element interconnect bus (EIB) end-to-end control mechanism. Ainsworth and Pinkston characterized the performance of the EIB in [1] and highlighted the end-to-end control mechanism as the main CBEA bottleneck. The EIB is a pipelined circuit-switched ring-topology network. The EIB's data arbiter does not allow a packet to be transferred along more than six hops (half the diameter of the ring). If a request is greater than six hops, the requester must wait until a data ring operating the opposite direction becomes free. This limits concurrently communicating elements to six. When

	JUICEnext (IBM QS22)		CellBuzz (IBM QS20)	
# SPEs	Wall clock	Speedup	Wall clock	Speedup
0	**8756.225**	100.00	6423.170	100.00
1	6338.917	138.13	6930.159	**92.68**
2	3297.401	265.55	3595.270	178.66
3	2301.286	380.49	2508.892	256.02
4	1796.828	487.32	1957.122	328.19
5	1515.433	577.80	1644.511	390.58
6	1325.122	660.79	1416.271	453.53
7	1243.254	704.30	1346.096	477.17
8	1126.720	**777.14**	**1124.675**	571.11

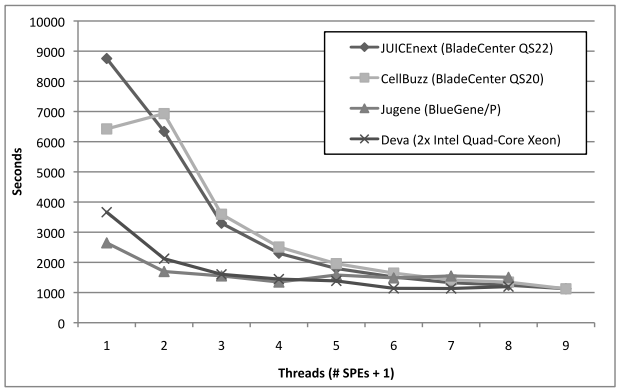
Table 3 Wall clock runtime in seconds and percentage speedup for the 3D transport on two CBEA systems. The domain is double-precision ozone (O3) concentrations on a 60km^2 $\times$ 1.2km area mapped to a grid of $600 \times 600 \times 12$ points and integrated for 12 hours with a $\Delta t = 50$ seconds timestep. Global extremes are emphasized

	Jugene (IBM BlueGene/P)		Deva (2x Quad-Core Xeon)	
# Threads	Wall clock	Speedup	Wall clock	Speedup
1	2647.364	100.00	**3664.473**	100.00
2	1696.784	**156.02**	2121.569	172.72
3	1551.104	170.68	1603.652	228.51
4	1347.977	196.40	1446.672	253.30
5	1581.657	167.38	1385.619	264.46
6	1485.877	178.17	1139.558	321.57
7	1550.012	170.80	**1134.279**	**323.07**
8	1506.631	175.71	1196.477	306.27

Table 4 Wall clock runtime in seconds and percentage speedup for the 3D transport module on two homogeneous multicore systems. The domain is double-precision ozone (O3) concentrations on a 60km^2 $\times$ 1.2km area mapped to a grid of $600 \times 600 \times 12$ points and integrated for 12 hours with a $\Delta t = 50$ seconds timestep. Global extremes are emphasized



(a) Wall clock runtime



(b) Speedup

Fig. 8 Wall clock runtime in seconds and speedup for the 3D transport module on three CBEA systems and two homogeneous multicore systems

the MIC and five SPEs communicate on the EIB – as occurs when five SPEs stream data simultaneously – the arbiter stalls communication from all other processing elements. The EIB's 204.8 GB/second peak bandwidth is enough to support streaming on all SPEs, so the control mechanism bottleneck is significant.

Fundamental hardware differences make a fair comparison between homogeneous and heterogeneous chipsets difficult. If equal effort were spent optimizing FIXEDGRID for both BlueGene/P and CBEA, it is likely the BlueGene/P would achieve better performance. However, if compiler technologies were as mature for the CBEA as they are for the BlueGene/P, the CBEA's performance would likely overtake the BlueGene/P's performance again. A comparison of thousands of BlueGene/P nodes with thousands of CBEA nodes would be a stronger test of scalability. The experimental results strongly sug-

gest that, on a per-node basis, heterogeneous multicore chipsets out-perform homogeneous multicore chipsets.

8 Conclusions and Future Work

The two- and three-dimensional chemical constituent transport modules in FIXEDGRID, a prototypical atmospheric model, were optimized for the Cell Broadband Engine Architecture (CBEA). Geophysical models are comprehensive multiphysics simulations and frequently contain a high degree of parallelism. By using the multiple layers of heterogeneous parallelism available in the CBEA, the 3D transport module achieved performance comparable to two nodes of the IBM BlueGene/P system at Forschungszentrum Jülich, ranked 6th on the current Top 500 supercomputing list, with only one CBEA chip.

Function offloading was examined as an approach for porting a 2D transport module written in Fortran 90 to the CBEA. The results show that function offloading is applicable to many scientific codes and can produce good results. Function offloading does not always leverage the streaming capabilities of the CBEA. Codes with a high degree of instruction-level parallelism (ILP) will see a good speedup with this approach, due to the SPE's SIMD ISA. Codes with data-level parallelism (DLP) are less likely to achieve maximum performance, since there are a relatively small number of SPEs. When both ILP and DLP are present, maximum performance will only be achieved when every level of parallelism in the CBEA is used, as in vector stream processing.

By using DMA lists to transfer incontiguous matrix column data, 2D transport scalability was improved by over 700%. This approach is applicable to any code transferring incontiguous data between main memory and the SPE local storage. DMA list overhead limits application performance when random access to both rows and columns of a matrix are required. Matrix representations providing efficient random access to incontiguous data will be explored. "Strided" DMA commands for the MFC would solve these issues entirely, but at a cost of increased hardware complexity.

Vector stream processing seeks to use all available parallelism in the CBEA by overlapping memory I/O with SIMD stream kernels on multiple cores. This method was implemented in a few light-weight static inline functions accompanied by user-defined types. Together, these constructs combine DMA intrinsics, DMA lists, and a triple-buffering scheme to stream data through the SPE local storage at the maximum rate permitted by the EIB data arbiter. By using the MFC proxy command queue and MFC instruction barriers instead of mailbox registers or busy-wait synchronization, vector stream processing achieves almost zero SPE idle time and maximizes SPE throughput. Careful consideration of data organization and data contiguity in multi-dimensional arrays reduces data reordering in the stream to only one dimension (i.e. the contiguous dimension must be vectorized). All levels of heterogeneous parallelism in the CBEA are used: the SPE's SIMD ISA, the MFC's nonblocking DMA intrinsics, and the many cores of the CBEA. Fully-optimized, FIXED-

GRID calculates ozone (O_3) transport on a domain of $600 \times 600 \times 12$ grid cells for twelve hours with a 50 second time step in approximately 18.34 minutes.

Future work will address the limitations of the Element Interconnect Bus. By pairing SPEs and using one for data transfer and one for computation, it may be possible to achieve better scalability for more than five SPEs by spreading DMA requests over a wider area on the EIB ring. Rafique et al. explored similar methods for I/O-intensive applications with good results [29]. Nodes in an IBM BladeCenter installation can use off-chip SPEs for a maximum of 16 SPEs. Off-chip SPEs have their own EIB, so it may be advantageous to use them for file I/O or checkpointing. Spreading memory requests over multiple EIBs also be explored.

The stream kernel discards 50% of the calculated values in favor of higher throughput. A method for reducing useless arithmetic operations will be explored. The PPE can quickly determine the sign of a wind vector, so perhaps the SPEs could be grouped into positive- and negative-sign groups. The PPE could then assign blocks to an SPE according to wind vector sign rather than starting address.

Programming language technologies for heterogeneous multicore chipsets are still relatively immature. The bulk of the performance improvement came from vectorized kernel functions, so better support for compiler auto-vectorization should be explored. While some auto-vectorization support is available, the performance of the auto-generated kernel was far below that of the hand-tuned kernel. A comparison of Listing 3 with Listing 4 should convince any programmer that more intelligent compilers are needed. Indeed, the majority of low-level optimizations we did by hand could have been done automatically by an advanced compiler.

Immediate future work concerns accelerating FIXEDGRID's chemical kinetics module for the CBEA. Large-scale production models, such as STEM [8], WRF/Chem [20], and CMAQ [27], spend as much as 90% of their computation time in the chemical kinetics module. All of these models use chemical kinetics generated by the Kinetics Pre-Processor (KPP) [14]. We are currently examining approaches for accelerating KPP-generated mechanisms for a range of multicore technologies, including the CBEA. KPP for accelerated chipsets will be immediately beneficial to three production geophysical models.

Acknowledgements The authors wish to thank Forschungszentrum Jülich and the Jülich Supercomputing Centre for the IBM BlueGene/P and IBM BlaceCenter resouces that have contributed to this research. The authors also acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research. The authors particularly wish to thank Dr. Willi Homberg of Forschungszentrum Jülich and Prof. Dr. Felix Wolf of RWTH Aachen for their help and support.

References

1. Ainsworth TW, Pinkston TM (2007) On characterizing performance of the Cell Broadband Engine Element Interconnect Bus. In: Proceedings of the First International Symposium on Networks-on-Chip (NOCS '07), Princeton, NJ, pp 18–29, DOI 10.1109/NOCS.2007.34
2. Alam SR, Agarwal PK (2007) On the path to enable multi-scale biomolecular simulations on petaFLOPS supercomputer with multi-core processors. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '07), Long Beach, CA, pp 1–8, DOI 10.1109/IPDPS.2007.370443
3. Bader DA, Patel S (2008) High performance MPEG-2 software decoder on the Cell Broadband E. In: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS '08), Miami, FL, pp 1–10, DOI 10.1109/IPDPS.2008.4536234
4. Baik H, Sihn KH, il Kim Y, Bae S, Han N, Song HJ (2007) Analysis and parallelization of H.264 decoder on Cell Broadband Engine Architecture. In: Proceedings of the IEEE International Symposium on Signal Processing and Information Technology, Giza, pp 791–795, DOI 10.1109/ISSPIT.2007.4458128
5. Benthin C, Wald I, Scherbaum M, Friedrich H (2006) Ray tracing on the cell processor. In: Proceedings of the IEEE Symposium on Interactive Ray Tracing, Salt Lake City, UT, pp 15–23, DOI 10.1109/RT.2006.280210
6. Blagojevic F, Nikolopoulos DS, Stamatakis A, Antonopoulos CD (2007) Dynamic multigrain parallelization on the Cell Broadband Engine. In: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07), ACM, New York, NY, USA, pp 90–100, DOI 10.1145/1229428.1229445
7. Buttari A, Dongarra J, Langou J, Langou J, Luszczek P, Kurzak J (2007) Mixed precision iterative refinement techniques for the solution of dense linear systems. J High Perform Comput Appl 21(4):457–466, DOI 10.1177/1094342007084026
8. Carmichael GR, Peters LK, Kitada T (1986) A second generation model for regional scale transport / chemistry / deposition. Atmos Env 20:173–188
9. Carter WPL (1990) A detailed mechanism for the gas-phase atmospheric reactions of organic compounds. Atmos Env 24A:481–518
10. Chen L, Hu Z, Lin J, Gao GR (2007) Optimizing the fast fourier transform on a multi-core architecture. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '07), Long Beach, CA, pp 1–8, DOI 10.1109/IPDPS.2007.370639
11. Chen T, Raghavan R, Dale J, Iwata E (2006) Cell Broadband Engine Architecture and its first implementation. IBM developerWorks
12. Corporation IBM (2006) PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual. URL http://www-306.ibm.com/chips/techlib, 2nd edn

13. Dally WJ, Labonte F, Das A, Hanrahan P, Ahn JH, Gummaraju J, Erez M, Jayasena N, Buck I, Knight TJ, Kapasi UJ (2003) Merrimac: Supercomputing with streams. In: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC '03), IEEE Computer Society, Washington, DC, p 35
14. Damian V, Sandu A, Damian M, Potra F, Carmichael GR (2002) The Kinetic Preprocessor KPP – a software environment for solving chemical kinetics. Comput Chem Eng 26:1567–1579
15. Douillet A, Gao GR (2007) Software-pipelining on multi-core architectures. In: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07), Brasov, Romania, pp 39–48, DOI 10.1109/PACT.2007.4336198
16. Erez M, Ahn JH, Gummaraju J, Rosenblum M, Dally WJ (2007) Executing irregular scientific applications on stream architectures. In: Proceedings of the 21st Annual International Conference on Supercomputing (ICS '07), ACM, New York, NY, USA, pp 93–104, DOI 10.1145/1274971.1274987
17. Fatahalian K, Knight TJ, Houston M, Erez M, Horn DR, Leem L, Park JY, Ren M, Aiken A, Dally WJ, Hanrahan P (2006) Sequoia: Programming the memory hierarchy. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)
18. Flachs B, Asano S, Dhong SH, Hofstee HP, Gervais G, Kim R, Le T, et al (2006) The microarchitecture of the synergistic processor for a cell processor. IEEE J Solid State Circuits 41(1):63–70
19. Gedik B, Andrade H, Wu KL, Yu PS, Doo M (2008) SPADE: The System S declarative stream processing engine. In: Proceedings of the 2008 International Conference on Management of Data (SIGMOD '08), ACM, New York, NY, USA, pp 1123–1134, DOI 10.1145/1376616.1376729
20. Grell GA, Peckham SE, Schmitz R, McKeen SA, Frost G, Skamarock WC, Eder B (2005) Fully coupled online chemistry within the WRF model. Atmos Env 39:6957–6975
21. Hieu NT, Keong KC, Wirawan A, Schmidt B (2008) Applications of heterogeneous structure of Cell Broadband Engine Architecture for biological database similarity search. In: Proceedings of the The 2nd International Conference on Bioinformatics and Biomedical Engineering (ICBBE '08), Shanghai, pp 5–8, DOI 10.1109/ICBBE.2008.8
22. Hirsch C (1988) Numerical Computation of Internal and External Flows 1: fundamentals and numerical discretization. John Wiley and Sons, Chichester
23. Hundsdorfer W (1996) Numerical solution of advection-diffusion-reaction equations. Tech. rep., Centrum voor Wiskunde en Informatica
24. Ibrahim KZ, Bodin F (2008) Implementing wilson-dirac operator on the Cell Broadband Engine. In: Proceedings of the 22nd Annual International Conference on Supercomputing (ICS '08), ACM, New York, NY, USA, pp 4–14, DOI 10.1145/1375527.1375532

25. Kurzak J, Dongarra J (2007) Implementation of mixed precision in solving systems of linear equations on the cell processor. Concurr Comput: Pract Exp 19(10):1371–1385, URL http://cscads.rice.edu/Presentations/fulltext-kurzak.pdf
26. Li B, Jin H, Shao Z, Li Y, Liu X (2008) Optimized implementation of ray tracing on Cell Broadband Engine. In: Proceedings of the International Conference on Multimedia and Ubiquitous Engineering (MUE '08), Busan, pp 438–443, DOI 10.1109/MUE.2008.83
27. Meng BZ, Gbor P, Wen D, Yang F, Shi C, Aronson J, Sloan J (2007) Models for gas/particle partitioning, transformation and air/water surface exchange of PCBs and PCDD/Fs in CMAQ. Atmos Env 41(39):9111–9127
28. Muta H, Doi M, Nakano H, Mori Y (2007) Multilevel parallelization on the Cell/B.E. for a motion JPEG 2000 encoding server. In: Proceedings of the 15th International Conference on Multimedia (MULTIMEDIA '07), ACM, New York, NY, USA, pp 942–951, DOI 10.1145/1291233.1291442
29. Rafique MM, Butt AR, Nikolopoulos DS (2008) DMA-based prefetching for I/O-intensive workloads on the cell architecture. In: Proceedings of the 2008 Conference on Computing Frontiers (CF '08), ACM, New York, NY, USA, pp 23–32, DOI 10.1145/1366230.1366236
30. Ray J, Kennedy C, Lefantzi S, Najm H (2003) High-order spatial discretizations and extended stability methods for reacting flows on structured adaptively refined meshes. In: Proceedings of Third Joint Meeting of the U.S. Sections of the Combustion Institute, Chicago, USA
31. Sandu A, Daescu D, Carmichael G, Chai T (2005) Adjoint sensitivity analysis of regional air quality models. J Comp Phys 204:222–252
32. Strang G (1968) On the construction and comparison of difference schemes. SIAM J Numer Anal 5(3):506–517, URL http://www.jstor.org/stable/2949700
33. Williams S, Shalf J, Oliker L, Kamil S, Husbands P, Yelick K (2006) The potential of the cell processor for scientific computing. In: Proceedings of the 3rd Conference on Computing Frontiers (CF '06), ACM, New York, NY, USA, pp 9–20, DOI 10.1145/1128022.1128027
34. Zhu Z, Wang Q, Feng B, Shao L (2007) Speech codec optimization based on Cell Broadband Engine. In: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '07), Honolulu, HI, vol 2, pp 805–808, DOI 10.1109/ICASSP.2007.366358