Multi-core Acceleration of Chemical Kinetics for Simulation and Prediction

John C. Linford
Virginia Polytechnic Institute and State University
Blacksburg, VA
jlinford@vt.edu

John Michalakes
National Center for Atmospheric Research
Boulder, CO
michalak@ucar.edu

Manish Vachharajani
University of Colorado at Boulder
Boulder, CO
manishv@colorado.edu

Adrian Sandu
Virginia Polytechnic Institute and State University
Blacksburg, VA
sandu@cs.vt.edu

ABSTRACT

This work implements a computationally expensive chemical kinetics kernel from a large-scale community atmospheric model on three multi-core platforms: NVIDIA GPUs using CUDA, the Cell Broadband Engine, and Intel Quad-Core Xeon CPUs. A comparative performance analysis for each platform in double and single precision on coarse and fine grids is presented. Platform-specific design and optimization is discussed in a mechanism-agnostic way, permitting the optimization of many chemical mechanisms. The implementation of a three-stage Rosenbrock solver for SIMD architectures is discussed. When used as a template mechanism in the the Kinetic PreProcessor, the multi-core implementation enables the automatic optimization and porting of many chemical mechanisms on a variety of multi-core platforms. Speedups of 5.5× in single precision and 2.7× in double precision are observed when compared to eight Xeon cores. Compared to the serial implementation, the maximum observed speedup is 41.1× in single precision.

Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: Processor Architectures—*multiple data stream architectures (multi-processors)*; D.1.3 [**Programming Techniques**]: Concurrent Programming, Parallel Programming; J.2 [**Computer Applications**]: Physical Sciences and Engineering

General Terms

Design, Performance

Keywords

Multi-core, NVIDIA CUDA, Cell Broadband Engine, OpenMP, Chemical Kinetics, Atmospheric Modeling, Kinetic PreProcessor

1. INTRODUCTION

Chemical kinetics models trace the evolution of chemical species over time by solving large numbers of partial differential equations. The Weather Research and Forecast with Chemistry model (WRF-Chem) [20], the Community Multiscale Air Quality Model (CMAQ) [7], and the Sulfur Transport and dEposition Model (STEM) [8] approximate the chemical state of the Earth's atmosphere by applying a chemical kinetics model over a regular grid. Computational time is dominated by the solution of the coupled and stiff[1] equations arising from the chemical reactions, which may involve millions of variables [13].

These models are embarrassingly parallel on a fixed grid since changes in concentration of species y_i at any grid point depend only on concentrations and meteorology at the same grid point. Yet chemical kinetics models may be responsible for over 90 percent of an atmospheric model's computational time. For example, a RADM2 kinetics mechanism combined with the SORGAM aerosol scheme (RADM2SORG chemistry kinetics option in WRF-Chem) involves 61 species in a network of 156 reactions. On a 40×40 grid with 20 horizontal layers, the meteorological part of the simulation (the WRF weather model itself) is only 160×10^6 floating point operations per time step, about 2.5 percent the cost of the full WRF-Chem with both chemical kinetics and aerosols.

The new generations of multi-core processors mass produced for commercial IT and "graphical computing" (i.e. video games) achieve high rates of performance for highly-parallel applications, such as atmospheric models which contain abundant coarse- and fine-grained parallelism. Successful use of these novel architectures as accelerators on each node of large-scale conventional compute clusters will enable not only larger, more complex simulations, but also reduce

[1]For chemistry, "stiff" means that the system comprises widely varying reaction rates and cannot be solved using explicit numerical methods.

the time-to-solution for a range of earth system applications.

This paper presents significantly reduced time-to-solution for a chemical kinetics kernel from WRF-Chem and demonstrates a portable method for achieving similar performance for similar chemical mechanisms. We use Intel Quad-Core Xeon chipsets with OpenMP, NVIDIA GPUs with CUDA, and the Cell Broadband Engine Architecture (CBEA) to solve the RADM2 [31] mechanism for two domain resolutions in both double and single precision. RADM2 was chosen as a representative instance of a chemical mechanism for this work. Computational specifics of the RADM mechanism, benefits, and costs relative to other similar chemical mechanism codes are incidental at this stage of our research to uncover opportunities and challenges for efficient implementation of a "typical" chemical kinetics code. Our intent is to generalize to other similar chemical kinetics codes in future work. Detailed descriptions of porting and tuning RADM2 per platform are given in a mechanism-agnostic way so these approaches can be applied to any chemical mechanism. When our implementations are used as templates in the Kinetic PreProcessor [13] highly-optimized platform-specific code for any chemical mechanism can be generated. Benchmarks demonstrate the strong scalability of these new multi-core architectures over coarse- and fine-grain grids in both double and single precision. Speedups of 5.5× in single precision and 2.7× in double precision are observed when compared to eight Xeon cores. Compared to the state-of-the-art serial implementation, the speedup is 41.1×.

A brief formulation of a chemical kinetics model is given in Section 3. A typical homogeneous multi-core chipset, GPUs, NVIDIA CUDA, and the CBEA are outlined in Section 4. Multi-core ports of RADM2 and performance benchmarks are presented in Section 5. A prototype of the Kinetic PreProcessor extended for multi-core is given in Section 6. Conclusions and future work are given in Section 7.

2. RELATED WORK

Implementing chemical kinetics models on emerging multi-core technologies can be unusually difficult because expertise in kinetics and atmospheric modeling must be combined with a strong understanding of various multi-core paradigms. For this reason, existing literature tends to focus on the model sub-components, such as basic linear algebra operations [17, 33, 35], and does not comprehensively address this problem domain for chemical kinetics and related problems on real-world domains.

Research into using GPUs to accelerate the transport and diffusion of atmospheric constituents is ongoing. Fan et al. [15] used a 35-node cluster to simulate the dispersion of airborne contaminants in the Times Square area of New York City with the lattice Boltzmann model (LBM). For only $12,768, they were able to add a GPU to each node and boost the cluster's performance by 512 gigaFLOPS to achieve a 4.6x speedup in their simulation. Perumalla [30] used an NVIDIA GeForce 6800 Go GPU to explore time-stepped and discrete event simulation implementations of 2D diffusion. Large simulations saw a speedup of up to 16x on the GPU as compared to the CPU implementation. As previously mentioned, transport forms a relatively small fraction of the computational cost of comprehensive atmospheric simulation with chemistry.

Like many scientific codes, linear algebra operations are a core component of chemical kinetics simulations. The literature of the last four years abounds with examples of significantly improved linear algebra performance for both GPUs and the CBEA. Williams et al. [35, 36] achieved a maximum speedup of 12.7× and power efficiency of 28.3× with double precision general matrix multiplication, sparse matrix vector multiplication, stencil computation, and Fast Fourier Transform kernels on the CBEA as compared to AMD Opteron and Itanium2 processors. Bolz et al. [6] implemented a sparse matrix conjugate gradient solver and a regular-grid multigrid solver on NVIDIA GeForce FX hardware. The GPU performed 120 unstructured (1370 structured) matrix multiplies per second, while an SSE implementation achieved only 75 unstructured (750 structured) matrix multiplies per second on a 3.0GHz Pentium 4 CPU. Krüger and Westermann [26] investigated solvers for Navier-Stokes equations on GPUs. They represented matrices as a set of diagonal or column vectors, and vectors as 2D texture maps to achieve good basic linear algebra operator performance on NVIDIA GeForce FX and ATI Radeon 9800 GPUs.

Our work uses the heterogeneous parallelism of the CBEA in a way similar to that introduced by Ibrahim and Bodin in [23]. They introduced runtime data fusion for the CBEA which dynamically reorganizes finite element data to facilitate SIMD-ization while minimizing shuffle operations. Using this method, they achieved a sustained 31.2 gigaFLOPS for an implementation of the Wilson-Dirac Operator. Runtime data fusion is not applicable to the chemical kinetic models, but we use the PPU in a similar manner to reorganize data from the WRF-Chem model to facilitate SIMD-ization in a way specific to WRF-Chem.

3. ATMOSPHERIC CHEMICAL KINETICS

This section describes the formulation of a chemical kinetics kernel for use in large-scale atmospheric models. Many atmospheric models, including WRF-Chem and STEM, support a number of chemical kinetics solvers that are automatically generated at compile time by the Kinetic PreProcessor (KPP) [13]. KPP is a general analysis tool that facilitates the numerical solution of chemical reaction network problems. It automatically generates Fortran or C code that computes the time-evolution of chemical species, the Jacobian, and other quantities needed to interface with numerical integration schemes, and incorporates a library of several widely-used atmospheric chemistry mechanisms. KPP has been successfully used to treat many chemical mechanisms from tropospheric and stratospheric chemistry, including CBM-IV [19], SAPRC [10], and NASA HSRP/AESA. The Rosenbrock methods implemented in KPP typically outperform backward differentiation formulas, like those implemented in SMVGEAR [25, 14]

A chemical mechanism may be so stiff that it will not converge without full double precision floating point computation. One example is the SAPRC mechanism. RADM2 converges in both double and single precision, yet convergence is typically achieved in 6-8 fewer solver iterations when working in double precision. The overhead of double precision computation dictated by hardware design often outweighs any solver iteration savings, so it is preferable to work in single precision when possible. This work considers both double and single precision in order to be applicable to mechanisms requiring double precision.

3.1 Forming the Chemical System

Given a reaction network and initial concentrations as input files, KPP generates code to solve the differential equation of mass action kinetics to determine the concentrations at a future time. The derivation of this equation is given at length in [13] and summarized here.

Consider a system of n chemical species with R chemical reactions, $r = [r_1, \ldots, r_R]^T$. Let y be the vector of concentrations of all species involved in the chemical mechanism, $y = [y_1, \ldots, y_n]^T$. The concentration of species i is denoted by y_i. We define $k_j \in k = [k_1, \ldots, k_R]^T$ to be the rate coefficient of reaction r_j.

The stoichiometric coefficients $s_{i,j}$ are defined as follows. $s^-_{i,j}$ is the number of molecules of species y_i that react (are consumed) in reaction r_j. Similarly, $s^+_{i,j}$ is the number of molecules of species y_i that are produced in reaction r_j. If y_i is not involved in reaction r_j then $s^-_{i,j} = s^+_{i,j} = 0$.

The principle of mass action kinetics states that each chemical reaction progresses at a rate proportional to the concentration of the reactants. Thus, the jth reaction in the model is stated as

$$(r_j) \qquad \sum s^-_{i,j} y_i \xrightarrow{k_j} \sum s^+_{i,j} y_i, \qquad 1 \le j \le R, \tag{1}$$

where k_j is the proportionality constant. In general, the rate coefficients are time dependent: $k_j = k_j(t)$.

The reaction velocity (the number of molecules performing the chemical transformation during each time step) is given in molecules per time unit by

$$\omega_j(t, y) = k_j(t) \prod_{i=1}^{n} y_i^{s^-_{i,j}}. \tag{2}$$

y_i changes at a rate given by the cumulative effect of all chemical reactions:

$$\frac{d}{dt} y_i = \sum_{j=1}^{R} (s^+_{i,j} - s^-_{i,j}) \omega_j(t, y), \qquad i = 1, \ldots, n \tag{3}$$

If we organize the stoichiometric coefficients in two matrices,

$$S^- = (s^-_{i,j})_{1 \le i \le n, 1 \le j \le R}, \qquad S^+ = (s^+_{i,j})_{1 \le i \le n, 1 \le j \le R},$$

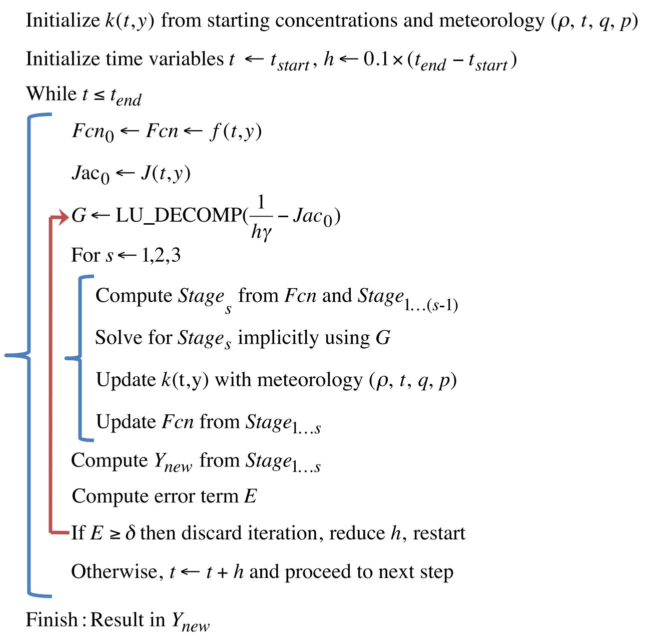
then Equation 3 can be rewritten as

$$\frac{d}{dt} y = (S^+ - S^-)\omega(t, y) = S\omega(t, y) = f(t, y), \tag{4}$$

where $S = S^+ - S^-$ and $\omega(t, y) = [\omega_1, \ldots, \omega_R]^T$ is the vector of all chemical reaction velocities.

Equation 4 gives the time derivative function in aggregate form. Depending on the integration method, other forms, such as a split production-destruction form may be preferred. KPP can produce both aggregate and production-destrucion forms. Implicit integration methods also require the evaluation of the Jacobian of the derivative function:

$$J(t, y) = \frac{\partial}{\partial y} f(t, y) = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} & \cdots & \frac{\partial f_1}{\partial y_n} \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} & \cdots & \frac{\partial f_2}{\partial y_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial y_1} & \frac{\partial f_n}{\partial y_2} & \cdots & \frac{\partial f_n}{\partial y_n} \end{bmatrix}$$



```
Initialize $k(t,y)$ from starting concentrations and meteorology ($\rho$, $t$, $q$, $p$)
Initialize time variables $t \leftarrow t_{start}$, $h \leftarrow 0.1 \times (t_{end} - t_{start})$
While $t \le t_{end}$
    $Fcn_0 \leftarrow Fcn \leftarrow f(t,y)$
    $Jac_0 \leftarrow J(t,y)$
    $G \leftarrow$ LU_DECOMP($\frac{1}{h\gamma} - Jac_0$)
    For $s \leftarrow 1,2,3$
        Compute $Stage_s$ from $Fcn$ and $Stage_{1\ldots(s-1)}$
        Solve for $Stage_s$ implicitly using $G$
        Update $k$(t,y) with meteorology ($\rho$, $t$, $q$, $p$)
        Update $Fcn$ from $Stage_{1\ldots s}$
    Compute $Y_{new}$ from $Stage_{1\ldots s}$
    Compute error term $E$
    If $E \ge \delta$ then discard iteration, reduce $h$, restart
    Otherwise, $t \leftarrow t + h$ and proceed to next step
Finish: Result in $Y_{new}$
```

Figure 1: A general outline of the three-stage Rosenbrock solver for chemical kinetics. t is the system time, h is the small time step, $Stage_s$ is the result of Rosenbrock stage s, δ is an error threshold. $k(t, y)$, $f(t, y)$, and $J(t, y)$ are as given in Section 3.1. Y_{new} is the new concentration vector.

3.2 Solving the Chemical System

The solution of the ordinary differential equations is advanced in time by a numerical integration method. KPP can generate a variety of implicit solvers to solve the stiff system of ODEs. We consider a Rosenbrock integrator with three Newton stages as generated by KPP [21]. Each Newton stage is solved implicitly. The implementation takes advantage of sparsity as well as trading exactness for efficiency when reasonable. An outline of the implementation is shown in Figure 1. If the system is autonomous, the reaction rates do not depend on the time variable t and may be computed only once. Otherwise they must be recomputed during the integration stages as shown.

In WRF-Chem, a KPP-generated Rosenbrock solver is applied to every point on a fixed domain grid. Chemical kinetics are embarrassingly parallel between cells, so there is abundant data parallelism (DLP). Within the solver itself, the ODE system is coupled so that, while there is still some data parallelism available in lower-level linear algebra operations, parallelization is limited largely to the instruction level (ILP). Thus, a three-tier parallelization is possible: ILP on each core, DLP using single-instruction-multiple-data (SIMD) features of a single core, and DLP across multiple cores (using multi-threading) or nodes (using MPI). The coarsest tier of MPI and OpenMP parallelism is already supplied by WRF-Chem.

3.3 The RADM2 Chemical Kernel

RADM2 was developed by Stockwell et. al. [31] for the Regional Acid Deposition Model version 2 [11]. It is widely used in atmospheric models to predict concentrations of oxidants and other air pollutants. The RADM2 kinetics mechanism combined with the SORGAM aerosol scheme involves 61 species in a network of 156 reactions. It treats inorganic species, stable species, reactive intermediates and abundant stable species (O_2, N, H_2O). Atmospheric organic chemistry is represented by 26 stable species and 16 peroxy radicals. Organic chemistry is represented through a reactivity aggregated molecular approach [27]. Similar organic compounds are grouped together into a limited number of model groups (HC_3, HC_5, and HC_8) through reactivity weighting. The aggregation factors for the most emitted VOCs are given in [27].

The KPP-generated RADM2 mechanism uses a three-stage Rosenbrock integrator. Four working copies of the concentration vector (three stages and output), an error vector, the ODE function value, the Jacobian function value, and the LU decomposition of $\frac{1}{h\gamma} - Jac_0$ are needed for each grid cell. This totals at least 1,890 floating point values per grid cell, or approximately 15KB of double precision data.

4. MULTI-CORE ARCHITECTURES

This section introduces the three multi-core architectures used in this study: homogeneous multi-core chipsets, general purpose graphics processing units (GPGPUs), and the Cell Broadband Engine Architecture (CBEA).

4.1 Homogeneous Multi-core Chipsets

Homogeneous multi-core design has supplanted single-core design in commercial servers, workstations, and laptops. The Intel Xeon 5400 Series [1] is typical of this design. It comprises two or four Xeon cores on a single die and is intended for general-purpose server and workstation computing. Each core has a private 16KB L1 data cache and shares 6MB of on-chip L2 cache with one other core. The 5400 series implements the Intel 64 architecture with Streaming SIMD Extensions 4 (SSE4) [34] in 45nm technology. A quad-core chip at 3GHz has a theoretical peak floating point performance of 48 gigaFLOPS with nominal 90W dissipation. It achieves 40.5 gigaFLOPS (0.5 gigaFLOPS/watt) in the LINPACK benchmark [18] . We include it in this study as an example of the current industry standard.

4.2 GPGPUs and NVIDIA CUDA

Graphics Processing Units (GPUs) are low-cost, massively-parallel homogeneous microprocessors designed for visualization and gaming. Because of their power, these special-purpose chips are being used for non-graphics "general-purpose" applications, hence the term GPGPU. The NVIDIA Tesla C1060 (Figure 2) has 4GB of GDDR3 device memory and 240 1.2GHz processing units on 30 multiprocessors. Each multi-processor has 16KB of fast shared memory and a 16K register file. The C1060's theoretical peak performance is 933 single precision gigaFLOPS (3.95 gigaFLOPS/watt) or 76 double precision gigaFLOPS (0.38 gigaFLOPS/watt) [4]. GPU performance is often an order of magnitude above that of comparable CPUs, and GPU performance has been increasing at a rate of 2.5x to 3.0x annually, compared with 1.4x for CPUs [22]. GPU technology has the additional advantage of being widely-depoloyed in modern computing systems. Many desktop workstations have GPUs which can be harnessed for scientific computing at no additional cost.

NVIDIA GPUs are programmed in CUDA [29]. Expressed as an extension of C and C++, CUDA is a model for parallel programming that provides a number of abstractions that allow the programmer to focus on algorithmic efficiency and develop scalable parallel applications. The programmer writes a serial program which calls parallel *kernels*, which may be simple functions or full programs. Kernels execute across a set of parallel lightweight threads to operate on data in the GPU's memory. Threads are organized into *thread blocks*. Threads in a block can cooperate among themselves through barrier synchronization and the shared memory. A collection of independent blocks forms a *grid*. The programmer specifies the number of threads per block and number of blocks per grid. Applications that perform well on the GPU do so by structuring data and computation to exploit registers and shared memory and have large numbers of threads to hide latency to device memory. Recent versions of NVIDIA GPUs incorporate a hardware double precision floating point unit in addition to eight SIMD streaming processors on each multi-processor. Since double precision operations must be pipelined through this unit instead of executing on the streaming processors, the GPU has a penalty for double precision beyond just the doubling of data volumes. In practice this is highly application specific.

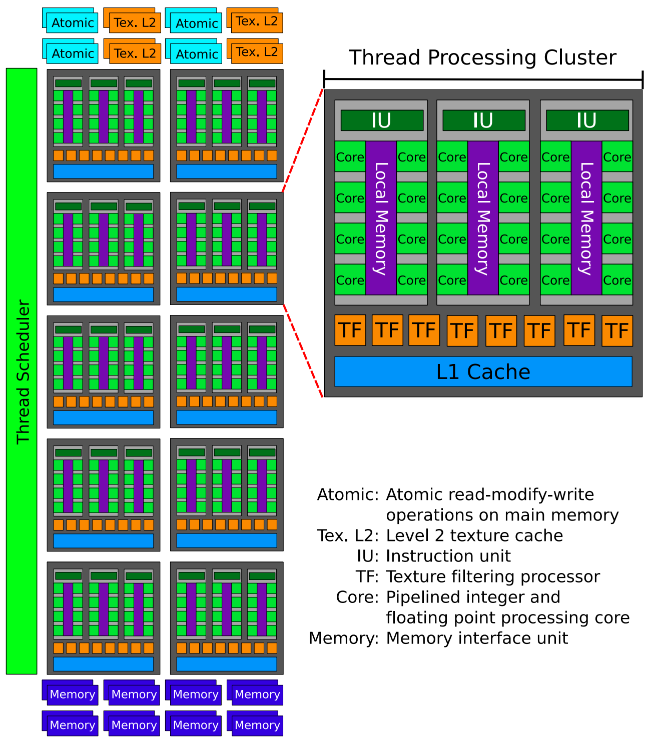
Since CUDA's release in 2007, it has become the focus of development activity by a community of tens of thousands of registered CUDA developers. Hundreds of scalable parallel programs for a wide range of applications, including computational chemistry, sparse matrix solvers, sorting, searching, and physics models have been developed [28]. Many of these applications scale transparently to hundreds of processor cores and thousands of concurrent threads. The CUDA model is also applicable to other shared-memory parallel processing architectures, including multi-core CPUs [32].

4.3 The CBEA

The CBEA describes a heterogeneous multi-core processor that has drawn considerable attention in both industry and academia (Figure 3). It was originally designed for the game box market, and therefore it has a low cost and low power requirements. The main components of the CBEA are a multithreaded Power Processing element (PPE) and eight Synergistic Processing elements (SPEs) [16]. 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 [24] and two levels of on-chip cache. Each SPE is a 128-bit SIMD processor with two major components: a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC). All SPE instructions are executed on the SPU. The SPE includes 128 registers of 128 bits and 256 KB of software-controlled local storage.

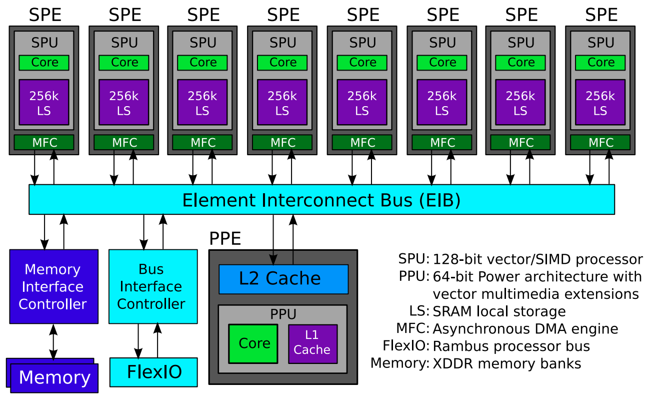
The MFC's DMA commands are subject to size and alignment restrictions. Data transferred between SPE local storage and main memory must be 8-byte aligned, at most 16 KB large, and in blocks of 1, 2, 4, 8, or multiples of 16 bytes. Multiples of 128 bytes are most efficient. Incontiguous (i.e. strided) data cannot be transferred with a single command and must be handled by *DMA lists*. A DMA list is a list of commands, each specifying a starting address and size. It is formed in local storage by the SPU and passed to the MFC.

The Cell Broadband Engine (Cell BE) is the game box implementation of the CBEA and may have either six or eight SPEs. It is primarily a single precision floating point processor. Its peak single precision FP performance is 230.4 gi-



Atomic
Tex. L2
Thread Scheduler
Thread Processing Cluster
IU
Core
Local Memory
TF
L1 Cache
Memory
Atomic: Atomic read-modify-write operations on main memory
Tex. L2: Level 2 texture cache
IU: Instruction unit
TF: Texture filtering processor
Core: Pipelined integer and floating point processing core
Memory: Memory interface unit

Figure 2: The parallel computing architecture features of the NVIDIA Tesla C1060. The thread scheduler hardware automatically manages thousands of threads executing on 240 cores. Hiding memory latency by oversubscribing the cores produces maximum performance.



SPE
SPU
Core
256k LS
MFC
Element Interconnect Bus (EIB)
Memory Interface Controller
Bus Interface Controller
PPE
L2 Cache
PPU
Core
L1 Cache
Memory
FlexIO
SPU: 128-bit vector/SIMD processor
PPU: 64-bit Power architecture with vector multimedia extensions
LS: SRAM local storage
MFC: Asynchronous DMA engine
FlexIO: Rambus processor bus
Memory: XDDR memory banks

Figure 3: The Cell Broadband Engine Architecture. Every element is explicitly controlled by the programer and is highly configurable. Careful consideration of data alignment and use of the SPU vector ISA produces maximum performance.

gaFLOPS (2.45 gigaFLOPS/watt) but peak performance for double precision is only 21.03 gigaFLOPS (0.22 gigaFLOPS/watt) [12]. The PowerXCell 8i processor is a second implementation of the CBEA intended for high-performance, double precision floating point intensive workloads that benefit from large-capacity main memory. It has nominal dissipation of 92W and a double precision theoretical peak performance of 115.2 gigaFLOPS (1.25 gigaFLOPS/watt) [3]. Roadrunner at Los Alamos, the first computer to achieve a sustained petaFLOPS, uses the PowerXCell 8i processor [5], and the top seven systems on the November 2008 Green 500 list [2] use the PowerXCell 8i.

5. KERNELS AND RESULTS

This section discusses porting, parallelizing, and benchmarking the RADM2 chemical kernel from WRF-Chem on three multi-core platforms. Both the CUDA and CBEA versions began with a translation of the serial Fortran code to C. CUDA is an extension of C and C++, and although two Fortran compilers exist for the CBEA (gfortran 4.1.1 and IBM XL Fortran 11.1), these compilers have known issues that make fine-tuning easier in C. Since KPP can generate code in both Fortran and C, we were able to automatically regenerate the majority of the RADM2 kernel by changing KPP's input parameters. A manual translation of the WRF-Chem/KPP interface was required.

Workloads.

WRF-Chem was configured, compiled, and run using two test cases: a **coarse grid** of 40×40 with 20 layers and a 240 second timestep, and a **fine grid** of 134×110 with 35 layers and a 90 second timestep. The input data to the RADM2 solver was written to files at the call site. The unmodified Fortran source files for the RADM2 chemical kinetics solver (generated by KPP during WRF-Chem compilation), along with a number of KPP-generated tables of indices and coefficients used by the solver, were isolated into a standalone program that reads in the input files and invokes the solver. The timings and output from the original solver running under the standalone driver for one time step comprised the baseline performance benchmark. It was compiled with Intel compiler 10.1 with options "-O3 -xT".

A by-hand analysis of the computational and memory access characteristics of the kernel was conducted. For the course grid, the kernel involves approximately 610,000 floating point operations and 10^6 memory accesses. These numbers vary with the number of solver iterations, which is dependent on the time step. Although the operation count is high, computational intensity is low: 0.60 operations per 8-byte word and .08 operations per byte. WRF meteorology is only about 5,000 operations per time step.

Table 1 shows the baseline serial performance in seconds based on ten runs of the benchmark on a single core of an Intel Quad-Core Xeon 5400 series. "Rosenbrock" indicates the inclusive time required to advance chemical kinetics one time step for all points in the domain. It corresponds to the process described in Figure 1 and includes "LU Decomp.", "LU Solve", "ODE Function", and "ODE Jacobian", timings for which are also reported. "LU Decomp" and "LU Solve" are used to solve a linear system within the Rosenbrock integrator. "ODE Function" and "ODE Jacobian" are the time spent computing the mechanism's ODE function $f(t, y)$, and Jacobian function $J(t, y)$, respectively.

Table 1: RADM2 timing (seconds) of the serial chemical kernel executed for one time step on a single core of an Intel Quad-Core Xeon 5400 series. The category labels are described in detail at the end of Section 5.

	Double		Single	
Rosenbrock	67.2134	20.8971	67.8416	21.0944
LU Decomp.	30.7513	10.1084	31.2645	10.2868
LU Solve	9.9441	3.1102	10.0512	3.1366
ODE Function	8.3253	2.3785	8.3453	2.3761
ODE Jacobian	9.3343	2.5421	9.3463	2.5430
	Fine	Coarse	Fine	Coarse

Table 2 shows the performance in seconds of the RADM2 multi-core ports. The labels are identical to those in Table 1. Several operations in the fine-grid double precision case took so long on the PlayStation 3 that the SPU hardware timer overflowed before they could complete. Accurate timings cannot be supplied in this case. Only the overall solver time was available for the "Tesla (a)" implementation because the entire solver is a single CUDA kernel. The GeForce GTX280 GPU has a higher clock rate (1.46 GHz) than the Tesla; it is otherwise identical. However, only the single-precision version of the chemistry kernel worked on this processor for unknown reasons. We are working with NVIDIA to determine why double precision fails on this particular card. The benchmark results are discussed in Section 5.4.

5.1 Intel Quad-Core Xeon

Since the chemistry at each WRF-Chem grid cell is independent, the outermost iteration over cells in the RADM2 kernel became the thread-parallel dimension; that is, a one-cell-per-thread decomposition. The Quad-Core Xeon port implements this with OpenMP. Attention had to be given to all data references, since the Rosenbrock integrator operates on pointers to global data structures. In some cases, simply declaring variables in the `private` or `shared` blocks of the parallel constructs did not prevent unwanted data sharing between threads. In these cases, data was copied from global structures to `threadprivate` variables.

5.2 NVIDIA CUDA

The CUDA implementation takes advantage of the very high degree of parallelism and independence between cells in the domain, using a straightforward cell-per-thread decomposition. The first CUDA version implemented the entire Rosenbrock mechanism (Figure 1) as a single kernel. This presented some difficulties and performance was disappointing (Table 2). The amount of storage per grid cell, precluded using the fast but small (16KB per multi-processor) shared memory to speed up the computation. On the other hand, the Tesla GPU has 384K registers which can be used to good effect, since KPP generated fully unrolled loops as thousands of assignment statements. The resulting CUDA-compiled code could use upwards of a hundred registers per thread, though this severely limited the number of threads that could be actively running, even for large parts of the Rosenbrock code that could use many more.

The second CUDA implementation addressed this by moving the highest levels of the Rosenbrock solver back onto the CPU: time loops, Runge-Kutta loops, and error-control branch-back logic. The lower levels of the Rosenbrock call tree – LU decomp., LU solve, ODE function evaluation, and Jacobi matrix operations, as well as vector copy, daxpy, etc. – were coded and invoked as separate kernels on the GPU. As with the single-kernel implementation, all data for the solver was device-memory resident and arrays were stored with cell-index stride-one so that adjacent threads access adjacent words in memory. This coalesced access best utilizes bandwidth to device memory on the GPU.

Other advantages of this multi-kernel implementation were: (1) it was easier to debug and measure timing since the GPU code was spread over many smaller kernels with control returning frequently to the CPU, (2) it was considerably faster to compile, and (3) it limited the impact of resource bottlenecks (register pressure, shared-memory usage) to only those affected kernels. Performance critical parameters such as the size of thread blocks and shared-memory allocation were adjusted and tuned separately, kernel-by-kernel, without subjecting the entire solver to worst-case limits. For example, using shared memory for temporary storage improved performance of the LU decomposition kernel but limited thread-block size to only 32 or 64 threads (depending on floating point precision). The LU code with fully rolled loops (a couple dozen lines) used only 16 registers per thread; the fully unrolled version of the code, as generated by KPP, used 124 registers per thread.[2] Since the number of threads per block was already limited by threads using shared memory, it did no harm to use the fully unrolled version of the kernel and take advantage of the large register file as well. The combined improvement for the LU decomposition kernel on the GTX 280 was a factor of 2.16 (0.799 seconds for a small number of threads per block using shared memory and large numbers of registers, versus 1.722 seconds for a larger number of threads per block but no shared memory and few registers).

One disadvantage of moving time and error control logic to the CPU was that all cells were forced to use the minimum time step and iterate the maximum number of times, even though only a few cells required that many to converge. For the WRF-Chem workload, 90 percent of the cells converge in 30 iterations or fewer. The last dozen or so cells required double that number. While faster by a factor of 3 to 4 on a per-iteration basis, the increase in wasted work limited performance improvement to less than a factor of two. On the Tesla, the improvement was only 9.5 seconds down to about 5 seconds for the new kernel. The "Tesla (b)" and "GTX 280" results in Table 2 were for the multi-kernel version of the solver, but with an additional refinement: time, step-length, and error were stored separately for each cell and vector masks were used to turn off cells that were converged. The solver still performed the maximum number of iterations; however, beyond the half-way mark, most thread-blocks did little or no work and relinquished the GPU cores very quickly. During the latter half of the solve, only a small percentage of thread-blocks had any active cells, so that branch-divergence – which occurs when a conditional evaluates differently for threads in a SIMD block, forcing both paths of the branch to be executed – was not a factor.

[2]Register per thread and other metrics were obtained reading hardware counters through the profiler distributed with CUDA by NVIDIA.

Table 2: RADM2 timings (seconds) of the multi-core kernels in a 200 Watt envelope. Each time shown is the minimum over several successive runs. Category labels are explained in detail at the end of Section 5. Tesla (a) is single-kernel implementation, (b) is multi-kernel. For a socket-to-socket comparison, double the time of all systems *except* Tesla C1060 and GTX 280.

		QS22		QS20		PS3		Xeon		Tesla C1060		GTX
		16 SPEs		16 SPEs		6 SPEs[a]		8 Cores		(a)[b]	(b)	280[c]
Single	Rosenbrock	2.070	0.783	1.652	0.787	7.003	2.012	9.083	2.925	9.479	3.285	2.475
	LU Decomp.	1.645	0.509	1.065	0.509	2.633	1.266	3.9659	1.358	****	1.022	0.799
	LU Solve	0.199	0.093	0.198	0.093	1.597	0.252	1.288	0.420	****	0.928	0.735
	ODE Fun.	0.040	0.017	0.040	0.017	0.496	0.049	1.061	0.312	****	0.333	0.236
	ODE Jac.	0.036	0.015	0.036	0.015	0.409	0.043	1.129	0.316	****	0.392	0.291
Double	Rosenbrock	5.822	1.150	10.150	2.003	****	3.412	9.358	3.115	10.774	6.900	****
	LU Decomp.	3.255	0.654	5.277	1.062	****	1.769	4.324	1.542	****	2.686	****
	LU Solve	1.048	0.210	2.180	0.439	****	0.700	1.397	0.463	****	2.150	****
	ODE Fun.	0.254	0.045	0.473	0.086	****	0.206	0.936	0.279	****	0.431	****
	ODE Jac.	0.335	0.057	0.732	0.124	****	0.168	1.038	0.292	****	0.572	****
		Fine	Coarse	Fine	Coarse	Fine	Coarse	Fine	Coarse	Coarse	Coarse	Coarse

[a]The poor fine grid PlayStation 3 performance due to the grid data (380MB) being larger than main memory.
[b]Timing detail was not available with single kernel (first implemenation) RADM2 on GPU.
[c]Only single-precision available.

5.3 Cell Broadband Engine Architecture

The heterogeneous Cell Broadband Engine Architecture forces a carefully-architected approach. The PPU is capable of general computation on both scalar and vector types, but the SPUs diverge significantly from general processor design [16]. An SPU cannot access main memory directly; it must instruct the MFC to copy data to the 256KB local store before it can operate on the data. It has no dynamic branch predication hardware and static misprediction costs 18 cycles, a loss equivalent to 12 single precision floating point operations. An SPU implements a vector instruction set architecture so scalar operations immediately discard at least 50 percent of the core's performance. These architecture features strongly discourage a homogeneous one-cell-per-thread decomposition across all cores.

We chose a master-worker approach for the CBEA port. The PPU, with full access to main memory, is the master. It prepares WRF-Chem data for the SPUs which process them and return them to the PPU. In order to comply with size and alignment restrictions (see Section 4.3), a user-defined grid cell type, `cell_t`, completely describing the state of a single gridpoint, was defined. The PPU manages a buffer of `cell_t` objects, filling it with data from the incontiguous WRF-Chem data. Macros pad `cell_t` to a 128-byte boundary at compile time, and the PPU uses the `__attribute__((aligned(128)))` compiler directive to place the first array element on a 128-byte boundary. Thus, every element of the buffer is correctly aligned and can be accessed by the MFC. The buffer is made arbitrarily large to ensure SPUs never wait for the buffer to fill. Since the solver is computation-bound, the PPU has ample time to maintain the buffer, and in fact, the PPU may idle while waiting for results from the SPUs. To recover these wasted cycles, we configured the PPU to switch to a "worker" mode and apply the solver routine to the buffer while waiting. In the fully optimized code, the PPU can process approximately 12 percent of the domain while waiting for SPUs to complete.

The SPU's floating point SIMD ISA operates on 128-bit vectors of either four single precision or two double precision floating point numbers. To take advantage of SIMD, we extended `cell_t` to contain the state of either two or four grid points, depending on the desired precision. The PPU interleaves the data of two (four) grid points into a vector cell, implemented as `vcell_t`, which is padded, aligned, buffered and transfered exactly as in the scalar case. This achieves a four-cells-per-thread (two-cells-per-thread in double precision) decomposition.

Only one design change in the Rosenbrock integrator was necessary to integrate a vector cell. As shown in Figure 1, the integrator iteratively refines the Newton step size h until the error norm is within acceptable limits. This will cause an intra-vector divergence if different vector elements accept different step sizes. However, it is numerically sound to continue to reduce h even after an acceptable step size is found. The SIMD integrator reduces the step size until the error for every vector element is within tolerance. Conventional architectures would require additional computation under this scheme, but because all operations in the SPU are SIMD this actually recovers lost FLOPS. This enhancement doubled (quadrupled for single precision) the SPU's throughput with no measurable overhead on the SPU.

CBEA Benchmarks.

The performance of the CBEA port is shown in Table 2. Benchmarks were performed on an in-house PlayStation 3 system, an IBM BladeCenter QS22 at Forschungszentrum Jülich, and an IBM BladeCenter QS20 at Georgia Tech. The PlayStation 3 and the QS20 use the Cell Broadband Engine and lack hardware support for pipelined double precision arithmetic. The QS22 uses the PowerXCell 8i and includes hardware support for pipelined double precision arithmetic. The PlayStation 3 has 256MB XDRAM, the QS20 has 1GB XDRAM, and the QS22 has 8GB XDRAM. Both the QS20 and the QS22 are configured as "glueless" dual processors: two CBEA chipsets are connected through their FlexIO interfaces to appear as a single chip with 16 SPEs and 2 PPEs. The PS3 has only six available SPEs for yield reasons. We include it here because it is representative of "small memory" systems: computers with a low ratio of memory to cores. Our benchmarks used a single thread on a single PPE to service the maximum number of available SPEs. We did not use the multi-threading capabilities of the PPE, and a whole PPE is completely unused in the BladeCenter systems. Further optimizations are possible.

Every benchmark was performed with executables from two compilers: GCC from the Cell SDK 3.1 and IBM XLC 11.1. For single precision, the best performance was from GCC with only "-O5" as arguments. XLC's optimizations introduced too many precision errors to achieve single precision solver convergence; we only achieved convergence with the arguments "-qhot=level=0:novector -qstrict=all". For double precision, the best performance was from XLC with arguments "-O5 -qarch=edp" on the QS22 system. QS20 systems achieved best performance with GCC and "-O5".

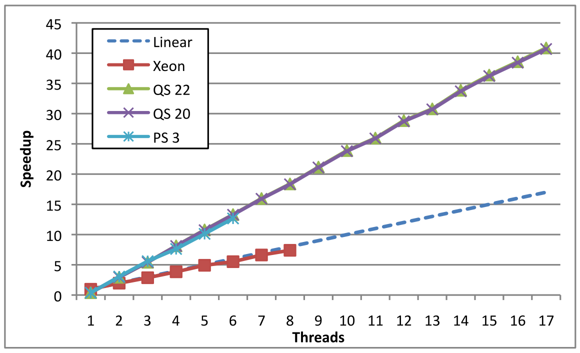
5.4 Performance Analysis and Discussion

Table 2 shows the performance of all the parallel implementations. On a 200 Watt power budget, two Quad-Core Xeon chips, two CBEA chips, or one Tesla C1060 GPU can be allocated as shown. For a socket-to-socket comparison, the numbers in Table 2 should be doubled for all but the GPU case.

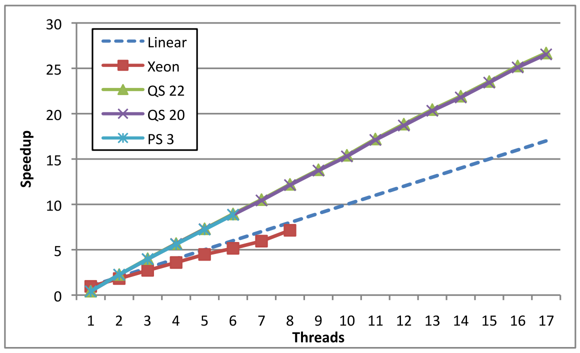
Of the three platforms investigated, Quad-Core Xeon with OpenMP was by far the easiest to program. A single address space and single ISA meant only one copy of the integrator source was necessary. Also, OpenMP tool chains are more mature than those for GPUs or the CBEA. As shown in Figure 4, this port achieved nearly linear speedup over eight cores. However, it may be unreasonable to expect this trend to continue for similar architectures with hundreds of cores. Current interconnect designs cannot provide the necessary memory bandwidth in this paradigm.

The CBEA implementation achieves the best performance. On a fine-grain double precision grid, two PowerXCell 8i chipsets are 11.5× faster than the serial implementation, and 32.9× faster in single precision. Coarse grained single precision grids see a speedup of 28.0×. The CBEA's explicitly-managed memory hierarchy and fast on-chip memory provide this performance. Up to 40 grid cells can be stored in SPE local storage, so the SPU never waits for data. Because the memory is explicitly managed, data can be intelligently and asynchronously prefetched. However, the CBEA port was difficult to implement. Two optimized copies of the solver code, one for the PPU and one for the SPU, were required. On-chip memory must be explicitly managed and careful consideration of alignment and padding are the programmer's responsibility. Compiler technologies for the Cell have not matured as rapidly as hoped, leaving many menial tasks up to the programmer. At this time, programming the Cell is prohibitively difficult without a deep understanding of the Cell's architecture. Research into better tool chains and code generators is certainly justified.

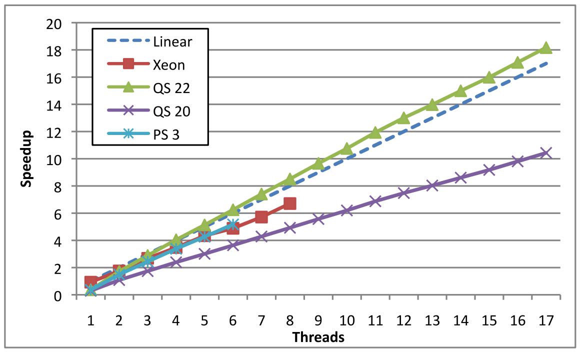
The NVIDIA CUDA implementation was also straightforward to program, however it proved to be the most difficult to optimize. CUDA's automatic thread management and familiar programming environment improve programmer productivity: our first implementation of RADM2 on GPU was simple to conceive and implement. However, a deep understanding of the underlying architecture is still required in order to achieve good performance. For example, memory access coalescing is one of the most powerful features of the GPU architecture, yet CUDA neither hinders nor promotes program designs that leverage coalescing.



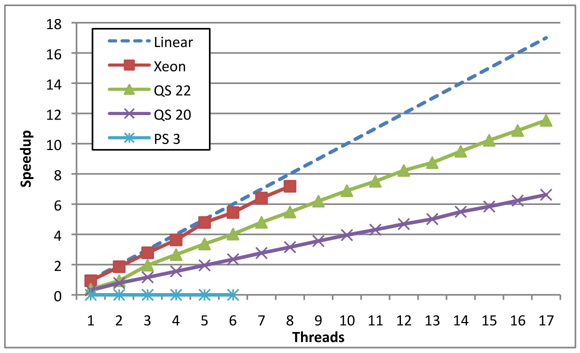
(a) Coarse - Single



(b) Fine - Single



(c) Coarse - Double



(d) Fine - Double

Figure 4: OpenMP and CBEA speedup as compared to the original serial code. GPU speedup is omitted since the thread models of the GPU and conventional architectures are not directly comparable.

In our case, the GPU required the most effort to achieve acceptable performance. On a single-precision coarse grid, this implementation achieves an 8.5× speedup over the serial implementation. The principal limitation is the size of the on-chip shared memory and register file, which prevent large-footprint applications from running sufficient numbers of threads to expose parallelism and hide latency to the device memory. The entire concentration vector must be available to the processing cores, but there is not enough on-chip storage to achieve high levels of reuse, so the solver is forced to fetch from the slow GPU device memory. Because the ODE system is coupled, a per-species decomposition is generally impossible. However, for certain cases, it may be possible to decompose by species groups. This will reduce pressure on the shared memory and boost performance. The alternative is to wait until larger on-chip shared memories are available.

6. MULTI-CORE KPP

In order to generalize our work with the RADM2 kernel, we explored extensions of KPP that would allow it to generate highly-optimized code in multiple languages for a range of multi-core platforms. Before this work, it was unknown what a highly-optimized multi-core implementation of chemical kinetics "looked like". While porting to each platform, care was taken to avoid any design which was specific to RADM2. Thus, the multi-core RADM2 implementations form templates for any chemical kinetics mechanism running on these architectures.

KPP generates code in two ways: complete function generation using language trees, and template file specification. In template file specification, source code templates are copied from a library and then "filled in" with code appropriate to the chemical mechanism being generated. This is the method used to generate the core Rosenbrock integration functions. Language trees describing language-independent source code are generated while forming the chemical system by the method in Section 3.1. This is the method used to generate the ODE function and Jacobian function.

Five extensions to KPP for multi-core architectures were identified. First, architectures with SIMD features, such as the Cell's SPE, require `vector double` and `vector float` types. Language tree functionality will need to be extended to generate code using these types by describing vector types and their algebra to KPP. Secondly, template code for a vectorized Rosenbrock solver and it's associated BLAS functions needs to be added to KPPs library. The vectorized integrator from our RADM2 implementation is easily converted to a general template. Third, template code for multi-threaded communication and synchronization for every target multi-core platform needs to be added to KPP. Again, our RADM2 implementation can be used as the template. Fourth, the KPP input file processor needs to be updated to accept a parameterization of the target multi-core platform. This could be as simple as a target name (i.e. GPGPU) or as complex as a short machine description. Finally, KPP's makefile generation routines need to be updated to target a variety of tool chains.

To prove the feasibility of this method, a partial extension of KPP for the CBEA was completed using our RADM2 implementation as a template. The partial implementation cannot generate Fortran, nor can it target GPUs or homogeneous multi-core. Furthermore, it does not optimize SIMD code generation so a great many unnecessary scalar-to-vector conversions are performed. All these issues will be addressed in future work. Nevertheless, the extended KPP was successfully used to generate a SAPRC [9] mechanism involving 93 species in 235 reactions. Once KPP was set up with the template, virtually no effort was required to generate the multi-core mechanism.

7. CONCLUSIONS AND FUTURE WORK

We have presented optimized ports of the RADM2 chemical kinetics mechanism from WRF-Chem for three multi-core platforms: NVIDIA CUDA, the Cell Broadband Engine Architecture (CBEA), and OpenMP. The methodology for porting to each platform was described in a mechanism-agnostic way. Thus, any chemical mechanism can be ported to these platforms by using our implementation as a template.

A detailed performance analysis for each platform was given. The CBEA achieves the best performance due to its fast, explicitly-managed on-chip memory: a maximum speedup of 41.1× as compared to the serial implementation. The GPU's performance is severely hampered by the limited amount of on-chip memory, yet it achieves a single-precision speedup of 8.5× as compared to the serial implementation. The OpenMP implementation achieved almost linear speedup for up to eight cores with only moderate programming effort: a maximum speedup of 7.5×.

The Kinetic PreProcessor (KPP) was extended to demonstrate the feasibility of using our implementation as a template for automatic code generation. A fully-extended KPP will automatically generate Fortran or C code for many chemical mechanisms targeting a variety of multi-core platforms. Future work will complete this extension of KPP. Only the KPP-generated chemical kinetics solver is the subject of acceleration here; the SORGAM aerosol kernel will be addressed in future work.

8. ACKNOWLEDGMENTS

The authors wish to thank Georg Grell and Steve Peckham of the NOAA Earth Systems Research Laboratory for their advice and assistance with WRF-Chem; Paulius Micikevicius and Gregory Ruetsch at NVIDIA Corporation; Prof. Dr. Felix Wolf of RWTH Aachen; and the Jülich Supercomputing Centre, and Willi Homberg of the Jülich Supercomputing Centre. The authors 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. This work was partially supported by the National Center for Supercomputing Applications which made available the NCSA's experimental GPU cluster.

9. REFERENCES

[1] Platform brief: Intel xeon processor 5400 series. `http://download.intel.com/products/processor/xeon/dc54kprodbrief.pdf`, 2007.

[2] The Green500 list. `http://www.green500.org/`, November 2008.

[3] PowerXCell 8i processor technology brief. `http://www-03.ibm.com/technology/cell/`, August 2008.

[4] Technical brief: NVIDIA GeForce GTX 200 GPU architectural overview. Technical Report TB-04044-001_v01, NVIDIA Corporation, May 2008.

[5] 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 (SC'08)*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[6] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *ACM SIGGRAPH 2005 Courses (SIGGRAPH'05)*, page 171, New York, NY, USA, 2005. ACM.

[7] D. W. Byun and J. K. S. Ching. Science algorithms of the EPA models-3 community multiscale air quality (CMAQ) modeling system. Technical Report U.S. EPA/600/R-99/030, U.S. EPA, 1999.

[8] G. R. Carmichael, L. K. Peters, and T. Kitada. A second generation model for regional scale transport / chemistry / deposition. *Atmos. Env.*, 20:173–188, 1986.

[9] W. P. L. Carter. Documentation of the SAPRC atmospheric photochemical mechanism preparation and emissions processing programs for implementation in airshed models. Report to the California Air Resourcs Board Contract No. A5-122-32, California Air Resources Board, 1988.

[10] W. P. L. Carter. A detailed mechanism for the gas-phase atmospheric reactions of organic compounds. *Atmos. Env.*, 24A:481–518, 1990.

[11] J. Chang, F. Binowski, N. Seaman, J. McHenry, P. Samson, W. Stockwell, C. Walcek, S. Madronich, P. Middleton, J. Pleim, and H. Lansford. The regional acid deposition model and engineering model. State-of-Science/Technology Report 4, National Acid Precipitation Assessment Program, Washington D.C., 1989.

[12] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation. *IBM developerWorks*, June 2006.

[13] V. Damian, A. Sandu, M. Damian, F. Potra, and G. R. Carmichael. The Kinetic Preprocessor KPP – a software environment for solving chemical kinetics. *Comput. Chem. Eng.*, 26:1567–1579, 2002.

[14] P. Eller, K. Singh, A. Sandu, K. Bowman, D. K. Henze, and M. Lee. Implementation and evaluation of an array of chemical solvers in the Global Chemical Transport Model GEOS-Chem. *Geosci. Model Dev.*, 2:89–96, 2009.

[15] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on supercomputing (SC'04)*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.

[16] B. Flachs, S. Asano, S. H. Dhong, H. P. Hofstee, G. Gervais, R. Kim, T. Le, and et. al. The microarchitecture of the synergistic processor for a cell processor. *IEEE J. Solid State Circuits*, 41(1):63–70, 2006.

[17] N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha. Lu-gpu: Algorithms for dense linear systems on graphics hardware. In *Proceedings of the 2005 ACM/IEEE conference on supercomputing (SC'05)*, Piscataway, NJ, USA, 2005. IEEE Press.

[18] P. Gepner, D. L. Fraser, and M. F. Kowalik. Second generation quad-core intel xeon processors bring 45 nm technology and a new level of performance to hpc applications. In *Proceedings of the 8th International Conference on Computational Science (ICCS'08)*, volume 5101/2008 of *Lecture Notes in Computer Science*, pages 417–426, Kraków, Poland, 23–25 June 2008. Springer Berlin / Heidelburg.

[19] M. W. Gery, G. Z. Whitten, J. P. Killus, and M. C. Dodge. A photochemical kinetics mechanism for urban and regional scale computer modeling. *J. Geophys. Res.*, 94(D10):12925–12956, 1989.

[20] G. A. Grell, S. E. Peckham, R. Schmitz, S. A. McKeen, G. Frost, W. C. Skamarock, and B. Eder. Fully coupled online chemistry within the WRF model. *Atmos. Env.*, 39:6957–6975, 2005.

[21] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential Algebraic Problems*, volume 14 of *Springer Series in Comput. Mathematics*. Springer-Verlag, 2nd edition edition, 1996.

[22] B. Himawan and M. Vachharajani. Deconstructing hardware usage for general purpose computation on GPUs. In *Fith Annual Workshop on Duplicating, Deconstructing, and Debunking (in conunction with ISCA-33)*, 2006.

[23] K. Z. Ibrahim and F. Bodin. Implementing wilson-dirac operator on the Cell Broadband Engine. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS '08)*, pages 4–14, New York, NY, USA, 2008. ACM.

[24] International Business Machines Corporation. *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*, 2.07c edition, October 2006.

[25] M. Z. Jacobson and R. Turco. SMVGEAR: A sparse-matrix, vectorized gear code for atmospheric models. *Atmos. Environ.*, 28:273–284, 1994.

[26] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM SIGGRAPH 2005 Courses (SIGGRAPH'05)*, page 234, New York, NY, USA, 2005. ACM.

[27] P. Middleton, W. Stockwell, and W. Carter. Aggregation and analysis of volatile organic compound emissions for regional modeling. *Atmos. Environ.*, 24A:1107–1133, 1990.

[28] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. In *ACM SIGGRAPH 2008 Courses (SIGGRAPH'08)*, pages 1–14, New York, NY, USA, 2008. ACM.

[29] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide Version 2.0*, 2008.

[30] K. S. Perumalla. Discrete-event execution alternatives on general purpose graphical processing units (GPGPUs). In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation (PADS'06)*, pages 74–81, Washington, DC, USA,

2006. IEEE Computer Society.

[31] W. R. Stockwell, P. Middleton, J. S. Chang, and X. Tang. The second generation regional acid deposition model chemical mechanism for regional air quality modeling. *J. Geophys. Res.*, 95:16343–16367, 1990.

[32] J. Stratton, S. Stone, and W. mei Hwu. MCUDA: An efficient implementation of CUDA kernels on multi-cores. Technical Report IMPACT-08-01, University of Illinois at Urbana-Champaign, March 2008.

[33] V. Volkov and J. Demmel. Lu, qr and cholesky factorizations using vector capabilities of gpus. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.

[34] O. Wechsler. Inside intel core microarchitecture. White paper, Intel Corporation, 2006.

[35] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on supercomputing (SC'07)*, pages 1–12, New York, NY, USA, 2007. ACM.

[36] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *Proceedings of the 3rd Conference on Computing Frontiers (CF '06)*, pages 9–20, New York, NY, USA, 2006. ACM.