Automatic Generation of Multicore Chemical Kernels

John C. Linford, John Michalakes, Manish Vachharajani, and Adrian Sandu

Abstract—This work presents the Kinetics PreProcessor: Accelerated (KPPA), a general analysis and code generation tool that achieves significantly reduced time-to-solution for chemical kinetics kernels on three multicore platforms: NVIDIA GPUs using CUDA, the Cell Broadband Engine, and Intel Quad-Core Xeon CPUs. A comparative performance analysis of chemical kernels from WRF-Chem and the Community Multiscale Air Quality Model (CMAQ) is presented for each platform in double and single precision on coarse and fine grids. We introduce the multicore architecture parameterization that KPPA uses to generate a chemical kernel for these platforms and describe a code generation system that produces highly tuned platform-specific code. Compared to state-of-the-art serial implementations, speedups exceeding $25\times$ are regularly observed, with a maximum observed speedup of $41.1\times$ in single precision.

Index Terms—KPPA, multicore, NVIDIA CUDA, Cell Broadband Engine, OpenMP, chemical kinetics, atmospheric modeling, Kinetics PreProcessor, WRF-Chem, CMAQ.

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) [1], the Community Multiscale Air Quality Model (CMAQ) [2], the Sulfur Transport and dEposition Model (STEM) [3], and GEOS-Chem [4] 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 equations arising from the chemical reactions, which may involve millions of variables [5]. The stiffness of these equations, arising from the widely varying reaction rates, prohibits their solution through explicit numerical methods.

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, an 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 typically sized 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. As a second example, Fig. 1 shows the performance of serial and parallel runs of the global tropospheric model GEOS-Chem. The cost of chemical kinetics dominates in both runs, even as the number of threads increases. A strong scaling approach is needed to improve performance.

The new generations of multicore 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 the time-to-solution for a range of earth system applications.

Writing chemical kinetics code is often tedious and error-prone work, even for conventional scalar architectures. Emerging multicore architectures, particularly heterogeneous emerging architectures such as the Cell Broadband Engine Architecture (CBEA) and General Purpose Graphics Processing Units (GPGPUS), are much harder to program than their scalar predecessors. For these architectures, a deep understanding of the problem domain is required to achieve good performance. General analysis tools like the Kinetic PreProcessor (KPP) [5] make it possible to rapidly generate correct and efficient chemical kinetics code on scalar architectures, but these generated codes cannot be easily ported to strong-scaling emerging architectures.

This work presents the Kinetics PreProcessor: Accelerated (KPPA), a general analysis and code generation tool that achieves significantly reduced time-to-solution for chemical kinetics kernels. KPPA facilitates the numerical solution of chemical reaction network problems and generates code targeting OpenMP, NVIDIA GPUs with CUDA, and the CBEA, in C and Fortran, and in double and single precision. KPPA-generated mechanisms leverage platform-specific multilayered heterogeneous parallelism to achieve strong

- *J.C. Linford and A. Sandu are with the Department of Computer Science, Virginia Polytechnic Institute and State University, Room 2201, Knowledgeworks II, 2202, Kraft Drive, Blacksburg, VA 24061. E-mail: jlinford@vt.edu, sandu@cs.vt.edu.*
- *J. Michalakes is with the Mesoscale and Microscale Meteorology Division, National Center for Atmospheric Research, 3456 Mitchell Lane, Boulder, CO 80307. E-mail: michalak@ucar.edu.*
- *M. Vachharajani is with the Department of Electrical, Computer, and Energy Engineering, University of Colorado, 425 UCB, Boulder, CO 80309. E-mail: manishv@colorado.edu.*

Manuscript received 24 Sept. 2009; revised 22 Feb. 2010; accepted 26 Mar. 2010; published online 19 May 2010.
Recommended for acceptance by D.A. Bader, D. Kaeli, and V. Kindratenko.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDSSI-2009-09-0448.
Digital Object Identifier no. 10.1109/TPDS.2010.106.

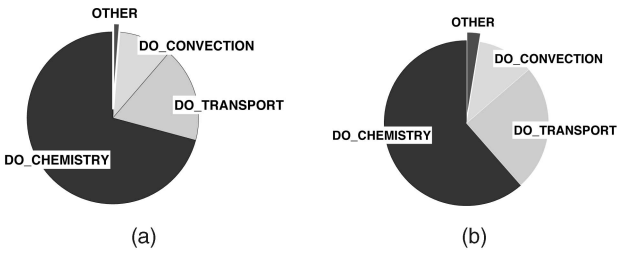

Fig. 1. Performance breakdown of GEOS-Chem. DO_CHEMISTRY, DO_CONVECTION, and DO_TRANSPORT are the wall-clock time spent fully computing chemical kinetics, convective transport, and advective transport, respectively. (a) Serial (b) Parallel with eight threads.

scalability. Compared to state-of-the-art serial implementations, speedups as high as $41.1\times$ are observed.

The rest of this paper is organized as follows: Related work is shown in Section 2. An overview of KPPA is presented in Section 3. Formulation of a chemical kinetics model is briefly discussed in Section 4. A typical homogeneous multicore chipset, NVIDIA GPUs, and the CBEA are outlined in Section 5. We conduct preliminary work with the RADM2 kinetics kernel on three multicore platforms in Section 6. Multicore code generation is described in Section 7, and the performance of three KPPA-generated codes is given in Section 8. We conclude this paper in Section 10.

2 Related Work

Implementing chemical kinetics models on emerging multicore technologies can be unusually difficult because expertise in kinetics and atmospheric modeling must be combined with a strong understanding of various multicore paradigms. For this reason, existing literature tends to focus on the model subcomponents, such as basic linear algebra operations [6], [7], [8], and does not comprehensively address this problem domain for chemical kinetics and related problems on real-world domains.

2.1 Atmospheric Modeling and Linear Algebra

Research into using GPUs to accelerate the transport and diffusion of atmospheric constituents is ongoing. Fan et al. [9] 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 $4.6\times$ speedup in their simulation. Perumalla [10] 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 $16\times$ 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. [8], [11] achieved a maximum speedup of $12.7\times$ and power efficiency of $28.3\times$ 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. Dongarra et al. [12] report up to 328 single-precision gigaflops when computing a left-looking block Cholesky factorization on a prereleased NVIDIA T10P. Bolz et al. [13] implemented a sparse matrix conjugate gradient solver and a regular-grid multigrid solver on NVIDIA GeForce FX hardware. The GPU performed 120 unstructured (1,370 structured) matrix multiplies per second, while an SSE implementation achieved only 75 unstructured (750 structured) matrix multiplies per second on a 3.0 GHz Pentium 4 CPU. Krüger and Westermann [14] 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. More recent efforts include the MAGMA project [15] which is developing a successor to LAPACK but for heterogeneous/hybrid architectures.

Our work uses the heterogeneous parallelism of the CBEA in a way similar to that introduced by Ibrahim and Bodin in [16]. 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 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.

2.2 General Analysis, Code Generation, and Tuning

Autotuning techniques like those found in ATLAS [17] and FFTW [18] are being applied to linear algebra on GPUs. Li et al. [19] designed a GEMM autotuner for NVIDIA CUDA-enabled GPUs that improved the performance of a highly tuned GEMM kernel by 27 percent.

KPP [5] 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. KPP has been successfully used to treat many chemical mechanisms from tropospheric and stratospheric chemistry, including CBM-IV [20], SAPRC [21], and NASA HSRP/AESA. The Rosenbrock methods implemented in KPP typically outperform backward differentiation formulas, like those implemented in SMVGEAR [22], [23].

3 Overview of KPPA

KPPA (Fig. 2) combines a general analysis tool for chemical kinetics with a code generation system for scalar, homogeneous multicore, and heterogeneous multicore architectures. It is written in object-oriented C++ with a clearly defined upgrade path to support future multicore architectures as they emerge. KPPA has all the functionality of KPP 2.1, the latest version of KPP, and generates mechanism code in C, Fortran, or other platform-specific languages, such as CUDA.

KPPA's input files and lexical parser are enhanced versions of the same components from KPP. Several new

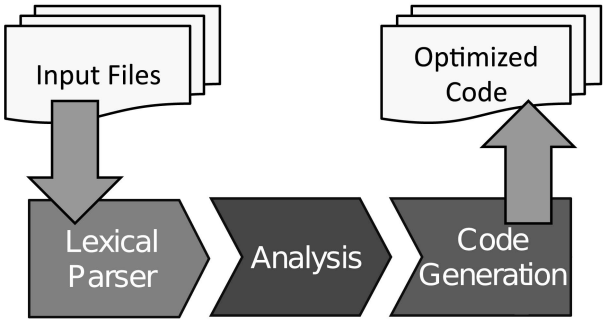

Fig. 2. Principal KPPA components and its program flow.

keywords describing the target architecture, loop unrolling parameters, and other new features have been added to the parser, but all original KPP functionality is retained.

The general analysis component was rewritten from scratch but borrows heavily from KPP. It is used to formulate the chemical system as described in Section 4. Many atmospheric models, including WRF-Chem and STEM, support a number of chemical kinetics solvers that are automatically generated at compile time by KPP. Reusing these analysis techniques in KPPA insures its accuracy and applicability. KPPA is backwards-compatible with KPP and can be used as a drop-in replacement in many situations. KPPA is released under the GNU General Public License (GPL) and can be downloaded from http://people.cs.vt.edu/jlinford/kppa.

The code generation component is written from scratch to accommodate the 2D design space of programming language/target architecture combinations (Table 1). Given the model description from the analytical component and a description of the target architecture, the code generation component produces a time-stepping integrator, the ODE function, and ODE Jacobian of the system, and other quantities required to interface with an atmospheric model. It can generate code in several languages, and can be extended to new target languages as desired. Its key feature is the ability to generate fully unrolled, platform-specific sparse matrix/matrix and matrix/vector operations which achieve very high levels of efficiency. Code generation is described in Section 7.

4 Atmospheric Chemical Kinetics

Regardless of computational architecture or modeling application, the solution to the chemical reaction network problem is calculated in the same way. A time-stepping method advances the system of coupled and stiff[1] ODEs through time.

4.1 Forming the Chemical System

Given a reaction network and initial concentrations as input files, KPPA generates code to solve the differential equation of mass action kinetics to determine the concentration at any future time. The derivation of this equation is given at length in [5] and summarized here.

1. Lambert defines stiffness in [24]: "If a numerical method is forced to use, in a certain interval of integration, a step length which is excessively small in relation to the smoothness of the exact solution in that interval, then the problem is said to be stiff in that interval."

TABLE 1
Language/Architecture Combinations Supported by KPPA

	Serial	OpenMP	GPGPU	CBEA
C	κ^*	κ	κ	κ
FORTRAN77	κ^*	κ		κ
Fortran 90	κ^*	κ		κ
MATLAB	κ^*			

κ^* *indicates functionality offered by existing tools.*

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

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

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

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

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

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

If we organize the stoichiometric coefficients in two matrices,

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

then (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. KPPA can produce both aggregate and production-destrucion forms. Implicit integration methods also require the evaluation of the Jacobian of the derivative function:

Initialize $k(t,y)$ from starting concentrations and meteorology (ρ, 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 (ρ, 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}

Fig. 3. A general outline of the three-stage Rosenbrock solver for chemical kinetics. Here, 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, and $k(t,y)$, $f(t,y)$, and $J(t,y)$ are as given in Section 4.1. Y_{new} is the new concentration vector.

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

4.2 Solving the Chemical System

The solution of the ordinary differential equations is advanced in time by a numerical integration method. We focus on the Rosenbrock integrator with three implicitly solved Newton stages as an example, but KPPA also supports Runge-Kutta methods [25]. The Rosenbrock implementation takes advantage of sparsity as well as trading exactness for efficiency when reasonable. An outline of the implementation is shown in Fig. 3.

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 in Fig. 3. Autonomicity is largely irrelevant to the implementation of both serial and multicore solver implementations since the reaction rates are calculated from data external to the integrator. Multicore architectures with limited per-thread memory (such as the CBEA) can take advantage of this independence by overlaying the integration routine with the reaction rate updates. In our experience, however, there is sufficient per-thread memory in any architecture to make this optimization unnecessary.

A chemical mechanism may be so stiff that it will not converge without double-precision floating-point computation. One example is the SAPRCNOV mechanism [21]. KPPA can generate code in either double or single precision. The choice of target architecture may have a profound impact on performance if double precision is required.

Once an iterative solver has been generated, an atmospheric model applies the solver to every point on a fixed-domain grid. Chemical kinetics are embarrassingly parallel between cells, so there is abundant data parallelism (DLP). Generally, 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). Some specific chemical mechanisms are only partially coupled and can be separated into a small number of subcomponents, but such intermodule decomposition is rare under the numerical methods examined in this work. Thus, a three-tier parallelization is generally possible: ILP on each core, DLP using single-instruction multiple-data (SIMD) features of a single core, and DLP across multiple cores (using multithreading) or nodes (using MPI). The coarsest tier of MPI and OpenMP parallelism is supplied by the atmospheric model.

5 Multicore Architectures

Once the general analysis component has formulated the chemical system as in Section 4, the code generation component constructs the architecture- and language-specific implementation of the numerical integration method. KPPA can generate code for homogeneous multicore chipsets, NVIDIA GPUs with CUDA, and the CBEA. Since power consumption and thermal issues have become the principle limitation to computing power, both the peak gigaflops and the gigaflops per dissipated watt should be considered when investing in a new technology. This section reviews these architectures.

5.1 Homogeneous Multicore Chipsets

Homogeneous multicore design has supplanted single-core design in commercial servers, workstations, and laptops. The Intel Xeon 5400 Series [26] is typical of this design. A quad-core chip at 3 GHz has a theoretical peak floating-point performance of 48 gigaflops with nominal 90 W dissipation. It achieves 40.5 gigaflops (0.5 gigaflops/watt) in the LINPACK benchmark [27]. We include it in this study as an example of the current industry standard and the baseline performance metric.

5.2 GPGPUs and NVIDIA CUDA

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 nongraphics "general-purpose" applications, hence the term GPGPU. The NVIDIA Tesla C1060 has 4 GB of GDDR3 device memory and 240 1.2 GHz processing units on 30 multiprocessors. Each multiprocessor has 16 KB of fast shared memory and a 16 K register file. The C1060's theoretical peak performance is 933 single-precision gigaflops (4.96 gigaflops/watt) or 76 double-precision gigaflops (0.41 gigaflops/watt) [28]. GPU performance is often an order of magnitude above that of comparable CPUs, and GPU performance has been increasing at a rate of $2.5\times$ to $3.0\times$ annually, compared with $1.4\times$ for CPUs [29]. GPU technology has the additional advantage of being widely deployed in modern computing systems. Many desktop workstations have GPUs which can be harnessed for scientific computing at no additional cost.

Recent versions of NVIDIA GPUs incorporate a hardware double-precision floating-point unit in addition to

eight SIMD streaming processors on each multiprocessor. 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. 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 device memory latency.

5.3 The Cell Broadband Engine Architecture (CBEA)

The CBEA describes a heterogeneous multicore processor that has drawn considerable attention in both industry and academia [30]. It consists of a multithreaded Power Processing element (PPE) and eight Synergistic Processing elements (SPEs) [31]. 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. 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.

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 with a peak single-precision FP performance of 230.4 gigaflops (2.45 gigaflops/watt) and a double-precision peak of only 21.03 gigaflops (0.22 gigaflops/watt) [30]. The PowerXCell 8i processor is the latest 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 92 W and a double precision theoretical peak performance of 115.2 gigaflops (1.25 gigaflops/watt) [32]. Roadrunner at Los Alamos, the first computer to achieve a sustained petaflop, uses the PowerXCell 8i processor [33], and the top seven systems on the November 2008 Green 500 list use the PowerXCell 8i [34].

5.4 Benchmark Systems

The benchmarks presented in this work assume a 200 watt power budget for the principal computing chipset. Within this envelope, two Quad-Core Xeon chips, two CBEA chips, or one Tesla C1060 GPU can be allocated. Supporting systems, such as memory and I/O, are not included in the budget, and since the Tesla GPU coprocesses for a CPU, it exceeds the budget by at least 90 watts. The quad-Core Xeon benchmarks are performed on a Dell Precision T5400 workstation with two Intel E5410 CPUs and 16 GB of memory on a 667 MHz bus. NVIDIA GPU benchmarks are performed on the NCSA's experimental GPU cluster. Each cluster node has two dual-core 2.4 GHz AMD Opteron CPUs and 8 GB of memory (only one Opteron is used in this study). CBEA benchmarks are 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 256 MB XDRAM, the QS20 has 1 GB XDRAM, and the QS22 has 8 GB 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 SPEs available for yield reasons.

6 Multicore Chemical Kinetics

Serial implementations of chemical kinetics mechanisms have been studied for decades. However, the uniqueness of the architectures described in Section 5 makes it difficult to anticipate implementation details of a general mechanism, that is, it was unknown what a highly optimized multicore implementation of chemical kinetics "looked like." In order to develop KPPA code generation functionality, a detailed exploration of a specific mechanism on every target architecture was therefore required.

6.1 RADM2 on Multicore

We begin our investigation by hand-porting and benchmarking the RADM2 chemical kernel from WRF-Chem on three multi-core platforms. RADM2 was developed by Stockwell et al. [35] for the Regional Acid Deposition Model version 2 [36]. 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 [37]. 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 [37].

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 15 KB of double-precision data. While porting to each platform, care was taken to avoid any design which was specific to RADM2. Thus, our hand-tuned multicore RADM2 implementations form templates the KPPA code generator can reuse when targeting these architectures.

Input data to the RADM2 solver was written to files from WRF-Chem 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 unmodified Fortran source files for the RADM2 chemical kinetics solver (generated by KPP during WRF-Chem compilation), along

TABLE 2
RADM2 Timing (Seconds) of the Serial Fortran Chemical Kernel Executed for One Time Step on a Single Core of an Intel Quad-Core Xeon 5410

	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

with a number of KPP-generated tables of indices and coefficients used by the solver, were isolated into a stand-alone program that reads in the input files and invokes the solver. The timings and output from the original solver running under the stand-alone driver for one time step comprised the baseline performance benchmark.

For the accelerated architectures, it was necessary to translate 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. Once the C model was developed, Fortran code for the CBEA could be generated automatically following the same design.

Table 2 shows the baseline serial performance in seconds based on 10 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 Fig. 3 and includes "LU Decomp.," "LU Solve," "ODE Function," and "ODE Jacobian," 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 3 shows the performance in seconds of the RADM2 multicore ports. The labels are identical to those in Table 2. 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 benchmark results are discussed in Section 6.2.

6.1.1 Intel Quad-Core Xeon with OpenMP

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 global pointers to global data structures. These pointers were specified as `threadprivate` to prevent unwanted data sharing between threads without duplicating the large global data structures. Other variables could simply be declared in the `private` or `shared` blocks of the `parallel` constructs.

6.1.2 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 ("a" in Table 3) implemented the entire Rosenbrock mechanism (Fig. 3) as a single kernel. This presented some difficulties and performance was disappointing (Table 3). The amount of storage per grid cell precluded using the fast but small (16 KB per multiprocessor) 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 upward of one 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

TABLE 3
RADM2 Timings (Seconds) of the Multicore Kernels

		QS22		QS20		PS3		Xeon		Tesla C1060	
		16 SPEs		16 SPEs		6 SPEs[a]		8 Cores		(a)[b]	(b)
Single	Rosenbrock	2.070	0.783	1.652	0.787	7.003	2.012	9.083	2.925	9.479	3.285
	LU Decomp.	1.645	0.509	1.065	0.509	2.633	1.266	3.9659	1.358	****	1.022
	LU Solve	0.199	0.093	0.198	0.093	1.597	0.252	1.288	0.420	****	0.928
	ODE Fun.	0.040	0.017	0.040	0.017	0.496	0.049	1.061	0.312	****	0.333
	ODE Jac.	0.036	0.015	0.036	0.015	0.409	0.043	1.129	0.316	****	0.392
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

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.

Each time shown is the minimum over several successive runs. Category labels are explained in detail at the end of Section 6.1.

the CPU. The lower levels of the Rosenbrock call tree 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. This design was also easier to debug and benchmark since the GPU code was spread over many smaller kernels with control returning frequently to the CPU, and it compiled considerably faster. Most importantly, it limited the impact of resource bottlenecks to only those affected kernels. Performance-critical parameters such as the size of thread blocks and shared memory allocation were tuned kernel-by-kernel without subjecting the entire solver to worst case limits.

The principal disadvantage of moving time and error control logic to the CPU is that all cells are forced to use the same minimum time step and iterate the maximum number of times, even though only a few cells required that many to converge. For the benchmark workloads, 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 "(b)" results in Table 3 were for the multikernel 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, resulting in a reduced wall-clock time.

6.1.3 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 [31]. Recognizing this, we chose a master-worker approach for the CBEA port. The PPU, with full access to main memory, is the master. It prepares grid data for the SPUs which process them and return them to the PPU. In order to comply with size and alignment restrictions (see Section 5.3), the PPU buffers the incontiguous WRF-Chem data into an aligned and padded array so that it can be safely accessed by the MFC.

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, the PPU interleaves the data of four (or two) grid points into an array of 128-bit vectors, 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 Fig. 3, the integrator iteratively refines the Newton step size h until the error norm is within acceptable limits. This will cause an intravector 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.

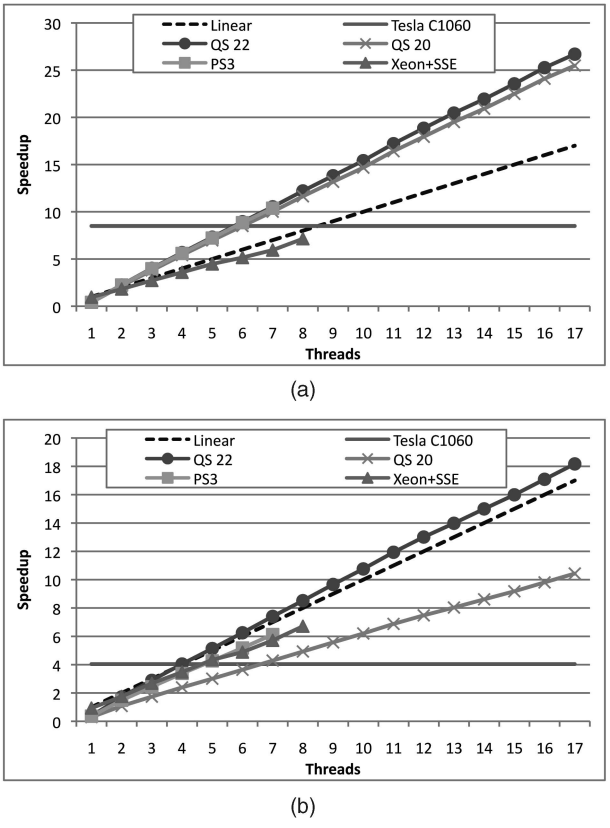

Fig. 4. RADM2 speedup as compared to the original serial code. A line indicates the maximum GPU speedup since the thread models of the GPU and conventional architectures are not directly comparable. (a) Single precision. (b) Double precision.

6.2 RADM2 Performance Analysis and Discussion

Table 3 shows the performance of all the parallel implementations. The benchmark systems are described in Section 5.4. Plots of fine-grid performance and further discussion can be found in [38].

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 Fig. 4, this port achieved nearly linear speedup over eight cores.

The CBEA implementation achieves the best performance. On a fine-grain grid, two PowerXCell 8i chipsets are $11.5\times$ faster in double precision than the serial implementation, and $41.1\times$ faster in single precision. Coarse grained single-precision grids see a speedup of $26.6\times$. The CBEA's explicitly managed memory hierarchy and fast on-chip memory provide this performance. Up to 40 RADM2 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.

The NVIDIA CUDA implementation was 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. In our case, the GPU required the most effort to achieve acceptable performance. On a single-precision coarse grid, this implementation achieves an $8.5\times$ 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.

7 Multicore Code Generation

KPPA's code generation module generalizes the hand-tuned RADM2 kernels developed in Section 6. Initially, we sought to extend KPP to support multicore architectures; however, KPP's design does not facilitate such an approach. KPP is a highly tuned procedural C code that uses a complex combination of function pointers and conditional statements to translate chemical kinetics concepts into a specific language. While efficient, this design is not extensible. Adding new features to KPP requires careful examination of many hundreds of lines of complex source code. Targeting multiple architectures in multiple languages creates a 2D design space (Table 1), increasing the complexity of the KPP source code to unmaintainable levels. Furthermore, the design space will only continue to grow as future architectures are developed. Hence, a completely new code generation module was developed from scratch.

KPPA's code generation component is object-oriented to enable extensibility in multiple design dimensions. Abstract base classes for Language objects and Architecture objects define the interfaces a Model object uses to produce an optimized chemical code. The Bridge Pattern [39] connects an Architecture to a Language, facilitating future architecture or language implementations. Adding a new language or architecture is often as simple as inheriting the abstract class and implementing the appropriate member functions and template files.

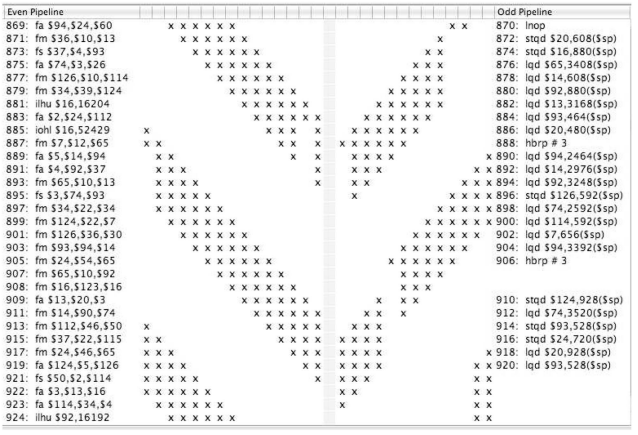

Fig. 5. SPU pipeline status while calculating the ODE function of the SAPRCNOV mechanism as shown by the IBM Assembly Visualizer for CBEA. No stalls are observed for all 2,778 cycles. The pattern repeats with minor variations for 2,116 of 2,778 cycles.

7.1 Language-Specific Code Generation

KPPA generates code in two ways: complete function generation using lexical trees, and template file specification. Complete function generation builds a language-independent expression tree describing a sparse matrix/matrix or matrix/vector operation. For example, the aggregate ODE function of the mechanism is calculated by multiplying the left-side stoichiometric matrix by the concentration vector, and then adding the result to elements of the stoichiometric matrix. KPPA performs these operations symbolically at code generation time, using the matrix formed by the analytical component and a symbolic vector, which will be calculated at runtime. The result is an expression tree of language-independent arithmetic operations and assignments, equivalent to a rolled-loop sparse matrix/vector operation, but in completely unrolled form. The language-independent lexical tree is translated to a specific language by an instance of the abstract Language class. Each concrete Language subclass defines how assignments, arithmetic operations, and type casts are performed in a specific language.

KPPA uses its knowledge of the target architecture to generate highly efficient function code. Language-specific vector types are preferred when available, branches are avoided on all architectures, and parts of the function can be rolled into a tight loop if KPPA determines that on-chip memory is a premium. An analysis of four KPPA-generated ODE functions and ODE Jacobians targeting the CBEA showed that, on average, both SPU pipelines remain full for over 80 percent of the function implementation. Pipeline stalls account for less than 1 percent of the cycles required to calculate the function. For example, in the SAPRCNOV mechanism on CBEA, there are only 20 stalls in the 2,989 cycles required by the ODE function (0.66 percent), and only 24 stalls in the 5,490 cycles required for the ODE Jacobian (0.43 percent). Clever use of the `volatile` qualifier can reduce the number of cycles to 2,778 and eliminate stalls entirely.

Fig. 5 shows the SPU pipelines during the SAPRCNOV ODE function execution as given by the IBM Assembly Visualizer for CBEA. This tool displays the even/odd SPU pipelines in the center, with an "X" marking a full pipeline stage. Stalls are shown as red blocks (no stalls are visible in

TABLE 4
Parameterizations of Three Multicore Platforms in Single and Double (DP) Precision

		Xeon 5400	Tesla C1060	CBEA
SP	Architecture Name	OpenMP	CUDA	CBEA
	Instruction Cardinality	1	32	4
	Integrator Cardinality	1	∞	4
	Scratch Size	0KB	16KB	256KB
DP	Architecture Name	OpenMP	CUDA	CBEA
	Instruction Cardinality	1	32	2
	Integrator Cardinality	1	∞	2
	Scratch Size	0KB	16KB	256KB

Fig. 5). When the pipelines achieve 100 percent utilization, smooth chevrons of "X"s are formed, similar to the patterns shown. Code of this caliber often requires meticulous hand optimization, but KPPA is able to generate this code automatically in seconds.

When template file specification is used, source code templates written in the desired language are copied from a library and then "filled in" with code appropriate to the chemical mechanism being generated and the target platform. This is the method used to generate the outer loop of the Rosenbrock integrator, BLAS wrapper functions, and other boilerplate methods. The platform-specific codes from our RADM2 implementations, such as the vectorized Rosenbrock solver and its associated BLAS wrapper functions, were easily converted to templates and added to the KPPA library. Boilerplate code for platform-specific multicore communication and synchronization was also imported from the RADM2 implementations.

Template file specification enables the rapid reuse of pretested and debugged code, and it allows easy generation of architecture-specific code (such as the producer-consumer integrator for CBEA) without making KPPA overly complex. However, it necessitates a copy of every template file translated in every language. In practice, this has not hindered the addition of new languages to KPPA, since most languages are in some way descended from Fortran and/or C. This fact, combined with the small size of the template files, makes by-hand translation of a C or Fortran template to the new language straightforward. Another alternative is to generate code from the original C or Fortran templates and then link against language-specific codes, if supported.

7.2 Multicore Support

General computing approaches require a fairly sophisticated machine parameterization, perhaps describing the memory hierarchy of the machine, the processor topology, or other hardware details [40]. Fortunately, because KPPA targets a specific problem domain, only a few architectural parameters are required, and because KPPA has domain-specific knowledge, it is able to generate highly optimized platform-specific code.

Our experiences in Section 6 show that only four parameters are required to generate a chemical mechanism with multilayered heterogeneous parallelism (see Table 4). The target *architecture name* must be specified. The architecture name is passed as a string to the code generator and is used to locate the correct code template files and determine the scalar word size of the architecture. The architecture's *instruction cardinality* must be known. Instruction cardinality specifies how many (possibly heterogeneous) instructions of a given precision can be executed per processor per cycle. For the CBEA, this is dictated by the size of a vector instruction, i.e., two in double precision and four in single. CUDA-enabled devices execute instructions in fixed sizes called *warps*. A single-precision warp is 32 threads on the GTX 200 architecture, making its instruction cardinality 32. In double precision, one DP unit is shared between 32 threads, so the cardinality is 1; however, this architectural detail is hidden from the CUDA developer. Therefore, we still consider the cardinality to be 32 in this case. Scalar cores have an instruction cardinality of 1. Keeping the instruction cardinality independent of the architecture name allows for improvements in existing architectures.

An architecture's *integrator cardinality* is closely related to its instruction cardinality. It is the optimal number of grid cells that are processed simultaneously by one instance of the integrator. In theory, integrator cardinality could be arbitrarily large for any platform, but, in practice, the optimal number of cells per integrator is dictated by the instruction cardinality. In single precision, the CBEA is most efficient when processing four cells per integrator instance. On the other hand, the sophisticated thread scheduling hardware in CUDA devices encourages a very large integrator cardinality to hide latency to device memory, so CUDA's integrator cardinality is limited only by the size of device memory.

The *scratch size* is the amount of (usually on-chip) memory which can be explicitly controlled by a single accelerator thread. KPPA uses this information to generate multibuffering and prefetching code. For an SPE thread on the CBEA, the scratch size is equivalent to the size of SPE local storage. CUDA devices share 16 KB of on-chip memory with every thread in a block (up to 512 threads for current architectures). Traditional architectures have an implicitly managed memory hierarchy, so there is no explicitly controlled cache.

In addition to the architecture parameterization, KPPA must be aware of the language features specific to the target architecture. Compilers targeting the CBEA have 128-bit vector types as first-level language constructs, and CUDA introduces several new language features to simplify GPU programming. In KPPA, a language is encapsulated in a C++ class that exposes methods for host- and device-side function declaration, variable manipulation, and other foundational operations. Support for platform-specific language features is achieved by inheriting a language class (i.e., C for CUDA) and then overriding member functions as appropriate.

8 KPPA Mechanism Performance

We used KPPA to explore the performance of an automatically generated RADM2 kernel and three other popular kernels: SAPRC'99, SMALL_STRATO, and SAPRCNOV. The benchmark systems are described in Section 5.4. The kernels were applied on the coarse domain grid from Section 6.1 with a KPPA-generated Rosenbrock integrator with six stages for 24 simulation hours. Except for SAPRCNOV, all mechanisms are calculated in single precision.

To test KPPA's extensibility, we used it to generate SSE-enhanced versions of state-of-the-art serial and OpenMP codes for each of these kernels. To generate SSE-enhanced

code, we specified an instruction and integrator cardinality of 4 for single precision (2 for double), described SSE intrinsics to the language generation module, and provided vector math functions from the Intel Math Kernel Library [41] in place of the scalar math library functions. In total, this effort took less than one week. Except for SAPRCNOV, the SSE-enhanced codes are approximately $2\times$ faster than the state-of-the-art serial codes. SAPRCNOV requires double-precision calculations, which limited the speedup to only $1.2\times$. It may be possible to improve on these speedups by using more advanced SSE intrinsics.

The Community Multiscale Air Quality Model (CMAQ) [2] is an influential model with a large user base including government agencies responsible for air quality policy, and leading atmospheric research labs. CMAQ uses a SAPRC mechanism [21] with 79 species in a network of 211 reactions to calculate photochemical smog. The performance of the KPPA-generated SAPRC mechanism from CMAQ is shown in Fig. 6a. The CBEA implementation of SAPRC '99 achieves the highest speedup of $38.6\times$ when compared to the state-of-the-art production code, or $19.3\times$ when compared with the SSE-enhanced serial implementation. However, this mechanism's large code size pushes the architecture's limits. Only four grid cells can be cached in SPU local storage, and its extreme photosensitivity results in an exceptionally large number of exponentiation operations during peak sunlight hours. The GPU code achieves a maximum speedup of only $13.7\times$ ($6.8\times$ compared to SSE) due to high memory latencies and a shortage of on-chip fast shared memory, just as described in Section 6.2.

SMALL_STRATO (Fig. 6b) is a stratospheric mechanism with seven species in a network of 10 reactions. It represents both small mechanisms and mechanisms with a separable Jacobian permitting domain decomposition within the mechanism itself. Its program text and over 100 grid points can be held in a single SPE's local store, resulting in a maximum speedup of $40.7\times$ on the CBEA ($20.5\times$ compared to the SSE-enhanced serial implementation). On the GPU, even though this mechanism is exceedingly small, it still cannot fit into the 64 KB shared memory since a thread block size of at least 128 is recommended, leaving only 512 bytes of shared memory per thread. This results in a maximum speedup of $11.2\times$ ($4.4\times$ compared to SSE). NVIDIA's new Fermi architecture [42] with larger on-chip cache may produce better results.

SAPRCNOV is a particularly complex mechanism with 93 species in a network of 235 reactions. Its stiffness necessitates a double-precision solver, and its size makes it an excellent stress test for on-chip memories. The compiled program code alone is over 225 KB large, and each grid cell comprises over 19 KB of data. Fig. 6c shows this mechanism's performance. The Quad-Core Xeon system, with its large L2 cache is not notably affected; however, the performance of the CBEA is drastically reduced. In order to accommodate the large program text, the ODE Function, ODE Jacobian, and all BLAS functions are overlayed in the SPE local storage by the compiler. When any of these functions are called, the SPE thread pauses and downloads the program text from main memory to an area of local storage shared by the overlayed functions. LU solve and the ODE function are called several times per integrator iteration, which multiplies the pause-and-swap overhead. Even with overlays, there is only enough room for two grid cells in local store, so triple-buffering is not an option. The CBEA achieves only $10.4\times$ speedup ($8.7\times$ compared to the SSE-enhanced serial implementation). The GPU achieves the highest speedup of $13.3\times$ ($11.1\times$ compared to SSE).

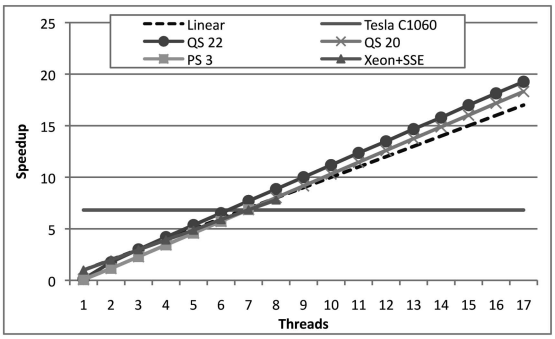

Fig. 6. Speedup of KPPA-generated chemical kernels as compared to SSE-enhanced serial code. A line indicates the maximum GPU speedup since the thread models of the GPU and conventional architectures are not directly comparable. Speedups in (a), (b), and (d) approximately double when compared to state-of-the-art serial codes. (a) SAPRC '99 from CMAQ. (b) SMALL_STRATO from KPP. (c) SAPRCNOV from STEM. (d) RADM2 from WRF-Chem.

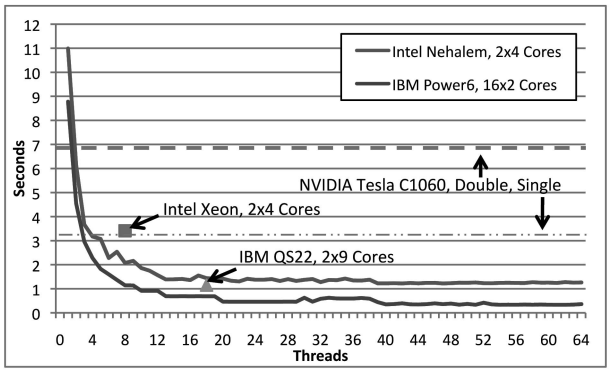

Fig. 7. Wall clock OpenMP performance of the RADM2 chemical mechanism on two cutting edge homogeneous multicore chipsets compared with emerging multicore architectures.

The performance of the KPPA-generated RADM2 kernel was similar to the performance of the hand-tuned code discussed in Section 6. This is not surprising, since the hand-tuned code is the template KPPA uses in template instantiation, and the hand-tuned RADM2 code was developed from the KPP-generated serial code.

9 Future Work

Future work will concentrate on alternate numerical methods, frameworks for calling KPPA-generated mechanisms remotely, and extending KPPA to other multicore platforms. The numerical methods presented have strong data dependencies and cannot adequately use fast on-chip memories. Other methods, such as Quasi-Steady-State-Approximation [43], may be appropriate if a high level of accuracy is not required. Installing multicore chipsets as accelerators in traditional clusters is viable [9]; however, not every cluster can be easily upgraded. Clusters of ASIC nodes, such as IBM BlueGene [44], do not support accelerator cards, or thermal dissipation and power issues may prohibit additional hardware in the installation rack. These systems can still benefit from accelerated multicore chipsets by offloading computationally-intense kernels to remote systems. If a kernel is $20\times$ or $30\times$ faster on a specific system, as demonstrated in this work, then the overhead of a remote procedure call may be acceptable, even for large data sets. We are investigating existing solutions, such as IBM Dynamic Application Virtualization [45], and will extend KPPA to generate model interfaces for remote kernels.

The recent acceptance of the OpenCL standard [46] makes OpenCL support an obvious next step for KPPA. Chemical codes targeting OpenCL will be more portable than CUDA- or CBEA-specific mechanisms, and may be supported on as-yet undeveloped architectures. A study comparing KPPA to other tools, such as Intel Ct [47] (formally RapidMind), is also a future work.

The results and analysis presented here are a snapshot in time; for example, newer versions of homogenous multicore processors (e.g., Intel's i7, IBM's Power6, and others) are already showing significant improvement in speed and multicore efficiency over their previous generations (see Fig. 7). Similarly, NVIDIA is moving forward with their 300-series Fermi GPUs. Currently, none has a clear advantage. However, given that chemical kinetics is a challenging benchmark in terms of sheer size and complexity per grid cell, performance and cost performance (both monetary and electrical efficiency) of new homogenous and heterogeneous multicore architectures will be important gating factors for climate chemistry, air quality, wildfire, and other earth science simulation in the coming decade.

10 Conclusion

We have presented KPPA, a general analysis tool and code generator for serial, homogeneous multicore, and heterogeneous multicore architectures. KPPA generates time-stepping codes for general chemical reaction networks in several languages, and is well-suited for use in atmospheric modeling. Optimized ports of four chemical kinetics kernels (RADM2 from WRF-Chem, SAPRC from CMAQ, SAPRCNOV, and SMALL_STRATTO) for three multicore platforms (NVIDIA CUDA, the Cell Broadband Engine Architecture (CBEA), and OpenMP)) were presented.

A detailed performance analysis for each platform was given. The CBEA achieves the best performance due to its fast, explicitly managed on-chip memory. Compared to the state-of-the-art serial implementation, RADM2[2] from WRF-Chem achieves a maximum speedup of $41.1\times$, SAPRC '99 from CMAQ achieves $38.6\times$ speedup, SAPRCNOV achieves $8.2\times$ speedup, and SMALL_STRATTO achieves $28.1\times$ speedup. OpenMP implementations achieve almost linear speedup for up to eight cores. Additional optimizations, such as SSE and cache manipulation, are in development. The GPU's performance is severely hampered by the limited amount of on-chip memory. The CBEA's performance is limited when the size of the code of a chemical kernel exceeds 200 KB; however, most atmospheric chemical kernels will fit within this envelope.

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 US National Science Foundation (NSF), 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.

References

[1] 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," *Atmospheric Environment*, vol. 39, no. 37, pp. 6957-6975, 2005.

[2] D.W. Byun and J.K.S. Ching, "Science Algorithms of the EPA Models-3 Community Multiscale Air Quality (CMAQ) Modeling System," Technical Report EPA/600/R-99/030, US Environment Protection Agency, 1999.

2. Additional information on the multicore RADM2 kernel with updated results and codes is maintained at http://www.mmm.ucar.edu/wrf/WG2/GPU.

[3] G.R. Carmichael, L.K. Peters, and T. Kitada, "A Second Generation Model for Regional Scale Transport/Chemistry/Deposition." *Atmospheric Environment,* vol. 20, no. 1, pp. 173-188, 1986.

[4] I. Bey, D.J. Jacob, R.M. Yantosca, J.A. Logan, B.D. Field, A.M. Fiore, Q. Li, H.Y. Liu, A.M. Mickley, and A.M. Schultz, "Global Modeling of Tropospheric Chemistry with Assimilated Meteorology: Model Description and Evaluation," *J. Geophysical Research,* vol. 106, no. D19, pp. 23073-23095, 2001.

[5] V. Damian, A. Sandu, M. Damian, F. Potra, and G.R. Carmichael, "The Kinetic Preprocessor KPP—a Software Environment for Solving Chemical Kinetics," *Computers and Chemical Eng.,* vol. 26, no. 1, pp. 1567-1579, 2002.

[6] N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha, "LU-GPU: Algorithms for Dense Linear Systems on Graphics Hardware," *Proc. ACM/IEEE Conf. Supercomputing (SC '05),* 2005.

[7] V. Volkov and J. Demmel, "LU, QR and Cholesky Factorizations Using Vector Capabilities of GPUs," Technical Report UCB/EECS-2008-49, EECS Dept., Univ. of California, Berkeley, http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.html, May 2008.

[8] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," *Proc. ACM/IEEE Conf. Supercomputing (SC '07),* pp. 1-12, 2007.

[9] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU Cluster for High Performance Computing," *Proc. ACM/IEEE Conf. Supercomputing (SC '04),* p. 47, 2004.

[10] K.S. Perumalla, "Discrete-Event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs)," *Proc. 20th Int'l Workshop Principles of Advanced and Distributed Simulation (PADS '06),* pp. 74-81, 2006.

[11] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The Potential of the Cell Processor for Scientific Computing," *Proc. Third Int'l Conf. Computing Frontiers (CF '06),* pp. 9-20, 2006.

[12] J. Dongarra, S. Moore, G. Peterson, S. Tomov, J. Allred, and V. Natoli, "Exploring New Architectures in Accelerating CFD for Air Force Applications," *Proc. High Performance Computing Modernization Program (HPCMP) Users Group Conf. '08,* pp. 472-478, July 2008.

[13] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," *Proc. ACM SIGGRAPH '05 Courses,* 2005.

[14] J. Krüger and R. Westermann, "Linear Algebra Operators for GPU Implementation of Numerical Algorithms," *Proc. ACM SIGGRAPH '03,* pp. 908-916, 2003.

[15] M. Baboulin, J. Demmel, J. Dongarra, S. Tomov, and V. Volkov, "Enhancing the Performance of Dense Linear Algebra Solvers on GPUs [in the MAGMA]," Poster at Supercomputing '08, Nov. 2008.

[16] K.Z. Ibrahim and F. Bodin, "Implementing Wilson-Dirac Operator on the Cell Broadband Engine," *Proc. 22nd Ann. Int'l Conf. Supercomputing (ICS '08),* pp. 4-14, 2008.

[17] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick, "Self Adapting Linear Algebra Algorithms and Software," *Proc. IEEE,* special issue on program generation, optimization, and adaptation, vol. 93, no. 2, pp. 293-312, June 2005.

[18] M. Frigo and S. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *Proc. IEEE Int'l Conf. Acoustics, Speech and Signal Processing,* vol. 3, pp. 1381-1384, 1998.

[19] Y. Li, J. Dongarra, and S. Tomov, "A Note on Auto-Tuning GEMM for GPUs," *Lecture Notes in Computer Science,* pp. 884-892, Springer, May 2009.

[20] 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. Geophysical Research,* vol. 94, no. D10, pp. 12925-12956, 1989.

[21] W.P.L. Carter, "A Detailed Mechanism for the Gas-Phase Atmospheric Reactions of Organic Compounds," *Atmospheric Environment,* vol. 24A, pp. 481-518, 1990.

[22] M.Z. Jacobson and R. Turco, "SMVGEAR: A Sparse-Matrix, Vectorized Gear Code for Atmospheric Models," *Atmospheric Environment,* vol. 28, pp. 273-284, 1994.

[23] 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," *Geoscientific Model Development,* vol. 2, pp. 89-96, 2009.

[24] J.D. Lambert, *Numerical Methods for Ordinary Differential Systems.* John Wiley and Sons, 1991.

[25] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II: Stiff and Differential Algebraic Problems,* second ed., vol. 14. Springer-Verlag, 1996.

[26] "Platform Brief: Intel Xeon Processor 5400 Series," http://www.intel.com/cd/channel/reseller/asmo-na/eng/products/server/processors/q5400/feature/index.htm, Sept. 2009.

[27] 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," *Proc. Eighth Int'l Conf. Computational Science (ICCS '08),* pp. 417-426, June 2008.

[28] "Technical Brief: NVIDIA GeForce GTX 200 GPU Architectural Overview,"Technical Report TB-04044-001_v01, NVIDIA Corporation, May 2008.

[29] B. Himawan and M. Vachharajani, "Deconstructing Hardware Usage for General Purpose Computation on GPUs," *Proc. Fith Ann. Workshop Duplicating, Deconstructing, and Debunking (in conjunction with ISCA-33),* 2006.

[30] T. Chen, R. Raghavan, J. Dale, and E. Iwata, "Cell Broadband Engine Architecture and Its First Implementation," *IBM developerWorks,* June 2006.

[31] B. Flachs et al., "The Microarchitecture of the Synergistic Processor for a Cell Processor," *IEEE J. Solid State Circuits,* vol. 41, no. 1, pp. 63-70, Jan. 2006.

[32] "PowerXCell 8i Processor Technology Brief," http://www-03.ibm.com/technology/cell/, Aug. 2008.

[33] 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," *Proc. ACM/IEEE Conf. Supercomputing (SC '08),* pp. 1-11, 2008.

[34] "The Green500 list," http://www.green500.org/, Nov. 2008.

[35] 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. Geophysical Research,* vol. 95, pp. 16343-16367, 1990.

[36] 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, Nat'l Acid Precipitation Assessment Program, 1989.

[37] P. Middleton, W. Stockwell, and W. Carter, "Aggregation and Analysis of Volatile Organic Compound Emissions for Regional Modeling," *Atmospheric Environment,* vol. 24A, pp. 1107-1133, 1990.

[38] J.C. Linford, J. Michalakes, M. Vachharijani, and A. Sandu, "Multi-Core Acceleration of Chemical Kinetics for Modeling and Simulation," *Proc. ACM/IEEE Conf. Supercomputing (SC '09),* Nov. 2009.

[39] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[40] K. Fatahalian, T.J. Knight, M. Houston, M. Erez, D.R. Horn, L. Leem, J.Y. Park, M. Ren, A. Aiken, W.J. Dally, and P. Hanrahan, "Sequoia: Programming the Memory Hierarchy," *Proc. ACM/IEEE Conf. Supercomputing (SC),* 2006.

[41] "Intel Math Kernel Library," Reference Manual 630813-031US, Intel Corporation, Mar. 2009.

[42] "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," Whitepaper v1.1, NVIDIA Corporation, 2009.

[43] L.O. Jay, A. Sandu, F.A. Potra, and G.R. Carmichael, "Improved Quasi-Steady-State-Approximation Methods for Atmospheric Chemistry Integration," *SIAM J. Scientific Computing,* vol. 18, no. 1, pp. 182-202, 1997.

[44] The IBM BlueGene Team, "An Overview of the BlueGene/La Supercomputer," *Proc. ACM/IEEE Conf. Supercomputing (SC '02),* Nov. 2002.

[45] M. Purcell, O. Callanan, and D. Gregg, "Streamlining Offload Computing to High Performance Architectures," *Proc. Int'l Conf. Computational Science—ICCS '09,* G. Allen, J. Nabrzyski, E. Seidel, G.D. van Albada, J. Dongarra, and P.M. Sloot, eds., pp. 974-983. Springer, 2009.

[46] A. Munshi, *The OpenCL Specification Version 1.0 Rev 43,* Khronos OpenCL Working Group, May 2009.

[47] A. Ghuloum, T. Smith, G. Wu, X. Zhou, J. Fang, P. Guo, B. So, M. Rajagopalan, Y. Chen, and B. Chen, "Future-Proof Data Parallel Algorithms and Software on Intel Multi-Core Architecture," *Intel Technology J.,* Nov. 2007.

John C. Linford received the BS degree in 2005 in computer science and mathematics from Weber State University, Ogden, Utah, and the PhD degree in computer science in 2010 from Virginia Polytechnic Institute and State University, Blacksburg, Virginia. He is a High Performance Computing Modernization NDSEG fellow funded by the US Department of Defense, and a US National Science Foundation (NSF) CESRI fellow. He is a member of the Virtual Institute for High Productivity Supercomputing, Forschungszentrum Jülich. His current research interests include accelerated computing architectures, signal processing, atmospheric modeling, and code generation methods.

John Michalakes received the MS degree in computer science in 1988 from Kent State University, Kent, Ohio. He is currently working toward the PhD degree in computer science at the University of Colorado at Boulder, under the guidance of Dr. Manish Vachharajani. He is a senior software engineer at the National Center for Atmospheric Research. His current research interests include programmability, performance, and scaling of atmospheric and related applications on HPC systems.

Manish Vachharajani received the BS degree in 1998 in electrical and computer engineering from Rutgers, The State University of New Jersey, and the PhD degree in electrical engineering in 2004 from Princeton University, New Jersey. He is an assistant professor of electrical computer and energy engineering at the University of Colorado at Boulder. His current research interests include parallel systems, work in languages, compilers, operating systems, and hardware design.

Adrian Sandu received the engineer diploma in electrical engineering with specialty in control systems in 1990 from Technical University Bucharest, Romania, and the PhD degree in 1997 in applied mathematical and computational sciences from The University of Iowa. He is an associate professor of computer science and director of the Computational Science Laboratory, Virginia Polytechnic Institute and State University, Blacksburg, Virginia. His current research interests include numerical methods, high-performance computing, sensitivity analysis, and inverse modeling.