Accelerating Atmospheric Modeling Through Emerging Multi-core Technologies

John Christian Linford

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Adrian Sandu, Chair
Dimitrios S. Nikolopoulos
Linsey C. Marr
Calvin J. Ribbens
Paul E. Plassmann

May 5, 2010
Blacksburg, Virginia

Keywords: multi-core architectures, atmospheric modeling, KPPA, NVIDIA CUDA, Cell Broadband Engine Architecture
Copyright 2010, John C. Linford

Accelerating Atmospheric Modeling Through Emerging Multi-core Technologies

John Christian Linford

(ABSTRACT)

The new generations of multi-core chipset architectures achieve unprecedented levels of computational power while respecting physical and economical constraints. The cost of this power is bewildering program complexity. Atmospheric modeling is a grand-challenge problem that could make good use of these architectures if they were more accessible to the average programmer. To that end, software tools and programming methodologies that greatly simplify the acceleration of atmospheric modeling and simulation with emerging multi-core technologies are developed. A general model is developed to simulate atmospheric chemical transport and atmospheric chemical kinetics. The Cell Broadband Engine Architecture (CBEA), General Purpose Graphics Processing Units (GPGPUs), and homogeneous multi-core processors (e.g. Intel Quad-core Xeon) are introduced. These architectures are used in case studies of transport modeling and kinetics modeling and demonstrate per-kernel speedups as high as 40×. A general analysis and code generation tool for chemical kinetics called "KPPA" is developed. KPPA generates highly tuned C, Fortran, or Matlab code that uses every layer of heterogeneous parallelism in the CBEA, GPGPU, and homogeneous multi-core architectures. A scalable method for simulating chemical transport is also developed. The Weather Research and Forecasting Model with Chemistry (WRF-Chem) is accelerated with these methods with good results: real forecasts of air quality are generated for the Eastern United States 65% faster than the state-of-the-art models.

This research was made with Government support under and awarded by DoD, Air Force Office of Scientific Research, National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFR 168a. Summer work in Jülich was supported by the Central Europe Summer Research Institute fellowship.

Dedication

To all the friends and family members who cheered me on and up. Particularly to my good friends and "adopted parents" Andrew and Denyse Sanderson. Without their generous support, kindness, and cups of tea this would not have been possible. And to my fiancée Katherine Wooten, who ran with me and fed me.

Acknowledgments

This work was made possible by a National Defense Science and Engineering Graduate Fellowship provided by the United States Department of Defense and managed by the High Performance Computing Modernization Program. It was also supported by a Central European Summer Research Institute Fellowship funded by the National Science Foundation and the generous financial support of Dr. Adrian Sandu and Dr. Linsey Marr of Virginia Tech.

I would like to thank several people for a fruitful collaboration during this research. John Michalakes of the National Center for Atmospheric Research is the senior architect of WRF and made significant contributions to the GPGPU-accelerated atmospheric chemical kinetics described in this work, including the first implementation of the RADM2 chemical kinetics mechanism on GPGPU. Manish Vachharajani of the University of Colorado at Boulder offered guidance and advisement to John and I during the development of the accelerated RADM2 chemical kernels. Georg Grell and Steve Peckham of the NOAA Earth Systems Research Laboratory offered advice and assistance with WRF-Chem. Paulius Micikevicius and Gregory Ruetsch of NVIDIA Corporation provided important insights into the NVIDIA GTX 280 and NVIDIA Fermi architectures. Prof. Dr. Felix Wolf of RWTH Aachen and the Jülich Supercomputing Centre (JSC) advised me during my visits to JSC and supported me financially in 2008. Dr. Daniel Becker, Dr. Markus Geimer, and Marc-André Hermanns of JSC also advised me at JSC. Willi Homberg of JSC supported me on the JUICEnext QS22 cluster. Dr. Greg Newby of the Arctic Region Supercomputing Center supported me on the Quasar QS22 cluster. I also acknowledge the 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.

Contents

Dedication **iii**

Acknowledgments **iv**

Contents **v**

List of Figures **viii**

List of Tables **x**

1 Introduction **1**

2 Multi-Core Architectures **3**

2.1 Homogeneous Multi-core Chipsets 4

2.2 The Cell Broadband Engine Architecture (CBEA) 5

2.2.1 The Element Interconnect Bus (EIB) 7

2.2.2 Programming Considerations 7

2.2.3 Case Studies and Applications 8

2.3 General Purpose Graphics Processing Units 9

2.3.1 NVIDIA CUDA 10

2.3.2 ATI Stream SDK 11

2.3.3 Case Studies and Applications 12

2.4 Other chipset designs 13

2.5 Multi-core Languages, Tools and Methodologies 14

Chapter Summary 15

3 Atmospheric Modeling **16**

3.1 A Comprehensive Chemistry Transport Model . . . 17

3.2 Three State-of-the-art Atmospheric Models . . . 18

3.2.1 WRF and WRF-Chem . . . 19

3.2.2 GEOS-Chem . . . 20

3.2.3 CMAQ . . . 20

Chapter Summary . . . 21

4 Multi-core Simulation of Chemical Kinetics **23**

4.1 The Chemical Kinetics Model . . . 24

4.1.1 Example Chemical Mechanisms . . . 25

4.2 Solving the Chemical System . . . 25

4.2.1 Implicit Rosenbrock Time Integration . . . 27

4.2.2 An Implementation of the Rosenbrock Integrator . . . 28

4.3 RADM2 on Three Multi-Core Architectures [5] . . . 31

4.3.1 Intel Quad-Core Xeon with OpenMP . . . 36

4.3.2 NVIDIA CUDA . . . 36

4.3.3 Cell Broadband Engine Architecture . . . 38

4.4 KPPA: A Software Tool to Automatically Generate Chemical Kernels [6] . . . 40

4.4.1 Language-specific Code Generation . . . 44

4.4.2 Multi-core Support . . . 45

4.4.3 KPPA-Generated Mechanism Performance . . . 47

4.4.4 Future Work . . . 49

Chapter Summary . . . 50

5 Multi-core Simulation of Atmospheric Chemical Transport **52**

5.1 The Chemical Transport Model . . . 53

5.1.1 Boundary Conditions . . . 54

5.2 Solving the Transport Equations In Parallel [11] . . . 54

5.2.1 Implicit Crank-Nicholson Methods . . . 56

5.2.2 Explicit Runge-Kutta-Chebyshev Methods . . . 56

5.3 FIXEDGRID . . . 60
5.3.1 Chemical Transport in FIXEDGRID . . . 61
5.4 Function Offloading for Transport [7] . . . 64
5.4.1 Naive Function Offload . . . 64
5.4.2 Improved Function Offload . . . 66
5.4.3 Scalable Incontiguous DMA . . . 67
5.4.4 Analysis and Comparison . . . 67
5.5 Vector Stream Processing for Transport [9] . . . 72
5.5.1 Streaming Vectorized Data with the CBEA . . . 73
5.5.2 The Vectorized Transport Kernel Function . . . 74
5.5.3 Analysis and Comparison . . . 74
5.6 Future Work . . . 77
Chapter Summary . . . 77

6 Multi-core Accelerated WRF-Chem **79**
6.1 Baseline . . . 81
6.2 Quasar . . . 81
6.3 Deva + PS3 . . . 83
6.4 Accelerated Deva . . . 85
Chapter Summary . . . 87

7 Conclusions **88**

Bibliography **90**

A Program Source Code **102**

B Annotated List of Figures **107**

List of Figures

2.1 Theoretical peak performance of four multi-core architectures. 4
2.2 The Cell Broadband Engine Architecture. 6
2.3 Parallel computing architecture features of the NVIDIA Tesla C1060. 10
2.4 Overview of CUDA. 11
2.5 Example Sequoia memory trees. 14

3.1 Runtime profiles of the Sulfur Transport and dEposition Model (STEM). . . 17
3.2 Runtime profile of WRF-Chem on an Intel Xeon workstation. 19
3.3 Runtime profile of serial and parallel runs of GEOS-Chem. 21

4.1 Outline of the three-stage Rosenbrock solver for chemical kinetics. 29
4.2 Accelerated RADM2 speedup as compared to the original serial code. 34
4.3 CBEA implementation of RADM2. 39
4.4 Principle KPPA components and KPPA program flow. 41
4.5 Abbreviated UML diagram of KPPA. 43
4.6 SPU pipeline status while calculating the SAPRCNOV ODE function. 45
4.7 Speedup of KPPA-generated code as compared to SSE-enhanced serial code. 48
4.8 Performance of OpenMP accelerated RADM2. 50

5.1 Data dependencies for implicit and explicit time-stepping. 55
5.2 Stability regions of Runge-Kutta-Chebyshev explicit time-stepping method. . 59
5.3 STEM speedup on System X. 60
5.4 3D discretization stencil for explicit upwind-biased advection/diffusion. . . . 62
5.5 Second order time splitting methods for 2D and 3D transport discretization. 63

5.6 FixedGrid data storage scheme for the 3D transport module. 63
5.7 Domain decomposition for two-dimensional transport. 65
5.8 2D transport module performance executing on IBM BladeCenter Q22. . . . 68
5.9 Runtime and speedup for the 2D transport module. 71
5.10 Runtime and speedup for the 3D transport module. 76

6.1 Ground-level concentrations and temperatures from the WRF-Chem case study. 80
6.2 Runtime profile of WRF-Chem executing on an Intel Xeon workstation. . . . 81
6.3 Runtime profile of WRF-Chem executing on an IBM BladeCenter QS22. . . 83
6.4 Runtime profile of WRF-Chem executing on an Intel Xeon/Sony PS3 cluster. 85
6.5 Predicted runtime profile of shared memory accelerated WRF-Chem. 86

List of Tables

4.1 RADM2 chemical kernel operation counts. 32

4.2 Runtime profile of serial RADM2 on an Intel Xeon workstation. 33

4.3 Runtime profile of accelerated RADM2 on five multi-core platforms. 35

4.4 Language/architecture combinations supported by KPPA. 42

4.5 Parameterizations of three multi-core platforms. 46

5.1 STEM execution times on System X. 59

5.2 Runtime and speedup for the 2D transport module on three CBEA systems. 70

5.3 Runtime and speedup for the 2D transport module on two homogeneous multi-core systems. 71

5.4 Runtime and speedup for the 3D transport module on two CBEA systems. . 75

5.5 Runtime and speedup for the 3D transport module on two homogeneous multi-core systems. 75

6.1 WRF-Chem execution times on an Intel Xeon workstation. 82

6.2 WRF-Chem execution times on an IBM BladeCenter QS22. 84

6.3 WRF-Chem execution times on an Intel Xeon/Sony PS3 cluster. 86

6.4 Predicted execution times of shared memory accelerated WRF-Chem. 87

Chapter 1

Introduction

Weather, air quality, and global climate change modeling have been official grand-challenge problems since the creation of the Federal High Performance Computing Program in 1987 [46]. These geophysical models solve systems of equations involving millions or billions of floating point values and can consume hours or days of computer time. They influence important government policy and thousands of daily decisions, yet the state-of-the-art models are not always accurate. Computational complexity prevents geophysical models from producing *accurate* and *timely* results for high resolutions, over long periods of time, or on global scales.

Traditional supercomputer designs cannot provide the necessary computing power because power consumption, heat dissipation, and data synchronization issues place a restrictive upper bound on the capabilities of semiconductor-based microelectronic technology. In essence, we have reached the point where the cost of up-scaling this technology is unsustainable. However, the new generations of low-power 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. Technologies such as the Cell Broadband Engine Architecture (CBEA) [74] and General Purpose Graphics Processing Units (GPGPUs) [129, 85] combine multiple processing cores, memory controllers, and other critical components in a unified physical package. This greatly increases processing power and memory throughput while respecting physical and economical bounds.

Supercomputers based on these architectures have achieved record-breaking floating point performance while maintaining unprecedented power efficiency. Roadrunner at Los Alamos National Laboratory [23], the first supercomputer to achieve a sustained petaflops, is prominent evidence of the potential of heterogeneous multi-core chipsets. According to the "Top 500" lists from November 2009 [133, 130], seven of the top-ten most power efficient supercomputers in the world are based on either CBEA or GPGPU technologies, the second fastest supercomputer in the world is based on CBEA technology, and the fifth fastest in the world

is based on ATI GPGPU technology.

The next generation of atmospheric models must exploit complex parallelism in order to tap the full potential of "emerging" multi-core chipset designs. The CBEA and GPGPUs represent a paradigm shift in circuit design, and although they offer unprecedented computing power, the cost is staggering software complexity. They can be extremely difficult to program in general, and to date, these powerful architectures have not been used effectively in atmospheric models. Successful use of these novel architectures will enable not only larger, more complex simulations, but also reduce the time-to-solution for a range of earth system applications.

This work develops methods and software tools to simplify the application of emerging multi-core technologies to atmospheric modeling. The remainder of this work is structured as follows:

Methodology The literature review in Chapter 2 finds many demonstrations of the power of emerging multi-core technologies but few examples of atmospheric models benefiting from these developments. What I call the "personality" of emerging multi-core architectures is covered here. Chapter 3 uses the basic mathematical formulation of an atmospheric model to identify and categorize critical processes. In general, a comprehensive atmospheric model consists of two general processes: chemistry and transport.

Development Multi-core methods for chemistry and transport are discussed in Chapters 4 and 5. Chemistry is covered in Chapter 4 with the development of methods for computing chemical kinetics on multi-core SIMD architectures [8], implicit time integration for chemical kinetics on various multi-core architectures [1, 5], and the development of a multi-platform multi-lingual code generation tool for chemical kinetics [6]. Chapter 5 develops an explicit time integration scheme for chemical transport [11, 3], applications of multi-core platforms to chemical transport [7, 12], and a scalable method for calculating transport on the CBEA [9, 10].

Applications Case studies of the newly-developed methods and software tools to real-world climatological and air quality models are presented in Chapter 6.

Concluding remarks are made in Chapter 7.

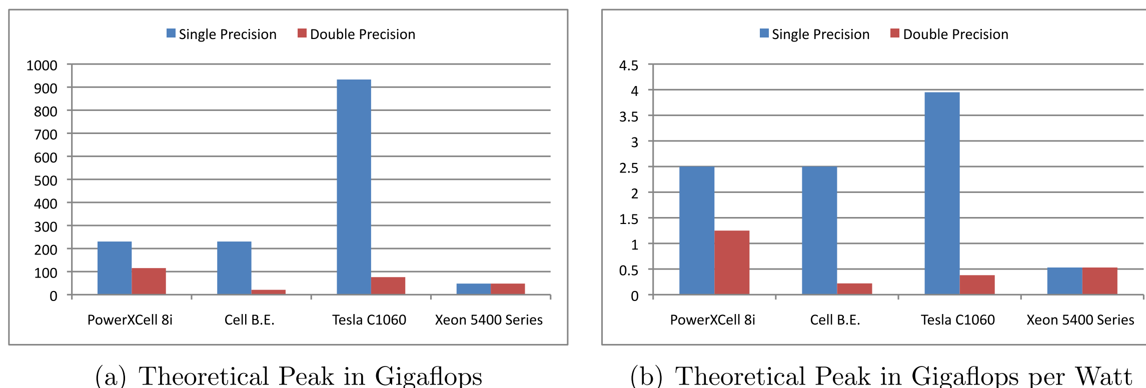
Chapter 2

Multi-Core Architectures

This chapter introduces emerging multi-core architectures and their associated systems, languages, and tools. Case studies of application performance on each architecture are also presented. Implementing atmospheric models on emerging multi-core technologies can be unusually difficult, so the existing literature tends to focus on the model sub-components, such as basic linear algebra operations [51, 117, 119], and does not comprehensively address this problem domain.

Multi-core chipsets combine two or more independent processing cores into a single integrated circuit to implement parallelism in a single physical package. This pushes against physical constraints by allowing many processes to be assigned to the same node, thus reducing the machine's physical footprint. Cores may share on-chip resources (such as a level of coherent cache) and/or have their own private resources (such as a high-speed local data store). An architecture which consistently repeats one core design for all cores is called *homogeneous*. If multiple core designs are used, the architecture is called *heterogeneous*. The defining characteristics of "emerging" multi-core technologies include massive heterogeneous parallelism, exposed or explicit communication, multiple on-chip address spaces, and an explicitly-managed memory hierarchy.

Many instances of the same core on a single chip may be economically or physically infeasible, particularly if caches are shared between cores. Heterogeneous (somtimes called *accelerated*) multi-core chipsets specify subsets of cores for particular applications for optimal use of on-chip space. For example, cores may be specialized for memory I/O or compute-intensive liner algebra. The specialized cores are called *accelerators* since they rely a traditional CPU for general computation. An accelerator's high flops and additional memory I/O efficiency hides memory latency as the chip is oversubscribed. Therefore, accelerated multi-core chipsets exhibit much stronger scalability than homogeneous multi-core chipsets while facilitating good physical scalability [60].



(a) Theoretical Peak in Gigaflops

(b) Theoretical Peak in Gigaflops per Watt

Figure 2.1: The theoretical peak performance of four multi-core architectures in gigaflops and gigaflops-per-watt.

2.1 Homogeneous Multi-core Chipsets

Homogeneous multi-core design has now supplanted single-core design in commercial servers, workstations, and laptops. The Intel Xeon 5400 Series [136] is a typical example of this design. A quad-core Xeon 5400 chip at 3GHz has a theoretical peak single-precision floating point performance of 48 gigaflops with nominal 90W dissipation. It achieves 40.5 gigaflops (0.5 gigaflops/watt) in the LINPACK benchmark [54] . Similar designs not only reduce power consumption, but can also lower communication overhead. Alam and Agarwal [19] characterized computation, communication and memory efficiencies of bio-molecular simulations on a Cray XT3 system with dual-core Opteron processors. They found that the communication overhead of using both cores in the processor simultaneously can be as low as 50% when compared to the single-core execution times.

Most homogeneous multi-core chips have between two and eight cores and support up to sixteen threads, though some like the Sun UltraSPARC T2 (codename Niagara) support up to 64 threads on eight cores [76]. Niagara is representative of the state-of-the-art in commercial multi-core technologies that target service computing. It features eight in-order cores, each capable of executing four simultaneous threads, and a shared cache. Commercial server workloads are ideal for Niagara, but the chip lacks architectural elements critical to floating-point intensive scientific workloads, such as SIMD floating-point execution and multi-core scatter-gather.

A more extreme example a homogeneous chip that support over 100 threads is the IBM Cyclops-64 (C64). This multi-core chip was developed by IBM and its collaborators for the Cyclops-64 petaflop computer system [64]. C64 consists of 80 500MHz CPUs, each with two thread units, a floating-point unit, and two SRAM memory banks of 32KB each. A 32KB instruction cache is shared among five processors. C64 seeks to be the first “supercomputer

on a chip", though it's theoretical peak performance is only 80 gigaflops. Verification testing and system software development is being done at the University of Delaware. Chen et al. [35] optimized the Fast Fourier Transform on the Cyclops-64 and achieved over four times the performance of an Intel Xeon Pentium 4 processor using only one Cyclops-64 chip. A thorough understanding of the problem and of the underlying hardware were critical to achieving good performance.

Douillet and Gao [42] developed a method for automatically generating a multi-threaded software-pipelined schedule from parallel or non-parallel imperfect loop nests written in a standard sequential language such as C or FORTRAN. Their method exhibits good scalability on the IBM Cyclops-64 multi-core architecture up to 100 cores, showing that enhanced compilers can produce efficient multi-core code, even if the programmer lacks domain-specific knowledge. However, this method is not easily transferable to heterogeneous architectures, as it assumes every core is equally applicable to every task.

2.2 The Cell Broadband Engine Architecture (CBEA)

The Cell Broadband Engine Architecture (CBEA) [74] is a heterogeneous multi-core processor architecture which has drawn considerable attention in both industry and academia. It is the result of over four years of development by the STI consortium made up of Sony, Toshiba, and IBM. For Sony, a CBEA-based chip is the central processing unit for the Sony PlayStation 3 gaming console. Toshiba uses the CBEA for media processing in its high-definition television products. IBM provides several CBEA-based server products for scientific computing applications. In general, STI developed Cell to rapidly process large amounts of parallel data. The processing capabilities of the CBEA are made possible by its heterogeneous processors and its on-chip network architecture. In spite of its low cost and low power requirements, it has archived unprecedented peak single-precision and double-precision floating point performance, making it suitable for high-performance computing. Furthermore, the CBEA standards and design are "open", which greatly simplifies research and development.

As shown in Figure 2, the main components of the CBEA are a Power Processing element (PPE), eight Synergistic Processing elements (SPEs), a Memory Interface Controller (MIC), and a RAMBUS Flex I/O interface split into two separate elements (IOF0 and IOF1). These twelve elements are connected by the on-chip Element Interconnect Bus (EIB). The PPE is a 64-bit dual-thread in-order PowerPC [68] processor with Vector/SIMD Multimedia extensions and 32KB L1 + 512KB L2 cache. Each SPE is a 128-bit processor with two major components: a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC) [49]. The SPU executes all instructions in the SPE with new VMX-like instruction set architecture (ISA). The SPE includes 128 registers of 128 bits and 256 KB of software-controlled local storage. The MIC provides memory access of up to 512MB of RAMBUS XDR RAM. Finally, the RAMBUS FlexIO interface is a bus controller with twelve 1-byte lanes operating at 5

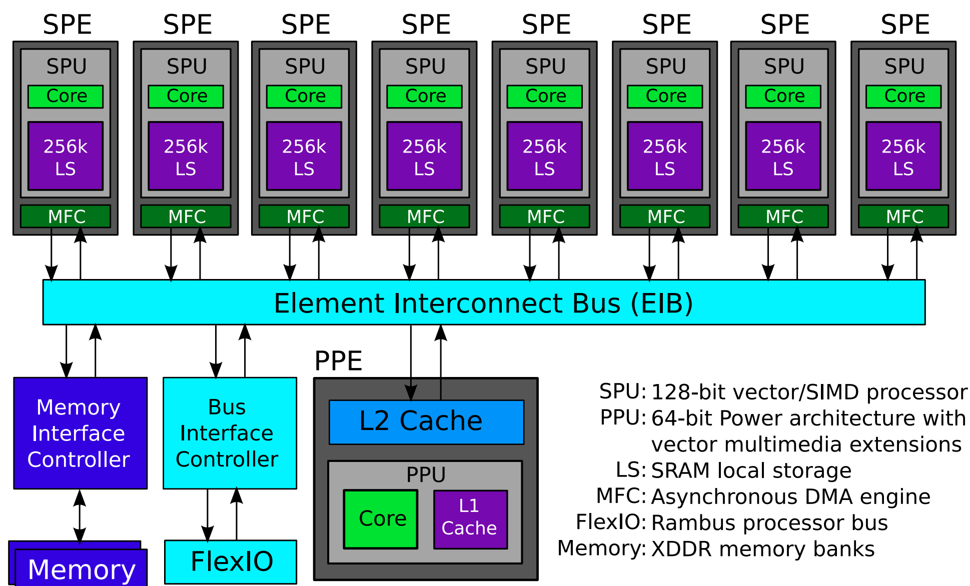


Figure 2.2: The Cell Broadband Engine Architecture. All elements are explicitly controlled by the programmer. Careful consideration of data alignment and use of the SPU vector ISA produces maximum performance. End-to-end control of the EIB is the main limiter of overall performance.

GHz. Seven lanes are dedicated to outgoing data; five lanes handle incoming data. The lanes are divided between the non-coherent IOIF0 and IOIF1 interfaces. IOIF0 can also be used as a coherent interface by using the Broadband Interface (BIF) protocol. The BIF protocol supports access to other CBEA processors in multiprocessor system configurations, creating a "glueless" multi-processor which gives the appearance of a single CBEA chip with two PPEs and sixteen SPEs. This is the configuration used by IBM in it's high-performance computing products.

The Cell Broadband Engine (Cell BE) is the first implementation of the CBEA and is primarily a single-precision floating point processor for media applications. Its peak single-precision FP performance is 230.4 Gflops, but peak performance for double-precision is only 21.03 [36]. Mixed-precision methods should be considered when solving double-precision problems on this chipset [30]. The Cell BE appears in the Sony PlayStation 3 and in IBM QS20 and QS21 servers.

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 92W and a double precision theoretical peak performance of 115.2 gigaflops [128]. At 1.25 gigaflops/watt, the PowerXCell 8i achieves the best double-precision flops/watt of any chipset on the market. Roadrunner at Los Alamos uses the PowerXCell 8i processor [23], and the top six systems on the November 2009 Green

500 list use the PowerXCell 8i [130]. The PowerXCell 8i appears most frequently in IBM QS22 blades.

2.2.1 The Element Interconnect Bus (EIB)

The CBEA places a number of requirements on its on-chip network. The EIB must support twelve elements, which is greater than the typical two- to eight-node interconnect support needed by most other commercially available multi-core processors to date. In addition, each element is capable of 51.2 GB/s of aggregate throughput, so the EIB must be able to support high data rates. In contrast, mainstream multi-core processors such as those from Intel and AMD support far less: typically, less than 15 GB/s per core. To achieve this, the EIB consists of four 16 byte-wide data rings (two in each direction), a shared command bus, and a central data arbiter to connect the twelve elements [36]. Each ring is capable of handling up to three concurrent non-overlapping transfers, allowing the network to support up to twelve data transfers at a time across the EIB for a peak bandwidth of 204.8 Gigabytes/second. The central data arbiter controls access to the EIB data rings on a per transaction basis. Because of the different caching and coherency approaches in the PPE and SPEs, the EIB also supports memory coherent and non-coherent data transfers. Ainsworth and Pinkston characterized the performance of the CBEA's Element Interconnect Bus in [18]. They found that the end-to-end control of the EIB influenced by software running on the cell has inherent scaling problems and serves as the main limiter to overall network performance.

2.2.2 Programming Considerations

The CBEA architecture is subject to stringing alignment and memory access requirements. The only memory directly available to an SPE is its own on-chip local storage. To operate on data in main memory, an SPU places a DMA request on a MFC's request queue. The SPE typically send the request to it's own MFC, but can also make requests of another SPE's MFC. Data transferred by a MFC must be 8-byte aligned, 1, 2, 4, 8, or a multiple of 16 bytes in size, and no more than 16 KB total per DMA request. Data blocks that are a multiple of 128 bytes in size and 128-byte aligned are transfered most efficiently. Noncontiguous (i.e. strided) data cannot be transferred in a single DMA request and must either be handled by many DMA requests of perhaps different sizes or a single *DMA list*. A DMA list is a list of MFC commands, each specifying a starting address and size. It is formed in local storage by the SPU and passed to the MFC. DMA lists overcome the 16 KB limit and enable scalable transfers of strided data.

The CBEA's heterogeneous design and non-blocking data access capabilities allow for a variety of programming methodologies. A *traditional approach*, which views each SPE as an independent processor, is certainly possible. This is the common approach for codes which are predominately composed of homogeneous parallel routines and codes ported directly from

homogeneous multi-core architectures. The traditional approach is often easier to program and may reduce debugging and optimization times, but it sacrifices much of the CBEA's processing power. Another popular approach is the *function offload model*. In this case, functions with SIMD operations or stream-like properties are offloaded to the SPEs, while the PPE performs the bulk of general processing. Sourcery VSIPL++ is a commercial package that depends heavily on function offloading [37]. The CBEA also excels as a *stream processor*. When used in this way, the SPEs apply kernel functions to data provided by the MFC. SPEs can be pipelined to rapidly apply multiple kernels to the same stream of data. The CBEA's polymorphic parallelism allows combinations of all these methods to be dynamically applied as needed. Regardless of methodology, maximum performance is obtained through the use of asynchronous memory I/O and the vector ISAs of the SPUs and PPU.

2.2.3 Case Studies and Applications

The Cell BE was developed as a multimedia processor, so it's no surprise that it achieves excellent performance as an H.264 decoder [22], JPEG 2000 encoding server [91], speech codec processor [125], ray tracing engine [83, 24], high performance MPEG-2 decoder [21], and spatial domain filter [100]. It is also a promising platform for high-performance scientific computing. Williams et al. achieved a maximum speedup of 12.7x and power efficiency of 28.3x with double-precision general matrix multiplication, sparse matrix vector multiplication, stencil computation, and Fast Fourier Transform kernels on the Cell BE, compared with AMD Opteron and Itanium2 processors [120]. Their single precision kernels achieved a peak speedup of 37.0x with 82.4x power efficiency when compared with the AMD Opteron.

Blagojevic et al. [26] developed a runtime system and scheduling policies to exploit polymorphic and layered parallelism on the CBEA. By applying their methods to a Maximum Likelihood method for phylogenetic trees, they realized a 4x performance increase on the Cell BE in comparison with a dual-processor Hyperthreaded Xeon system. Hieu et al. [59] combined the heterogeneous SIMD features of the SPEs and PPE to achieve an 8.8x speedup for the Striped Smith-Waterman algorithm as compared to an Intel multi-core chipset with SSE2 and a GPGPU implementation executed on NVIDIA GeForce 7800 GTX.

Ibrahim and Bodin [67] introduced *runtime data fusion* for the CBEA to dynamically reorganize finite element data to facilitate SIMD-ization while minimizing shuffle operations. By combining this method with hand-optimized DMA requests and buffer repairs, they achieved a sustained 31.2 gigaflops for an implementation of the Wilson-Dirac Operator. Furthermore, by using the heterogeneous features of the PPE to analyze data frames, they were able to reduce memory bandwidth by 26%.

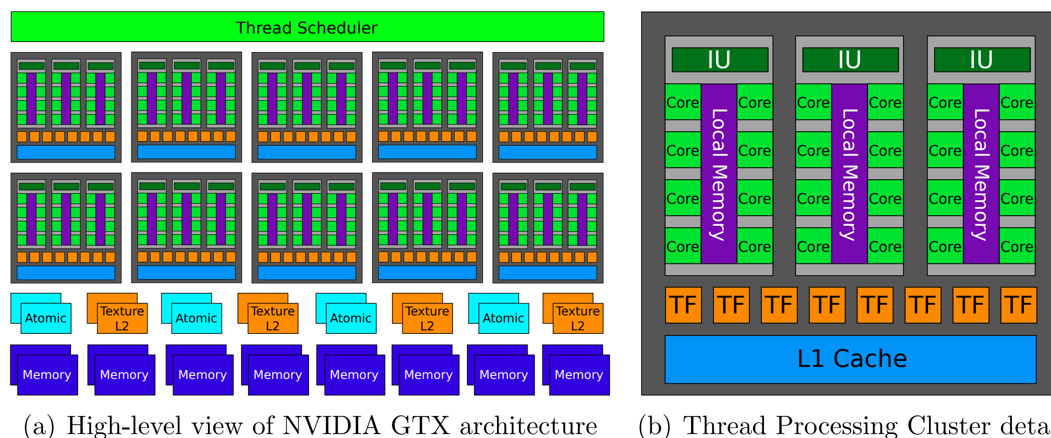
2.3 General Purpose Graphics Processing Units

Graphics Processing Units (GPUs) are low-cost, low-power (in terms of watts per flop), massively-parallel homogeneous microprocessors designed for visualization and gaming. Although NVIDIA boldly claims to have "invented the GPU in 1999" [132], specialized hardware for graphics programming was first seen in commercial products from Atari in the mid 1970s, and the first commercial instance of a graphics-dedicated co-processor card was the IBM Professional Graphics Controller (PGC) card introduced in 1984 [43]. Accelerator chips dedicated to either 2D or 3D acceleration (but not both) were commodity items in the mid 1990s, appearing in the Nintendo 64, Sony PlayStation, and PC graphics cards from 3dfx, S3, and ATI. Later generations of 3dfx Voodoo and Rendition Verite graphics cards combined 2D and 3D acceleration on a single board in the late 1990s. NVIDIA did in fact coin the term "GPU" with the release of the GeForce 256 in 1999, the first accelerator card to combine transform, lighting, triangle setup/clipping, and rendering engines in a single chip. Transform and lighting (or T&L) then became the defining characteristic of GPUs. The advent of OpenGL API and DirectX 9.0 in early 2000 popularized pixel-by-pixel processing in GPU hardware and lead to an order-of-magnitude performance improvement over CPUs by 2002 [105].

Because of their power, these special-purpose chips have received a lot of attention recently as general purpose computers (GPGPUs). Examples include the NVIDIA GTX family (a.k.a "Tesla" and "Fermi"), and ATI Firestream GPUs. One popular example, the NVIDIA Tesla C1060 (Figure 3), 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) [129]. The ATI Radeon HD 4870 competes with the C1060. It has 800 750MHz processing units and a single-precision theoretical peak of 1.2 teraflops [85]. GPUs are often an order of magnitude faster than CPUs and consume 2–3 times more power. GPU performance has been increasing at a rate of 2.5x to 3.0x annually, compared with 1.4x for CPUs [61] and GPU technology has the distinct 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 to the user.

Recent GPUs incorporate hardware double precision floating point units in addition to their single-precision cores. The double precision unit is shared between multiple processing cores, so double precision operations must be pipelined through this unit instead of executing on the streaming processors. Hence, 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.

The early success of GPUs as general purpose computers has inspired GPU-like hardware



(a) High-level view of NVIDIA GTX architecture (b) Thread Processing Cluster detail

Figure 2.3: 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.

with explicit support for high performance computing. The latest NVIDIA GPU codename "Fermi" extends the GTX 200 architecture with improved double precision performance, Error Correcting Code (ECC) support, increased shared memory capacity, and a true cache hierarchy [132]. Fermi GPUs are expected to be available in 2010.

2.3.1 NVIDIA CUDA

NVIDIA GPUs are programmed in CUDA [93]. Since CUDA's release in 2007, 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 [92]. 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 [114].

Expressed as a minimal extension of the C and C++ programming languages, CUDA is a model for parallel programming that provides a few easily understood 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 three-dimensional *thread blocks*. Threads in a block can cooperate among themselves through barrier synchronization and fast, private shared memory. A collection of independent blocks forms a *grid*. An application may execute multiple grids independently (in parallel) or dependently (sequentially). The programmer specifies the number of threads per block (up to 512) and number of blocks

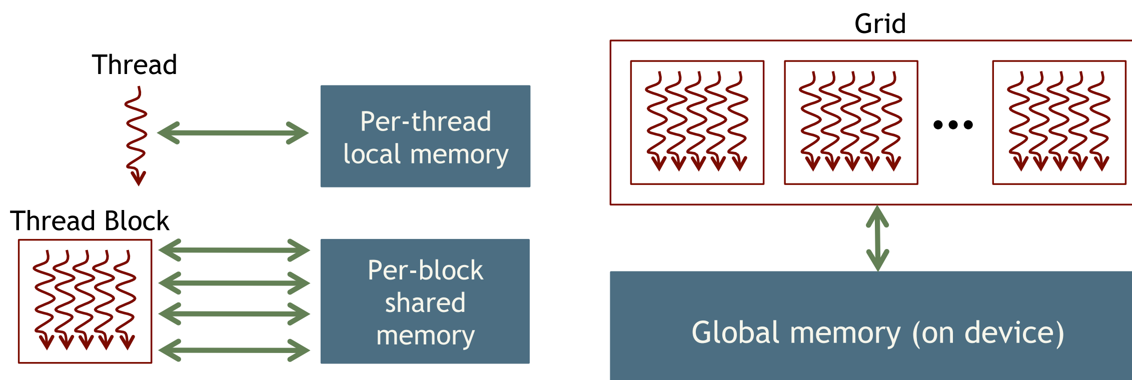


Figure 2.4: Overview of CUDA. A thread is the basic unit of parallelism. Threads are grouped into blocks, which are aggregated into grids. Threads within a block can communicate via the shared memory; threads in separate blocks neither communicate nor access each other's shared memory.

per grid. Threads from different blocks cannot communicate directly, however they may coordinate their activities by using atomic memory operations on the global memory visible to all threads.

The CUDA programming model is a superset of the Single Program Multiple Data (SPMD) model. The ability to dynamically create a new grid with the right number of thread blocks and threads for each application step grants greater flexibility than the SPMD model. The concurrent threads express fine-grained data- and thread-level parallelism, and the independent thread blocks express course-grained data parallelism. Parallel execution and thread management in CUDA are automatic. All thread creation, scheduling, and termination are handled for the programmer by the GPU hardware.

2.3.2 ATI Stream SDK

ATI uses stream processing in its GPGPU products. *Stream processing* is a programming paradigm for exploiting parallelism that works particularly well in a heterogeneous environment. A stream is a collection of data which can be operated on in parallel. Stream environments apply a set of computational kernels (functions or operations) to each data element. Often the kernels are pipelined. This approach is particularly useful when programming several separate computational units without explicitly managing allocation, synchronization, and communication. As an abstraction, this model has been successful on CBEA, GPGPUs, and multi-core CPUs.

GPGPU products from ATI have tried several stream processing approaches, beginning in 2006 with the introduction of ATI Close To Metal (CTM), arguably the first commercial

instance of GPGPU computing [126]. Although announced in 2006, CTM wasn't fully supported until December 2008. CTM formed the base layer of the ATI Stream SDK, a software stack topped by Brook+, AMD's implementation of the Brook stream programming language for ATI and AMD GPGPUs [127]. Brook, developed for programming streaming processors such as the Merrimac [38] and Imagine processors [75], uses extensions to standard ANSI C to incorporate the ideas of data parallel computing and arithmetic intensity into a familiar and efficient language. On GPUs, the Brook source-to-source compiler maps Brook kernels onto Cg shaders, which are translated by vendor-supplied shader compilers, and emits code which uses the Brook Runtime to invoke kernels [28]. CTM's esoteric design discouraged wide-scale adoption and in 2009 the ATI Stream SDK adopted the OpenCL standard [90] in favor of CTM.

2.3.3 Case Studies and Applications

Fan et al. [47] used a GPU cluster to simulate the dispersion of airborne contaminants in the Times Square area of New York City with the lattice Boltzmann model (LBM). They added an NVIDIA GeForce FX 5800 Ultra with 128MB memory to each node of a 35-node cluster of 2.4GHz dual-core Pentium Xeon nodes. For only $12,768, they were able to boost the cluster's performance by 512 gigaFLOPS and achieve a 4.6x speedup in their simulation. Their implementation of an explicit numerical method on a structured grid showed how a range of simulations may be done on a GPU cluster.

Michalakes and Vachharajani [87] applied NVIDIA GPUs to a computationally intensive portion of the Weather Research and Forecasting (WRF) model. Their modest investment in programming effort yielded an order of magnitude performance improvement in a small but performance-critical module of WRF.

Perumalla [95] used the Brook stream programming language and an NVIDIA GeForce 6800 Go GPU to explore time-stepped and discrete event simulation implementations of 2D diffusion, as well as a hybrid implementation which used both time-stepped and discrete event simulation. Although the GPU was slower than the CPU for small simulations, large simulations saw a speedup of up to 16x on the GPU as compared to the CPU implementation.

Krüger and Westermann [77] and Bolz et al. [27] investigated solvers for Navier-Stokes equations on GPUs. These works represent matrices as a set of diagonal or column vectors, and vectors as 2D texture maps to achieve considerably improved basic linear algebra operator performance on NVIDIA GeForce FX and ATI Radeon 9800 hardware. Hardware shortcomings, such as the lack of a continuous floating-point pipeline in the Radeon GPU limited computation accuracy, however these issues have been resolved in modern GPUs.

Bolz et al. [27] 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. Both the GPU and CPU implementations were bandwidth limited.

Dongarra et al. [41] report up to 328 single-precision gigaflops when computing a left-looking block Cholesky factorization on a pre-released NVIDIA T10P.

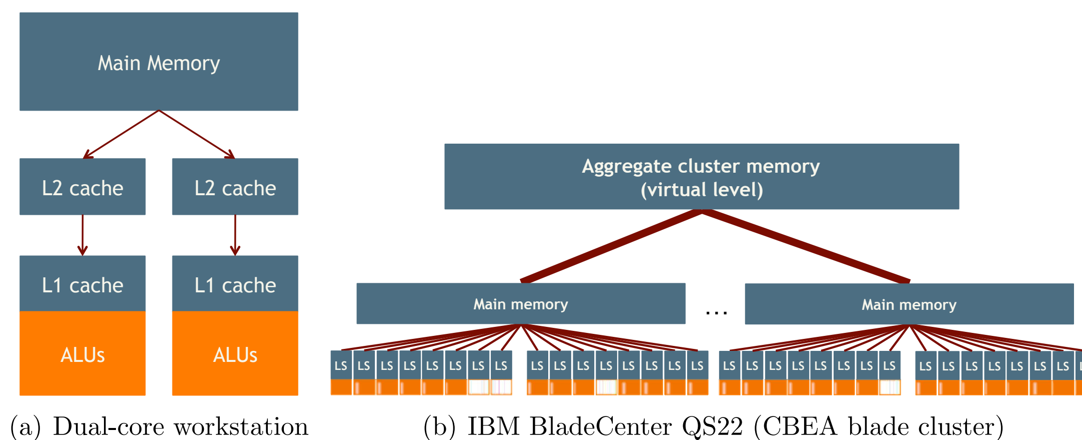
Auto-tuning techniques like those found in ATLAS [40] and FFTW [50] are being applied to linear algebra on GPUs. Li et al. [84] designed a GEMM auto-tuner for NVIDIA CUDA-enabled GPUs that improved the performance of a highly-tuned GEMM kernel by 27%. More recent efforts include the MAGMA project [20] which is developing a successor to LAPACK but for heterogeneous/hybrid architectures.

Although not an example of GPU performance, Erez and Ahn [45] used Brook to map unstructured mesh and graph algorithms to the Merrimac [38] stream processing infrastructure and achieved between 2.0x and 4.0x speedup as compared to a baseline architecture that stresses sequential access.

2.4 Other chipset designs

Intel is developing a new multi-core chipset for general purpose and visual computing code-named *Larrabee* [106]. Larrabee uses multiple in-order x86 CPU cores augmented with 16-wide vector processing units (VPUs) and fixed-function logic blocks to provide dramatically higher performance-per-watt as compared to out-of-order CPUs. Larrabee's design combines features of the CBEA, such as vector accelerators and a bi-directional ring network, with the many-core design of GPUs, but using a familiar x86 instruction set and core-shared L2 cache. Larrabee was featured in the SC'09 keynote with a demonstration of over a sustained single-precision teraflops on a single chip.

In addition to the general purpose chipsets discussed in this chapter, many heterogeneous and homogeneous multi-core Application-Specific Integrated Circuits (ASICs) have been developed. These chipsets may be unsuitable for general-purpose computation, but they demonstrate the power of multi-core design and influence multi-core microprocessor design. ASICs commonly appear in cellular phones, automobiles, aircraft, etc. Rather than cite dozens of multi-core ASICs, two representative examples are given. Anton is a special-purpose parallel machine, currently under construction, that is expected to dramatically accelerate molecular dynamics computations relative to other parallel solutions [107]. The IBM Blue Gene/L and Blue Gene/P supercomputers utilize system-on-a-chip techniques to integrate all system functions including a PowerPC processor, communications processor, three cache levels, and multiple high speed interconnection networks onto a single ASIC [115].



(a) Dual-core workstation (b) IBM BladeCenter QS22 (CBEA blade cluster)

Figure 2.5: Example Sequoia memory trees. Sequoia facilitates portability by abstracting the memory hierarchy.

2.5 Multi-core Languages, Tools and Methodologies

The principal difficulty in programming multi-core technology is the steep learning curve for efficient use of heterogeneous parallelism. Most emerging multi-core platforms support the C programming language or some variation of C. Fortran compilers are available for the CBEA and NVIDIA GPUs, however these compilers are not as mature as single-core compilers or even homogeneous multi-core compilers. Traditional parallelization approaches, such as OpenMP and MPI, can be used on heterogeneous and massively-parallel homogeneous chipsets [94, 118, 78, 79], yet this rarely achieves peak performance since these approaches were developed under a different chipset paradigm.

Sequoia is a popular stream processing programming language with implementations for CBEA, GPGPU, distributed memory clusters, and other platforms [48]. Sequoia addresses the complexity of programming *exposed-communication* chipsets (those which require software to move data in between distinct off-chip and on-chip address spaces) through first class language mechanisms. The Sequoia model abstracts parallel machines as trees of distinct memory modules. This facilitates portability, since the algorithm description is strictly separate from machine-specific optimizations. Task abstractions describe self-contained units of computation isolated on a local address space and form the smallest unit of computation (i.e. the granularity of parallelism in Sequoia is the task). A Sequoia *task* is a generalization of a stream programming kernel. Depending on the application, Sequoia can achieve excellent performance on many multi-core platforms and has the distinct advantage of simplifying multi-core programming.

SPADE is a declarative stream processing engine for the System S stream processing middleware [53]. It targets versatility and scalability to be adequate for future-generation stream

processing platforms. A prototype implementation for the CBEA is under development.

Chapter Summary

Emerging multi-core architectures and their associated systems, languages, and tools were introduced. Multi-core chipsets combine two or more independent processing cores into a single integrated circuit to implement parallelism in a single physical package. Physical constraints drive multi-core chipset design. An architecture which consistently repeats one core design for all cores is called *homogeneous*. If multiple core designs are used, the architecture is called *heterogeneous*. The defining characteristics of "emerging" multi-core technologies include massive heterogeneous parallelism, exposed or explicit communication, multiple on-chip address spaces, and an explicitly-managed memory hierarchy.

Case studies exhibiting good application performance on both homogeneous and heterogeneous multi-core architectures were presented. The Cell Broadband Engine Architecture (CBEA) excels in processing media, commercial, and scientific applications, even when the flops-per-byte ratio is less than one. General Purpose Graphics Processors (GPGPUs) make excellent high performance processors when the flops-per-byte ratio is high. These chipsets lead the market in single-precision flops and flops-per-watt. Special-purpose chipsets and ASICs were also discussed.

Chapter 3

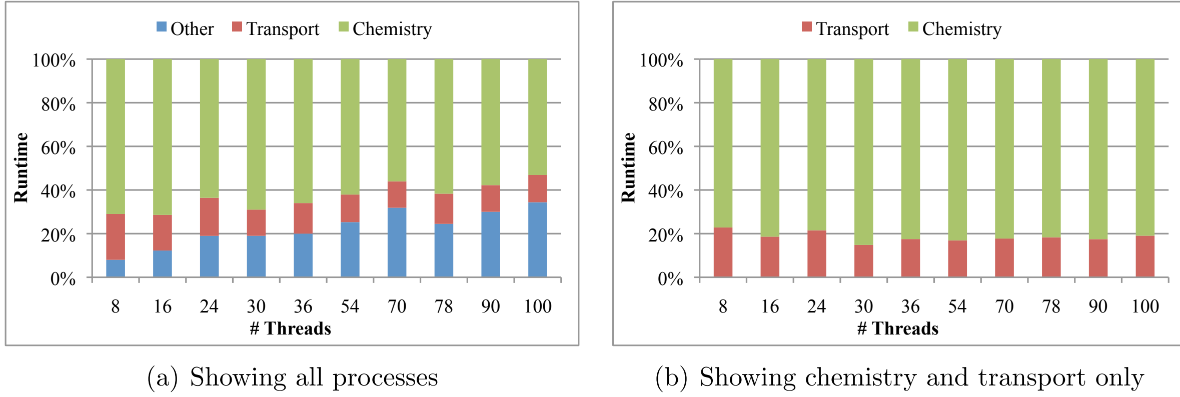
Atmospheric Modeling

Earth's atmospheric state is the result of a complex and dynamic combination of hundreds of interacting processes. In order to accurately describe atmospheric state, a model must consider the chemical interactions of air-born chemical species together with their sources, long range transport, and deposition. This is done by combining a *chemical kinetics model* with a *chemical transport model*. These processes are often abbreviated as *chemistry* and *transport*, respectively. On a realistic domain, chemistry and transport together form systems of equations involving between 10^6 and 10^9 floating point values. Accurate solutions of these systems often require hours or days of computer time.

Chemical transport modules (CTMs) solve mass balance equations for concentrations of trace species in order to determine the fate of pollutants in the atmosphere [98]. The mass balance equations are generally separable through operator splitting; for example dimension splitting separates calculation of north-south transport from east-west transport. Both implicit and explicit methods are appropriate. In a comprehensive model, the transport component (the CTM itself) may be responsible for as much as 30% of the computational time. When chemistry is ignored, transport is often responsible for over 90% of the computational time.

Chemical kinetics models solve large numbers of coupled partial differential equations to analyze the interactions of chemical species. Studies of air pollution, acid rain, smog, combustion (such as wild fires), and climate change require chemical modeling. Due to the stiffness[1] of the coupled ODE system, chemistry is often the most computationally expensive component of a comprehensive model. There is, however, abundant data parallelism in the chemical model. On a fixed grid, changes in a concentration c_i of chemical species i depend only on species concentrations at that same grid point; there are no point-to-point data dependencies. Hence, chemical models are embarrassingly parallel on a regular fixed grid. There is little or no parallelism at each grid point since the ODE system is coupled,

[1] "Stiffness" is defined by Lambert in [81]: "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."



(a) Showing all processes (b) Showing chemistry and transport only

Figure 3.1: Runtime profiles of the Sulfur Transport and dEposition Model (STEM) executing on the System X supercomputer. Chemistry dominates even as the number of threads increases.

so strong scaling is achieved mainly by processing more than one grid point per core, rather than increasing the number of cores assigned to a single a grid point. As shown in Figures 1, 2, 3, stepping the system of coupled and stiff ODEs through time often consumes 60-95% of the model runtime, regardless of the number of threads.

3.1 A Comprehensive Chemistry Transport Model

The principle concern of all atmospheric models is the time evolution of concentrations of chemical species. The descriptions of this process given in [104, 62] are summarized here. In general, the time evolution of a concentration $c_i \in \mathbf{c}$ of atmospheric chemical species i $(1 \leq i \leq s)$ is described by the material balance equations

$$\frac{\partial c_i}{\partial t} = -\mathbf{u} \cdot \nabla c_i + \frac{1}{\rho}\nabla \cdot (\rho K \nabla c_i) + \frac{1}{\rho} f_i(\rho \mathbf{c}) + E_i, \tag{3.1}$$

$$c_i(t^0, x) = c_i^0(x), \tag{3.2}$$

$$c_i(t, x) = c_i^{IN}(t, x) \quad \text{for } x \in \Gamma^{IN}, \tag{3.3}$$

$$K\frac{\partial c_i}{\partial \mathbf{n}} = 0, \tag{3.4}$$

$$K\frac{\partial c_i}{\partial \mathbf{n}} = V_i^{dep} c_i - Q_i \quad \text{for all } 1 \leq i \leq s. \tag{3.5}$$

Here $t^0 \leq t \leq T$ is time, $\mathbf{u}$ is the wind field vector, ρ is air density, K is the turbulent diffusivity tensor, f_i is the rate of chemical transformation in species i, Q_i and E_i are surface

and elevated emissions of species i respectively, and V_i^{dep} is the deposition velocity of species i. The domain Ω coves a region of the atmosphere and has the boundary $\partial\Omega$ with $\mathbf{n}$ the outward normal vector on each boundary point. The boundary can be partitioned into $\partial\Omega = \Gamma^{IN} \cup \Gamma^{OUT} \cup \Gamma^{GR}$ where Γ^{GR} is the ground level portion of the boundary, Γ^{IN} (inflow boundary) is the set of boundary points where $u \cdot \mathbf{n} \leq 0$, and Γ^{OUT} (outflow boundary) is the set where $u \cdot \mathbf{n} > 0$.

Equations 3.1–3.5 are called the *forward or direct model* and its solution is uniquely determined once the model parameters are specified. It is solved by a sequence of N timesteps of length Δt taken between t^0 and $t^N = T$. At each step, a numerical approximation $c^k(x) \approx c(t^k, x)$ at $t^k = t^0 + k\Delta t$ is calculated such that

$$c^{k+1} = \Phi_{[t^k,t^{k+1}]} \circ c^k, \quad c^N = \prod_{k=0}^{N-1} \Phi_{[t^k,t^{k+1}]} \circ c^0 \tag{3.6}$$

where Φ is the numerical solution operator. Standard practice is to form Φ through an operator splitting approach where the transport steps along each direction and the chemistry steps are taken successively [82, 66, 65]. Operator splitting permits the application of different time stepping methods to different parts of Equation 3.1. For example, chemical processes are stiff, which calls for an implicit ODE method, but explicit methods are more suitable for space-discretized advection. If A is the numerical operator for directional transport, and B is the solution operator for chemistry:

$$\Phi_{[t^k,t^{k+1}]} = A_X^{\Delta t/2} \circ A_Y^{\Delta t/2} \circ A_Z^{\Delta t/2} \circ B^{\Delta t} \circ A_Z^{\Delta t/2} \circ A_Y^{\Delta t/2} \circ A_X^{\Delta t/2}. \tag{3.7}$$

The numerical errors introduced by splitting should not be overlooked [110], however it is reasonable to assume that the errors are small. For example, in computational air pollution modeling the splitting errors oscillate with the diurnal cycle and are bounded in growth for evolving time [82].

3.2 Three State-of-the-art Atmospheric Models

This section gives three examples of widely-used atmospheric models useful for investigating weather, climate change, storm formation and tracking, and urban and regional air quality. WRF [109] and it's variants (WRF-Chem, WRF-Fire, etc.) are popular models with a large user base. The national weather service, Alaskan wildfire prediction, and many state governments rely on WRF for weather, wildfire, and air quality prediction. GEOS-Chem [25] is likewise a popular model with a user community extending well outside academia. Many global climate change predictions rely on GEOS-Chem. CMAQ [31] is an important air quality model that influences state and local air quality regulations across the United States.

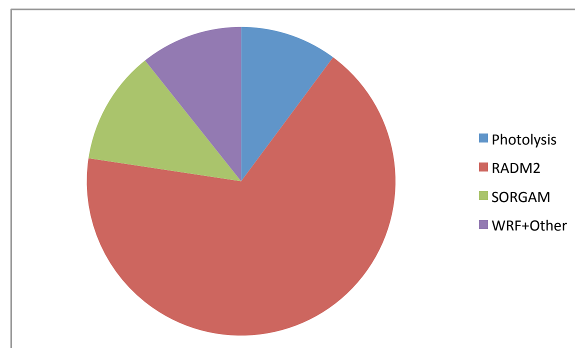


Figure 3.2: Runtime profile of WRF-Chem running on an Intel Quad-core Xeon 5500 series CPU. The data set is the RADM2 chemical mechanism combined with the SORGAM aerosol mechanism on a typically-sized $40 \times 40 \times 20$ domain grid.

3.2.1 WRF and WRF-Chem

The Weather Research and Forecasting model (WRF) is a limited-area model of the atmosphere for mesoscale research and operational numerical weather prediction [109]. WRF is in widespread use over a range of applications including real-time weather prediction, tropical cyclone/hurricane research and prediction, regional climate, atmospheric chemistry and air quality, and basic atmospheric research. WRF includes a moist thermodynamic equation, making it appropriate for precipitation processes, and is fully nonhydrostatic, so it is appropriate for deep convection and gravity wave breaking. [108] describes the continuous equations solved in WRF. These are Euler equations cast in a flux (conservative) form where the vertical coordinate is defined by a normalized hydrostatic pressure. WRF state variables are discretized over regular Cartesian grids and the model solution is computed using an explicit high-order Runge-Kutta time-split integration scheme.

WRF-Chem is WRF combined with anthropogenic, biogenic, photolysis, and aerosol modeling schemes [57]. The Kinetic PreProcessor (KPP) [39] provides Rosenbrock solvers for five different gas-phase chemical reaction calculations and two aerosol schemes. Other schemes, such as the MOSAIC aerosol scheme [121], MADE-SORGAM aerosol scheme [111] and various NASA Goddard atmospheric radiation schemes are also supported. Figure 2 shows the runtime profile of WRF-Chem running on an Intel Quad-core Xeon 5500 series CPU. The data set is the RADM2 chemical mechanism combined with the SORGAM aerosol mechanism on a typically-sized $40 \times 40 \times 20$ domain grid. Chemistry is clearly the dominant process.

[86] describes a WRF "nature run" that provides a very high-resolution standard against which more coarse simulations and parameter-sweeps may be compared. The calculation is neither embarrassingly parallel, nor completely floating-point dominated, but memory bandwidth limited and latency-bound with respect to interprocessor communication. 32,768

CPUs of BlueGene/L at IBM Watson (BGW) and 65,536 CPUs of BlueGene/L at Lawrence Livermore National Laboratory (LLNL) are used to calculate an idealized high resolution rotating fluid on the earth's hemisphere. The nature run is a good benchmark for comparing multi-core technologies.

The computational complexity of WRF motivates grid enablement research in [101]. Sadjadi et al. present the LA Grid WRF Portal which allows meteorologists and business owner communities to simulate WRF runs and visualize the output in combination with Google Maps based projections. LA Grid WRF Portal is designed as the front-end for WRF ensembles executing on a heterogeneous grid. This work-in-progress found that grid enablement is non-trivial with no well-established methodology, and that grid-enabling a program developed for a local cluster has many repercussions. The impact of moving from local to wide area networks can be counterproductive even when processing power increases. This suggests that fast, inexpensive, power-efficient clusters that can provide the computational power required to run WRF at high resolutions are preferable to large distributed grids of today's technology.

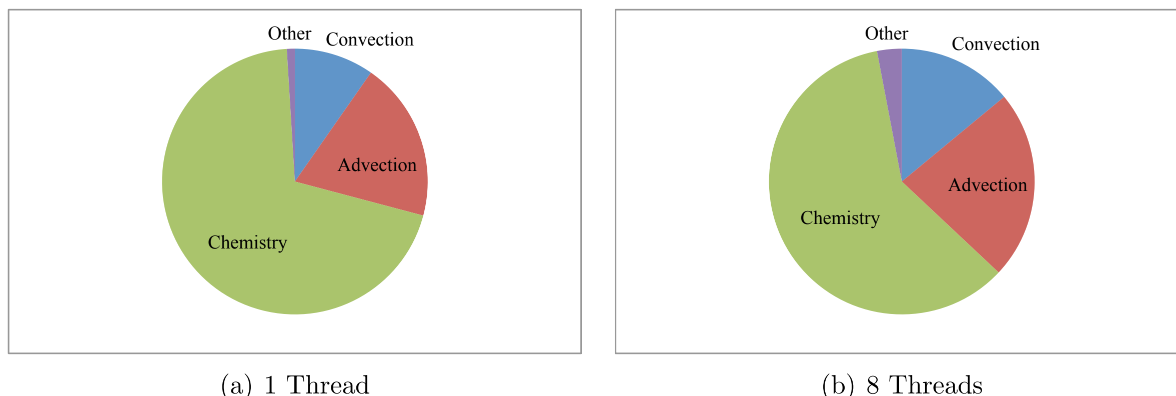
3.2.2 GEOS-Chem

GEOS-Chem [25] is a global 3-D chemical transport model for atmospheric composition. It principally uses meteorological input from the Goddard Earth Observing System (GEOS) of the NASA Global Modeling and Assimilation Office [52], but can use other meteorological inputs as well. The model uses detailed inventories for fossil fuel, biomass burning, biofuel burning, biogenic, and aerosol emissions, and includes state-of-the-art transport and photolysis routines. Chemistry in GEOS-Chem is calculated by SMVGEAR II [71] or by a combination of SMVGEAR II and KPP [39]. All model components have been parallelized using the OpenMP compiler directives, and scale well when running across multiple CPU's. However it does not yet support distributed memory systems with MPI. Figure 3 shows the runtime profile of serial and parallel runs of GEOS-Chem using the $4^\circ \times 5^\circ$ GEOS grid data. In both cases, chemistry is the dominant process.

GEOSChem also contributes to the broader modeling community by serving as an outer nest for several regional air quality models. It has a highly modular structure to facilitate the swapping of code with other atmospheric composition models and the integration into Earth System models. GEOSChem is a grass-roots cooperative model owned by its user community.

3.2.3 CMAQ

The Community Multiscale Air Quality model (CMAQ) is an active open-source development project of the U.S. Environmental Protection Agency (EPA) Atmospheric Science Modeling



(a) 1 Thread (b) 8 Threads

Figure 3.3: Runtime profile of serial and parallel runs of GEOS-Chem using the $4° \times 5°$ GEOS grid data.

Division that consists of a suite of programs for conducting air quality model simulations [31]. CMAQ combines current knowledge in atmospheric science and air quality modeling with multi-processor computing techniques in an open-source framework to deliver fast, technically-sound estimates of ozone, particulates, toxics, and acid deposition. It addresses pollutant issues in the context of a "one atmosphere" perspective where complex interactions between atmospheric pollutants and regional and urban scales are confronted.

[122] describes a study in which CMAQ was used to investigate transport and photochemical transformation of tropospheric ozone over East Asia during the Transport and Chemical Evolution of the Pacific (TRACE-P) field campaign [70]. CMAQ's predictions were compared with data gathered via two aircraft flying over the East Asian Pacific region. The average simulated value was generally in good agreement with observations and simulated levels are within a factor of two of their observations. Simulated vertical distribution of chemical species was very similar to observations, verifying CMAQ as a investigative tool for atmospheric constituents.

Chapter Summary

A general atmospheric model describing atmospheric chemical transport and kinetics was developed. Chemical transport modules (CTMs) solve mass balance equations for concentrations of trace species in order to determine the fate of pollutants in the atmosphere. In a comprehensive model, the transport component (the CTM itself) may be responsible for as much as 30% of the computational time. Chemical kinetics models solve large numbers of coupled partial differential equations to analyze the interactions of chemical species. Due to the stiffness of the coupled ODE system, chemistry is often the most computationally

expensive component of a comprehensive model. Yet chemical models are embarrassingly parallel on a regular fixed grid. Stepping the system of coupled and stiff ODEs through time often consumes 60-95% of the model runtime.

Three state-of-the-art atmospheric models were introduced. WRF-Chem is widely used to predict weather, study storms (such as hurricanes) and forecast regional air quality. GEOS-Chem is an influential global climate model. CMAQ is a popular air quality model useful for local and regional air quality forecasts. In all three models, chemistry overwhelmingly dominates computational time, marking it as the target for multi-core acceleration.

Chapter 4

Multi-core Simulation of Chemical Kinetics

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) [57], the Community Multiscale Air Quality Model (CMAQ) [31], the Sulfur Transport and dEposition Model (STEM) [32], and GEOS-Chem [25] 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 [39]. 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 c_i of species 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% 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 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% the cost of the full WRF-Chem with both chemical kinetics and aerosols.

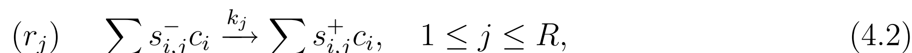
4.1 The Chemical Kinetics Model

In general, the concentration of a chemical species $c_i \in \mathbf{c} = [c_1, \ldots, c_n]^T$ can be determined at any future time by solving a generalized mass action model (GMA) [63]

$$\frac{\partial c_i}{\partial t} = \sum_{j=1}^{N_i} a_{ij} \prod_{k=1}^{n} c_k^{g_{ijk}} \quad (i = 1, \ldots, n) \tag{4.1}$$

where a_{ij} are the rate constants and g_{ijk} are the kinetic orders. To describe the time evolution of the concentrations as given by mass action kinetics, the time derivative function of the chemical system is required. The derivation of this function as given in [39] is summarized here.

Consider a system of n chemical species and R chemical reactions, $r = [r_1, \ldots, r_R]^T$. $k_j \in k = [k_1, \ldots, k_R]^T$ is defined as the rate coefficient of reaction r_j. The stoichiometric coefficients $s^-_{i,j}$ and $s^+_{i,j}$ are defined as the number of molecules of species c_i that are consumed in reaction r_j or are produced in reaction r_j, respectively. If c_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



$$(r_j) \qquad \sum s^-_{i,j} c_i \xrightarrow{k_j} \sum s^+_{i,j} c_i, \quad 1 \le j \le R, \tag{4.2}$$

where k_j is the proportionality constant. In general, the rate coefficients are time dependent: $k_j = k_j(t)$. For example, photolysis reactions are strongest at noon and zero at night.

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

$$\omega_j(t, \mathbf{c}) = k_j(t) \prod_{i=1}^{n} c_i^{s^-_{i,j}} \quad \text{(molecules per time unit).} \tag{4.3}$$

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

$$\frac{\mathrm{d}c_i}{\mathrm{d}t} = \sum_{j=1}^{R} (s^+_{i,j} - s^-_{i,j}) \omega_j(t, \mathbf{c}) \quad (i = 1, \ldots, n) \tag{4.4}$$

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 4.4 can be rewritten as

$$\frac{\mathrm{d}\mathbf{c}}{\mathrm{d}t} = (S^+ - S^-)\omega(t, \mathbf{c}) = S\omega(t, \mathbf{c}) = f(t, \mathbf{c}), \tag{4.5}$$

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

Equation 4.5 gives the time derivative function in aggregate form. Depending on the integration method, a split production-destruction form may be preferred. The production rates vector is formed by considering only those reactions that produce c_i:

$$S^+ \omega(t, \mathbf{c}) = P(t, \mathbf{c}). \tag{4.6}$$

In forming the destruction (consumption) vector, note that c_i is consumed only if it is a reactant in a reaction that consumes $\mathbf{c}$. Therefore, the destruction rate contains $\mathbf{c}$ as a factor:

$$D_i(t, \mathbf{c}) = \sum_{j=1}^{R} \frac{s^-_{i,j} \omega_j(t, \mathbf{c})}{c_i}. \tag{4.7}$$

The diagonal matrix of destruction terms

$$D(t, \mathbf{c}) = \mathrm{diag}[D_1(t, \mathbf{c}), \ldots, D_n(t, \mathbf{c})]$$

allows the loss rates to be expressed as:

$$S^- \omega(t, \mathbf{c}) = D(t, \mathbf{c})\mathbf{c} \tag{4.8}$$

leading to the production-destruction form of the equation:

$$\frac{\partial \mathbf{c}}{\partial t} = P(t, \mathbf{c}) - D(t, \mathbf{c})\mathbf{c}. \tag{4.9}$$

4.1.1 Example Chemical Mechanisms

RADM2 RADM2 was developed by Stockwell et. al. [112] for the Regional Acid Deposition Model version 2 [34]. It is widely used in atmospheric models to predict concentrations of oxidants and other air pollutants. RADM2 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 [88]. 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 [88].

4.2 Solving the Chemical System

The solution of the ordinary differential equations is advanced in time by a numerical integration method. The chemical system's extreme stiffness discourages explicit methods so this is

typically an implicit method such as Runge-Kutta or Rosenbrock [58]. Implicit integration methods require the evaluation of the Jacobian of the derivative function:

$$J(t, \mathbf{c}) = \frac{\partial}{\partial \mathbf{c}} f(t, \mathbf{c}) = \begin{bmatrix} \frac{\partial f_1}{\partial c_1} & \frac{\partial f_1}{\partial c_2} & \cdots & \frac{\partial f_1}{\partial c_n} \\ \frac{\partial f_2}{\partial c_1} & \frac{\partial f_2}{\partial c_2} & \cdots & \frac{\partial f_2}{\partial c_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial c_1} & \frac{\partial f_n}{\partial c_2} & \cdots & \frac{\partial f_n}{\partial c_n} \end{bmatrix} \tag{4.10}$$

Rosenbrock methods are attractive for a number of reasons. Like fully implicit methods, they preserve exact conservation properties due to the use of the analytic Jacobian matrix. However, they do not require an iteration procedure as for truly implicit methods and are therefore easier to implement. They can be developed to possess optimal linear stability properties for stiff problems. They are of one-step type, and thus can rapidly change step size. This is of particular importance in chemical kinetic modeling in view of the many operator-split restarts (Equation 3.7).

There is no "one size fits all" chemical model implementation approach since chemical mechanisms vary greatly in number, time dependence, and reactivity of species. If the system is autonomous, the rate constants do not depend on the time variable t and may be computed only once. Otherwise they must be recomputed during the integration stages. Autonomicity is largely irrelevant to the integrator's implementation since the reaction rates are calculated from data external to the integrator. However, the computational expense of a time-dependent system is usually more than an order of magnitude greater than that of an autonomous system.

Additionally, a chemical mechanism may be so stiff that it will not converge without full double precision floating point computation. One example is the SAPRC [33] mechanism. If time integration converges in single precision then it usually converges in only slightly 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.

Because of this variety in mechanism size and characteristics, models that support several mechanisms typically use a preprocessor to generate the mechanism's ODE function, Jacobian, and the most appropriate time integration method. The models in Section 3.2 support a number of chemical kinetics solvers automatically generated at compile time by the Kinetic PreProcessor (KPP) [39]. 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 [55], SAPRC [33], and NASA HSRP/AESA. The Rosenbrock methods implemented

in KPP typically outperform backward differentiation formulas, like those implemented in SMVGEAR [71, 44].

4.2.1 Implicit Rosenbrock Time Integration

The brief summary of Rosenbrock methods given in [103] is repeated here. For more more detail, see [58]. Rosenbrock methods are usually considered in conjunction with the stiff autonomous ODE system

$$\dot{y} = f(y), \quad t > t_0, \quad y(t_0) = y_0. \tag{4.11}$$

Note that every non-autonomous system $\dot{y} = f(t, y)$ can be put in the form of 4.11 by treating time t also as a dependent variable, i.e. by augmenting the system with the equation $t = 1$.

Usually stiff ODE solvers use some form of implicitness in the discretization formula for reasons of numerical stability. The simplest implicit scheme is the backward Euler method

$$y_{n+1} = y_n + hf(y_{n+1}) \tag{4.12}$$

where $h = t_{n+1} - t_n$ is the step size and y_n the approximation to $y(t)$ at time $t = t_n$. Since y_{n+1} is defined implicitly, this numerical solution itself must also be approximated. Usually some modification of the iterative Newton method is used. Suppose that just one iteration per time step is applied. If we then assume that y_n is used as the initial iterate, the following numerical result is found

$$y_{n+1} = y_n + k \tag{4.13}$$

$$k = hf(y_n) + hJk \tag{4.14}$$

where J denotes the Jacobian matrix $f'(y_n)$ (Equation 4.10).

The numerical solution is now effectively computed by solving the system of linear algebraic equations that defines the increment vector k, rather than a system of nonlinear equations. Rosenbrock in [99] proposed to generalize this linearly implicit approach to methods using more stages so as to achieve a higher order of consistency. The crucial conseration put forth was to no longer use the iterative Newton method, but instead to derive stable formulas by working the Jacobian matrix directly into the integration formula. His idea has found widespread use and a generally accepted formula [58] for a so-called s-stage Rosenbrock method is

$$y_{n+1} = y_n + \sum_{i+1}^{s} b_i k_i, \tag{4.15}$$

$$k_i = hf(y_n + \sum_{j=1}^{i-1} \alpha_{ij} k_j) + hJ \sum_{j=1}^{i} \gamma_{ij} k_j, \tag{4.16}$$

where s and the formula coefficients b_i, α_{ij} and γ_{ij} are chosen to obtain a desired order of consistency and stability for stiff problems. In implementation, the coefficients y_{ij} are actually taken equal for all stages, i.e. $\gamma_{ij} = \gamma$ for all $i = 1, \ldots, s$. For $s = 1$, $\gamma = 1$ linearized implicit Euler formula is recovered.

For the non-autonomous system $\dot{y} = f(t, y)$, the definition of k_i is changed to

$$k_i = hf(t_n + \alpha_i h, y_n + \sum_{j=1}^{i-1} \alpha_{ij} k_j) + \gamma_i h^2 \frac{\partial f}{\partial t}(t_n, y_n) + hJ \sum_{j=1}^{i} \gamma_{ij} k_j \tag{4.17}$$

where

$$\alpha_i = \sum_{j=1}^{i-1} \alpha_{ij}, \quad \gamma_i = \sum_{j=1}^{i} \gamma_{ij}.$$

Like Runge-Kutta methods, Rosenbrock methods successively form intermediate results

$$Y_i = y_n + \sum_{j=1}^{i-1} \alpha_{ij} k_j, \quad 1 \le i \le s, \tag{4.18}$$

which approximate the solution at the intermediate time points $t_n + \alpha_i h$. Rosenbrock methods are therefore also called Runge-Kutta-Rosenbrock methods. Observe that if we identify J with the zero matrix and omit the $\partial f / \partial t$ term, a classical explicit Runge-Kutta method results.

4.2.2 An Implementation of the Rosenbrock Integrator

As an example implementation of the Rosenbrock integrator, consider a Rosenbrock integrator with three Newton stages as generated by KPP [102, 103, 39]. 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. t, h, and s are as in Section 4.2.1, $Stage_s$ is the result of Rosenbrock stage s (k_j in Section 4.2.1) and δ is the error threshold (typically 1.0). $k(t, y)$ is the vector of reaction rates, $f(t, y)$ is the time derivative ODE function, and $J(t, y)$ is the Jacobian. Y_{new} is the new concentration vector at time $t = t_{end}$. If the system is autonomous, the reaction rates $k(t, y)$ 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.

Each time step requires an evaluation of the Jacobian J, s solutions of a linear system with the matrix $I - \gamma hJ$, and s derivative evaluations. The multiplications with J are not actually performed since the LU-decomposition of $\frac{1}{h\gamma} - J$ can be reused during each stage calculation. Because of its multistage nature, the computational costs of a Rosenbrock method, spent within one time step, are often considered to be higher than the costs of a linear multistep method of the BDF type. In particular, the Jacobian update and solution of the s linear

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 $t \leq t_{end}$

$Fcn_0 \leftarrow Fcn \leftarrow f(t,y)$

$Jac_0 \leftarrow J(t,y)$

$G \leftarrow \text{LU_DECOMP}(\frac{1}{h\gamma} - Jac_0)$

For $s \leftarrow 1,2,3$

Compute $Stage_s$ from Fcn and $Stage_{1\ldots(s\text{-}1)}$

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 $E \geq \delta$ then discard iteration, reduce h, restart

Otherwise, $t \leftarrow t + h$ and proceed to next step

Finish : Result in Y_{new}

Figure 4.1: A general outline of the three-stage Rosenbrock solver for chemical kinetics.

systems, requiring one LU-decomposition and s forward-backward substitutions typically account for most of the implementation's CPU time. However, if a Rosenbrock code solves the problem efficiently in fewer steps than a BDF code (the common case [102]) then the CPU time for a whole integration using a Rosenbrock method is significantly reduced.

J in large chemical models is frequently over 90% zeros. Exploiting the high level of sparsity greatly reduces the cost of linear algebra calculations. In the KPP-generated integrators, the ODE function, the Jacobian evaluation, LU-decomposition, and forward-backward substitution are highly-optimized automatically generated sparse implementations.

The Rosenbrock integrator adapts it's step size h automatically to enable small steps at times when the solution gradients are large and large steps when the solution gradients are small. The basis of the adaptation is the so-called embedded formula

$$\tilde{y}_{n+1} = y_n + \sum_{i=1}^{s} \tilde{b}_i (Stage_i), \tag{4.19}$$

The approximation $\tilde{y}_{n+1}$ thus differs only in the choice of weights $\tilde{b}_i$ and is available at no extra cost. The weights are chosen such that the order of consistency of $\tilde{y}_{n+1}$ is $\tilde{p} = p - 1$, where p is the order of y_{n+1}. Hence $Est = \tilde{y}_{n+1} - y_{n+1}$ is the local error estimator.

For an ODE system of dimension m, that is, a chemical mechanism with m variable species, the step size is adjusted as follows. Let $Tol_k = atol + rtol|y_{n+1,k}|$, where $atol$ and $rtol$ are user-specified absolute and relative error tolerance (typically 1.0 and 1e-3, respectively) and $y_{n+1,k}$ is the k-th component of y_{n+1}. Then

$$Err = \sqrt{\frac{1}{m} \sum_{k=1}^{m} \left(\frac{Est_k}{Tol_k} \right)^2}. \tag{4.20}$$

The integration step is accepted if $Err < \delta$; otherwise the step is rejected and redone. The step size for the new step, both in the rejected and accepted case, is estimated by the usual step size prediction formula

$$h_{new} = h \ \min(10, \max(0.1, \frac{0.9}{Err^{1/(\tilde{p}+1)}})) \tag{4.21}$$

At the first step after a rejection, the maximal growth factor of 10 is set to 1.0. Furthermore, h is constrained by a minimum h_{min} and a maximum h_{max}. To help avoid wasted steps in the implementation, h is the last step size of the previous operator-split integration interval for every new integration and h_{max} is effectively unbounded. A rejection of the first step is followed by a ten times reduction of h. Because the maximal growth factor is 10, the step size adjusts very rapidly and quickly attains large values if the solution is sufficiently smooth and $h = h_{start}$ is chosen small.

Once an iterative solver has been generated by KPP, 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). 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 sub-components, but such inter-module decomposition is rare under the numerical methods examined in this work. 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 supplied by the atmospheric model.

4.3 RADM2 on Three Multi-Core Architectures [5]

Serial implementations of chemical kinetics mechanisms have been studied for decades, so their design is well established. However, the uniqueness of emerging multi-core architectures makes it difficult to anticipate implementation details of a general mechanism, that is, it was unknown what a highly-optimized multi-core implementation of chemical kinetics "looked like." A detailed exploration of a specific mechanism on every target architecture is required.

This section presents a case study demonstrating significantly reduced time-to-solution for a chemical kinetics kernel from WRF-Chem. Intel Quad-Core Xeon chipsets with OpenMP, NVIDIA GPUs with CUDA, and the Cell Broadband Engine Architecture (CBEA) are used to solve the RADM2 [112] 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. 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. Section 4.4 generalizes these results to other similar chemical kinetics codes, enabling the rapid development of multi-core accelerated chemical kinetics kernels. 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 maximum observed speedup is 41.1×.

The investigation began by hand-porting 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. (Once the C model was developed, Fortran code for the CBEA could be generated automatically following the same design.) 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.

Table 4.1: Typical operation counts for a single invocation of the RADM2 chemical kernel. Although the operation count is high, computational intensity is low: 0.60 operations per 8-byte word and .08 operations per byte.

	Coarse	**Fine**
FP Ops	609,226	4,264,528
Load/Store	1,021,227	17,360,859

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 floating-point data. While porting to each platform, care was taken to avoid any design which was specific to RADM2. Thus, our hand-tuned multi-core RADM2 implementations form templates a code generator can reuse when targeting these architectures.

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.

Benchmarks

The benchmarks presented in this work assume a 200W power budget for the principle computing chipset. Within this envelope, two Quad-Core Xeon chips, two CBEA chips, or one Tesla C1060 GPU can be allocated. (Note that the GPU is a co-processor for a CPU, so in fact the GPU cases break the 200 watt budget by at least 45 watts.) Quad-Core Xeon benchmarks are performed on a Dell Precision T5400 workstation with two Intel E5410 CPUs and 16GB of memory on a 667MHz bus. NVIDIA GPU benchmarks are performed on the NCSA's experimental GPU cluster and an in-house Intel Quad-Core Xeon workstation with an NVIDA GeForce GTX280 GPU. Each NCSA 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 256MB XDRAM, the QS20 has 1GB XDRAM, and the QS22 has 8GB XDRAM. Both the QS20 and

Table 4.2: 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.

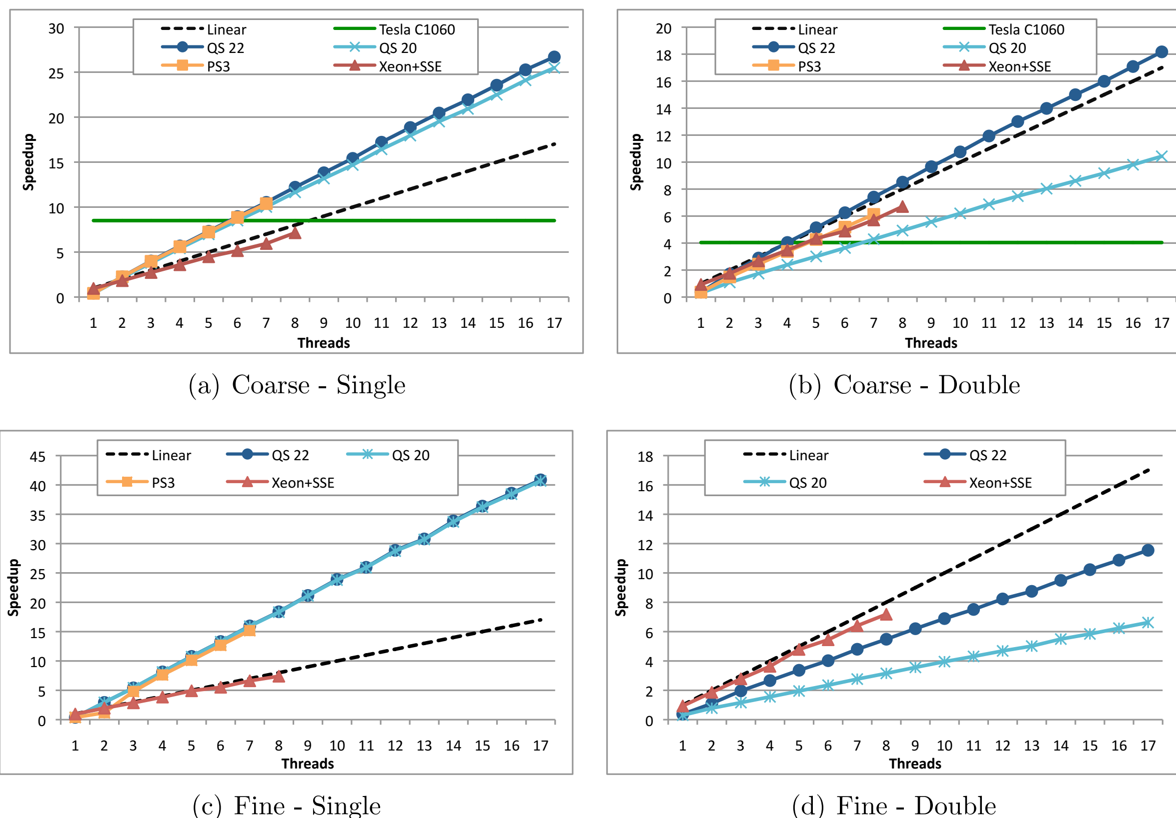
	Double		**Single**	
Wall Clock	72.1439	21.1622	75.7696	22.3848
Rosenbrock	67.2134	20.8971	67.8416	21.0944
LU Decomp.	30.7513	10.1084	31.2645	10.2868
LU Solve	9.9441	3.1102	10.0512	3.1366
ODE Function	8.3253	2.3785	8.3453	2.3761
ODE Jacobian	9.3343	2.5421	9.3463	2.5430
File I/O	3.2563	0.0546	3.3986	1.0607
Data Packing	2.1925	0.0885	2.1790	0.0920
	Fine	Coarse	Fine	Coarse

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.

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 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".

Table 4.2 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", 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 4.3 shows the performance in seconds of the RADM2 multi-core ports. The labels are identical to those in Table 4.2. 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



(a) Coarse - Single

(b) Coarse - Double

(c) Fine - Single

(d) Fine - Double

Figure 4.2: Accelerated RADM2 speedup as compared to the original serial code. A horizontal line indicates the maximum GPU speedup since the thread models of the GPU and conventional architectures are not directly comparable. Fine grid tests could not be conducted on some platforms due to memory limitations.

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. Figure 2 shows the speedup of the RADM2 ports as compared to the original serial code. 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 the GTX280. Engineers at NVIDIA have proposed that our GTX memory contains a hardware error. Since rendering applications are highly tolerant of small errors, the current generation of GTX GPUs do not support error checking in memory. However, stiff chemical kinetics demand 100% accuracy. The next generation of NVIDIA GPUs codenamed "Fermi" supports memory ECC.

Table 4.3: 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 4.3.

		QS22		**QS20**		**PS3**		**Xeon**		**Tesla C1060**	
		16 SPEs		**16 SPEs**		**6 SPEs**[a]		**8 Cores**		**(a)**[b]	**(b)**
Single	Wall Clock	9.9176	1.2696	11.4860	1.1690	342.9870	2.6361	10.9354	3.0118	****	4.945
	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
	File I/O	1.9895	0.1757	3.8953	0.0781	38.2561	0.1329	0.7188	0.0553	****	0.0620
	Data Packing	4.3952	0.2516	4.1078	0.2492	276.9783	0.3211	0.2697	0.0099	****	0.6132
Double	Wall Clock	13.2182	1.7221	22.7739	2.6015	518.3893	4.3059	11.4476	3.2026	****	9.457
	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
	File I/O	1.6012	0.1702	4.4391	0.0785	35.8189	0.1245	0.7270	0.0563	****	0.0589
	Data Packing	4.3641	0.2862	5.6773	0.3359	385.8060	0.3344	0.2819	0.0101	****	0.6992
		Fine	Coarse	Fine	Coarse	Fine	Coarse	Fine	Coarse	Coarse	Coarse

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

4.3.1 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.

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 2, this port achieved nearly linear speedup over eight cores.

4.3.2 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 4.3). 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.[1] 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 4.3 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.

The NVIDIA CUDA implementation was straight-forward to program, but 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 required 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.

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 con-

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

centration 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.

4.3.3 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 [49]. 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 (Figure 3). 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 2.2), 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.

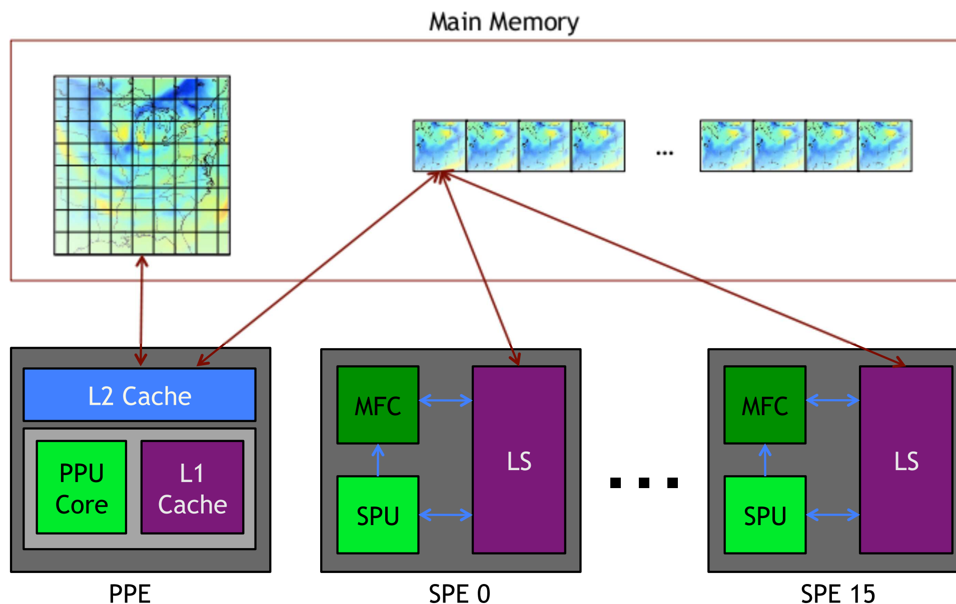


Figure 4.3: CBEA implementation of RADM2. The master (PPE) prepares model data for processing by the workers (SPEs).

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.

The final version of the kernel adds two more optimizations to the SPU and vectorizes the solver on the PPU. Because mis-predicted branches are expensive on the SPU, loops were unrolled whenever possible and static branch prediction hints were added when the common branch outcome was known. A triple-buffering scheme was added to the local store data, allowing the memory flow controller to simultaneously write out old results to main memory and read in fresh data while the SPU processes data in the local store. Because the SPU's ISA is very similar to Altivec, it was simple to back-port the SIMD integrator to the PPU. The PPU uses a scalar integrator for double precision data.

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". 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.

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.

4.4 KPPA: A Software Tool to Automatically Generate Chemical Kernels [6]

Writing chemical kinetics code is often tedious and error-prone work, even for conventional scalar architectures. Emerging multi-core architectures, particularly heterogeneous architectures, are much harder to program than their scalar predecessors and require a deep understanding of the problem domain to achieve good performance. General analysis tools like the Kinetic PreProcessor (KPP) [39] make it possible to rapidly generate correct and efficient chemical kinetics code on scalar architectures, but these generated codes cannot be easily ported to emerging architectures.

This section presents KPPA (the Kinetics PreProcessor: Accelerated), 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 multi-layered heterogeneous parallelism to achieve strong scalability. Compared to state-of-the-art serial implementations, speedups of 20×–40× are regularly observed in KPPA-generated code.

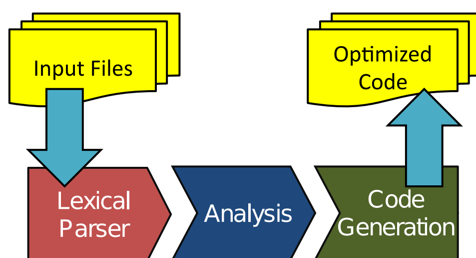


Figure 4.4: Principle KPPA components and KPPA program flow.

KPPA combines a general analysis tool for chemical kinetics with a code generation system for scalar, homogeneous multi-core, and heterogeneous multi-core architectures. It is written in object-oriented C++ with a clearly-defined upgrade path to support future multi-core architectures as they emerge. KPPA has all the functionality of KPP 2.1, the latest version of KPP, and maintains backwards compatibility with KPP. Though based on KPP, KPPA is written entirely from scratch and is a new software tool maintained separately. KPPA is released under the GNU General Public License (GPL) and can be downloaded from `http://people.cs.vt.edu/~jlinford/kppa`.

Initially we sought to extend KPP to support multi-core architectures, however its 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 two-dimensional design space, 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 software tool was developed from scratch.

In Figure 4, the analysis component formulates the chemical system with the same methods first implemented in KPP. 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.

The code generation component shown in Figure 4 accommodates a two-dimensional design space of programming language / target architecture combinations (Table 4.4). 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. It's key feature is the ability to generate fully-unrolled, platform-specific sparse matrix/matrix and matrix/vector operations which achieve very high levels

Table 4.4: Language/architecture combinations supported by KPPA. κ^* indicates functionality offered by existing tools.

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

of efficiency.

KPPA's code generation component is object-oriented to enable extensibility in multiple design dimensions (see Figure 5). Abstract base classes for Language objects and Architecture objects define the interfaces a Model object uses to produce an optimized chemical code. Adding a new language or architecture is often as simple as inheriting the abstract class and implementing the appropriate member functions and template files. Subclasses of the Model_Chem class define legal pairs of languages and architectures and override any Model_Chem methods that need special handling for that specific language/architecture pair. In practice, very few methods need special handling. Most of the code found in Model_Chem subclasses pertains to additional files that need to be generated, for example, math library wrappers or device-specific utility functions.

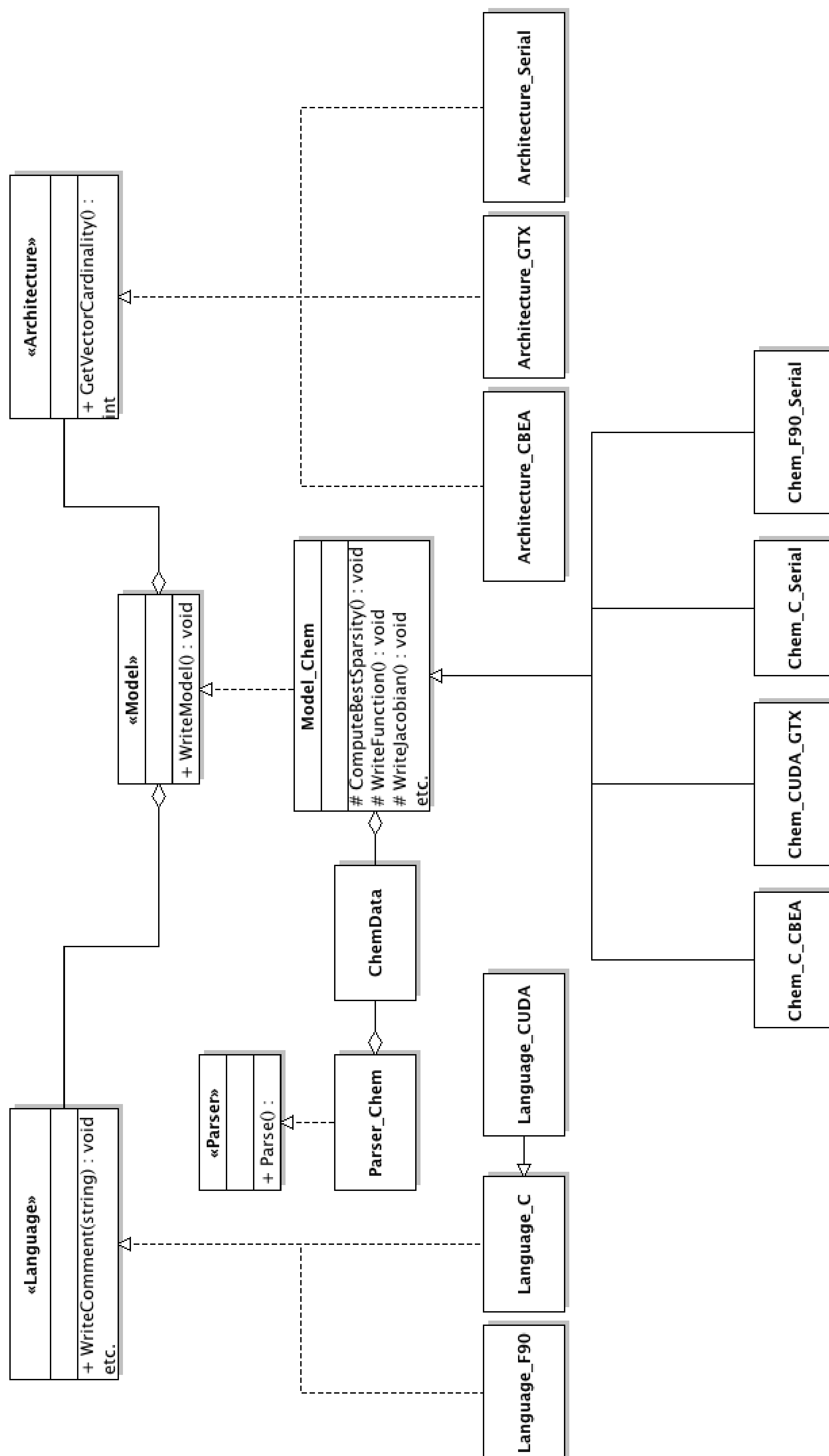


Figure 4.5: Abbreviated UML diagram of KPPA. Abstract base classes allow future languages and architectures to be added. Subclasses of the `Model_Chem` class define legal pairs of languages and architectures.

4.4.1 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 run-time. 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. A Language object 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% of the function implementation. Pipeline stalls account for less than 1% of the cycles required to calculate the function. For example, in the SAPRCNOV mechanism on CBEA, there are only 20 stalls in the 2989 cycles required by the ODE function (0.66%), and only 24 stalls in the 5490 cycles required for the ODE Jacobian (0.43%). Clever use of the `volatile` qualifier can reduce the number of cycles to 2778 and eliminate stalls entirely.

Figure 6 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 Figure 6). When the pipelines achieve 100% 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 it's associated BLAS wrapper functions, were easily converted to templates and added to the KPPA library. Boilerplate code for platform-specific multi-core communication and synchronization was also imported from the RADM2 implementations.

Template file specification enables the rapid reuse of pre-tested and debugged code, and

Figure 4.6: 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 2778 cycles. The pattern repeats with minor variations for 2116 of 2778 cycles.

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 straight-forward. Another alternative is to generate code from the original C or Fortran templates and then link against language-specific codes, if supported.

4.4.2 Multi-core 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 [48]. 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 4.3 show that only four parameters are required to generate a chemical mechanism with multi-layered heterogeneous parallelism (see Table 4.5). 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

Table 4.5: Parameterizations of three multi-core platforms in single (SP) 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

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 it's 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 process. KPPA uses this information to generate multi-buffering and prefetching code. For an SPE process on the CBEA, the scratch size is equivalent to the size of SPE local storage. CUDA devices share 16KB 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.

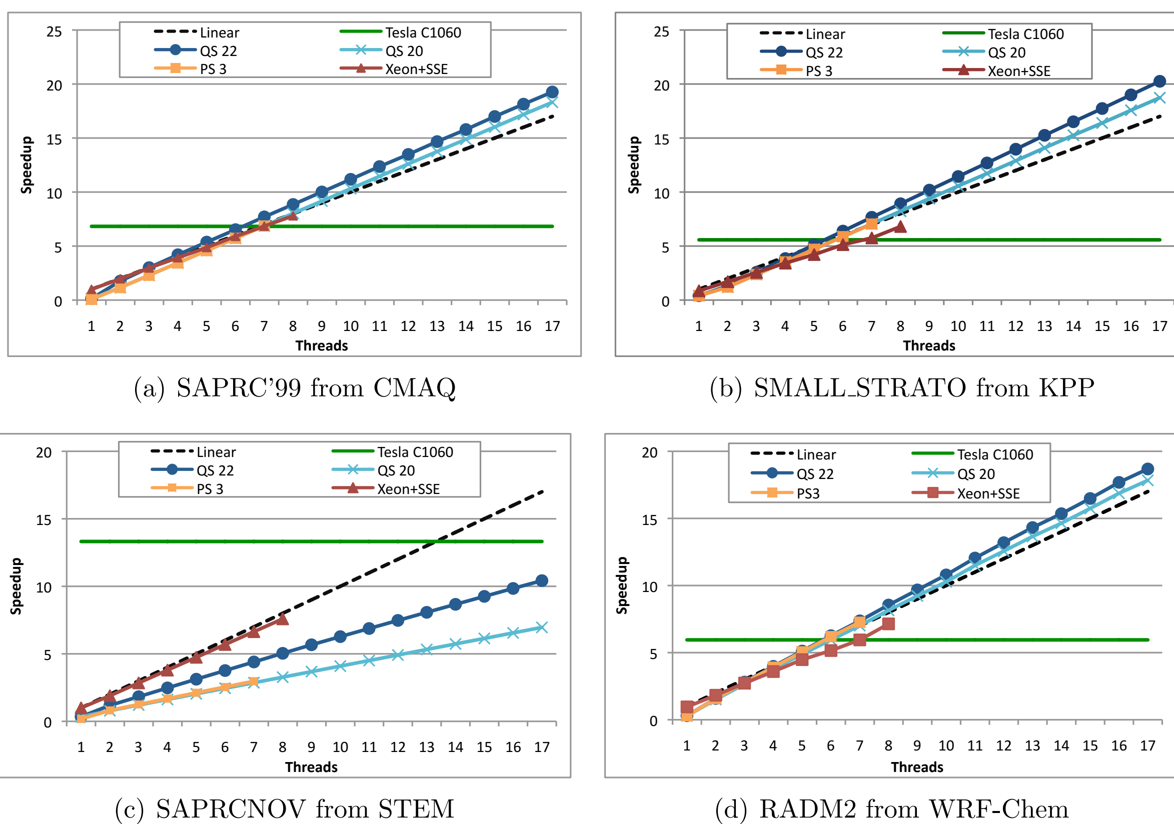
4.4.3 KPPA-Generated Mechanism Performance

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 4.3. The kernels were applied on the coarse domain grid from Section 4.3 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 [131] 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× faster than the state-of-the-art serial codes. SAPRCNOV requires double precision calculations, which limited the speedup to only 1.2×. It may be possible to improve on these speedups by using more advanced SSE intrinsics.

The Community Multiscale Air Quality Model (CMAQ) [31] 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 [33] 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 Figure 4.8(a). The CBEA implementation of SAPRC'99 achieves the highest speedup of 38.6× when compared to the state-of-the-art production code, or 19.3× 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 it's 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× (6.8× compared to SSE) due to high memory latencies and a shortage of on-chip fast shared memory, just as described in Section 4.3.

SMALL_STRATO (Figure 4.8(b)) is a stratospheric mechanism with 7 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. It's program text and over



(a) SAPRC'99 from CMAQ

(b) SMALL_STRATO from KPP

(c) SAPRCNOV from STEM

(d) RADM2 from WRF-Chem

Figure 4.7: Speedup of KPPA-generated chemical kernels as compared to SSE-enhanced serial code. A horizontal 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.

100 grid points can be held in a single SPE's local store, resulting in a maximum speedup of 40.7× on the CBEA (20.5× compared to the SSE-enhanced serial implementation). On the GPU, even though this mechanism is exceedingly small, it still cannot fit into the 64KB 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× (4.4× compared to SSE). NVIDIA's new Fermi architecture [132] 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 225KB large, and each grid cell comprises over 19KB of data. Figure 4.8(c) shows this mechanism's performance. The Quad-Core Xeon system, with it's 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× speedup (8.7× compared to the SSE-enhanced serial implementation). The GPU achieves the highest speedup of 13.3× (11.1× compared to SSE).

The performance of the KPPA-generated RADM2 kernel was similar to the performance of the hand-tuned code discussed in Section 4.3. 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.

4.4.4 Future Work

Future work will concentrate on alternate numerical methods, frameworks for calling KPPA-generated mechanisms remotely, and extending KPPA to other multi-core 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 [72], may be appropriate if a high level of accuracy is not required. Installing multi-core chipsets as accelerators in traditional clusters is viable [47], however not every cluster can be easily upgraded. Clusters of ASIC nodes, such as IBM BlueGene [115], 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 multi-core chipsets by offloading computationally-intense kernels to remote systems. If a kernel is 20× or 30× faster on a specific system, as demonstrated in this work, then the overhead of a remote

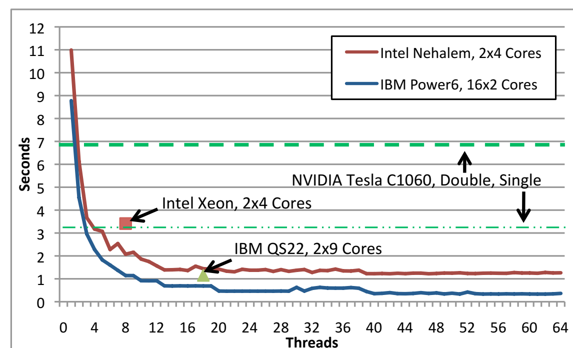


Figure 4.8: Wall clock OpenMP performance of the RADM2 chemical mechanism on two cutting edge homogeneous multi-core chipsets compared with emerging multi-core architectures.

procedure call may be acceptable, even for large data sets. We are investigating existing solutions, such as IBM Dynamic Application Virtualization [96], and will extend KPPA to generate model interfaces for remote kernels.

The recent acceptance of the OpenCL standard [90] 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 [56] (formally RapidMind) is also a future work.

Chapter Summary

KPPA (KPP: Accelerated) is a general analysis tool and code generator for serial, homogeneous multi-core, and heterogeneous multi-core 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 multi-core 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×, SAPRC'99 from CMAQ achieves 38.6× speedup, SAPRCNOV achieves 8.2× speedup,

[2]Additional information on the multi-core RADM2 kernel with updated results and codes is maintained at `http://www.mmm.ucar.edu/wrf/WG2/GPU`.

and SMALL_STRATTO achieves 28.1× 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 200KB, however most atmospheric chemical kernels will fit within this envelope.

The results and analysis presented here are a snapshot in time; for example, newer versions of homogeneous multi-core processors (e.g. Intel's i7, IBM's Power6, and others) are already showing significant improvement in speed and multi-core efficiency over their previous generations (see Figure 8). 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 multi-core architectures will be important gating factors for climate-chemistry, air-quality, wild-fire, and other earth science simulation in the coming decade.

Chapter 5

Multi-core Simulation of Atmospheric Chemical Transport

Chemical transport modules (CTMs) solve mass balance equations for concentrations of trace species in order to determine the fate of pollutants in the atmosphere [98]. The mass balance equations are generally separable through operator splitting; for example dimension splitting separates calculation of north-south transport from east-west transport. Both implicit and explicit methods are appropriate. In a comprehensive model, the transport component (the CTM itself) may be responsible for as much as 30% of the computational time. When chemistry is ignored, transport is often responsible for over 90% of the computational time.

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

The experimental results show that function offloading is a good method for porting existing scientific codes, but it may not achieve maximum performance. Vector stream processing produces better scalability for computationally-intense constituent transport simulation. By using vector stream processing to leverage the polymorphic parallelism of the CBEA, the

3D transport module achieves the same performance as two nodes of an IBM BlueGene/P supercomputer, or eight cores of an Intel Xeon homogeneous multi-core system, with one CBEA chip. A version of this chapter was published as [10]

5.1 The Chemical Transport Model

The one-dimensional transport equation in the x dimension is

$$\frac{\partial \mathbf{c}}{\partial t} = -u\frac{\partial \mathbf{c}}{\partial x} + \frac{1}{\rho}\frac{\partial}{\partial x}\left(\rho K \frac{\partial \mathbf{c}}{\partial x}\right) \tag{5.1}$$

where u is the horizontal component of the wind vector only. By considering a simple space discretization on a uniform grid ($x_i = ih$) with a mesh width $h = \frac{1}{m}$, solutions to Equation 5.1 can be approximated by finite difference methods [65]. This produces the third order upwind-biased horizontal advection discretization

$$-\left(u\frac{\partial \mathbf{c}}{\partial x}\right)\Bigg|_{x=x_i} = \begin{cases} u_i\left(\dfrac{-\mathbf{c}_{i-2} + 6\mathbf{c}_{i-1} - 3\mathbf{c}_i - 2\mathbf{c}_{i+1}}{6\Delta x}\right) & \text{if } u_i \geq 0 \\ u_i\left(\dfrac{2\mathbf{c}_{i-1} + 3\mathbf{c}_i - 6\mathbf{c}_{i+1} + \mathbf{c}_{i+2}}{6\Delta x}\right) & \text{if } u_i < 0 \end{cases} \tag{5.2}$$

and the second order central difference diffusion discretization

$$\frac{1}{\rho}\frac{\partial}{\partial x}\left(\rho K\frac{\partial \mathbf{c}}{\partial x}\right)\Bigg|_{x=x_i} = \frac{(\rho_{i+1}K_{i+1} + \rho_i K_i)(\mathbf{c}_{i+1} - \mathbf{c}_i) - (\rho_i K_i + \rho_{i-1}K_{i-1})(\mathbf{c}_i - \mathbf{c}_{i-1})}{2\rho_i \Delta x^2}. \tag{5.3}$$

The vertical convective term is discretized by first order upwind-biased finite differences on a non-uniform grid

$$-\left(w\frac{\partial \mathbf{c}}{\partial z}\right)\Bigg|_{z=z_i} = \begin{cases} -w_i\left(\dfrac{\mathbf{c}_i - \mathbf{c}_{i-1}}{z_i - z_{i-1}}\right) & \text{if } w_i \geq 0 \\ -w_i\left(\dfrac{\mathbf{c}_{i+1} - \mathbf{c}_i}{z_{i+1} - z_i}\right) & \text{if } w_i < 0 \end{cases} \tag{5.4}$$

where w is the vertical component of the wind vector. Vertical diffusion is also discretized by second order central differences

$$\frac{1}{\rho}\frac{\partial}{\partial z}\left(\rho K\frac{\partial \mathbf{c}}{\partial z}\right)\Bigg|_{z=z_i} = \frac{(\rho_{i+1}K_{i+1} + \rho_i K_i)(\mathbf{c}_{i+1} - \mathbf{c}_i) - (\rho_i K_i + \rho_{i-1}K_{i-1})(\mathbf{c}_i - \mathbf{c}_{i-1})}{2\rho_i \Delta z_i^2} \tag{5.5}$$

where Δz_i is the height of vertical layer i.

5.1.1 Boundary Conditions

First order differences can be used on the horizontal inflow and outflow boundaries to maintain quadratic order of consistency for the whole scheme on the interior points of the domain. Suppose $u_1 \leq 0$ so x_1 is an inflow boundary point with $\mathbf{c}_{IN}$ the corresponding (Dirichlet) boundary concentration. The inflow discretization is

$$\frac{\mathrm{d}\mathbf{c}_1}{\mathrm{d}t} = -u_1 \frac{\mathbf{c}_1 - \mathbf{c}_{IN}}{\Delta x} + \frac{(\rho_2 K_2 + \rho_1 K_1)(\mathbf{c}_2 - \mathbf{c}_1) - (3\rho_1 K_1 - \rho_2 K_2)(\mathbf{c}_1 - \mathbf{c}_{IN})}{2\rho_1 \Delta x^2}. \tag{5.6}$$

Suppose $u_N > 0$ so x_N is an outflow boundary point. In this case, the (Neumann) boundary condition of zero diffusive flux across the outflow boundary gives the discretization

$$\frac{\mathrm{d}\mathbf{c}_N}{\mathrm{d}t} = -u_N \frac{\mathbf{c}_N - \mathbf{c}_{N-1}}{\Delta x} - \frac{(\rho_N K_N + \rho_{N-1} K_{N-1})(\mathbf{c}_N - \mathbf{c}_{N-1})}{2\rho_N \Delta x^2}. \tag{5.7}$$

Similar to horizontal inflow and outflow boundaries, the vertical top boundary condition is Dirichlet for inflow and Neumann for outflow. The ground boundary condition considers the flow of material given by surface emission rates $\mathbf{Q}$ and by deposition processes with velocity $\mathbf{V}^{dep}$. Hence the ground boundary condition is

$$-K \frac{\partial \mathbf{c}}{\partial z}\bigg|_{z=0} = \mathbf{Q} - \mathbf{V}^{dep} \mathbf{c}^T \tag{5.8}$$

and is discretized as

$$\frac{\mathrm{d}\mathbf{c}_1}{\mathrm{d}t} = \frac{(\rho_2 K_2 + \rho_1 K_1)(\mathbf{c}_2 - \mathbf{c}_1)}{2\rho_1 (z_2 - z_1)\Delta z_1} - \frac{\mathbf{V}^{dep}\mathbf{c}_1^T - \mathbf{Q}}{\Delta z_1} \tag{5.9}$$

where Δz_1 is the height of the first layer.

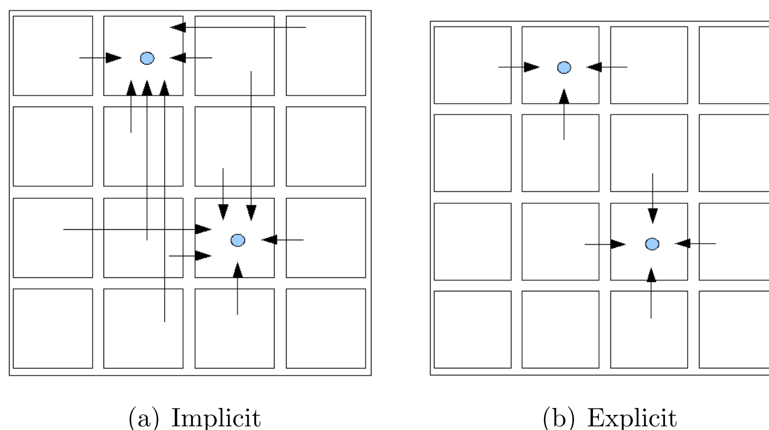
5.2 Solving the Transport Equations In Parallel [11]

The horizontal (Equation 5.2) and vertical (Equation 5.4) space semi-discretizations lead to linear ordinary differential equations. In the horizontal case

$$\frac{\mathrm{d}\mathbf{c}}{\mathrm{d}t} = A(t)\mathbf{c} + \mathbf{b}(t) \tag{5.10}$$

where matrix A(t) depends on the wind field, the diffusion tensor, and the air density but not the unknown concentrations. The vector $\mathbf{b}(t)$ represents the Dirichlet boundary conditions. In the vertical case

$$\frac{\mathrm{d}\mathbf{c}}{\mathrm{d}t} = A(t)\mathbf{c} + B(t)e_N + \frac{Q(t)}{\Delta z_1} e_1 \tag{5.11}$$



(a) Implicit (b) Explicit

Figure 5.1: Data dependencies for implicit and explicit time-stepping.

where the entry $A_{1,1}$ also accounts for the deposition velocity, B(t) accounts for the top boundary and Q(t) accounts for ground emissions. e_j is the jth column of the identity matrix. The solution is found by solving the pentadiagonal matrices and systems, which can be done very efficiently with an appropriate domain decomposition scheme. Both implicit and explicit iterative methods are appropriate.

The data dependencies of implicit and explicit methods must be considered when implementing the transport model (Figure 1). The horizontal spatial discretization uses a finite difference scheme with a stencil of $2w + 1$ grid points. In a two-dimensional decomposition, the calculation of the rate of change of the concentrations due to transport in the grid cell (i, j) depends on the concentrations in the neighbors $(i \pm k, j)$ and $(i, j \pm k)$ for $k = 0, 1, \cdots, w$. Dimension splitting separates transport calculation along the X-axis from transport calculation along the Y-axis.

An implicit time stepping scheme leads to strong data dependencies. In calculating transport along the X-axis, the concentration at (i, j) depends on all grid points in the ith row $(i, 1)$ through (i, NY). Similarly for the Y-axis, the concentration depends on all grid points in the jth column $(1, j)$ through (NX, j). Implicit time stepping schemes have the advantage of unconditional numerical stability. The simulation can be carried out with very large time steps, the only restrictions being imposed by the accuracy of the solution. On the other hand, implicit time stepping leads to a large domain of dependency, which negatively impacts parallelization.

An explicit time stepping scheme leads to relaxed data dependencies. During each stage of the explicit calculation the cell values within the stencil $(i \pm k, j)$ and $(i, j \pm k)$ for $k = 0, 1, \cdots, w$ are used. The domain of dependency of cell (i, j) includes sw neighbors in the north, south, east, and west directions, where w is the method half stencil and s is the number of stages of the time integration. Because the domain of dependency includes only a

small number of neighbors, explicit schemes are amenable to efficient parallelization. On the other hand they are stable only for a restricted range of step sizes; if the underlying problem is stiff the step size restrictions may prove severe and considerably degrade the efficiency of the simulation.

5.2.1 Implicit Crank-Nicholson Methods

The implicit Crank-Nicholson method advances the system forward in time from t^n to $t^{n+1} = t^n + \Delta t$. For horizontal transport the forward model reads

$$c^{n+1} = \left(I - \frac{\Delta t}{2} A(t^{n+1})\right)^{-1} \left[\left(I + \frac{\Delta t}{2}(t^n)\right) c^n + \Delta t \frac{B(t^n) + B(t^{n+1})}{2}\right]. \tag{5.12}$$

In the vertical case, forward Euler timestepping is used for the boundaries and the ground emissions and the forward model reads

$$c^{n+1} = \left(I - \frac{\Delta t}{2} A(t^{n+1})\right)^{-1} \left[\left(I + \frac{\Delta t}{2}(t^n)\right) c^n + \Delta t \left(B(t^n) e_N + \frac{Q(t^n)}{\Delta z_1} e_1\right)\right]. \tag{5.13}$$

Because of their unconditional numerical stability, implicit methods have been the standard approach for transport calculation for much of the last thirty years. They are not reviewed here. A good implementation is given in [89] where an *H-V partitioning* parallelization strategy for implicit time-stepping (Crank-Nicholson) based on partitioning the 4-D domain data in horizontal and vertical slices is implemented. During one stage of the computation the data is distributed in horizontal (H) slices; each processor can compute the horizontal (X and Y) transport in each horizontal slice. In the other stage of the computation the data is distributed in vertical (V) columns; each processor performs the radiation, Z-transport, chemistry, emission and deposition calculations in a vertical column. At each time iteration the data needs to be shuffled from H slices to V columns and back. This involves a total data exchange at each step; the volume of data transmitted is $NX \times NY \times NZ \times NS$ words.

5.2.2 Explicit Runge-Kutta-Chebyshev Methods

Explicit time-stepping methods are now a popular alternative to the implicit approach whenever massive hardware parallelism is available. When using the explicit method a *tiled partitioning* approach is possible. The horizontal dimension of the domain is divided in $m \times n$ tiles. Here $mn = p$ (the number of threads or processes). Each tile contains $NX/m \times NY/n \times NZ$ grid cells. Advancing the concentrations in a tile requires data from the immediate non-diagonal neighboring tiles. The data exchanged between neighbors at every iteration are of size $s \times w \times NT \times NZ \times NS$ where NT is the width or height of a tile, w is the width of the ghost cell (half stencil of the spatial discretization) and s is the number of stages in the

time stepping method (hence the number of times the exchange has to be performed per iteration). The width or height of a tile decreases as p increases – if we have p processors and a decomposition of the domain into $\sqrt{p} \times \sqrt{p}$ tiles then the tile size is $NT \sim NX/\sqrt{p}$ or $NT \sim NY/\sqrt{p}$.

The volume of data exchanged at each step is significantly smaller than that of an H-V partitioning since $w \ll NX$ (stencil much smaller than the domain), $WT \ll NY$ for a large number of processors, and $s = \mathcal{O}(1)$. Moreover, in the tiled decomposition the communication takes place only among the neighboring processors so a considerable overlap of communications between different processors in possible.

The second order explicit Runge-Kutta-Chebyshev method [73, 17, 116, 123] is reviewed here. For an ODE system,

$$w'(t) = F(t, w(t)) \quad t > 0, \quad w(0) = w_0$$

the second order explicit Runge-Kutta-Chebyshev method is defined as:

$$\begin{aligned} W_0 &= w_n \\ W_1 &= W_0 + \bar{\mu}_1 F_0 \\ W_j &= (1 - \mu_j - \nu_j)W_0 + \mu_j W_{j-1} + \nu_j W_{j-2} + \bar{\mu}_j \tau F_{j-1} + \bar{\gamma}_j \tau F_0 \\ w_{n+1} &= W_s \end{aligned} \tag{5.14}$$

where $j = 2...s$, $F_k = F(t_n - c_k\tau, W_k)$, w_n is the solution vector (concentrations) at t_n, and w_j are the internal stage solutions of the method.

The parameters for the formula are set so that for each stage vector $W_j = P_j(z)W_0$ the stability functions $P_j(z)$ [73, 17, 116] are nearly optimal for parabolic problems, the method is internally stable, and it has second order consistency for all $W_j(2 \leq j \leq s)$. This is achieved by using the first kind Chebyshev polynomial, satisfying the three-term recurrence relation

$$T_j(x) = 2xT_{j-1}(x) - T_{j-2}(x), j = 2, 3, ..., s \tag{5.15}$$

where $T_0(x) = 1$ $T_1(x) = x$.

Hence $P_j(z)$ (in the relation $W_j = P_j(z)W_0$) becomes,

$$P_j(z) = a_j + b_j T_j(\omega_0 + \omega_1 z) \tag{5.16}$$

where,

$$\begin{aligned} a_j &= 1 - b_j T_j(\omega_0) \\ b_0 &= b_2 \\ b_1 &= \frac{1}{\omega_0} \\ b_j &= T_j''(\omega_0)/(T_j'(\omega_0))^2, j = 2...s \\ \omega_0 &= 1 + \epsilon/s^2 \\ \omega_1 &= T_s'(\omega_0)/T_s''(\omega_0) \end{aligned}$$

Here, $\epsilon \geq 0$ is called a damping parameter. $\epsilon > 0$ can be varied to control the stability region along the negative real axis. Increasing $\epsilon > 0$ results in a wider strip of stability region along the negative real axis and vice versa.

The coefficients in Equation 5.14 for $j = 2 \ldots s$ are

$$\begin{aligned}
\bar{\mu_1} &= b_1\omega_1 \\
\mu_j &= \frac{2b_j\omega_0}{b_{j-1}} \\
\bar{\mu_j} &= \frac{2b_j\omega_1}{b_{j-1}} \\
\nu_j &= -\frac{b_j}{b_{j-2}} \\
\bar{\gamma_j} &= -a_{j-1}\bar{\mu_j}
\end{aligned}$$

Incrementing the values of s increases the extent of the stability region further left of the negative real axis.

The stability function $P_s(z)$ for the values of s are:

$$\begin{aligned}
P_2(z) &= \frac{899}{1681} + \frac{676}{1681}\left(2(\frac{27}{26} + \frac{41}{52}z)^2 - 1\right) \\
P_3(z) &= \frac{51427}{78400} + \frac{19773}{78400}\left(4(\frac{140}{351}z + \frac{27}{26})^3 - 3(\frac{140}{351}z + \frac{27}{26})\right) \\
P_5(z) &= .6552 + .2979\left(5(1.0062 + .1279z) - 20(1.0062 + .1279z)^3 + 16(1.0062 + .1279z)^5\right)
\end{aligned}$$

The stability regions shown in Figure 2 indicate that the maximum allowable time-step is approximately 10^4 seconds.

Performance of stabilized explicit time-stepping methods

The explicit time stepping scheme and the tiled decomposition strategy were implemented in the Sulfur Transport and dEposition Model (STEM) [32]. STEM is essentially a 3-D Eulerian model. It has been used to address a wide series of policy issues in U.S., Asia and Europe, related to acidification, cloud chemistry, tropospheric ozone, and aerosols formation.

The test problem used for numerical experiments is based on real meteorological and chemical conditions in East Asia for 24 hours on March 3, 2001. The data was collected in support of NASA's Trace-P experiment. The simulation is carried out on a computational grid with 90×60 horizontal grid points (at a resolution of $80{\times}80$ Km) and 18 vertical grid levels (of variable vertical resolution). The number of species the simulation was run for was 106. In this simulation there are over 10 million ($90{\times}60{\times}18{\times}106$) double precision variables that are computed at every time step, making the total size of the data to be 8.24×10^7 (bytes)

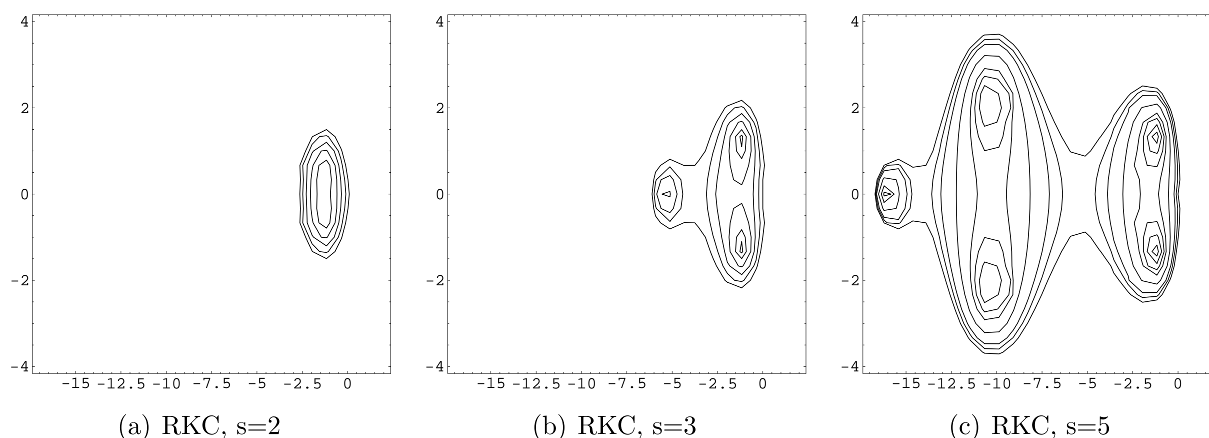


Figure 5.2: Stability regions of Runge-Kutta-Chebyshev explicit time-stepping method for three values of s.

Table 5.1: STEM execution times on System X (seconds) for 24 hours of simulation time.

p	RKC ($s = 5$)	RKC ($s = 3$)	RKC ($s = 2$)	Crank-Nicholson
1	14763	14448	14478	12985
8	1850	1806	1822	821
16	830	813	805	650
30	499	488	492	526
54	380	361	371	494
78	321	316	317	489
100	295	313	294	515

or approximately 82 MB. The results presented were obtained from the runs on System X [135], a cluster consisting of 1100 Apple Xserve G5 machines, with 4Gb of RAM per node connected with 4X InfiniBand core switches operating at 12.25 teraflops per second.

The parallel performance of the new tiled decomposition based on RKC explicit time stepping is compared against the HV partitioning of Miehe et al. [89] which supports Crank-Nicholson implicit time integration. The time steps were set to be the same for both the methods. The parallel runs were profiled using the Tuning and Analysis Utilities (TAU) [134], while the serial runs have been profiled using calls to `cpu_time()`. For each run we have measured the total execution time and the time spent in data distribution, communication, transport calculations and chemistry calculations.

The total execution times for the serial calculation and for parallel runs for up to 100 processors are shown in Table 5.1. For the serial code the total execution time is about the

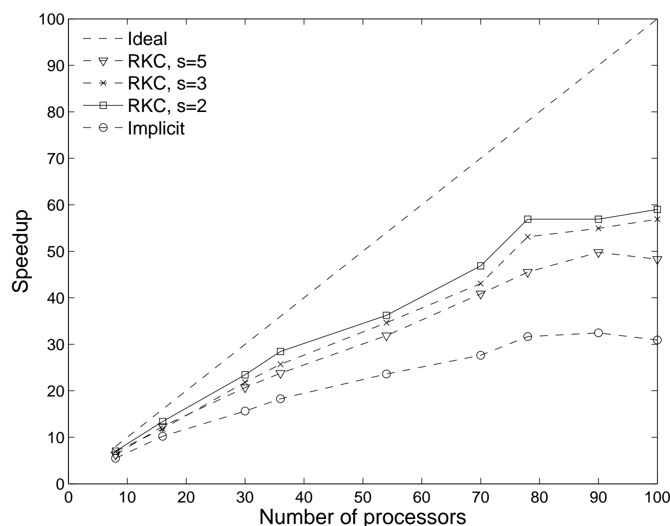


Figure 5.3: STEM speedup on System-X (RKC is parallel explicit, CN is parallel implicit)

same for both implicit and explicit time stepping approaches. The increase in the number of stages of the RKC method leads to only a small increase of the execution time. For large numbers of processors the total execution time is lower for the tiled parallelization based on RKC time stepping, due to the higher parallel efficiency allowed by this approach.

This conclusion is further substantiated by the speedup and efficiency curves shown in Figure 3. The performance of the HV decomposition with Crank-Nicholson time stepping starts to degrade above 80 processors. The execution time of the tiled decomposition with RKC time stepping continues to decrease for larger numbers of processors. A typical efficiency degradation for larger numbers of processors is observed due to the increase in the communication to computation ratio for this fixed problem size simulation. For the same number of processors the efficiency is slightly decreased with increasing number of RKC stages. This is also expected as one communication stage is needed for each RKC computational stage; more stages lead to more communication for only a very modest increase of the computational load.

There are s stages at every iteration of the RKC method, which implies exchanging ghost cells s times with the neighboring blocks at every iteration. Every time a ghost cell exchange takes place, the processors have to be synchronized. As observed from the speedup plots (Figure 3) the performance improves with lower values of s.

5.3 Fixedgrid

Fixedgrid is a comprehensive prototypical atmospheric model based the state-of-the-art

Sulfur Transport and dEposition Model version 3 (STEM-III) [32]. It is principally an experimental platform for geophysical modeling; its data sets are simplistic compared to production models, but the processes are the same. Properly-formatted real-world data may be used without issue. Mechanisms can be optimized and profiled in FIXEDGRID without confounding performance variables, and various programming methodologies can be reliably compared without rewriting a comprehensive model. It has been used successfully in several studies of emerging multi-core architectures [3, 7, 9, 12, 10].

FIXEDGRID describes chemical transport according to the equations in Section 5.1 and uses an implicit Rosenbrock method to integrate a 79-species SAPRC'99 [33] atmospheric chemical mechanism for VOCs and NOx (see Chapter 4). Chemistry and transport processes can be selectively disabled to observe their effect on monitored concentrations, and FIXEDGRID's modular design makes it easy to introduce new methods. 2D and 3D transport kernels allow FIXEDGRID to address a range of mass-balance problems.

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

Data organization in memory is critically important. Accelerator architectures derive much of their strength from SIMD operations and/or asynchronous explicit memory transfers. For these operations to succeed, the data must conform to the architecture requirements. When FIXEDGRID executes on the CBEA, data transferred between SPE local storage and main memory must be contiguous, 8-byte aligned, no larger than 16 KB, and in blocks of 1, 2, 4, 8, or multiples of 16 bytes. Transfers of less than 16 bytes are highly inefficient. To meet these requirements in three dimensions, each row of the data matrices is statically padded to a 16-byte boundary, effectively adding buffer y-z planes (see Figure 6) and aligned with `__attribute__((aligned(128)))`. Two-dimensional FIXEDGRID simulations are padded in the same way, but with a vector instead of a plane ($z = 1$).

5.3.1 Chemical Transport in FIXEDGRID

FIXEDGRUD uses *dimension splitting* to apply the transport equations to each dimension of a d-dimensional model independently. Parallelism is introduced by reducing a d-dimensional problem to a set of independent one-dimensional problems. Equations 5.2, 5.3 and 5.5 can then be implemented as individual routines and applied to each dimension of the concentration matrix individually and in parallel. This forms the *kernel* functions for use in streaming architectures. Dimension splitting is common in many transport models and other scientific applications. The 3D stencil for upwind-biased discretization is shown in Figure 4.

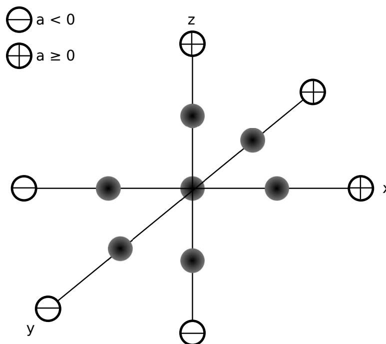


Figure 5.4: 3D discretization stencil for explicit upwind-biased advection/diffusion

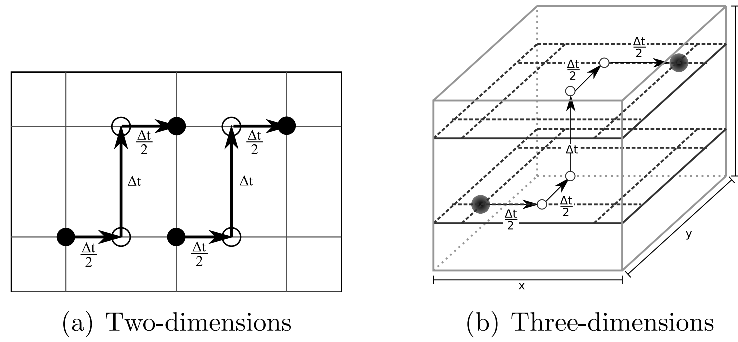
Dimension splitting introduces a local truncation error at each time step. This error can be reduced by a symmetric time splitting method [65]. Time splitting methods are frequently used when applying different time stepping methods to different parts of an equation. For example, chemical processes are stiff, which calls for an implicit ODE method, but explicit methods are more suitable for space-discretized advection. By interleaving time steps taken in each dimension, the truncation error is reduced. In brief, a linear ODE describing time-stepped chemical transport, $w'(t) = Aw(t)$ where $A = A_1 + A_2 + A_3$, can be approximated by

$$w_{n+1} = e^{\Delta t A} w_n \approx e^{\frac{\Delta t}{2} A_3} e^{\frac{\Delta t}{2} A_2} e^{\Delta t A_1} e^{\frac{\Delta t}{2} A_2} e^{\frac{\Delta t}{2} A_3} w_n, \tag{5.17}$$

where $e^B = I + B + \frac{1}{2}B^2 + \ldots + \frac{1}{k!}B^k + \ldots$ is the exponential function of a matrix, and Δt is the timestep. This is a multi-component second order Strang splitting method [113]. For three-dimensional discretization, take A_1, A_2, A_3 to be representative of discretization along the z, y, and x axes, respectively, or for two-dimensional discretization, take $A_3 = [0]$ and A_2, A_1 to be representative of discretization along the y and x axes, respectively. Figure 5 illustrates time splitting applied to 2D and 3D discretization. A half time step ($\frac{\Delta t}{2}$) is used when calculating mass flux along the minor axes, and a whole time step is used when calculating mass flux along the major axis. This process is equal to calculating the mass flux in one step, but reduces truncation error to $\mathbf{O}(\Delta t^3)$ [65].

Depending on the underlying architecture, single- or double-precision data types may be more efficient. Some accelerator architectures, such as 1st generation NVIDIA CUDA GPUs, support only single-precision floating-point arithmetic. To address this, FIXEDGRID uses C preprocessor macros to statically refactor the code for the most appropriate floating point data type.

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



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

Figure 5.5: Second order time splitting methods for 2D and 3D transport discretization.

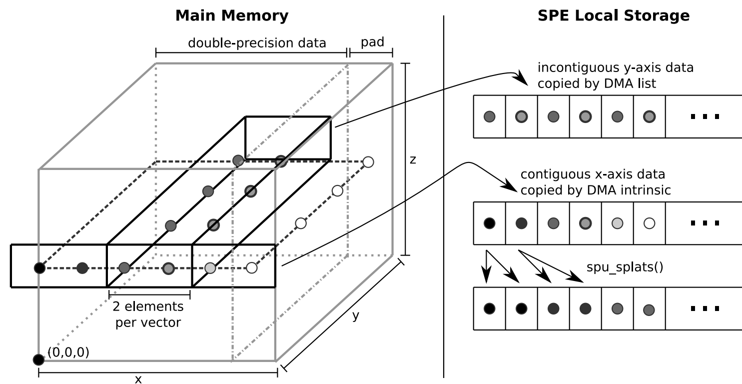


Figure 5.6: FIXEDGRID data storage scheme for the 3D transport module showing padding and vectorization with double-precision data types

extracting the desired element from the buffer. Preprocessor macros can be used to calculate basic vector metrics (i.e. number of elements per 16-byte vector) and statically support both 4-byte and 8-byte floating point types with a clear, simple syntax.

FIXEDGRID uses second order time splitting to reduce truncation error in 2D/3D transport (see Figure 5). A half time step ($\frac{\Delta t}{2}$) is used when calculating mass flux along the domain minor axes, and a whole time step is used when calculating mass flux along the domain major axis. Note that this reduces the truncation error but doubles the work required to calculate mass flux for the minor axes. It is therefore desirable for 2D row and 3D x/y-discretization to be highly efficient. Concentrations of the same chemical species are stored contiguously to maximize locality and avoid buffering in transport calculations.

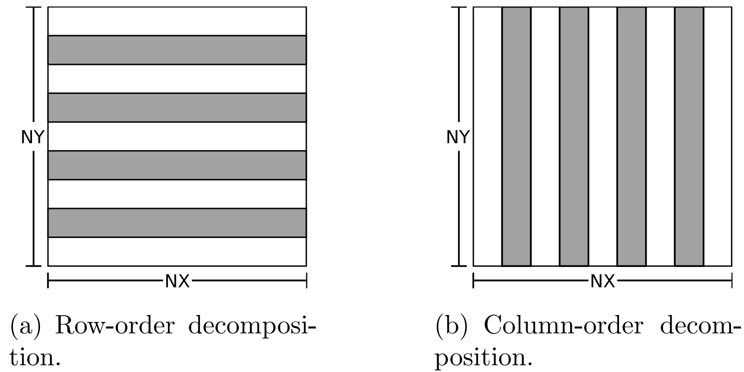
5.4 Function Offloading for Transport [7]

This section discusses the implementation, analysis, and optimization of a 2D FIXEDGRID transport model for for the CBEA using a *function offload* approach. The benefits of known optimization techniques are quantified and a novel method for general incontiguous data access is introduced and quantified. FIXEDGRID calculates transport in the concentration matrix in rows (discretization along the x-axis) and in columns (discretization along the y-axis), as shown in Figure 7. The 2D transport module was ported from the original Fortran, rather than written it from scratch, as an exercise in porting an existing scientific code to the CBEA. This was a nontrivial effort and resulted in three new versions of the transport module. Each new version was carefully profiled and analyzed to gage the benefit of the changes introduced. Performance is reported in Section 5.4.4.

5.4.1 Naive Function Offload

A *function offloading* approach was used to accelerate the 2D transport mechanism. Function offloading identifies computationally-intense functions and moves these functions to the SPEs. The main computational core for 2D transport is the `discretize()` function, which calculates mass flux in a single row or column. A naive offload was applied to move the unmodified function code to the SPEs while ignoring the architecture details of the SPE, such as the SPE's SIMD ISA and support for overlapping computation and communication. When `discretize()` is invoked, each SPE receives the main-memory address of a `fixedgrid_spe_argv_t` structure (Listing 5.1) containing the parameters to the `discretize()` function. The SPE fetches the parameters via DMA. `fixedgrid_spe_argv_t` is exactly 128 bytes long and aligned on a 128-byte boundary to maximize transfer speed. The PPU signals the SPE to invoke the offloaded function via mailbox registers.

The only memory available to the SPE directly is its own 256KB of local storage, which is shared between code and data. Considering code size and neglecting overlays, approximately



(a) Row-order decomposition.

(b) Column-order decomposition.

Figure 5.7: Domain decomposition for two-dimensional transport.

Listing 5.1: User defined data types for passing arguments to the SPE program

```
1  typedef union
2  {
3    double   dbl;
4    uint64_t u64;
5    uint32_t u32[2];
6  } arg_t;
7
8  typedef struct
9  {
10   arg_t arg[16];
11 } fixedgrid_spe_argv_t;
```

2000 double-precision variables can be held in SPE local storage. Any data exceeding this limit is processed piecewise. Data in main memory must be contiguous and aligned on a 16-byte boundary to be eligible for DMA transfer. Column data in the concentration matrix is problematic for the CBEA, since it is incontiguous and odd-indexed columns (for double-precision data) are never on a 16-byte boundary.

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

5.4.2 Improved Function Offload

Naive function offload neglects the SPE's SIMD ISA and asynchronous memory I/O capabilities. Offloaded function performance can be improved through triple-buffered asynchronous DMA, loop unrolling, and SIMD intrinsics. Tripple-buffered DMA enables each SPE to simultaneously load the next row or column into local storage, process the current row or column, and write previously-processed data back to main memory. Six buffers are required to triple-buffer incontiguous data: three in local storage and three for data reorganization in main memory by the PPE. Only three local storage buffers are required for contiguous data. Asynchronous DMA was used in the 2D transport function to reduce runtime by approximately 23%.

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

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

The byte-size of a data element must be carefully considered when optimizing the offloaded function. If single-precision floating point data is used, a vector on the SPU contains four elements, as opposed to just two for double-precision floating point data. Since data must be aligned on a 16-byte boundary to be eligible for DMA transfer, retrieving column elements of the concentration matrix which are not multiples of the vector size will result in a bus

error. Thus, two/four columns of data must be retrieved when using single/double precision vector intrinsics. This reduces runtime by increasing SPE bandwidth, but puts additional strain on the SPE local storage.

5.4.3 Scalable Incontiguous DMA

Incontiguous data can be transferred directly to SPE local storage via *DMA lists*. A DMA list is an array of main memory addresses and transfer sizes, stored as pairs of unsigned integers, which resides in SPE local storage. Each element describes a DMA transfer, so the addresses must be aligned on a 16-byte boundary and the transfer size may not exceed 16KB. DMA lists were originally perceived as a mechanism for surpassing the 16KB transfer limit imposed by the MFC hardware, and most literature and tutorials introduce them as such. Here we use DMA lists to significantly increase the scalability of incontiguous data access with only a linear increase in storage consumption while simultaneously freeing the SPU to process data.

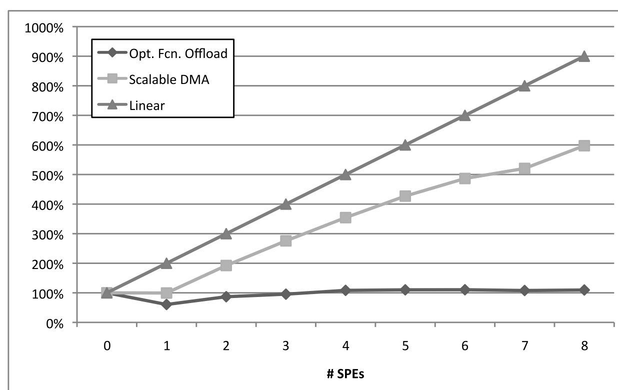
To transfer a column of y elements in the concentration matrix requires a DMA list of y entries. The same DMA list can be used for both writing to and reading from main memory. Single-/double-precision variables are four/eight bytes long, so at least four/two columns should be fetched simultaneously, even if only one column is desired. When the DMA list is processed, the two columns are copied to the same buffer in local storage with their elements interleaved. The desired column is then extracted by the SPE from the local storage and fed into the offloaded function. Scalability is greatly improved by moving the array copy loops to the SPEs, but $\mathbf{O}(y \times 2 \times 3)$ additional local storage elements are required. However, no additional buffers are required in main-memory.

Figure 8 shows how DMA lists improve 2D transport performance. "Row discret" and "Col discret" give the inclusive function time for transport calculations in the rows/columns of the concentration matrix, respectively. In Figure 5.9(b), "Row discret" is clearly decreasing as more SPEs are used, yet "Col discret" remains relatively constant. When DMA lists are used (Figure 5.9(c)) both "Col discret" and "Row discret" decrease as more SPEs are used.

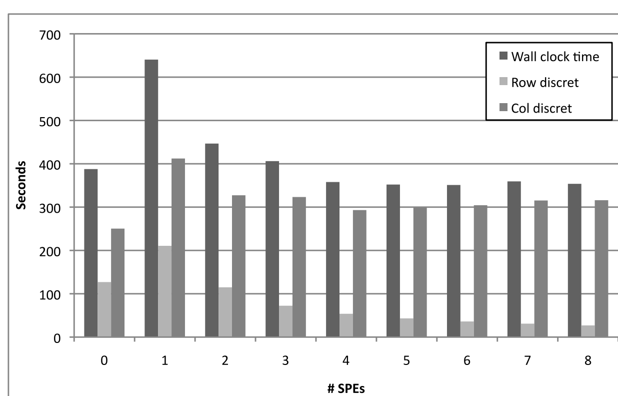
5.4.4 Analysis and Comparison

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

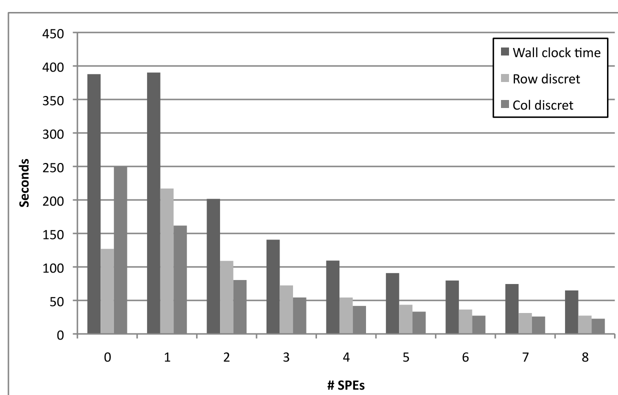
JUICEnext: a 35-node IBM BladeCenter QS22 at Forschungszentrum Jülich. Each node has 8GB XDRAM and four 3.2GHz PowerXCell 8i CPUs with eight SPEs each.



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



(b) Inclusive function times for optimized function offload.



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

Figure 5.8: Performance of the 2D transport module executing on JUICEnext (IBM BladeCenter Q22) when matrix columns are buffered by the PPU (Opt. Fcn. Offload) and when DMA lists are used (Scalable DMA)

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

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

The performance of the CBEA systems is compared with the performance of two homogeneous multi-core systems:

Jugene: a 16-rack IBM BlueGene/P at Forschungszentrum Jülich.[1] Each rack contains 32 nodecards, each with 32 nodes. Each node contains an 850MHz 4-way SMP 32-bit PowerPC 450 CPU and 2GB DDR RAM (512MB RAM per core). Jugene premiered as the second fastest computer on the Top 500 list and presently ranks in 6th position.

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

The experiments were compiled with the GCC on all CBEA platforms. IBM XLC was used on Jugene, and the Intel C Compiler (ICC) was used on Deva. All experiments were compiled with maximum optimization (-O5 or -O3 where appropriate) and double-precision floating point operations. The presented results are the average of three runs on each platform. The measurement error was insignificant, so error bars are not shown.

Tables 5.2 and 5.3 list the wall clock runtime performance on the CBEA systems and homogeneous multi-core systems, respectively. These experimental results show the importance of leveraging all levels of heterogeneous parallelism in the CBEA. Initial versions of the 2D transport module, which make little or no use of heterogeneous parallelism, perform poorly in comparison to the final version, which uses asynchronous memory I/O and SIMD. It is clear that the PPU can be a severe scalability bottleneck if it preprocesses for the SPEs. It is more efficient to to use methods which can be offloaded to the SPEs, even though this introduces additional copy operations and consumes SPE local storage. Figure 9 shows the wall clock runtime performance and speedup of the final version of FIXEDGRID's 2D transport module on all five systems.

The CBEA is intended for compute-intensive workloads with many fine-grained and course-grained parallel operations. By porting the original Fortran code, we were unable to fully exploit every level of heterogeneous parallelism in the CBEA. Furthermore, 2D transport in the chosen domain size does not challenge the capabilities of the test systems. We believe this explains the low scalability of the 2D transport module, compared to the homogeneous systems. The analysis of the 3D transport module in Section 5.5.3 corroborates this theory.

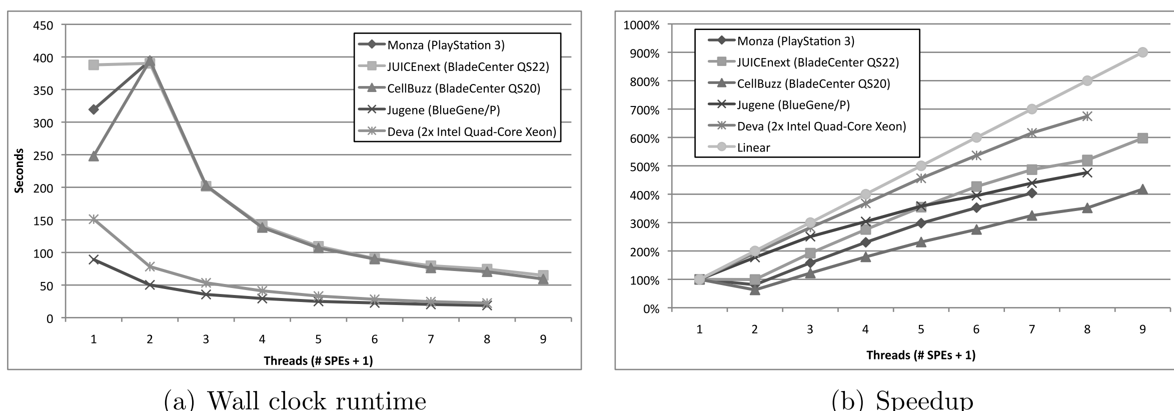
[1]We classify the BlueGene/P as a homogeneous multi-core system, even though it may be considered a heterogeneous multi-core system. The head nodes and "Double Hummer" dual FPUs introduce a degree of heterogeneity, however the BlueGene/P is homogeneous on the node-level.

Monza (PlayStation 3 Cluster)						
	Naïve Fcn. Offload		Opt. Fcn. Offload		Scalable DMA	
# SPEs	Wall clock	% Speedup	Wall clock	% Speedup	Wall clock	% Speedup
1	**2324.197**	13.74	677.282	47.16	394.100	81.04
2	1165.798	27.40	369.438	86.45	201.853	158.23
3	797.546	40.05	291.917	109.41	138.606	230.43
4	627.608	50.89	269.018	118.72	107.201	297.93
5	553.026	57.75	284.881	112.11	90.589	352.57
6	654.322	48.81	275.304	116.01	79.043	404.07
CellBuzz (IBM BladeCenter QS20)						
	Naïve Fcn. Offload		Opt. Fcn. Offload		Scalable DMA	
# SPEs	Wall clock	% Speedup	Wall clock	% Speedup	Wall clock	% Speedup
1	2066.130	**11.61**	545.152	45.48	394.487	62.84
2	1052.147	23.56	295.112	84.01	202.896	122.19
3	715.939	34.63	190.461	130.16	138.319	179.23
4	570.133	43.48	156.472	158.44	107.043	231.60
5	435.496	56.93	139.371	177.88	89.903	275.76
6	381.251	65.03	128.880	192.36	76.296	324.94
7	334.724	74.06	122.472	202.42	70.469	351.81
8	307.779	80.55	115.022	215.53	**59.325**	417.89
JUICEnext (IBM BladeCenter QS22)						
	Naïve Fcn. Offload		Opt. Fcn. Offload		Scalable DMA	
# SPEs	Wall clock	% Speedup	Wall clock	% Speedup	Wall clock	% Speedup
1	1617.694	23.97	640.491	60.55	390.172	99.39
2	858.288	45.18	446.678	86.82	201.607	199.29
3	654.199	59.28	406.151	95.48	140.669	296.89
4	648.914	59.76	357.931	108.35	109.456	391.73
5	642.692	60.34	352.095	110.14	90.817	486.46
6	642.466	62.10	351.066	110.46	79.706	587.38
7	644.576	60.16	359.384	107.91	74.491	634.91
8	639.315	60.66	353.773	109.62	64.920	**744.14**

Table 5.2: Wall clock runtime (seconds) and percentage speedup for the 2D transport module on three CBEA systems. The domain is double-precision ozone (O3) concentrations on a 60km^2 area mapped to a grid of 600×600 points and integrated for 12 hours with a $\Delta t = 50$ seconds timestep. Global extremes are emphasized

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

Table 5.3: Wall clock runtime (seconds) and percentage speedup for the 2D transport module on two homogeneous multi-core systems. The domain is double-precision ozone (O3) concentrations on a 60km^2 area mapped to a grid of 600×600 points and integrated for 12 hours with a $\Delta t = 50$ seconds timestep. Global extremes are emphasized



(a) Wall clock runtime (b) Speedup

Figure 5.9: Wall clock runtime (seconds) and speedup for the 2D transport module on three CBEA systems and two homogeneous multi-core systems.

The low runtimes on the shared memory machine are attributable to hardware support for double-precision arithmetic and more mature compiler technology. Kurzak and Dongarra found that the LINPACK benchmark is fourteen times slower on the Cell BE when double-precision arithmetic is used compared to single-precision arithmetic [80]. Their results and these experimental results suggest that a CBEA implementation with hardware support for double-precision arithmetic would out-perform a homogeneous processor of similar clock rate.

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

5.5 Vector Stream Processing for Transport [9]

This section discusses the implementation, analysis, and optimization of FIXEDGRID's 3D transport module on the CBEA using a *vector stream processing* approach. Transport is calculated according to the equations in Section 5.1 along the x-, y-, and z-axes using the stencil shown in Figure 4. The 3D module was written from scratch to achieve maximum performance. Performance is given in Section 5.5.3.

Stream processing is a programming paradigm for exploiting parallelism. Given a stream of data, a set of computational kernels (functions or operations) are applied to each element in the stream. Often the kernels are pipelined. This method is particularly useful when programming several separate computational units without explicitly managing allocation, synchronization, and communication. Stream processing is appropriate for programs using dimension splitting to reduce multidimensional operations to sets of 1D operations. The 1D operations can be implemented as kernel functions and the domain data streamed in parallel through the kernels.

Vector stream processing extends stream processing by making every element in the data stream a vector and using vectorized kernel functions. This approach uses every level of heterogeneous parallelism in the CBEA by overlapping memory I/O with SIMD stream kernels on multiple cores. In order to achieve this, data reorganization in the stream must be minimal. Interleaving caused by transferring incontiguous data via DMA lists (Section 5.4.3) implicitly arranges stream data into vectors, so DMA lists are an integral part of this approach. A SIMD kernel can be applied directly to this unorganized data stream. Contiguous data must either be vectorized or processed with a scalar kernel.

FIXEDGRID uses dimension splitting to calculate 3D transport separately along the x-, y-

, and z-axis with the same kernel function. Similar to the 2D module in Section 5.4.3, DMA lists are used to transfer incontiguous y- and z-axis data to SPE local storage in the 3D module. FIXEDGRID uses second order time splitting to reduce truncation error in 3D transport (see Figure 5). This doubles the work required to calculate mass flux along the x- and y-axis, so discretization in these directions should be highly efficient.

5.5.1 Streaming Vectorized Data with the CBEA

Streams of vector data are formed in the 3D module by combining DMA intrinsics with a triple-buffering scheme. Triple-buffering permits a single SPE to simultaneously fetch data from main memory, apply the kernel function, and write processed data back to main memory. A separate DMA tag is kept for each buffer, and MFC instruction barriers keyed to these tags prevent overwriting an active buffer while allowing simultaneous read/write from different buffers by the same MFC. A stream of data from main memory, through the SPU, and back to main memory is maintained. Redirecting the outward-flowing data stream to the inward-flowing stream of another SPE allows kernel function pipelining. The concentration matrix and associated data structures in main memory are divided into blocks, which are streamed simultaneously through the SPEs according to block starting address.

Scalar elements transfered via DMA lists are interleaved into SPE local storage as vectors, so data organization is not required for y- or z-axis data (see Figure 6). X-axis data is contiguous, so the data must either be vectorized manually or processed with a scalar kernel. There are two ways to vectorize scalar data: manually interleave multiple scalar data streams, or copy the scalar value to every vector element. Manually interleaving multiple streams requires copying four/two buffers of single-/double-precision data to SPE local storage and interlacing the data to form vectors before processing. This approach requires many scalar operations and an additional DMA transfer. The SPU is optimized for vector operations, so it is approximately 8% faster to copy the scalar value to every vector element via the `spu_splats` intrinsic, process with the vectorized kernel, and return to scalar data by discarding all vector elements except the first. By carefully manipulating the DMA tags associated with local storage buffers, the `spu_splats` operations for one buffer (i.e. chemical concentrations) can overlap with the next buffer fetch (i.e. wind field data). This improves performance by a further 4%.

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

zero SPE idle time.

The streaming operations were encapsulated in a small library of static inline functions and user-defined data types. Local storage buffers are described by a data type with fields for local storage address, main memory address, and buffer size. Functions for fetching, processing, and flushing buffers maximize the source code readability and ease debugging. The unabridged source code for x-axis mass flux calculation is shown in Listing A.1.

5.5.2 The Vectorized Transport Kernel Function

The 3D advection/diffusion kernel function was vectorized to use the SPE's SIMD ISA. Because the advection discretization is upwind-biased, different vector elements may apply different parts of Equation 5.2, based on the wind vector's sign. Disassembling the wind vector to test each element's sign introduces expensive branching conditionals and scalar operations. It is more efficient to calculate both parts of Equation 5.2 preemptively, resulting in two vectors of possible values, and then bitwise mask the correct values from these vectors into the solution vector. The `spu_cmpgt`, `spu_and`, and `spu_add` intrinsics, each mapping to a single assembly instruction, are the only commands required to form the solution vector. `spu_cmpgt` forms a bit-mask identifying elements with a certain sign, and `spu_and` and `spu_add` assemble the solution vector. Although half the calculated values are discarded, all branching conditionals in the kernel function are replaced with only a few assembly instructions and scalar operations are eliminated entirely. Performance of the vectorized kernel is approximately an order of magnitude above a scalar kernel with branches. The original C kernel is shown in Listing A.2 and the vectorized kernel is shown in Listing A.3.

5.5.3 Analysis and Comparison

This section discusses the performance of FIXEDGRID's 3D transport module on two CBEA systems, a distributed memory multi-core system, and a shared memory multi-core system. The experiments calculate ozone (O_3) concentrations on a 60km^2 $\times$ 1.2km area mapped to a grid of $600 \times 600 \times 12$ points and integrated for 12 hours with a $\Delta t = 50$ seconds timestep. With the exception of Monza, the 3D experiments were performed on the same systems and with the same compiler options as in Section 5.4.4. Monza results are not shown because the PlayStation 3 has insufficient RAM to calculate a domain of this size without paging. A single experiment took approximately 35 hours to complete on Monza, making the results incomparable to the other CBEA systems. Again, the presented results are the average of three runs on each platform. The measurement error was insignificant, so error bars are not shown. Tables 5.4 and 5.5 list the wall clock runtime performance on the CBEA systems and homogeneous multi-core systems, respectively.

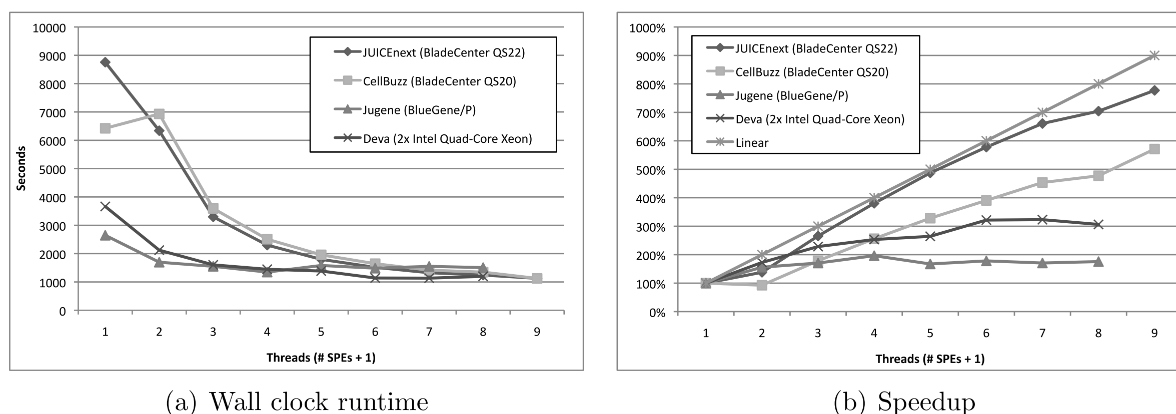
Figure 10 shows the wall clock runtime performance of FIXEDGRID on two CBEA systems

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

Table 5.4: Wall clock runtime (seconds) and percentage speedup for the 3D transport module on two CBEA systems. The domain is double-precision ozone (O3) concentrations on a $60\text{km}^2 \times 1.2\text{km}$ area mapped to a grid of $600 \times 600 \times 12$ points and integrated for 12 hours with a $\Delta t = 50$ seconds timestep. Global extremes are emphasized

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

Table 5.5: Wall clock runtime (seconds) and percentage speedup for the 3D transport module on two homogeneous multi-core systems. The domain is double-precision ozone (O3) concentrations on a $60\text{km}^2 \times 1.2\text{km}$ area mapped to a grid of $600 \times 600 \times 12$ points and integrated for 12 hours with a $\Delta t = 50$ seconds timestep. Global extremes are emphasized



(a) Wall clock runtime (b) Speedup

Figure 5.10: Wall clock runtime (seconds) and speedup for the 3D transport module on three CBEA systems and two homogeneous multi-core systems.

and two homogeneous multi-core systems. JUICEnext, with hardware support for fully-pipelined double precision arithmetic, achieves the best performance. Both CBEA systems achieve performance comparable to two nodes of the IBM BlueGene/P and eight Intel Xeon cores in a single chip.

There is a performance asymptote beginning around the 5 SPE mark. We believe this is due to the element interconnect bus (EIB) end-to-end control mechanism. Ainsworth and Pinkston characterized the performance of the EIB in [18] and highlighted the end-to-end control mechanism as the main CBEA bottleneck. The EIB is a pipelined circuit-switched ring-topology network. The EIB's data arbiter does not allow a packet to be transferred along more than six hops (half the diameter of the ring). If a request is greater than six hops, the requester must wait until a data ring operating the opposite direction becomes free. This limits concurrently communicating elements to six. When the MIC and five SPEs communicate on the EIB – as occurs when five SPEs stream data simultaneously – the arbiter stalls communication from all other processing elements. The EIB's 204.8 GB/second peak bandwidth is enough to support streaming on all SPEs, so the control mechanism bottleneck is significant.

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

5.6 Future Work

Future work should address the limitations of the Element Interconnect Bus. By pairing SPEs and using one for data transfer and one for computation, it may be possible to achieve better scalability for more than five SPEs by spreading DMA requests over a wider area on the EIB ring. Rafique et al. explored similar methods for I/O-intensive applications with good results [97]. Nodes in an IBM BladeCenter installation can use off-chip SPEs for a maximum of 16 SPEs. Off-chip SPEs have their own EIB, so it may be advantageous to use them for file I/O or checkpointing. Spreading memory requests over multiple EIBs should also be explored.

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

Programming language technologies for heterogeneous multi-core chipsets are still relatively immature. The bulk of the performance improvement came from vectorized kernel functions, so better support for compiler auto-vectorization should be explored. While some auto-vectorization support is available, the performance of the auto-generated kernel was far below that of the hand-tuned kernel. A comparison of Listing A.2 with Listing A.3 should convince any programmer that more intelligent compilers are needed. Indeed, the majority of low-level optimizations we did by hand could have been done automatically by an advanced compiler.

Chapter Summary

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

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

since there are a relatively small number of SPEs. When both ILP and DLP are present, maximum performance will only be achieved when every level of parallelism in the CBEA is used, as in vector stream processing.

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

Vector stream processing seeks to use all available parallelism in the CBEA by overlapping memory I/O with SIMD stream kernels on multiple cores. This method was implemented in a few light-weight static inline functions accompanied by user-defined types. Together, these constructs combine DMA intrinsics, DMA lists, and a triple-buffering scheme to stream data through the SPE local storage at the maximum rate permitted by the EIB data arbiter. By using the MFC proxy command queue and MFC instruction barriers instead of mailbox registers or busy-wait synchronization, vector stream processing achieves almost zero SPE idle time and maximizes SPE throughput. Careful consideration of data organization and data contiguity in multi-dimensional arrays reduces data reordering in the stream to only one dimension (i.e. the contiguous dimension must be vectorized). All levels of heterogeneous parallelism in the CBEA are used: the SPE's SIMD ISA, the MFC's nonblocking DMA intrinsics, and the many cores of the CBEA. Fully-optimized, FIXEDGRID calculates ozone (O_3) transport on a domain of $600 \times 600 \times 12$ grid cells for twelve hours with a 50 second time step in approximately 18.34 minutes.

Chapter 6

Multi-core Accelerated WRF-Chem

In this chapter, KPPA-generated RADM2 gas phase chemical mechanisms are used in WRF-Chem on homogeneous and heterogeneous multi-core architectures to generate real forecasts of air quality. This case study demonstrates the potential of accelerated architectures and realizes a 1.5× speedup in WRF-Chem. Speedups as high as 2.5× are predicted for today's architectures.

The simulation domain (Figure 1) spans the eastern half of the United States (approximately latitude 27.182 – 47.382, longitude -91.359 – -65.003) in a 40×40 grid with 19 vertical layers. The RADM2 gas phase chemistry scheme and SORGAM aerosol scheme were calculated for 40 minutes of simulation time beginning April 6, 2006 at 12:00 noon with a time step of 240 seconds. The test data was provided by John Michalakes of NCAR.

Benchmarks were conducted on three systems:

Quasar: an IBM BladeCenter QS22 installation at the Arctic Region Supercomputing Center (ARSC). It consists of twelve QS22 blades, each with two PowerXCell 8i CPUs and 8GB of DDR RAM, connected by 4X SDR Infiniband.

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

Deva+PS3: an ad-hoc heterogeneous cluster consisting of Deva and a single Sony PlayStation 3 connected by gigabit Ethernet. The PlayStation 3 has one Cell Broadband Engine CPU with six SPEs and 256 MB of XDR RAM.

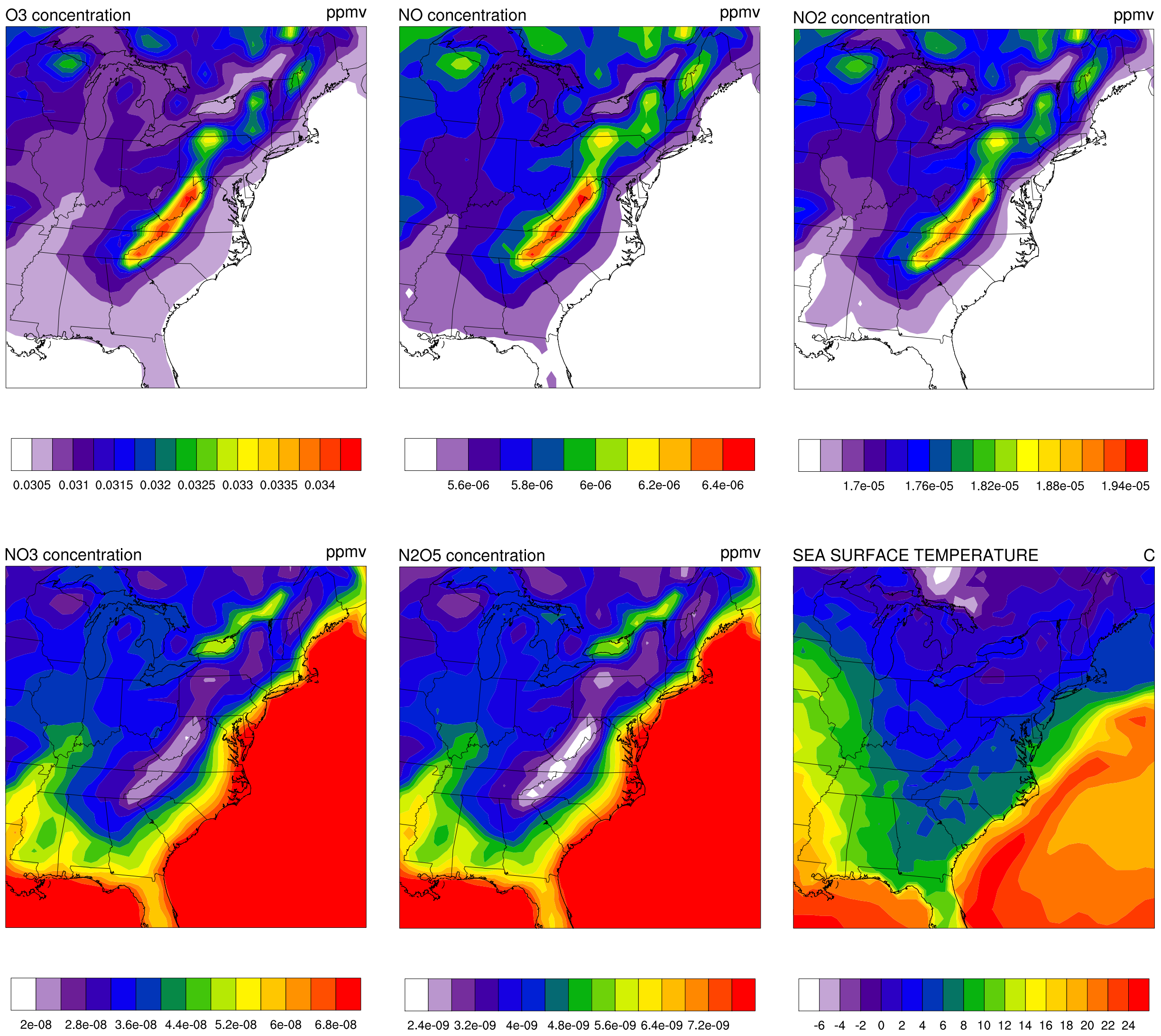
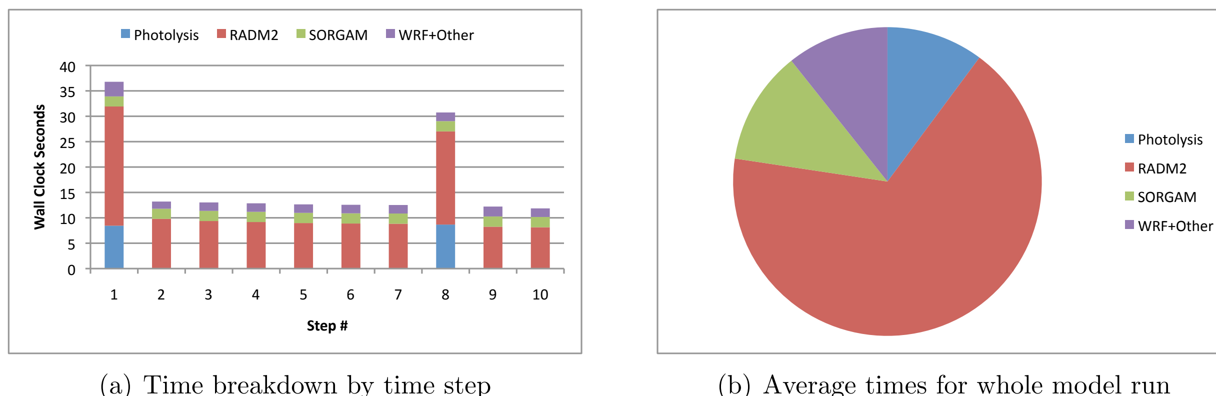


Figure 6.1: Ground-level concentrations and temperatures from the WRF-Chem case study.



(a) Time breakdown by time step

(b) Average times for whole model run

Figure 6.2: Runtime profile of WRF-Chem executing on an Intel Xeon workstation.

6.1 Baseline

A baseline profile of WRF-Chem was taken on Deva with the `ETIME` intrinsic. WRF-Chem 3.2 was built with the Intel Fortran compiler with shared memory parallelization ("smpar" option). Shared memory parallelization was chosen so that non-chemistry runs of WRF (i.e. the WRF model alone) could take advantage of all eight of Deva's cores. WRF-Chem does not support shared memory parallelization, hence the `OMP_NUM_THREADS` environment variable was set to "1" before running WRF-Chem. Deva's configuration is officially supported by WRF so compilation was successful with minimal effort. Known bugs in WRF-Chem had to be manually corrected before the code would compile.

Figure 2 shows the runtime profile of WRF-Chem on Deva. The serial, unaccelerated RADM2 chemical mechanism (generated by KPP 2.1) dominates computational time. Photolysis (chemical decomposition induced by light or other radiant energy) is also an expensive process, but it is not required on every time step. Chemical transport (the WRF model itself) accounts for only 10% of the model run time. Note that since chemistry is enabled, WRF executes serially on a single Xeon core; adding shared memory support to WRF-Chem will allow WRF to use all eight cores. Exact timings are given in Table 6.1.

6.2 Quasar

RADM2 computation is responsible for over two thirds of WRF-Chem's runtime. Chapter 4 developed a RADM2 implementation for the CBEA that is approximately 20× faster than WRF-Chem's current implementation, but in order to use the accelerated RADM2 the rest of the WRF model would need to be ported to the CBEA. Fortunately, the Power Processing Element (PPE) defined by the CBEA (see Chapter 2) is a standard PowerPC implementation

Table 6.1: WRF-Chem execution times (seconds) on an Intel Xeon workstation.

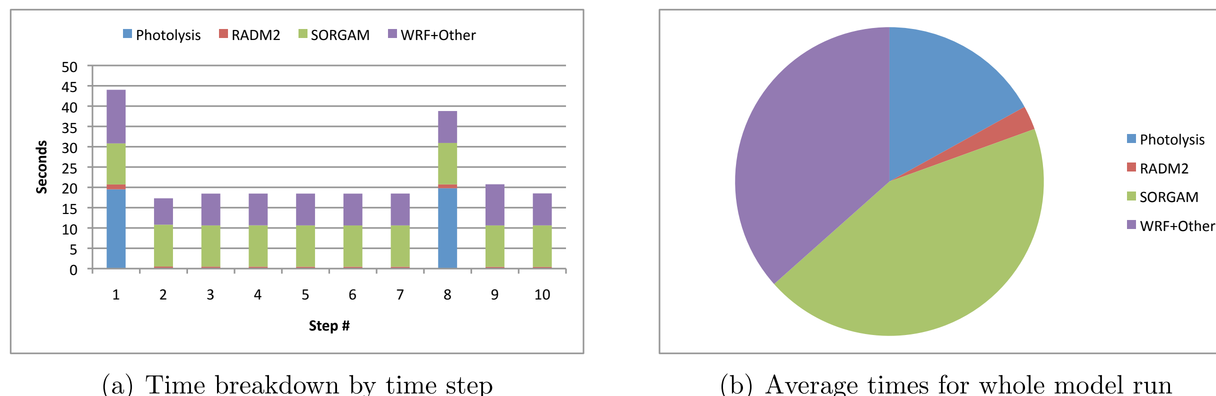
Step #	**Photolysis**	**RADM2**	**SORGAM**	**WRF**	**Total**
1	8.449	23.476	1.985	2.879	36.790
2	0	9.795	1.993	1.415	13.204
3	0	9.368	1.998	1.663	13.031
4	0	9.171	2.001	1.677	12.857
5	0	8.967	2.001	1.678	12.647
6	0	8.875	2.001	1.681	12.565
7	0	8.832	2.012	1.678	12.523
8	8.668	18.349	2.017	1.714	30.750
9	0	8.248	2.026	1.946	12.217
10	0	8.138	2.026	1.691	11.857
Mean	1.712	11.322	2.008	1.802	16.844
Median	0	9.070	2.008	1.680	12.752

so cross compiling WRF for PowerPC is sufficient. In this configuration, only RADM2 would take advantage of the SPEs and the rest of the model would run serially on the PPE.

Although straightforward in concept, cross compiling WRF-Chem to the CBEA was complex and frustrating. The WRF component of WRF-Chem was not hard to port, however the chemical driver code is brittle and doesn't port gracefully. Extensive code modifications were required in several files, and a complete rewrite of the KPP build process was needed. Approximately two days were required to compile WRF-Chem with gcc 4.3.2 for CBEA. WRF-Chem would not compile with IBM XLF compiler for CBEA.

Figure 3 shows the runtime profile of WRF-Chem on Quasar; Table 6.2 shows exact measurement times. RADM2 time is significantly reduced compared to the baseline, yet the overall model time has increased. WRF and SORGAM are 5× slower on the PPE than on the Xeon, and photolysis is 2.3× slower. This is because the PPE is a very weak CPU by today's standards and compilers for the CBEA have not matured. IBM has not developed a compiler that satisfactorily leverages all the hardware features of the CBEA and the community compiler (gcc 4.3.2) has fallen out of development. With today's tools, a complete rewrite of the entire WRF code base would be necessary to execute WRF effectively on the CBEA. Of course, this is both inadvisable and impractical.

Two observations arise from this data. First, reducing a subroutine's runtime by a factor of X does not mean that program runtime is reduced (under Amdahl's Law) by a function of X. Many multi-core acceleration success stories focus on only one kernel and ignore the impact of acceleration on the whole. The impact of acceleration must be considered on those parts of the code which are not accelerated directly. Secondly, heterogeneity enables rapid facilitation of existing codes in an accelerated environment with fairly low overhead.



(a) Time breakdown by time step

(b) Average times for whole model run

Figure 6.3: Runtime profile of WRF-Chem executing on Quasar, an IBM BladeCenter QS22.

Even though three of the four model components are severely slower on the PPE, the overall model runtime is increased by only 30%. Under Moore's Law, the number of cores-on-chip is expected to increase in future architectures, so a percentage of those cores may be used as accelerators without harming performance. This points to a convergence of accelerated architectures (such as GPUs) and traditional CPUs. Such a chip would execute legacy code at acceptable levels and, for a little additional effort, the accelerators could be engaged to achieve maximum performance. Compilers developed expressly for such chips would both reduce the negative impacts of accelerators and improve their utilization.

6.3 Deva + PS3

Because RADM2 performs well on the CBEA and WRF performs well on a standard CPU, we combined the two architectures in an ad-hoc heterogeneous cluster. In this configuration, everything runs on Deva except RADM2, which runs on a Sony PlayStation 3 (PS3). The two nodes communicate via gigabit Ethernet.

Invoking a remote accelerated kernel in a heterogeneous cluster is nontrivial. Accelerated computing introduces multiple instruction set architectures, different endians, and data type restrictions which are ignored by many remote invocation toolsets. Message Passing Interface (MPI) [29] addresses these issues but leads to ad-hoc implementations. Additionally, the communicators in legacy MPI codes must be split to support heterogeneity, a non-trivial process likely to introduce bugs. For a complete, comparative study of software support for hybrid computing, see [124].

Two software packages have been developed to directly address these issues: IBM Data Communication and Sychronization (DaCS) [69] and IBM Dynamic Application Virtualization (DAV) [96]. DaCS provides a set of services which ease the development of applications and

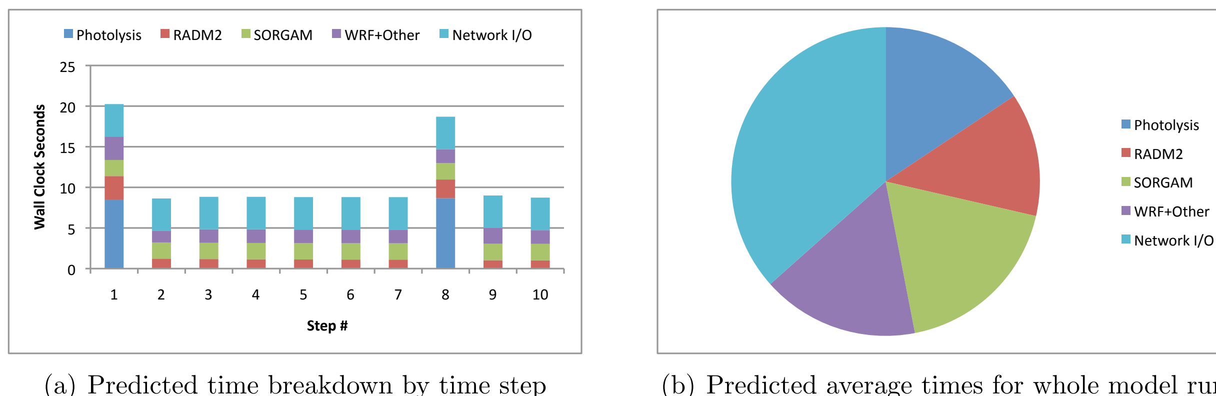
Table 6.2: WRF-Chem execution times (seconds) on Quasar, an IBM BladeCenter QS22.

Step #	**Photolysis**	**RADM2**	**SORGAM**	**WRF**	**Total**
1	19.513	1.04	10.105	13.197	44.019
2	0	0.502	10.307	6.498	17.307
3	0	0.480	10.147	7.830	18.457
4	0	0.470	10.178	7.819	18.467
5	0	0.460	10.173	7.829	18.461
6	0	0.455	10.165	7.935	18.455
7	0	0.453	10.178	7.842	18.473
8	19.794	0.941	10.192	7.862	38.789
9	0	0.423	10.205	10.129	20.757
10	0	0.417	10.219	7.877	18.513
Mean	3.931	0.581	10.1869	8.472	23.170
Median	0	0.465	10.178	7.839	18.470

application frameworks in a heterogeneous multi-tiered system. It has a fairly low-level API and may require non-trivial effort to implement, nevertheless it has been used successfully on the Roadrunner supercomputer [23]. DAV is a high-level toolset that enables applications to take advantage of accelerated libraries on remote systems while reducing time to deployment with minimal code changes and disruption to business. However, both these technologies have significant issues which limit their real-world applicability. Fortran support is weak in both DaCS and DAV, DAV is an alphaWorks project with shaky support for 64-bit applications, and neither project supports GPGPU or indeed any accelerator other than the CBEA. Furthermore, they both assume that users have elevated privileges, can stop and start software services, and have read-write access to files outside the user's home directory. DaCS applications require processor affinity at start, which requires the `CAP_SYS_NICE` permission. Naturally, most systems administrators deny requests for elevated permissions or system-wide software installation.

Strong Fortran support and 64-bit support was required in this test, a lightweight communication library was developed in preference to DaCS or DAV. The Kernel Offloading Interface (KOI) is based on the UNIX sockets interface so it does not require special user permissions. It is written in C and Fortran and integrates well with existing Fortran codes. KOI provides a small library of functions to connect to remote accelerated systems and call exposed functions on that system. The syntax is similar to CUDA's driver interface.

When WRF-Chem on Deva calls the RADM2 kernel, KOI serializes the model data and sends it to the PS3. The PS3 applies the RADM2 kernel and returns the data. Figure 4 shows the runtime performance of WRF-Chem on the heterogeneous cluster; Table 6.3 shows exact measurement times. This configuration achieves the best runtime performance: an overall



(a) Predicted time breakdown by time step (b) Predicted average times for whole model run

Figure 6.4: Runtime profile of WRF-Chem executing on Deva with RADM2 offloaded to a Sony Playstation 3.

average speedup of 1.5×. Network I/O overhead is disappointingly high: 36% of the model runtime. If the overhead were avoided entirely, the overall speedup would be 2.5×. KOI is not a high performance library and could certainly be improved. One could potentially overlap communication and computation by allowing the PS3 to apply RADM2 to domain data as it arrives rather than wait for the entire domain dataset before processing. Research into kernel offloading is ongoing.

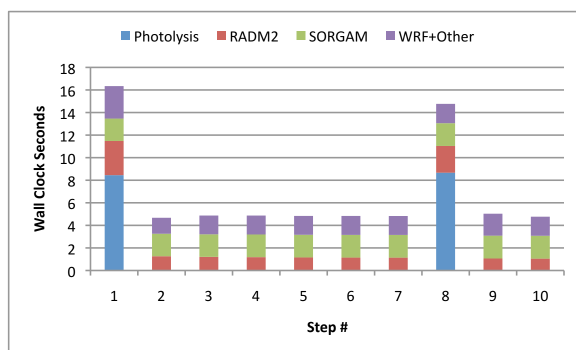
6.4 Accelerated Deva

KPPA can target familiar homogeneous multi-core chipsets as well as emerging multi-core architectures. In this section, we hypothesize on WRF-Chem's maximum performance if Deva could use all eight cores. The numbers presented here are the baseline benchmark taken on Deva but with the RADM2 timings replaced by timings from the stand-alone KPPA-generated RADM2 kernel from Chapter 4. Note that if WRF-Chem were modified to support OpenMP, then the WRF part of WRF-Chem would also use OpenMP, further reducing runtime.

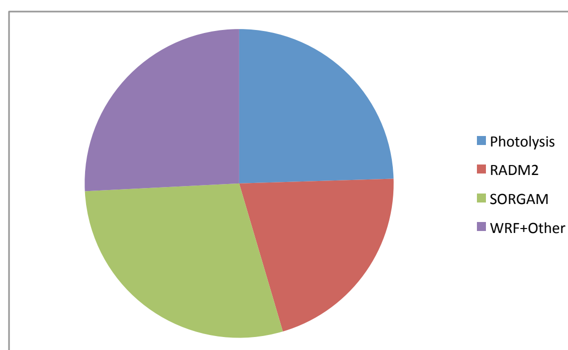
KPPA-generated kernels use both SSE extensions and OpenMP, making them significantly faster than state-of-the-art kernels. However, WRF-Chem does not support shared memory parallelization so it cannot take advantage of KPPA-generated RADM2 on Deva. If WRF-Chem were modified to support shared memory parallelization, the runtime profile would probably be similar to that shown in Figure 5. Exact timings are given in Table 6.2. These predicted results demonstrate KPPA's applicability to existing architectures.

Table 6.3: WRF-Chem execution times (seconds) on Deva with RADM2 offloaded to a Sony Playstation 3.

Step #	Photolysis	RADM2	SORGAM	WRF	Network	Total
1	8.449	2.935	1.985	2.879	3.914	20.162
2	0	1.224	1.993	1.415	3.919	8.552
3	0	1.171	1.998	1.663	4.004	8.838
4	0	1.146	2.001	1.677	3.979	8.812
5	0	1.121	2.001	1.678	3.935	8.735
6	0	1.109	2.001	1.681	3.964	8.764
7	0	1.104	2.012	1.678	3.923	8.717
8	8.668	2.294	2.017	1.714	3.901	18.595
9	0	1.031	2.026	1.946	3.989	8.988
10	0	1.017	2.026	1.691	3.976	8.703
Mean	1.712	1.415	2.008	1.802	3.949	10.887
Median	0	1.134	2.008	1.680	3.950	8.788



(a) Predicted time breakdown by time step



(b) Predicted average times for whole model run

Figure 6.5: Predicted runtime profile of WRF-Chem executing on Deva with KPPA-generated RADM2.

Table 6.4: Predicted WRF-Chem execution times (seconds) on Deva with KPPA-generated RADM2.

Step #	**Photolysis**	**RADM2**	**SORGAM**	**WRF**	**Total**
1	8.449	3.029	1.985	2.879	16.343
2	0	1.264	1.993	1.415	4.672
3	0	1.209	1.998	1.663	4.871
4	0	1.183	2.001	1.677	4.870
5	0	1.157	2.001	1.678	4.837
6	0	1.145	2.001	1.681	4.836
7	0	1.140	2.012	1.678	4.830
8	8.668	2.368	2.017	1.714	14.768
9	0	1.064	2.026	1.946	5.033
10	0	1.050	2.026	1.691	4.768
Mean	1.712	1.460	2.008	1.802	6.983
Median	0	1.170	2.008	1.680	4.853

Chapter Summary

WRF-Chem benchmarks from three systems, two accelerated clusters and one homogeneous multi-core workstation, were presented. The heterogeneous cluster combining an Intel Xeon node with a CBEA-based node achieves a 1.5× speedup in WRF-Chem when compared to an Intel Xeon workstation. When WRF-Chem is run entirely on a CBEA-based node, the unaccelerated parts of the model slow down such as to negate any benefit gained from acceleration in other parts of the model.

These results demonstrate the potential of accelerator architectures to significantly reduce runtimes. Ideally, an accelerated chipset would have a powerful generic CPU for general code execution and not rely wholly on its accelerators for performance. This points to a convergence of traditional CPU architectures and accelerated (i.e. GPU) architectures. Ignoring network overhead in Figure 4 gives an idea of the performance of such a chip.

Additionally, WRF-Chem performance was predicted for a version of WRF-Chem that supports shared memory parallelization. If OpenMP support were added to WRF-Chem, it could take advantage of the KPPA-generated accelerated mechanisms which are 2-7× faster than what is currently used.

Chapter 7

Conclusions

Multi-core architectures have now firmly established their dominance over traditional single core designs, both in high performance and end-user markets. Most desktop workstations now have at least one programmable GPU card in addition to a homogeneous multi-core CPU. Servers and computing clusters have enthusiastically adopted homogeneous multi-core chips in the trend towards increasing hardware parallelism. Adapting existing software for new architectures has never been easy, however this work proves that harnessing multi-core heterogeneous parallelism can be done and is well worth the effort. Furthermore, the methods and software tools developed here are "future-proof" in that they are designed to support as-yet undeveloped CPU and GPU architectures.

The survey of multi-core architectures in Chapter 2 is a snapshot of the current state-of-the-art in chipset design. In context, an important trend is apparent: CPUs and GPUs will certainly merge into single heterogeneous multi-core chipsets in the near future. Following GPU development back to 1984, we see separate boards for 2D and 3D acceleration, then separate 2D and 3D chips on the same board, then in 1999 the chips merged into modern GPU architectures connected by a fast bus to the CPU. On the CPU end, the number of homogeneous cores on a single die increased steadily over the last decade until, in the present day, CPU designers are adding heterogeneous accelerator cores to standard CPUs. CPU/GPU hybrids will capably execute legacy codes at acceptable speeds and optimized codes would far surpass the limits of general purpose computing cores, all while respecting physical and economical bounds.

This revolution in hardware must *not* drastically change existing software design. The CBEA, for all its power and potential, has fallen out of vogue with all but the most demanding of high performance computing customers. This is clearly due to a lack of software tools for managing its complex heterogeneous multi-layered parallelism; it is simply too difficult to program with the tools at hand. On the other hand, NVIDIA GPUs with CUDA are enjoying a growing share of the general purpose computing market. CUDA abstracts parallelism in an intuitive and practical way so programming a GPU is relatively easy. Yet

GPUs will never graduate from their position as co-processors while GPU design (both hardware and software) remains too "parallelism-centric" to support important serial applications. Compilers and software tools targeting future CPU/GPU hybrid architectures need to support existing program design (as the CBEA does) and accessibly support programmer directed parallelization (as CUDA does) if they are to succeed.

Atmospheric modeling and simulation will remain an important application for the foreseeable future. Accurate and comprehensive long-term climate models are urgently needed, as are real-time wildfire models and air quality models on all scales. The underlying mathematics are well established; it is only today's computing hardware that limits model development. Computing architectures with multi-layered heterogeneous parallelism are the next step in chipset design, and the methods and software tools developed in this work parameterize such architectures with good results. KPPA (Chapter 4) uses only four parameters to describe multi-core architectures with multiple layers of heterogeneous parallelism. The code it generates is not only faster on emerging architectures, but also significantly faster on familiar homogeneous architectures. When CPU/GPU hybrid architectures reach the market, KPPA will generate high performance chemical kernels for them as well. Similarly, the scalable method for chemical transport modeling given in Chapter 5 can be adapted to any chip consisting of SIMD accelerator cores connected to a general purpose CPU core with exposed communication. Scalability will play an important role as the number of cores-on-chip increases.

The developments in chemical kinetics modeling presented in Chapter 4 need not be limited to chemistry alone. KPPA generates chemical codes by combining it's domain-specific knowledge with a simple parameterization of a multi-core architecture. In essence, it is a glorified source-to-source compiler. Other problems (e.g. finite element methods, n-body simulation, visualization, etc.) could use a similar approach. Capturing the domain-specific algorithm in a KPPA-like program allows the programmer to write general algorithms to be translated into specific high-performance implementations. One could envision a comprehensive atmospheric model with application-specific code generators for physics, transport, chemistry, etc., that targets existing and future architectures. KPPA was non-trivial to develop so such a model would require enormous effort, but the method has been shown to be highly effective and quite possible.

Bibliography

[1] **J. C. Linford**. Accelerating the implicit integration of stiff chemical systems with emerging multi-core technologies. Workshop Presentation. NCAR IMAGe T-O-Y'09, August 18–20 2009.

[2] **J. C. Linford**, D. Becker, and F. Wolf. Implementation and validation of the extended controlled logical clock. Technical Report FZJ-JSC-IB-2007-11, Forschungszentrum Jülich, Jülich Germany, November 2007.

[3] **J. C. Linford**, E. Constantinescu, and A. Sandu. Computational performance of parallel air quality models. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Applications (SIAM PP'08)*, Atlanta, Georgia, March 12–14 2008.

[4] **J. C. Linford**, M.-A. Hermanns, M. Geimer, D. Böhme, and F. Wolf. Detecting load imbalance in massively parallel applications. Technical report, Forschungszentrum Jülich, Jülich Germany, 2008.

[5] **J. C. Linford**, J. Michalakes, M. Vachharijani, and A. Sandu. Multi-core acceleration of chemical kinetics for modeling and simulation. In *Proceedings of the 2009 ACM/IEEE conference on supercomputing (SC'09)*, Portland, OR, November 14–20 2009.

[6] **J. C. Linford**, J. Michalakes, M. Vachharijani, and A. Sandu. Automatic generation of multi-core chemical kernels. *IEEE TPDS: Special Issue on High-Performance Computing with Accelerators*, 2010.

[7] **J. C. Linford** and A. Sandu. Optimizing large scale chemical transport models for multicore platforms. In *Proceedings of the 2008 Spring Simulation Multiconference (SpringSim '08)*, Ottawa, Canada, April 2008.

[8] **J. C. Linford** and A. Sandu. Chemical kinetics on multi-core SIMD architectures. In G. Allen, J. Nabrzyski, E. Seidel, G. D. van Albada, J. Dongarra, and P. M. Sloot, editors, *Computational Science – ICCS 2009*, volume 5544 of *LNCS*. Springer, 2009.

[9] **J. C. Linford** and A. Sandu. Vector stream processing for effective application of heterogeneous parallelism. In *Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC'09)*, Honolulu, HI, March 8–12 2009.

[10] **J. C. Linford** and A. Sandu. Scalable heterogeneous parallelism for atmospheric modeling and simulation. *J. Supercomput.*, 10.1007/s11227-010-0380-8, 2010.

[11] **J. C. Linford**, A. Srivastava, and A. Sandu. Performance of stabilized explicit time integration methods for parallel air quality models. In *Proceedings of the 2007 High Performance Computing Symposium (HPC'07)*, Norfolk, VA, March 25–29 2007.

[12] S. Schneider, J.-S. Yeom, B. Rose, **J. C. Linford**, A. Sandu, and D. S. Nikolopoulos. A comparison of programming models for multiprocessors with explicitly managed memory hierarchies. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09)*, 2009.

[13] D. Becker, **J. C. Linford**, R. Rabenseifner, and F. Wolf. Replay-based synchronization of timestamps in event traces of massively parallel applications. In *Proceedings of the International Conference on Parallel Processing (ICPP'08)*, Portland, Oregon, September 8-12 2008.

[14] D. Becker, **J. C. Linford**, R. Rabenseifner, and F. Wolf. Replay-based synchronization of timestamps in event traces of massively parallel applications. *J. Scale. Comput. Pract. Exp.*, 10(1):49–60, 2009.

[15] D. Becker, R. Rabenseifner, F. Wolf, and **J. C. Linford**. Scalable timestamp synchronization for event traces of message-passing applications. In *J. Par. Comp.*, 2009.

[16] L. A. Schweitzer, L. C. Marr, **J. C. Linford**, and M. A. Darby. The sustainable mobility learning laboratory: Interactive web-based education on transportation and the environment. *App. Environ. Educ. Commun.*, 7(1):20–29, 2008.

[17] A. Abdulle and A. A. Medovikov. Second order chebyshev methods based on orthogonal polynomials. *Numeriche Mathematik*, 1(90):1–18, 2001.

[18] T. W. Ainsworth and T. M. Pinkston. On characterizing performance of the Cell Broadband Engine Element Interconnect Bus. In *Proceedings of the First International Symposium on Networks-on-Chip (NOCS '07)*, pages 18–29, Princeton, NJ, May 2007.

[19] S. R. Alam and P. K. Agarwal. On the path to enable multi-scale biomolecular simulations on petaFLOPS supercomputer with multi-core processors. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '07)*, pages 1–8, Long Beach, CA, Mar. 26–30, 2007.

[20] 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 2008, 18 November 2008.

[21] D. A. Bader and S. Patel. High performance MPEG-2 software decoder on the Cell Broadband E. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS '08)*, pages 1–10, Miami, FL, Apr. 2008.

[22] H. Baik, K.-H. Sihn, Y. il Kim, S. Bae, N. Han, and H. J. Song. Analysis and parallelization of H.264 decoder on Cell Broadband Engine Architecture. In *Proceedings of the IEEE International Symposium on Signal Processing and Information Technology*, pages 791–795, Giza, Dec. 2007.

[23] 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.

[24] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray tracing on the cell processor. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 15–23, Salt Lake City, UT, Sept. 2006.

[25] I. Bey, D. J. Jacob, R. M. Yantosca, J. A. Logan, B. D. Field, A. M. Fiore, Q. Li, H. Y. Liu, L. J. Mickley, and M. G. Schultz. Global modeling of tropospheric chemistry with assimilated meteorology: Model description and evaluation. *J. Geophys. Res*, 106(D19):23,073–23,095, 2001.

[26] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos. Dynamic multigrain parallelization on the Cell Broadband Engine. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07)*, pages 90–100, New York, NY, USA, 2007. ACM.

[27] 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.

[28] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[29] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[30] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *J. High Perform. Comput. Appl.*, 21(4):457–466, 2007.

[31] 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.

[32] G. R. Carmichael and L. K. Peters. A second generation model for regional scale transport / chemistry / deposition. *Atmos. Env.*, 20(1):173–188, 1986.

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

[34] 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.

[35] L. Chen, Z. Hu, J. Lin, and G. R. Gao. Optimizing the fast fourier transform on a multi-core architecture. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '07)*, pages 1–8, Long Beach, CA, Mar. 26–30, 2007.

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

[37] CodeSourcery, Inc., `http://www.codesourcery.com/vsiplplusplus`. *Sourcery VSIPL++ User's Guide 2.2-9*, 2009.

[38] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC '03)*, page 35, Washington, DC, 2003. IEEE Computer Society.

[39] 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.

[40] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick. Self adapting linear algebra algorithms and software. *Proceedings of the IEEE: special issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.

[41] J. Dongarra, S. Moore, G. Peterson, S. Tomov, J. Allred, and V. Natoli. Exploring new architectures in accelerating CFD for air force applications. In *Proceedings of HPCMP Users Group Conference 2008*, 14–17 July 2008.

[42] A. Douillet and G. R. Gao. Software-pipelining on multi-core architectures. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07)*, pages 39–48, Brasov, Romania, Sept. 15–19, 2007.

[43] K. Duke and W. A. Wall. A professional graphics controller. *IBM Sys. J.*, 24(1):14, 1985.

[44] 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.

[45] M. Erez, J. H. Ahn, J. Gummaraju, M. Rosenblum, and W. J. Dally. Executing irregular scientific applications on stream architectures. In *Proceedings of the 21st Annual International Conference on Supercomputing (ICS '07)*, pages 93–104, New York, NY, USA, 2007. ACM.

[46] Executive Office of the President, Office of Science and Technology Policy. *A Research and Development Strategy for High Performance Computing*, November 1987.

[47] 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.

[48] 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. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*, 2006.

[49] 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.

[50] M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384, 1998.

[51] 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.

[52] J. Gass. NASA goddard space flight center global modeling and assimilation office. `http://gmao.gsfc.nasa.gov/systems/geos5/`, March 2010.

[53] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: The System S declarative stream processing engine. In *Proceedings of the 2008 International Conference on Management of Data (SIGMOD '08)*, pages 1123–1134, New York, NY, USA, 2008. ACM.

[54] 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 / Heidelberg.

[55] 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.

[56] 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 Tech. J.*, November 2007.

[57] 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.

[58] 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.

[59] N. T. Hieu, K. C. Keong, A. Wirawan, and B. Schmidt. Applications of heterogeneous structure of Cell Broadband Engine Architecture for biological database similarity search. In *Proceedings of the The 2nd International Conference on Bioinformatics and Biomedical Engineering (ICBBE '08)*, pages 5–8, Shanghai, May 2008.

[60] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, July 2008.

[61] 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.

[62] C. Hirsch. *Numerical Computation of Internal and External Flows 1: fundamentals and numerical discretization.* John Wiley and Sons, Chichester, 1988.

[63] F. Horn and R. Jackson. General mass action kinetics. *Archive for Rational Mechanics and Analysis*, 47(2):81–116, January 1972.

[64] Z. Hu, G. Gerfin, B. Dobry, and G. R. Gao. Programming experience on cyclops-64 multi-core chip architecture. In *First Workshop on Software Tools for Multi-Core Systems (STMCS)*, March 2006.

[65] W. Hundsdorfer. Numerical solution of advection-diffusion-reaction equations. Technical report, Centrum voor Wiskunde en Informatica, 1996.

[66] W. Hundsdorfer and J. G. Verwer. *Numerical Solutions of Time-Dependent Advection-Diffusion-Reaction Equations.* Springer, 2003.

[67] 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.

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

[69] International Business Machines Corporation. *Data Communication and Synchronization for Hybrid-x86 Programmer's Guide and API Reference*, October 2007.

[70] D. J. Jacob, D. D. Davis, S. C. Liu, R. E. Newell, B. J. Huebert, B. E. Anderson, E. L. Atlas, D. R. Blake, E. V. Browell, W. L. Chameides, S. Elliott, V. Kasputin, E. S. Saltzman, H. B. Singh, and N. D. Sze. Transport and chemical evolution over the pacific (TRACE-P): A NASA/GTE aircraft mission. White paper, National Aeronautics and Space Administration, June 1999.

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

[72] L. O. Jay, A. Sandu, F. A. Potra, and G. R. Carmichael. Improved quasi-steady-state-approximation methods for atmospheric chemistry integration. *SIAM J. Sci. Comput.*, 18(1):182–202, 1997.

[73] W. J.G. Verwer, B.P. Sommeijer. RKC time-stepping for advection-diffusion-reaction problems. *J. Comp. Phys.*, 201:61–79, 2004.

[74] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5), 2005.

[75] U. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany. The Imagine stream processor. In *Proceedings 2002 IEEE International Conference on Computer Design*, pages 282–288, Sept. 2002.

[76] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, Mar./Apr. 2005.

[77] 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.

[78] A. Kumar, N. Jayam, A. Srinivasan, G. Senthilkumar, P. K. Baruah, S. Kapoor, M. Krishna, and R. Sarma. Feasibility study of mpi implementation on the heterogeneous multi-core cell be™ architecture. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures (SPAA'07)*, pages 55–56, New York, NY, USA, 2007. ACM.

[79] A. Kumar, G. Senthilkumar, M. Krishna, N. Jayam, P. K. Baruah, R. Sharma, A. Srinivasan, and S. Kapoor. A buffered-mode mpi implementation for the cell betm processor. In *Proceedings of the 7th international conference on Computational Science, Part I (ICCS'07)*, pages 603–610, Berlin, Heidelberg, 2007. Springer-Verlag.

[80] J. Kurzak and J. Dongarra. Implementation of mixed precision in solving systems of linear equations on the cell processor. *Concurr. Comput.: Pract. Exp.*, 19(10):1371–1385, 2007.

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

[82] D. Lanser and J. Verwer. Analysis of operator splitting for advection-diffusion-reaction problems from air pollution modeling. Technical Report MAS-R9805, Centrum voor Wiskunde en Informatica, 1998.

[83] B. Li, H. Jin, Z. Shao, Y. Li, and X. Liu. Optimized implementation of ray tracing on Cell Broadband Engine. In *Proceedings of the International Conference on Multimedia and Ubiquitous Engineering (MUE '08)*, pages 438–443, Busan, Apr. 2008.

[84] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. *Lecture Notes in Computer Science*, 5544/2009:884–892, May 2009.

[85] M. Mantor. AMD: Entering the golden age of heterogeneous computing. `http://ati.amd.com/technology/streamcomputing/IUCAA_Pune_PEEP_2008.pdf`, September 23–237 2008.

[86] J. Michalakes, J. Hacker, R. Loft, M. O. McCracken, A. Snavely, N. J. Wright, T. Spelce, B. Gorda, and R. Walkup. WRF nature run. In *Proceedings of the 2007 ACM/IEEE conference on supercomputing (SC'07)*, pages 1–6, New York, NY, USA, 2007. ACM.

[87] J. Michalakes and M. Vachharajani. GPU acceleration of numerical weather prediction. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08)*, pages 1–7, Miami, FL, Apr. 2008.

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

[89] P. Miehe, A. Sandu, G. R. Carmichael, Y. Tang, and D. Daescu. A communication library for the parallelization of air quality models on structured grids. *Atmospheric Environment*, 36(24):3917–3930, 2002.

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

[91] H. Muta, M. Doi, H. Nakano, and Y. Mori. Multilevel parallelization on the Cell/B.E. for a motion JPEG 2000 encoding server. In *Proceedings of the 15th International Conference on Multimedia (MULTIMEDIA '07)*, pages 942–951, New York, NY, USA, 2007. ACM.

[92] 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.

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

[94] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting openmp on cell. In *Proceedings of the 3rd international workshop on OpenMP (IWOMP'07)*, pages 65–76, Berlin, Heidelberg, 2008. Springer-Verlag.

[95] 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.

[96] M. Purcell, O. Callanan, and D. Gregg. Streamlining offload computing to high performance architectures. In G. Allen, J. Nabrzyski, E. Seidel, G. D. van Albada, J. Dongarra, and P. M. Sloot, editors, *Computational Science – ICCS 2009*, volume 5544 of *LNCS*, pages 974–983. Springer, 2009.

[97] M. M. Rafique, A. R. Butt, and D. S. Nikolopoulos. DMA-based prefetching for I/O-intensive workloads on the cell architecture. In *Proceedings of the 2008 Conference on Computing Frontiers (CF '08)*, pages 23–32, New York, NY, USA, 2008. ACM.

[98] J. Ray, C. Kennedy, S. Lefantzi, and H. Najm. High-order spatial discretizations and extended stability methods for reacting flows on structured adaptively refined meshes. In *Proceedings of Third Joint Meeting of the U.S. Sections of the Combustion Institute*, Chicago, USA, March 2003.

[99] H. H. Rosenbrock. Some general implicit processes for the numerical solution of differential equations. *Comput. J.*, 5:329–330, 1963.

[100] K. V. Ruchandani and H. Rawat. Implementation of spatial domain filters for cell broadband engine. In *First International Conference on Emerging Trends in Engineering and Technology (ICETET'08)*, pages 116–118, Nagpur, Maharashtra, July 2008.

[101] S. M. Sadjadi, L. Fong, R. M. Badia, J. Figueroa, J. Delgado, X. J. Collazo-Mojica, K. Saleem, R. Rangaswami, S. Shimizu, H. A. D. Limon, P. Welsh, S. Pattnaik, A. Praino, D. Villegas, S. Kalayci, G. Dasgupta, O. Ezenwoye, J. C. Martinez, I. Rodero, S. Chen, n. Javier Mu D. Lopez, J. Corbalan, H. Willoughby, M. McFail, C. Lisetti, and M. Adjouadi. Transparent grid enablement of weather research and forecasting. In *Proceedings of the 15th ACM Mardi Gras conference (MG'08)*, pages 1–8, New York, NY, USA, 2008. ACM.

[102] A. Sandu, J. G. Blom, E. Spee, J. G. Verwer, F. A. Potra, and G. R. Carmichael. Benchmarking stiff ODE solvers for atmospheric chemistry equations I – implicit vs. explicit. *Atmos. Env.*, 31:3151–3166, 1997.

[103] A. Sandu, J. G. Blom, E. Spee, J. G. Verwer, F. A. Potra, and G. R. Carmichael. Benchmarking stiff ODE solvers for atmospheric chemistry equations II – Rosenbrock solvers. *Atmos. Env.*, 31:3459–3472, 1997.

[104] A. Sandu, D. Daescu, G. Carmichael, and T. Chai. Adjoint sensitivity analysis of regional air quality models. *J. Comp. Phys.*, 204:222–252, 2005.

[105] B. Sanford. Integrated graphics solutions for graphics-intensive applications. White paper, Advanced Micro Devices, Inc., September 2002.

[106] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. on Graphics*, 27(3), 2008.

[107] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. C. Wang. Anton, a special-purpose machine for molecular dynamics simulation. In *Proceedings of the 34th annual international symposium on computer architecture (ISCA'07)*, pages 1–12, New York, NY, USA, 2007. ACM.

[108] W. C. Skamarock and J. B. Klemp. A time-split nonhydrostatic atmospheric model for weather research and forecasting applications. *J. Comput. Phys.*, 227(7):3465–3485, 2008.

[109] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, W. Wang, and J. G. Powers. A description of the advanced research WRF version 2. Technical report, National Center for Atmospheric Research, 2005.

[110] B. Sportisse. An analysis of operator splitting techniques in the stiff case. *J. Comp. Phys.*, (161):69–91, 2000.

[111] W. R. Stockwell, F. Kirchner, M. Kuhn, and S. Seefeld. A new mechanism for regional atmospheric chemistry modeling. *J. Geophys. Res.*, 102:15847–25879, 1997.

[112] 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.

[113] G. Strang. On the construction and comparison of difference schemes. *SIAM J. Numer. Anal.*, 5(3):506–517, 1968.

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

[115] The IBM BlueGene Team. An overview of the BlueGene/L supercomputer. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC'02)*, Nov. 2002.

[116] J. Verwer. Runge Kutta method for parabolic partial differential equations. *Applied Numerical Mathematics*, 22:359–379, 1996.

[117] 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.

[118] H. Wei and J. Yu. Loading OpenMP to Cell: An effective compiler framework for heterogeneous multi-core chip. In *Proceedings of the 3rd international workshop on OpenMP (IWOMP'07)*, pages 129–133, Berlin, Heidelberg, 2008. Springer-Verlag.

[119] 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.

[120] 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.

[121] R. A. Zaveri, R. C. Easter, J. D. East, and L. K. Peters. Model for simulating aerosol interactions and chemistry (MOSAIC). *J. Geophys. Res.*, 113(D13204), July 2008.

[122] M. Zhang, I. Uno, R. Zhang, Z. Han, Z. Wang, and Y. Pu. Evaluation of the models-3 community multi-scale air quality (CMAQ) modeling system with observations obtained during the TRACE-P experiment: Comparison of ozone and its related species. *Atmos. Env.*, 40(26):4874–4882, August 2006.

[123] Z. Zheng and L. Petzold. Runge-Kutta Chebyshev projection method. *J. Comp. Phys.*, 2005.

[124] S. Zhou. Software support for hybrid computing. In preperation., November 2009.

[125] Z. Zhu, Q. Wang, B. Feng, and L. Shao. Speech codec optimization based on Cell Broadband Engine. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '07)*, volume 2, pages 805–808, Honolulu, HI, Apr. 2007.

[126] AMD "Close to Metal" technology unleashes the power of stream computing. http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~114147,00.html, November 2006.

[127] ATI stream technology: GPU technology for accelerated computing. http://ati.amd.com/technology/streamcomputing/, November 2008.

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

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

[130] The Green500 list. http://www.green500.org/, November 2009.

[131] Intel math kernel library. Reference Manual 630813-031US, Intel Corporation, March 2009.

[132] NVIDIA's next generation CUDA compute architecture: Fermi. Whitepaper v1.1, NVIDIA Corporation, 2009.

[133] The Top500 list. http://www.top500.org/, November 2009.

[134] Tuning and analysis utilities. http://www.cs.uoregon.edu/research/tau/, April 2010.

[135] Virginia tech terascale computing facility. http://www.arc.vt.edu/arc/SystemX/, April 2010.

[136] Platform brief: Intel Xeon processor 5400 series. http://www.intel.com/cd/channel/reseller/asmo-na/eng/products/server/processors/q5400/feature/index.htm, Accessed 15 Sept. 2009.

Appendix A

Program Source Code

Listing A.1: X-axis mass flux calculation for blocks with more than one row where a whole row fits in local storage

```
1  /* Start buffer 0 transfer */
2  fetch_x_buffer(0, 0);
3
4  /* Start buffer 1 transfer */
5  fetch_x_buffer(1, NX_ALIGNED_SIZE);
6
7  /* Process buffer 0 */
8  transport_buffer(0, size, dt);
9
10 /* Loop over rows in this block */
11 for(i=0; i<block-2; i++)
12 {
13   w = i % 3;
14   p = (i+1) % 3;
15   f = (i+2) % 3;
16
17   /* Write buffer w back to main memory (nonblocking) */
18   write_x_buffer(w, i*NX_ALIGNED_SIZE);
19
20   /* Start buffer f transfer (nonblocking) */
21   fetch_x_buffer(f, (i+2)*NX_ALIGNED_SIZE);
22
23   /* Process buffer p */
24   transport_buffer(p, size, dt);
25 }
26
27 /* Discretize final row */
28 w = i % 3;
29 p = (i+1) % 3;
30
31 /* Write buffer w back to main memory (nonblocking) */
32 write_x_buffer(w, i*NX_ALIGNED_SIZE);
33
34 /* Process buffer p */
35 transport_buffer(p, size, dt);
36
37 /* Write buffer p back to main memory (nonblocking) */
38 write_x_buffer(p, (i+1)*NX_ALIGNED_SIZE);
39
40 /* Make sure DMA is complete before we exit */
41 mfc_write_tag_mask( (1<<w) | (1<<p) );
42 spu_mfcstat(MFC_TAG_UPDATE_ALL);
```

Listing A.2: Stream processing kernel

```
1  inline real_t
2  advec_diff(real_t cell_size,
3    real_t c2l, real_t w2l, real_t d2l,
4    real_t c1l, real_t w1l, real_t d1l,
5    real_t   c, real_t   w, real_t   d,
6    real_t c1r, real_t w1r, real_t d1r,
7    real_t c2r, real_t w2r, real_t d2r)
8  {
9    real_t wind, diff_term;
10   real_t advec_term, advec_termL, advec_termR;
11
12   wind = (w1l + w) / 2.0;
13   if(wind >= 0.0) advec_termL = (1.0/6.0) * ( -c2l + 5.0*c1l +
        2.0*c );
14   else advec_termL = (1.0/6.0) * ( 2.0*c1l + 5.0*c - c1r );
15   advec_termL *= wind;
16   wind = (w1r + w) / 2.0;
17   if(wind >= 0.0) advec_termR = (1.0/6.0) * ( -c1l + 5.0*c + 2.0*
        c1r );
18   else advec_termR = (1.0/6.0) * ( 2.0*c + 5.0*c1r - c2r );
19   advec_termR *= wind;
20   advec_term = (advec_termL - advec_termR) / cell_size;
21   diff_term = ( ((d1l+d)/2)*(c1l-c) - ((d+d1r)/2)*(c-c1r) ) / (
        cell_size * cell_size);
22   return advec_term + diff_term;
23 }
```

Listing A.3: Vectorized stream processing kernel

```
 1 inline vector real_t
 2 advec_diff_v(vector real_t cell_size,
 3   vector real_t c2l, vector real_t w2l, vector real_t d2l,
 4   vector real_t c1l, vector real_t w1l, vector real_t d1l,
 5   vector real_t   c, vector real_t   w, vector real_t   d,
 6   vector real_t c1r, vector real_t w1r, vector real_t d1r,
 7   vector real_t c2r, vector real_t w2r, vector real_t d2r)
 8 {
 9   vector real_t acc1, acc2, acc3;
10   vector real_t wind, diff_term, advec_term;
11   vector real_t advec_term_pos, advec_term_neg;
12   vector real_t advec_termR, advec_termL;
13
14   acc1 = spu_add(w1l, w);
15   wind = spu_mul(acc1, HALF);
16   acc1 = spu_mul(c1l, FIVE);
17   acc2 = spu_mul(c, TWO);
18   advec_term_pos = spu_add(acc1, acc2);
19   advec_term_pos = spu_sub(advec_term_pos, c2l);
20   acc1 = spu_mul(c1l, TWO);
21   acc2 = spu_mul(c, FIVE);
22   advec_term_neg = spu_add(acc1, acc2);
23   advec_term_neg = spu_sub(advec_term_neg, c1r);
24   acc1 = (vector real_t)spu_cmpgt(wind, ZERO);
25   acc1 = spu_and(acc1, advec_term_pos);
26   acc2 = (vector real_t)spu_cmpgt(ZERO, wind);
27   acc2 = spu_and(acc2, advec_term_neg);
28   advec_termL = spu_add(acc1, acc2);
29   advec_termL = spu_mul(advec_termL, SIXTH);
30   advec_termL = spu_mul(advec_termL, wind);
31   acc1 = spu_add(w1r, w);
32   wind = spu_mul(acc1, HALF);
33   acc1 = spu_mul(c, FIVE);
34   acc2 = spu_mul(c1r, TWO);
35   advec_term_pos = spu_add(acc1, acc2);
36   advec_term_pos = spu_sub(advec_term_pos, c1l);
37   acc1 = spu_mul(c, TWO);
38   acc2 = spu_mul(c1r, FIVE);
39   advec_term_neg = spu_add(acc1, acc2);
40   advec_term_neg = spu_sub(advec_term_neg, c2r);
41   acc1 = (vector real_t)spu_cmpgt(wind, ZERO);
42   acc1 = spu_and(acc1, advec_term_pos);
43   acc2 = (vector real_t)spu_cmpgt(ZERO, wind);
44   acc2 = spu_and(acc2, advec_term_neg);
```

```
45    advec_termR = spu_add(acc1, acc2);
46    advec_termR = spu_mul(advec_termR, SIXTH);
47    advec_termR = spu_mul(advec_termR, wind);
48    acc1 = spu_sub(advec_termL, advec_termR);
49    advec_term = VEC_DIVIDE(acc1, cell_size);
50    acc1 = spu_add(d1l, d);
51    acc1 = spu_mul(acc1, HALF);
52    acc3 = spu_sub(c1l, c);
53    acc1 = spu_mul(acc1, acc3);
54    acc2 = spu_add(d, d1r);
55    acc2 = spu_mul(acc2, HALF);
56    acc3 = spu_sub(c, c1r);
57    acc2 = spu_mul(acc2, acc3);
58    acc1 = spu_sub(acc1, acc2);
59    acc2 = spu_mul(cell_size, cell_size);
60    diff_term = VEC_DIVIDE(acc1, acc2);
61    return spu_add(advec_term, diff_term);
62  }
```

Appendix B

Annotated List of Figures

2.1 The theoretical peak performance of four multi-core architectures in gigaflops and gigaflops-per-watt. 4

2.2 The Cell Broadband Engine Architecture. All elements are explicitly controlled by the programmer. Careful consideration of data alignment and use of the SPU vector ISA produces maximum performance. End-to-end control of the EIB is the main limiter of overall performance. 6

2.3 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. 10

2.4 Overview of CUDA. A thread is the basic unit of parallelism. Threads are grouped into blocks, which are aggregated into grids. Threads within a block can communicate via the shared memory; threads in separate blocks neither communicate nor access each other's shared memory. 11

2.5 Example Sequoia memory trees. Sequoia facilitates portability by abstracting the memory hierarchy. 14

3.1 Runtime profiles of the Sulfur Transport and dEposition Model (STEM) executing on the System X supercomputer. Chemistry dominates even as the number of threads increases. 17

3.2 Runtime profile of WRF-Chem running on an Intel Quad-core Xeon 5500 series CPU. The data set is the RADM2 chemical mechanism combined with the SORGAM aerosol mechanism on a typically-sized $40 \times 40 \times 20$ domain grid. . 19

3.3 Runtime profile of serial and parallel runs of GEOS-Chem using the $4° \times 5°$ GEOS grid data. 21

4.1 A general outline of the three-stage Rosenbrock solver for chemical kinetics. . . 29

4.2 Accelerated RADM2 speedup as compared to the original serial code. A horizontal line indicates the maximum GPU speedup since the thread models of the GPU and conventional architectures are not directly comparable. Fine grid tests could not be conducted on some platforms due to memory limitations. . . 34

4.3 CBEA implementation of RADM2. The master (PPE) prepares model data for processing by the workers (SPEs). 39

4.4 Principle KPPA components and KPPA program flow. 41

4.5 Abbreviated UML diagram of KPPA. Abstract base classes allow future languages and architectures to be added. Subclasses of the `Model_Chem` class define legal pairs of languages and architectures. 43

4.6 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 2778 cycles. The pattern repeats with minor variations for 2116 of 2778 cycles. 45

4.7 Speedup of KPPA-generated chemical kernels as compared to SSE-enhanced serial code. A horizontal 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. 48

4.8 Wall clock OpenMP performance of the RADM2 chemical mechanism on two cutting edge homogeneous multi-core chipsets compared with emerging multicore architectures. 50

5.1 Data dependencies for implicit and explicit time-stepping. 55

5.2 Stability regions of Runge-Kutta-Chebyshev explicit time-stepping method for three values of s. 59

5.3 STEM speedup on System-X (RKC is parallel explicit, CN is parallel implicit) 60

5.4 3D discretization stencil for explicit upwind-biased advection/diffusion 62

5.5 Second order time splitting methods for 2D and 3D transport discretization. . 63

5.6 FIXEDGRID data storage scheme for the 3D transport module showing padding and vectorization with double-precision data types 63

5.7 Domain decomposition for two-dimensional transport. 65

5.8 Performance of the 2D transport module executing on JUICEnext (IBM BladeCenter Q22) when matrix columns are buffered by the PPU (Opt. Fcn. Offload) and when DMA lists are used (Scalable DMA) 68

5.9 Wall clock runtime (seconds) and speedup for the 2D transport module on three CBEA systems and two homogeneous multi-core systems. 71

5.10 Wall clock runtime (seconds) and speedup for the 3D transport module on three CBEA systems and two homogeneous multi-core systems. 76

6.1 Ground-level concentrations and temperatures from the WRF-Chem case study. 80

6.2 Runtime profile of WRF-Chem executing on an Intel Xeon workstation. 81

6.3 Runtime profile of WRF-Chem executing on Quasar, an IBM BladeCenter QS22. 83

6.4 Runtime profile of WRF-Chem executing on Deva with RADM2 offloaded to a Sony Playstation 3. 85

6.5 Predicted runtime profile of WRF-Chem executing on Deva with KPPA-generated RADM2. 86