Chemical Kinetics on Multi-core SIMD Architectures

John C. Linford and Adrian Sandu

Virginia Polytechnic Institute and State University
Blacksburg VA 24060, USA
jlinford@vt.edu, sandu@cs.vt.edu

Abstract. Chemical kinetics modeling accounts for a significant portion of the computational time of atmospheric models. Effective application of multiple levels of heterogeneous parallelism can significantly reduce computational time, but implementation on emerging multi-core technologies can be prohibitively difficult. We introduce an approach for chemical kinetics modeling on multi-core SIMD architectures we call *abstract vectors* which exposes multi-layered heterogeneous parallelism. The scalar types of elements of the model state data are replaced with abstract multi-dimensional types. An implementation of the vector type and its operations can then be optimized for a given multi-core platform. This approach exposes SIMD parallelism on a fixed grid by aggregating like operations. We implemented abstract vectors in the Kinetics Preprocessor (KPP) and used our modified KPP to implement SAPRC models optimized for SSE, OpenMP, and NVIDIA CUDA.

1 Introduction

Many atmospheric models approximate the chemical state of the Earth's atmosphere by applying a chemical kinetics model over a regular grid. The Community Multiscale Air Quality Modeling System (CMAQ) [1], the Weather Research and Forecasting with Chemistry model (WRF/Chem) [2], and the Sulfur Transport and dEposition Model (STEM) [3] are examples of such models. Chemical kinetics models trace the evolution of chemical species over time by solving large numbers of coupled partial differential equations. Computational time is dominated by the solution of the coupled and stiff equations arising from the chemical reactions, which may involve millions of variables. Such models often require hours or days of supercomputer time for realistic regional simulations, of which up to 90% may be accounted for by chemical kinetics alone.

Increasing power consumption, heat dissipation, and other issues have lead to the rapid prevalence of multi-core microprocessor chip architectures in recent years. Massively-parallel chipsets with many, possibly heterogeneous, cores and SIMD capabilities are becoming commonplace. Multi-layered heterogeneous parallelism offers an impressive number of FLOPS, but at the cost of staggering hardware and software complexity. Effective application of multiple levels of heterogeneous parallelism can significantly reduce the computational time of

chemical kinetics simulations, yet implementing these models on emerging multi-core technologies can be prohibitively difficult.

We introduce *abstract vectors* as a method for applying multi-layered heterogeneous parallelism to chemical kinetics simulations on a regular grid. Each scalar element of the model state data is replaced by an abstract vector type of configurable length. Operations on the state vectors are performed as a sequence of simple vector arithmetic operations (add, multiply, axpy, etc.) on the vector elements of the state data. This method applies the same operation to every element of the abstract vector, exposing SIMD parallelism and greatly simplifying parallel implementation of the model on SIMD architectures.

We implemented the abstract vector approach in the Kinetic Preprocessor (KPP) [4]. Our modified KPP generates arbitrary models in terms of abstract vectors. Implementing a model on a SIMD architecture is as easy as specifying the basic abstract vector arithmetic operations for the platform's preferred parallelization method, and a higher-level parallelization most appropriate for the platform can be applied. We used the modified KPP to implement a single-precision SAPRC [5] model of 79 species and 211 reactions on Streaming SIMD Extensions (SSE), OpenMP with SSE, and NVIDIA's Compute Unified Device Architecture (CUDA) [9]. The SAPRC model using both SSE and OpenMP parallelization achieves almost five times the speedup of a serial KPP-generated SAPRC model.

2 GPUs and the Compute Unified Device Architecture

Graphics Processing Units (GPUs) are low-cost, low-power (watts per flop), massively-parallel homogeneous microprocessors designed for visualization and gaming. Because of their power, these special-purpose chips have received a lot of attention as general-purpose computers (GPGPUs). Examples include the NVIDIA GeForce GTX 280 with 240 1296MHz processing units and a theoretical single-precision peak of 933 gigaFLOPS [6], and the ATI Radeon HD 4870 with 800 750MHz processing units and a single-precision theoretical peak of 1.2 teraFLOPS [7]. GPUs are often an order of magnitude faster than CPUs, and GPU performance has been increasing at a rate of 2.5x to 3.0x annually, compared with 1.4x for CPUs [8]. GPU technology has the additional advantage of being widely-depoloyed in modern computing systems. Many desktop workstations have GPUs which can be harnessed for scientific computing at no additional cost to the user.

Expressed as a minimal extension of the C and C++ programming languages, CUDA [9] 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 synchroniza-

tion 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 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 similar to 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 underlying system.

Since NVIDIA released the Compute Unified Device Architecture (CUDA) in 2007, CUDA has become the target of massive development activity with tens of thousands of registered CUDA developers. Hundreds of scalable parallel programs for a wide range of applications, including computational chemistry, sparse matrix solvers, sorting, searching, and physics models have been developed [10]. 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 [11].

3 The Kinetic Preprocessor (KPP)

The Kinetic Preprocessor (KPP) [4] is a general analysis tool that facilitates the numerical solution of chemical reaction network problems. KPP 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. It incorporates a library of several widely-used atmospheric chemistry mechanisms and has been successfully used to treat many chemical mechanisms from tropospheric and stratospheric chemistry, including CBM-IV [15], SAPRC [5], and NASA HSRP/AESA.

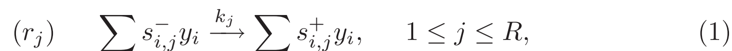
3.1 Chemical Kinetics Modeling

Given and the initial concentrations, KPP solves the differential equation of mass action kinetics to completely determine the concentrations at any future time. The derivation of this equation is given at length in [4] and summarized here.

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

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

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



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

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

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

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

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

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

If we organize the stoichiometric coefficients in two matrices,

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

then Equation 3 can be rewritten as

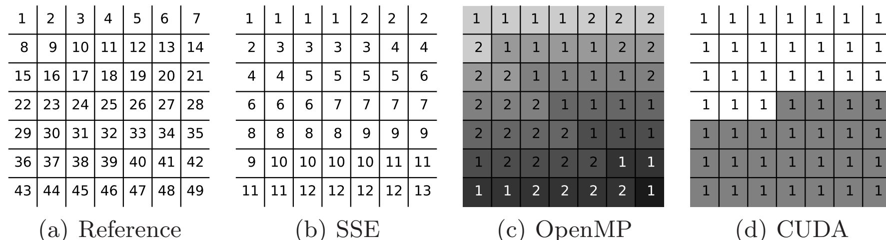
$$\frac{d}{dt} y = (S^+ - S^-) \omega(t, y) = S \omega(t, y) = f(t, y), \tag{4}$$

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

Equation 4 gives the time derivative function in standard aggregate form. Depending on the integration method, other forms, such as a split production-destruction form may be preferred. Given the time derivative function and the initial concentrations, the solution of the ordinary differential equations can be traced in time using a numerical integration method.

4 Vector KPP

In essence, an *abstract vector* is a pointer to floating point data on the heap. Vector operations are performed by applying a scalar operation while looping over the floating point data. By working with pointers, a chemical mechanism



(a) Reference (b) SSE (c) OpenMP (d) CUDA

Fig. 1. Parallelization in reference, SSE, and OpenMP with SSE, and CUDA versions of the SAPRC mechanism. Numbers indicate consecutive moments in time; background shading indicates level 1 parallelization.

implemented with abstract vectors can be applied to any contiguous data in an existing model.

We modified KPP to generate kinetics models in terms of abstract vectors. The abstract vector type is described by a C structure encapsulating a pointer to the first element, the vector length, and the vector pitch (or stride). As many members as needed can be added to the structure without altering code correctness. Fifteen element-wise vector operations – assign, negate, absolute value, addition, subtraction, multiplication, division, exponentiation, log10, square root, power, maximum, axpy, element get, element set – were defined. Several variations of these functions were required for vector-scalar operations, making a total of twenty-nine functions. We call these twenty-nine functions *level 0 functions* since they expose the finest granularity of SIMD parallelism. Our modified KPP is called KPP-VECT.

4.1 Model Code Generation

There are five major calculations performed many times during the time step integration: LU decomposition (`KPPDecomp()`), matrix solution by back substitution (`KPPSolve()`), the solution to the coupled ODE system (`Fun()`), calculation of the Jacobian (`Jac()`), and calculation of the reaction rates (`Update_RCONST()`). These calculations account for almost 95% of the model time, with `KPPDecomp()` and `Fun()` being the most expensive in the SAPRC model.

KPP generates large source files of completely unrolled instructions for each of these computations. In order to use abstract vectors, we replaced the infix grammar of the KPP code generator module with a prefix grammar that uses calls to the level 0 functions as operators. C code for `Fun()` as generated by KPP-VECT is shown in Listing 1.1. The code for numerical integration (i.e. a Rosenbrock method) is copied from pre-written source files. These files were modified to operate on abstract vectors to produce a second code library for KPP code generation.

KPP-VECT generates kinetics models in C. Implementation of the level 0 functions would have been much simpler for a high-level language with support

```
1  /* Original source:
2  Vdot[1] = A[127]+0.333*A[161]+0.351*A[166]+0.1*A[170]+0.37
3                 *A[185]+0.204*A[189]+0.103*A[193]+0.121*A[197]
4                 +0.185*A[204]+0.073*A[208];
5  */
6  mulsv(&Vdot[1], 0.333, &A[161]);
7  addvv(&Vdot[1], &A[127], &Vdot[1]);
8  mulsv(&_t7t_, 0.351, &A[166]);
9  addvv(&Vdot[1], &Vdot[1], &_t7t_);
10 mulsv(&_t6t_, 0.1, &A[170]);
11 addvv(&Vdot[1], &Vdot[1], &_t6t_);
12 mulsv(&_t5t_, 0.37, &A[185]);
13 addvv(&Vdot[1], &Vdot[1], &_t5t_);
14 mulsv(&_t4t_, 0.204, &A[189]);
15 addvv(&Vdot[1], &Vdot[1], &_t4t_);
16 mulsv(&_t3t_, 0.103, &A[193]);
17 addvv(&Vdot[1], &Vdot[1], &_t3t_);
18 mulsv(&_t2t_, 0.121, &A[197]);
19 addvv(&Vdot[1], &Vdot[1], &_t2t_);
20 mulsv(&_t1t_, 0.185, &A[204]);
21 addvv(&Vdot[1], &Vdot[1], &_t1t_);
22 mulsv(&_t0t_, 0.073, &A[208]);
23 addvv(&Vdot[1], &Vdot[1], &_t0t_);
```

Listing 1.1. Excerpt from the `Fun()` source code as generated by KPP-VECT

for operator or function overloading (such as C++), however many emerging multi-core architectures support only C or C variants. Future work will add support for other languages, such as Fortran, C++, and MATLAB. (The scalar implementation of KPP can generate C, FORTRAN 77, Fortran 90, and MATLAB code.)

4.2 Level 0 parallelization

The default implementation of the level 0 functions generated by KPP-VECT unrolls vector loops, but performs no further optimization. Optimization for a specific SIMD ISA is easily accomplished by modifying any or all of the twenty-nine level 0 functions. The default implementations are suitable for use on any platform, even those without branch prediction. The default implementation of the level 0 axpy operator is shown in Listing 1.2. SSE level 0 parallelization uses the XMM registers to compute four elements of the abstract vector simultaneously, as shown in Listing 1.3. Level 0 parallelization with NVIDIA CUDA divides the abstract vector into blocks and calculates each element of the solution vector with one lightweight thread. The CUDA implementation of the axpy operator is shown in Listing 1.4.

4.3 Level 1 parallelization

In general, a level 1 parallelization should subdivide the abstract vectors among several processes. Excluding specialized hardware, a level 1 parallelization should not be implemented in the twenty-nine level 0 functions, since the level 1 parallelization will generally incur more overhead than the level 0. Thread synchronization, explicit memory transfer, and vector packing may be involved, and the

```
1  void axpyvv(const float alpha, ref_vector_real_t X,
       ref_vector_real_t Y)
2  {
3    size_t i;
4    const size_t limit = Y->length;
5    const size_t blocklimit = (limit / 8) * 8;
6    float * const Xe = X->e;
7    float * const Ye = Y->e;

9    for(i=0; i<blocklimit; i+=8) {
10     Ye[i] += alpha*Xe[i];
11     Ye[i+1] += alpha*Xe[i+1];
12     Ye[i+2] += alpha*Xe[i+2];
13     Ye[i+3] += alpha*Xe[i+3];
14     Ye[i+4] += alpha*Xe[i+4];
15     Ye[i+5] += alpha*Xe[i+5];
16     Ye[i+6] += alpha*Xe[i+6];
17     Ye[i+7] += alpha*Xe[i+7];
18   }
19   switch(limit - i) {
20     case 7: Ye[i] += alpha*Xe[i]; ++i;
21     case 6: Ye[i] += alpha*Xe[i]; ++i;
22     case 5: Ye[i] += alpha*Xe[i]; ++i;
23     case 4: Ye[i] += alpha*Xe[i]; ++i;
24     case 3: Ye[i] += alpha*Xe[i]; ++i;
25     case 2: Ye[i] += alpha*Xe[i]; ++i;
26     case 1: Ye[i] += alpha*Xe[i];
27   }
28 }
```

Listing 1.2. Default implementation of level 0 axpy operator

level 0 functions are called thousands of times during a typical model run. We implement the OpenMP parallelization in an outer loop in the top-level integration function.

Figure 1 describes parallelization in serial reference, serial SSE, and OpenMP with SSE versions of the SAPRC mechanism. Level 0 parallelization is represented with positive integer numbers; level 1 parallelization is represented with different background shadings. Level 1 parallelization will vary from platform to platform, particularly for emerging multi-core technologies (i.e. Cell Broadband Engine).

5 Experimental Results

Figure 2 shows the speedup of two versions of the SAPRC99 model using abstract vectors. The "SSE" version uses Streaming SIMD Extensions to accelerate the basic abstract vector operations. This is only one level of parallelism. The "OpenMP" version uses SSE for basic abstract vector operations and OpenMP to parallelize the computation over the grid (two layers of parallelism). Eight OpenMP threads are used for all vector sizes.

A CUDA implementation was completed, however our experimental results are not available at this time. We fully expect to have these results in the final copy. Our initial results suggest that the CUDA implementation achieves better

```
size_t i;
const size_t limit = Y->length;
const size_t blocklimit = limit / 4;
__m128 m;
__m128    A = _mm_set1_ps(alpha);
__m128 * Xe = (__m128*)X->e;
__m128 * Ye = (__m128*)Y->e;
for(i=0; i<blocklimit; i++) {
  m = _mm_mul_ps(A, *Xe);
  *Ye = _mm_add_ps(*Ye, m);
  Xe++; Ye++;
}
i *= 4;
switch(limit-i) {
case 3: Y->e[i] += alpha*X->e[i]; ++i;
case 2: Y->e[i] += alpha*X->e[i]; ++i;
case 1: Y->e[i] += alpha*X->e[i];
}
```

Listing 1.3. SSE implementation of level 0 axpy operator

```
__global__ void axpyvv_k(const float alpha, float * X, float * Y,
    int len)
{
  int idx = threadIdx.x + blockDim.x * blockIdx.x;
  if(idx < len) Y[idx] += alpha*X[idx];
}
void axpyvv(const float alpha, ref_vector_real_t X,
    ref_vector_real_t Y) {
  int B = 256;
  axpyvv_k <<< VLEN / B, B >>> (alpha, X->e, Y->e, VLEN);
}
```

Listing 1.4. CUDA implementation of level 0 axpy operator

performance than the OpenMP version. The principle difficulty in developing the CUDA implementation is maximizing the ratio of floating-point operations to memory copies. To this end, the five principle computations (`KppDecomp()`, etc.) were converted to thread kernels. This approach is only feasible for GPUs with very large on-board memories, such as the NVIDIA GeForce GTX 280.

6 Conclusion and Future Work

We have presented a method for exposing SIMD parallelism in chemical kinetics modeling called *abstract vectors*. We implemented abstract vectors in the Kinetic PreProcessor (KPP) and used the modified KPP to generate versions of a SAPRC model which leverage Streaming SIMD Extensions (SSE), SSE and OpenMP, and NVIDIA CUDA. Our vector-based SAPRC mechanism is approximately five times faster than the KPP-generated SAPRC model currently used in the CMAQ [1], WRF/Chem [2], and STEM [3] production models.

There are several improvements that could be made to the code KPP-VECT generates. First, memory I/O overhead can be reduced by aggregating operators into special-purpose level 0 vector operators. For example, computations of the

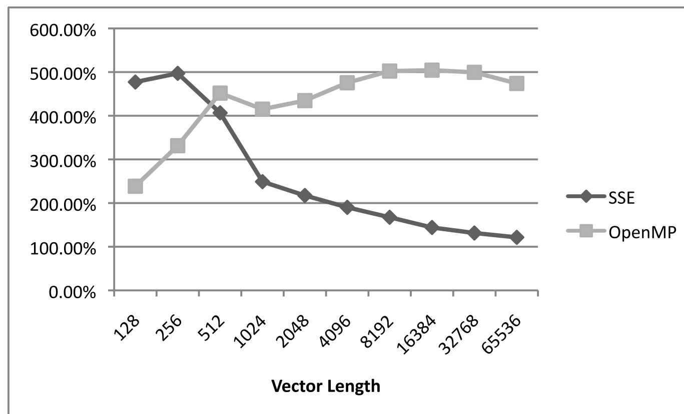


Fig. 2. Percent speedup over scalar KPP code for different vector lengths.

form $A = V_1 \times V_2 \times \ldots \times V_n$ occur frequently in the solution of the ODE function. An "n-multiply" kernel should be developed to reduce the number of memory copies. Furthermore, aggregating operations will reduce the number of function calls, thus reducing the code size and alleviating pressure on CBEA local storage.

The addition of a vector-of-vectors type would also improve performance. The current implementation of KPP-VECT uses arrays of abstract vectors in place of vectors of vectors, so contiguity of the data between vectors is not guaranteed. A vector-of-vector type would aggregate data, improving cache utilization and reducing the complexity of exposed communication processing.

Implementation on the Cell Broadband Engine Architecture (CBEA) requires further development. The performance of our initial CBEA implementation is unimpressive, due to the extremely small size of the synergistic processing element (SPE) local store. The SPE local store is shared between data and code. Ideally, a level 1 parallelization would offload the five principle computations (`KppDecomp()`, etc.) to the SPEs for processing. However, the compiled size of the five principle computations is much larger than the available 256 KB local storage. Portions of the program code must be manually swapped between main memory and local storage via overlays, and no more than four grid cells can be processed at a time. This overhead outweighs any benefit gained by moving computations to the SPE and reduces performance to that provided by the Power Processing Element (PPE), a standard PowerPC CPU. Alternately, the SPEs can be used to process the level 0 functions alone. However, the explicit communication overhead is prohibitive. We are exploring other approaches for the CBEA.

References

1. Byun, D.W., Ching, J.K.S.: 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)
2. Grell, G.A., Peckham, S.E., Schmitz, R., McKeen, S.A., Frost, G., Skamarock, W.C., Eder, B.: Fully coupled online chemistry within the WRF model. Atmos. Env. **39** (2005) 6957–6975
3. Carmichael, G.R., Peters, L.K., Kitada, T.: A second generation model for regional scale transport / chemistry / deposition. Atmos. Env. **20** (1986) 173–188
4. Damian, V., Sandu, A., Damian, M., Potra, F., Carmichael, G.R.: The Kinetic Preprocessor KPP – a software environment for solving chemical kinetics. Comput. Chem. Eng. **26** (2002) 1567–1579
5. Carter, W.P.L.: A detailed mechanism for the gas-phase atmospheric reactions of organic compounds. Atmos. Env. **24A** (1990) 481–518
6. : Technical brief: NVIDIA GeForce GTX 200 GPU architectural overview. Technical Report TB-04044-001_v01, NVIDIA Corporation (May 2008)
7. Mantor, M.: AMD: Entering the golden age of heterogeneous computing. http://ati.amd.com/technology/streamcomputing/IUCAA_Pune_PEEP_2008.pdf (September 23–237 2008)
8. Himawan, B., Vachharajani, M.: Deconstructing hardware usage for general purpose computation on GPUs. In: Fith Annual Workshop on Duplicating, Deconstructing, and Debunking (in conunction with ISCA-33). (2006)
9. Corporation, N.: NVIDIA CUDA compute unified device architecture: Programming guide version 2.0. http://www.nvidia.com/object/cuda_develop.html (2008)
10. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with cuda. In: ACM SIGGRAPH 2008 Courses (SIGGRAPH'08), New York, NY, USA, ACM (2008) 1–14
11. Stratton, J., Stone, S., mei Hwu, W.: MCUDA: An efficient implementation of CUDA kernels on multi-cores. Technical Report IMPACT-08-01, University of Illinois at Urbana-Champaign (March 2008)
12. Flachs, B., Asano, S., Dhong, S.H., Hofstee, H.P., Gervais, G., Kim, R., Le, T., et. al.: The microarchitecture of the synergistic processor for a cell processor. IEEE J. Solid State Circuits **41**(1) (2006) 63–70
13. Corporation, I.B.M.: PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual, http://www-306.ibm.com/chips/techlib. 2.07c edn. (October 2006)
14. Chen, T., Raghavan, R., Dale, J., Iwata, E.: Cell Broadband Engine Architecture and its first implementation. IBM developerWorks (June 2006)
15. Gery, M.W., Whitten, G.Z., Killus, J.P., Dodge, M.C.: A photochemical kinetics mechanism for urban and regional scale computer modeling. J. Geophys. Res. **94**(D10) (1989) 12925–12956