Development and Acceleration of Parallel Chemical Transport Models

Paul R. Eller

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

Master of Science
in
Computer Science and Applications

Dr.Adrian Sandu, Chair
Dr.Calvin J. Ribbens
Dr.Dimitris S. Nikolopoulos

July 14, 2009
Blacksburg, Virginia

Keywords: Chemical Transport Models, KPP, GEOS-Chem, STEM, Parallelization, GPU, CUDA

Development and Acceleration of Parallel Chemical Transport Models

Paul R. Eller

(ABSTRACT)

Improving chemical transport models for atmospheric simulations relies on future developments of mathematical methods and parallelization methods. Better mathematical methods allow simulations to more accurately model realistic processes and/or to run in a shorter amount of time. Parellization methods allow simulations to run in much shorter amounts of time, therefore allowing scientists to use more accurate or more detailed simulations (higher resolution grids, smaller time steps).

The state-of-the-science GEOS-Chem model is modified to use the Kinetic Pre-Processor, giving users access to an array of highly efficient numerical integration methods and to a wide variety of user options. Perl parsers are developed to interface GEOS-Chem with KPP in addition to modifications to KPP allowing KPP integrators to interface with GEOS-Chem. A variety of different numerical integrators are tested on GEOS-Chem, demonstrating that KPP provided chemical integrators produce more accurate solutions in a given amount of time than the original GEOS-Chem chemical integrator.

The STEM chemical transport model provides a large scale end-to-end application to experiment with running chemical integration methods and transport methods on GPUs. GPUs provide high computational power at a fairly cheap cost. The CUDA programming environment simplifies the GPU development process by providing access to powerful functions to execute parallel code. This work demonstrates the accleration of a large scale end-to-end application on GPUs showing significant speedups. This is achieved by implementing all relevant kernels on the GPU using CUDA. Nevertheless, further improvements to GPUs are needed to allow these applications to fully exploit the power of GPUs.

Acknowledgments

I would like to thank a number of people who have provided me with support and guidance over the last few years. I would like to thank my advisor Dr.Sandu for helping to guide my research over the last few years and for helping me to better understand the issues necessary to simulate large scale chemical transport models such as GEOS-Chem and STEM from both a mathematical and computational perspective.

I would also like to thank Kumaresh Singh for helping me understand and develop code for KPP and GEOS-Chem. I appreciate his advice when developing and accelerating code for GPUs using CUDA.

I would like to thank everyone in the Computational Science Lab for providing me with help, creating an enjoyable work environment, and for good company in the lab and on trips to conferences.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

As humanity has become more industrialized, we have produced a large variety of chemicals that both positively and negatively affect the atmosphere. Industrialism lead to pollution that contaminates the air and water, demonstrating that we need to take measures to better understand our effect on the environment as well as to protect the environment to ensure not only our health but to provide future generations with a clean and safe environment to live in.

Scientists have been studying the atmosphere for years and have developed mathematical models of the different naturally occurring processes in addition to the effects of humans on the environment. These mathematical models are implemented on computers to produce simulations of the atmosphere. The field of computational science has grown as scientists search for mathematical methods to more quickly and accurately simulate the atmosphere as well as other processes. Accurate simulations take very large amounts of computing power, limiting the detail that scientists can use when creating simulations that finish in a reasonable amount of time.

The development of multiprocessor systems provided scientists with access to larger amounts of computing power to run simulations through allowing computations to be done in parallel. This allows more accurate and detailed simulations to occur in a reasonable amount of time. However, these programs are significantly more complicated as programmers must specify how the computations and data are divided among the processors.

Therefore scientists need access to tools which are both very fast and very accurate to produce the most useful simulations. One approach to improve speed and accuracy of simulations is to replace slower and less accurate methods with faster and more accurate methods. We can provide users with more options to tune their simulation, which can allow them to produce more accurate simulations for a given problem. The second approach is to develop parallel codes. This can allow scientists to run simulations in less time, or more commonly to run more accurate simulations in a given amount of time.

1.2 Overview of Research

This research has provided the atmospheric modeling community with the tools needed to more quickly and accurately perform simulations. The state-of-the-science GEOS-Chem model is improved through adding functionality and further parallelizing the model. The test chemical transport model STEM is used to study the effects of parallelizing atmospheric processes for GPUs. The results produced by STEM provide insight into the best techniques for parallelizing larger models such as GEOS-Chem for GPUs as well as GPU bottlenecks that prevent chemical transport models from running efficiently on GPUs.

GEOS-Chem is modified to use KPP for the chemistry modeling. KPP provides users with access to a wide variety of numerical integrators and tuning parameters in order to accurately perform simulations. KPP achieves a high level of efficiency through producing models that implement sparse data structures and OpenMP parallelism. Additional tools are provided to quickly produce KPP input files based on GEOS-Chem input files, produce KPP code

capable of interfacing with GEOS-Chem, and modifying GEOS-Chem code to interface with KPP. GEOS-Chem is then tested for a number of different integrators to find the accuracy and running time.

STEM provides users with a simplified chemical transport model in order to test different numerical methods and parallelization methods. This allows users to more quickly and easily perform experiments which can then be applied to larger more complicated models such as GEOS-Chem. STEM uses a KPP generated chemistry routine with three one dimensional transport routines developed for STEM. Each of these routines is modified to run on GPUs and then accelerated to produce accurate results as quickly as possible. The data produced by these experiments provides valuable information for developing optimal GPU routines for larger models or finding bottlenecks on the GPU that need to be improved to allow chemical transport models to run quickly.

The large scale STEM model allows us to see the effects of accelerating an end-to-end real world application as opposed to the smaller idealized kernels used in most studies. This more clearly demonstrates the effectiveness of these numerical methods and parallelization methods for real world problems along with the challenges faced when developing applications for use in the real world.

1.3 Outline of Thesis

The rest of this thesis is organized as follows. Chapter 2 describes the integration of the KPP tools in the GEOS-Chem framework, along with results demonstrating the effectiveness of this implementation. Chapter 3 discusses the background for STEM, GPUs, and CUDA, along with development of a test version of STEM. Chapter 4 discusses development of GPU transport for STEM, including the mathematics and implementation details for implicit and explicit transport methods. Chapter 5 discusses the development of GPU chemistry for both the memory-intensive Rosenbrock method and the computation-intensive QSSA method.

Chapter 6 discusses the results produced by STEM for transport, chemistry, and full STEM. Chapter 7 provides conclusions based on this work.

Chapter 2

GEOS-Chem KPP

In this section we discuss the incorporation of the KPP generated gas-phase chemistry simulation code in GEOS-Chem. This research on GEOS-Chem KPP is published in [8]. Enriching GEOS-Chem with the Kinetic PreProcessor [32] chemical solvers will give GEOS-Chem users access to the high performance algorithms contained within KPP. KPP provides solvers for the computation of adjoints, which can be used for sensitivity analysis and data assimilation. Figure 2.1 shows the computational flow for using KPP to implement GEOS-Chem gas-phase chemistry. The geos2kpp_parser.pl is applied to the chemical mechanism description file globchem.dat to create KPP input files. KPP uses these input files to generate the KPP model GEOS-Chem uses for the gas-phase chemistry. This model is then copied into the GEOS-Chem v7-04-10 directory and gckpp_parser.pl is applied to this directory to create a KPP based version of GEOS-Chem.

A simple three step process modifies GEOS-Chem version 7-04-10 to use KPP. This process is summarized in Figure 2.1. First, the geos2kpp_parser.pl reads the chemical mechanism description file globchem.dat and creates three KPP input files - also containing information on the chemical mechanism, but in KPP format. Second, KPP processes these input files and generates Fortran90 chemical simulation code; the model files are then copied into the GEOS-Chem v7-04-10 directory. A special directive has been implemented in KPP to generate code

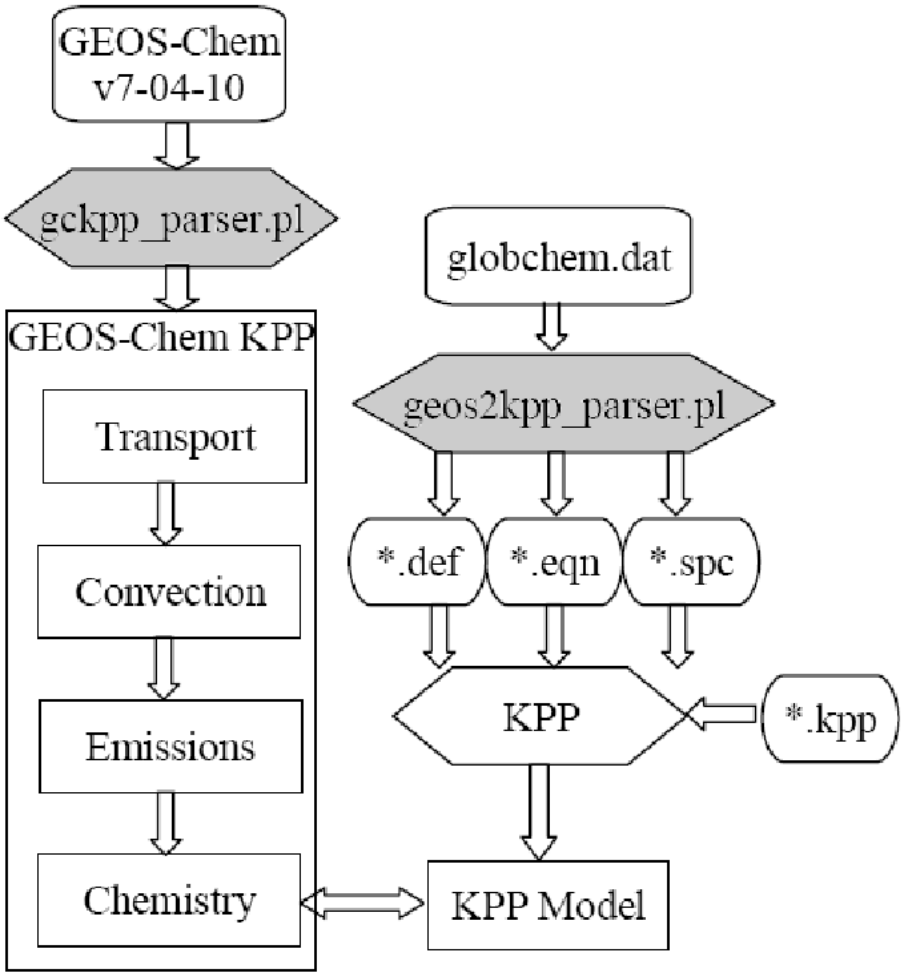


Figure 2.1: Flow diagram of implementing GEOS-Chem chemistry with KPP.

that interfaces with GEOS-Chem. The third step involves running the gckpp_parser.pl to modify GEOS-Chem source code to correctly call the KPP generated gas-phase chemistry simulation routines. The process is discussed in detail below.

2.1 Background

GEOS-Chem [12] is a state-of-the-science global 3-D model of atmospheric composition driven by assimilated meteorological observations from the Goddard Earth Observing System (GEOS) of the NASA Global Modeling Assimilation Office (GMAO). GEOS-Chem is used by the atmospheric research community for activities such as assessing intercontinental transport of pollution, evaluating consequences of regulations and climate change on air quality, comparison of model estimates to satellite observations and field measurements, and fundamental investigations of tropospheric chemistry. KPP was originally interfaced with GEOS-Chem and its adjoint in Henze et al. [18], see Appendices therein. Here we improve upon this implementation in terms of automation, performance, benchmarking, and documentation.

The GEOS-Chem native chemistry solver is the Sparse Matrix Vectorized GEAR II (SMVGEARII) code which implements backward differentiation formulas and efficiently solves first-order ordinary differential equations with initial value boundary conditions in large grid-domains (Jacobson and Turco [22]; Jacobson [21]). These sparse matrix operations reduce the CPU time associated with LU-decomposition and make the solver very efficient. This code vectorizes all loops about the grid-cell dimension and divides the domain into blocks of grid-cells and vectorizes around these blocks to achieve at least 90% vectorization potential. SMVGEARII uses reordering of grid cells prior to each time interval to group cells with stiff equation together and non-stiff equations together. This allows SMVGEARII to solve equations both quickly and with a high order of accuracy in multiple grid-cell models.

Calculation of tropospheric chemistry constitutes a significant fraction of the computational expense of GEOS-Chem. Given the push for running GEOS-Chem at progressively finer resolutions, there is a continual need for efficient implementation of sophisticated numerical methods. This requires a seamless, automated implementation for the wide range of users in the GEOS-Chem community.

The Kinetic PreProcessor KPP [32] is a software tool to assist computer simulations of chemical kinetic systems (Damian et al. [7]; Sandu and Sander [34]). The concentrations of a chemical system evolve in time according to the differential law of mass action kinetics. A computer simulation requires an implementation of the differential law and a numerical integration scheme. KPP provides a simple natural language to describe the chemical mechanism and uses a preprocessing step to parse the chemical equations and generate efficient output code in a high level language such as Fortran90, C, or Matlab. A modular design allows rapid prototyping of new chemical kinetic schemes and new numerical integration methods. KPP is being widely used in atmospheric chemistry modeling (Kerkweg et al. [24]; Hakami et al. [17]; Carmichael et al. [3]; Errera and Fonteyn [11]; Errera et al. [10]; Henze et al. [18]; Henze et al. [19]). The simulation code can be easily re-generated following changes to the chemical mechanism. KPP–2.2 is distributed under the provisions of the GNU license (`http://www.gnu.org/copyleft/gpl.html`) and the source code and documentation are available at [32].

KPP incorporates a comprehensive library of stiff numerical integrators that includes Rosenbrock, fully implicit Runge-Kutta, and singly diagonally implicit Runge-Kutta methods (Sandu et al. [35, 36]). These numerical schemes are selected to be very efficient in the low to medium accuracy regime. Users can select the method of choice and tune integration parameters (e.g., relative and absolute tolerances, maximal step size) via the optional input parameter vectors ICNTRL_U and RCNTRL_U. High computational efficiency is attained through fully exploiting sparsity. KPP computes analytically the sparsity patterns of the Jacobian and Hessian of the chemical ODE function, and outputs code that implements the sparse data structures. In this work we make the suite of KPP stiff solvers and the sparse linear algebra routines available in GEOS-Chem.

KPP provides tangent linear, continuous adjoint, and discrete adjoint integrators, which are needed in sensitivity analysis and data assimilation studies (Daescu et al. [6]; Sandu et al [33]). Flexible direct-decoupled and adjoint sensitivity code implementations are achieved with minimal user intervention. The resulting sensitivity code is highly efficient, taking full

advantage of the sparsity of the chemical Jacobians and Hessians. Adjoint modeling is an efficient tool to evaluate the sensitivity of a scalar response function with respect to the initial conditions and model parameters. 4D-Var data assimilation allows the optimal combination of a background estimate of the state of the atmosphere, knowledge of the interrelationships among the chemical fields simulated by the model, and observations of some of the state variables. An optimal state estimate is obtained by minimizing an objective function to provide the best fit to all observational data (Carmichael et al. [3]). Three dimensional chemical transport models whose adjoints have been implemented using KPP include STEM (Carmichael et al. [3]), CMAQ (Hakami et al. [17]), BASCOE (Errera et al. [10]; Errera et al. [11]), and an earlier version of GEOS-Chem (Henze et al. [18]; Henze et al. [19]).

2.2 OpenMP Parallelization

The GEOS-Chem/KPP parallel code is based on OpenMP and uses THREADPRIVATE variables to allow multiple threads to concurrently execute KPP chemistry for different grid cells.

The main chemistry loop found within the gckpp_driver subroutine in chemistry_mod.f iterates over each grid cell and performs the chemical integration. This loop is given an OpenMP parallel do loop in order to allow each iteration of this loop to be computed in parallel. This requires a number of KPP variables to be declared as THREADPRIVATE. This includes arrays containing the chemical concentrations, rate constants, fixed species, and variables for time and loop iteration. The temporary variable A used within the function evaluation also must be made THREADPRIVATE.

Similar parallelizations were developed for the adjoint integration subroutines, which required the previous changes in addition to making the checkpointing pointers within the integrator THREADPRIVATE.

2.3 Converting GEOS-Chem Inputs to KPP Inputs

KPP requires a description of the chemical mechanism in terms of chemical species (specified in the *.spc file), chemical equations (specified in the *.eqn file), and mechanism definition (specified in the *.def file). The syntax of these files is specific, but simple and intuitive for the user. Based on this input KPP generates all the Fortran90 (or C/Matlab) files required to carry out the numerical integration of the mechanism with the numerical solver of choice.

GEOS-Chem SMVGEARII uses the mechanism information specified in the "globchem.dat" file; this file contains a chemical species list and a chemical reactions list. The perl parser geos2kpp_parser.pl translates the chemical mechanism information from "globchem.dat" into the KPP syntax and outputs it in the files "globchem.def", "globchem.eqn", and "globchem.spc".

The globchem.dat chemical species list describes each species by their name, state, default background concentration, and additional characteristics. For example, consider the description of the following two species in the SMVGEARII syntax

globchem.dat:
A A3O2 1.00 1.000E-20 1.000E-20 1.000E-20 1.000E-20
I ACET 1.00 1.000E-20 1.000E-20 1.000E-20 1.000E-20

This information is parsed by geos2kpp_parser.pl and translated automatically into the KPP syntax. Specifically, the species are defined in the "globchem.spc" file and are declared as variable (active) or fixed (inactive). Variable refers to medium or short lived species whose concentrations vary in time and fixed refers to species whose concentrations do not vary too much. Species are set equal to the predefined atom IGNORE to avoid doing a species mass-balance checking for the GEOS-Chem implementation of KPP. The default initial concentrations are written in the file "globchem.def" as shown below.

globchem.spc:

```
#DEFVAR
   A302 = IGNORE;
#DEFFIX
   ACET = IGNORE;
```

globchem.def:

```
A3O2 = 1.000E-20;
ACET = 1.000E-20;
```

The globchem.dat chemical reactions list describes each reaction by their reactants, products, rate coefficients, and additional characteristics. This information is processed by geos2kpp_parser.pl to create the "globchem.eqn" file, which contains the chemical equation information in KPP format. KPP uses GEOS-Chem's calculated rate constants instead of using the kinetic parameters listed next to each chemical equation. A comparison of "globchem.dat" and "globchem.eqn" is shown below.

globchem.dat:

```
A 21 3.00E-12 0.0E+00 -1500 0 0.00 0 0
O3 + NO
=1.000NO2 + 1.000O2
```

globchem.eqn:

```
{1} O3 + NO = NO2 + O2 : ARR(3.00E-12, 0.0E00, -1500.0);
```

The perl parser geos2kpp_parser.pl makes the translation of the chemical mechanism information a completely automatic process. The user has to invoke the parser with the "globchem.dat" input file and obtains the KPP input files. The use of the perl parser is

illustrated below:

[user@local] $ perl -w geos2kpp_parser.pl globchem.dat
Parsing globchem.dat
Creating globchem.def, globchem.eqn, and globchem.spc

2.4 Updating KPP to Interface with GEOS-Chem

Once the input files are ready, KPP is called to generate all Fortran90 subroutines needed to describe and numerically integrate the chemical mechanism. We have slightly modified the KPP code generation engine to assist the integration of the KPP subroutines within GEOS-Chem. A new KPP input command (#GEOSCHEM) instructs to produce code that can interface with GEOS-Chem. This command is added to the KPP input file *.kpp, along with specifying details such as the name of the model created in the first step and integrator to use. A sample *.kpp file is shown below.

gckpp.kpp:

```
#MODEL globchem
#INTEGRATOR rosenbrock
#LANGUAGE Fortran90
#DRIVER none
#GEOSCHEM on
```

KPP is invoked with this input file

[user@local] $ kpp gckpp.kpp

and generates complete code to perform forward and adjoint chemistry calculations. The model files are named *gckpp_** or *gckpp_adj_**, respectively. These files are then copied into the GEOS-Chem code directory.

2.5 Updating GEOS-Chem to Interface with KPP

The last step involves running the gckpp_parser.pl to modify the GEOS-Chem source code to interface with KPP. This parser is run once in the GEOS-Chem code directory, modifies existing files to use KPP instead of SMVGEARII for the chemistry step, and adds new files with subroutines for adjoint calculations.

The GEOS-Chem code modifications are necessary for a correct data transfer to and from KPP. The chemical species concentrations are mapped between GEOS-Chem and KPP at the beginning of each timestep. This is done through copying data from the three-dimensional GEOS-Chem data structures to the KPP data structures for each grid cell. At the end of the time step the time-evolved chemical concentrations are copied from KPP back into GEOS-Chem. Since KPP performs a reordering of the chemical species to enhance sparsity gains, the species indices are different in KPP than in GEOS-Chem. The KPP-generated "kpp_Util.f90" file provides the subroutines Shuffle_user2kpp and Shuffle_kpp2user which maps a user provided array to a KPP array and viceversa. These subroutines are used to initialize the VAR and FIX species of KPP with the CSPEC array values from SMVGEARII (for each grid cell JLOOP).

GEOS-Chem calculates the reaction rates for each equation in each grid cell at the beginning of each chemistry step. These rate coefficients are mapped from the SMVGEARII reaction ordering to the KPP ordering via a reshuffling, and are saved. Prior to each call to the KPP integrator, the reaction rates for the particular grid cell are copied from the storage matrix into the local KPP integration data structures.

The use of the gckpp_parser.pl perl parser is illustrated below. The parser outputs a message

for each modified or created file.

[user@local] $ perl -w gckpp_parser.pl
Modifying and creating GEOS-Chem files
Modifying gckpp_Integrator.f90
...
Modifying chemistry_mod.f
Creating Makefile.ifort.4d
...
Creating checkpoint_mod.f
Done modifying GEOS-Chem files

The resulting GEOS-Chem code now uses KPP for gas-phase chemistry. Makefiles are included for the finite-difference driver, sensitivity driver, and 4D-Var driver.

2.6 Results and Analysis of GEOS-Chem KPP

We evaluate GEOS-Chem using global chemistry-only simulations. GEOS-Chem models a wide variety of chemical regimes including urban and rural, tropospheric and stratospheric, and day and night during each simulation. We use 106 chemical species (87 variables species and 19 fixed species), 311 reactions, and 43858 grid cells. This requires about 35MB of data to hold species concentrations and 104MB of data to hold the reaction rates. Experiments are performed on an 8-core Intel Xeon E5320 @ 1.86GHz. Additional details on the test machine are included in the appendix. This is a typical configuration for a small shared memory machine.

In order to illustrate the correctness of GEOS-Chem simulations using KPP for gas-phase chemistry we compare the results against the original simulation with SMVGEARII. GEOS-

Chem is run for 48 hours from 0 GMT on 7/1/2001 to 23 GMT on 7/2/2001. One simulation uses SMVGEARII chemistry and the second uses the KPP chemical code; the simulations are identical otherwise. Concentrations are checkpointed every hour and then compared. Plots of O_x concentrations (O3 + O1D + O3P) at the end of the 48 hours simulation are presented in Figure 2.2(a). Both simulations produce visually identical results. A difference plot is presented in Figure 2.2(b) which shows less than 1% error between the SMVGEARII computed concentrations and KPP computed concentrations.

A more stringent validation is provided by the scatter plot of SMVGEARII vs KPP/Rodas3 results. The simulation interval is one week, between 0 GMT on 2001/07/01 and 23 GMT on 2001/07/07, using an absolute tolerance of ATOL=10^{-1} and a relative tolerance of RTOL=10^{-3}. Both simulations are started with the same initial conditions and they differ only in the chemistry module. The results are shown in Figure 2.3. Each point in the scatter plot corresponds to the final O_x concentration in a different grid cell. The O_x concentrations obtained with KPP and with SMVGEARII are very similar to each other.

We compare the computed concentrations of the SMVGEARII and KPP solvers against a reference solution for each solver. Solutions are calculated using relative tolerances (RTOL) in the range $10^{-1} \leq \text{RTOL} \leq 10^{-3}$ and absolute tolerances ATOL = $10^4 \times$ RTOL molecules cm^{-3}. Reference solutions are calculated using RTOL=10^{-8} and ATOL=10^{-3} molecules cm^{-3}.

Following Henze et al. [18] and Sandu et al. [35, 36], we use significant digits of accuracy(SDA) to measure the numerical errors. We calculate SDA using

$$SDA = -\log_{10}\left(\max_k \left(\sum_{c_{k,j}\geq a} 1\right)^{-1} \cdot \sum_{c_{k,j}\geq a} \left|\frac{c^{\text{ref}}_{k,j} - \widehat{c}_{k,j}}{c^{\text{ref}}_{k,j}}\right|^2\right)$$

This uses a modified root mean square norm of the relative error of the solution ($\widehat{c}_{k,j}$) with respect to the reference solution ($c^{\text{ref}}_{k,j}$) for species k in grid cell j. A threshold value of a=10^6 molecules cm^{-3} avoids inclusion of errors from species concentrations with very small values.

Figure 2.4 presents a work-precision diagram where the number of accurate digits in the solution is plotted against the time needed by each solver to integrate the chemical mechanism. A seven day chemistry-only simulation from 0 GMT on 7/1/2001 to 0 GMT on 7/8/2001 is completed using the KPP Rosenbrock Rodas3, Rosenbrock Rodas4, Runge-Kutta, and Sdirk integrators and with the SMVGEARII integrator for each tolerance level. The results indicate that, for the same computational time, the KPP Rosenbrock integrators produce a more accurate solution than the SMVGEARII integrator. The Sdirk and Runge-Kutta integrators are slower at lower tolerances as shown in figure 2.4 in comparison to the Rosenbrock and SMVGEARII integrators. The Runge-Kutta and Sdirk integrators normally take longer steps, but they take fewer steps at higher tolerances. Since we normally use tolerances of 10^{-3} or lower for GEOS-Chem, we do not get these advantages. These results are similar to the results produced by Henze et al. [18] through demonstrating that the KPP Rosenbrock solvers are about twice as efficient as SMVGEARII for a moderate level of accuracy. Note that none of the KPP solvers uses vectorization. In addition to the solvers discussed above we have also tested LSODE (Radakrishan and Hindmarsh [31]), modified to use the KPP generated sparse linear algebra routines. LSODE is based on the same underlying mathematical formulas as SMVGEARII. The LSODE solution provides 3–4 accurate digits for a computational time of about 300 seconds (this would place LSODE in the upper left corner of Figure 2.4). Since LSODE does not reach the asymptotic regime for the low accuracy tolerances used here we do not report its results further.

Table 2.1 compares the reference solutions obtained with SMVGEARII, Rodas3, Rodas4, Runge-Kutta, and Sdirk integrators through showing the number of significant digits of accuracy produced when comparing two integrators. The results show that Rodas3, Rodas4, and Sdirk produce very similar references, while the SMVGEARII and Runge-Kutta solutions have between 0.5 and 4.0 digits of accuracy for each chemical species when compared to other integrators.

To illustrate the impact of the new chemistry code on parallel performance we consider a seven-day chemistry-only simulation from 0 GMT on 7/1/2001 to 0 GMT on 7/8/2001 using

Table 2.1: Direct comparison of the number of significant digits of accuracy produced by Rodas3, Rodas4, Runge-Kutta, Sdirk, and SMVGEAR reference solutions for NO_x, O_x, PAN, and CO for a one week chemistry-only simulation

Integrator	NO_x	O_x	PAN	CO
Rodas3–SMVGEARII	2.1	3.3	0.5	3.7
Rodas3–Rodas4	8.3	9.8	7.6	10.4
Rodas3–Runge-Kutta	2.1	3.3	2.7	4.0
Rodas3–Sdirk	8.5	9.5	7.8	10.2
Rodas4–SMVGEARII	2.1	3.3	0.5	3.7
Runge-Kutta–SMVGEARII	2.6	3.8	0.5	3.0
Sdirk–SMVGEARII	2.1	3.3	0.5	3.7

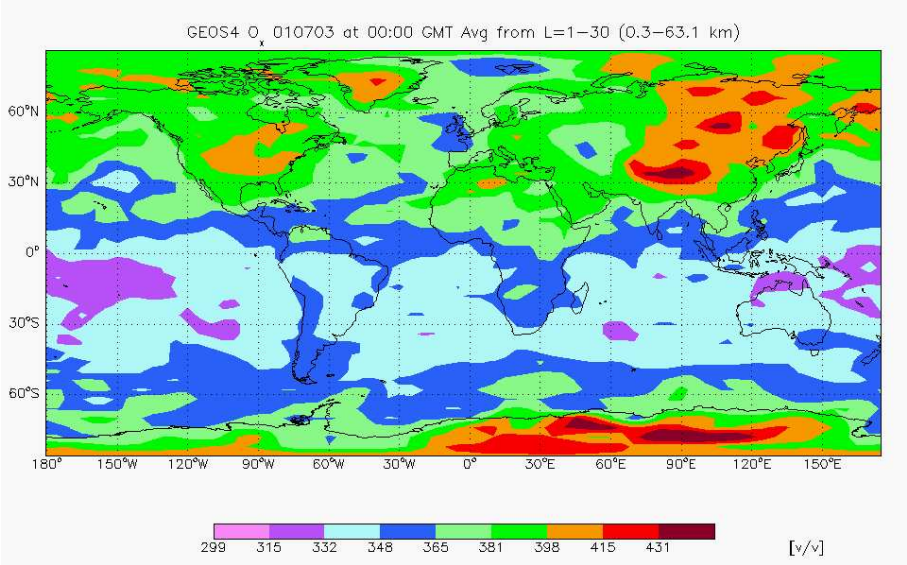
a relative tolerance of 3×10^{-2} and an absolute tolerance of 10^{-2} molecules/cm^3 and employ 1, 2, 4, and 8 processors. Figure 2.5 illustrates the parallel speedups for the KPP Rodas3 and Rodas4 integrators and for SMVGEARII; the code has close to linear speedups for all integrators.

Table 2.2 compares the average number of function calls made and average number of steps taken by each integrator per advection time step per grid cell. These results correlate well with those in Figure 2.4: the KPP integrators performing fewer function calls are faster.

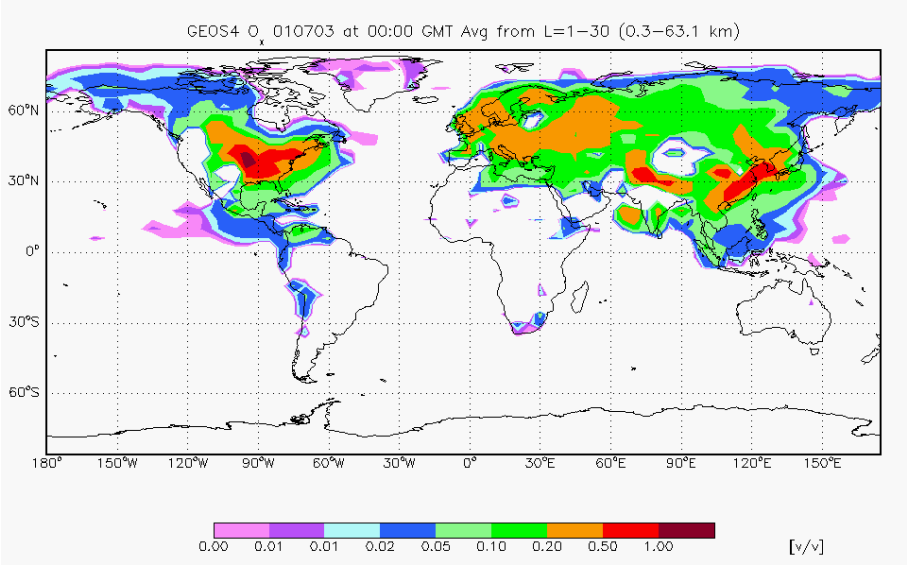
Overall, these results demonstrate that the KPP solvers and the native GEOS-Chem solver (SMVGEARII) produce accurate results as demonstrated through visual examination as well as difference plots. The KPP Rodas3 and Rodas4 solvers achieve a similar level of accuracy at a lower computational expense than the SMVGEARII solver as demonstrated through the SDA plot. Additionally the tested KPP solvers have comparable scalability for parallel processing as the SMVGEARII solver.

Table 2.2: Comparison of the average number of function calls and chemical time steps per advection time step per grid cell.

Integrator	Function Calls	Steps
Rodas3	9	3
Rodas4	18	3
Runge-Kutta	22	2
Sdirk	27	2
SMVGEARII	39	28



(a) O_x Plot



(b) Difference Plot

Figure 2.2: (a) SMVGEARII and KPP/Rodas3 computed O_x concentrations (Parts per billion volume ratio) for a 48 hours simulation (b) Difference between SMVGEARII and KPP/Rodas3 computed O_x concentrations.

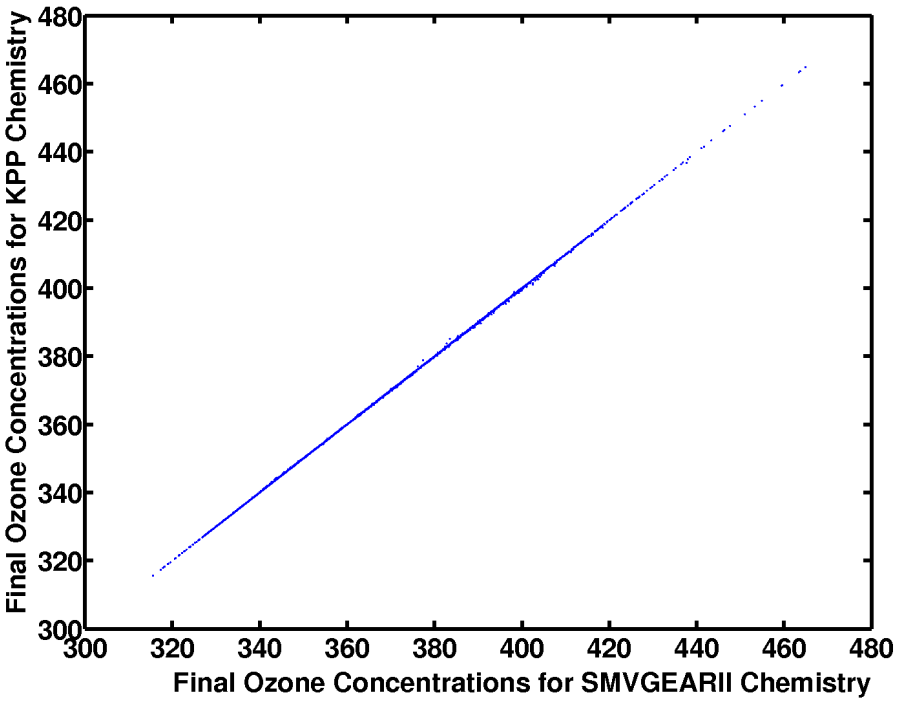


Figure 2.3: Scatterplot of O_x concentrations (molecules/cm^3) computed with SMVGEARII and KPP for a one week simulation (Mean Error = 0.05%, Median Error = 0.03%)

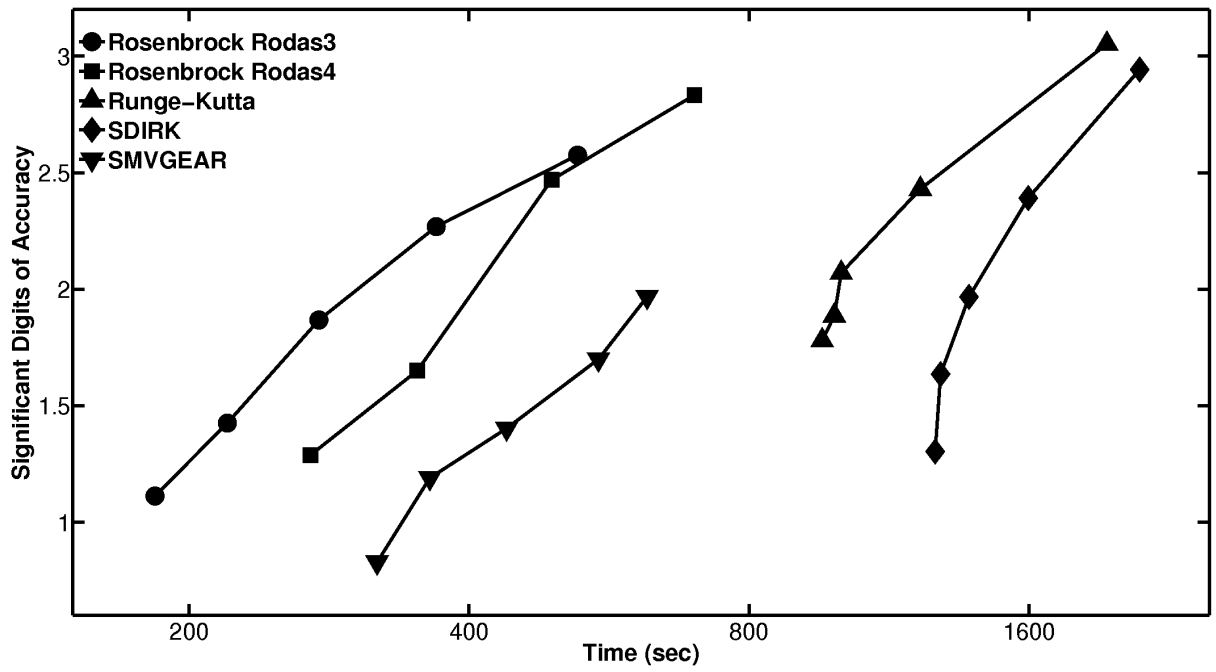


Figure 2.4: Work-precision diagram (Significant Digits of Accuracy versus run time) for a seven day chemistry-only simulation for RTOL=1.0E-1, 3.0E-2, 1.0E-2, 3.0E-3, 1.0E-3. Rodas4 does not have a point for RTOL=1.0E-1 due to not producing meaningful results.

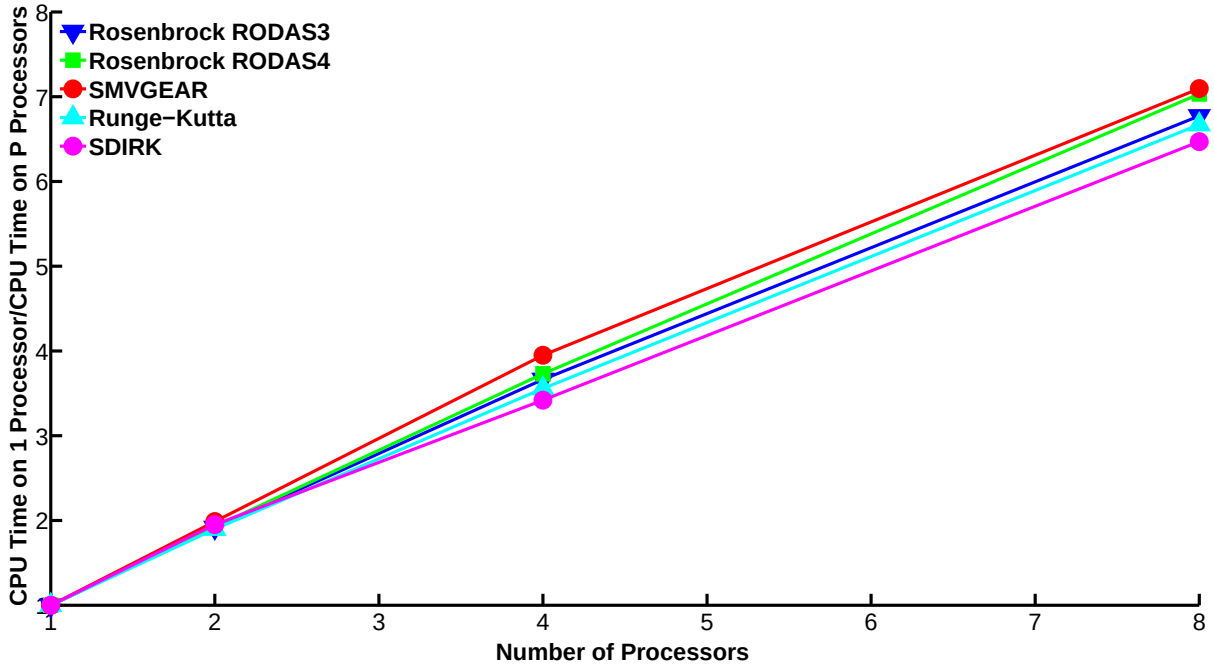


Figure 2.5: Speedup plot for Rosenbrock Rodas3, Rosenbrock Rodas4, SMVGEARII, Runge-Kutta, and Sdirk for a 7 day chemistry-only simulation

Chapter 3

GPU STEM Background

This work will investigate the benefits of running chemical transport models (CTMs) on Graphic Processing Units (GPUs) in order to assess the ability of GPUs to reduce the overall running time for CTMs.

To understand the benefits of running STEM on GPUs, we will first discuss the design and implementation of STEM, the potential benefits provided by GPUs, and the functionality provided by CUDA to simplify GPU programming. We will then look at modifications made to the test version of STEM used for this investigation.

3.1 Overview of STEM

The Sulfur Transport Eulerian Model (STEM) [2] has been developed as a simplified chemical transport model in order to experiment with new numerical methods and computing technologies prior to using these methods for larger more complicated models. STEM uses KPP to generate the chemical model for the SAPRCNOV chemical mechanism for the forward, tangent linear, and adjoint models. Transport routines are implemented in both the horizontal and vertical direction. STEM solves the mass-balance equations for concentrations of

trace species in order to determine the concentrations of chemicals in the atmosphere.

STEM contains a main loop which performs hour long simulations, loading simulation data at the beginning of the simulation, along with loading hourly data at the beginning of the main loop. Within this main loop four separate 15 minute time steps are computed. Each 15 minute time step uses an operator splitting approach, where transport steps and chemistry steps are taken one after another, allowing a modular program design to be used. The numerical solution N, where T is the directional transport operator and C is the chemistry operator, after each 15 minute interval is

$$N_{[t,t+\delta t]} = T_x^{\delta t/2} \cdot T_y^{\delta t/2} \cdot T_z^{\delta t/2} \cdot C^{\delta t} \cdot T_z^{\delta t/2} \cdot T_y^{\delta t/2} \cdot T_x^{\delta t/2} \tag{3.1}$$

This allows each transport routine and the chemistry routine to be developed and accelerated separately, simplifying the code development process while producing minimal amounts of error. This approach is frequently used throughout the atmospheric modeling community.

The test problem used during this investigation uses a 25x22x21 grid containing a total of 11550 grid cells. The layout of STEM is shown below.

Stem:

```
Load initial data

Time stepping loop (1 hour)

  Load hourly data

  Inner time stepping loop (15 minutes)

    Compute X–Transport
    Compute Y–Transport
```

```
    Compute Z–Transport

    Loop over grid cells

      Compute photolysis values
      Compute rate constants
      Compute chemistry

    End loop over grid cells

    Compute Z–Transport
    Compute Y–Transport
    Compute X–Transport

  End inner time stepping loop
End time stepping loop
```

The mathematical description of the transport and chemistry routines are discussed in chapters 4 and 5 respectively.

3.2 Overview of GPUs

Graphics processing units(GPUs) are highly parallel manycore processors [29] with a very large amount of computational horsepower along with very high memory bandwidth. GPUs devote more transistors to data processing and less to data caching in comparison to CPUs. This allows GPUs to solve data-parallel problems very quickly, as less flow control is needed due to each processor running the same program, along with hiding memory access latency with more computations.

Development of GPUs was driven by the high market demand for high-definition real time 3D graphics used especially by the video game industry. Due in part to the large audience of video gamers buying GPUs to play the latest video games, GPU prices have remained fairly low given their large computational power. Current GPUs can not only generate high quality 3D graphics, but they can also be used for computation intensive programs.

More recently scientists have recognized the capability of GPUs to run scientific computing applications quickly and cheaply. Initially scientists were forced to develop GPU programs using graphics processing languages. This required scientists to develop programs that are significantly different from their C or Fortran counterparts, resulting in large amounts of work to port the programs to graphics processing languages. This resulted in the development of programming models such as CUDA, which allows scientists to develop GPU applications using a fairly small number of extensions to the C programming language. This allows C programs to be modified very quickly to run on GPUs, with additional time being needed to optimize the code.

A number of GPU programming languages have been developed. NVIDIA has developed CUDA as a parallel programming model and software environment for developing CPU applications based on the C programming language. BrookGPU has been developed as an implementation of the Brook stream programming language for using GPUs for general purpose computing. Cg has been developed as a high level shading language for programming GPUs based on the C language. Extensions have been developed for MATLAB and Python to execute code on GPUs. This research will focus on developing code for CUDA due to the powerful parallel programming environment provided by CUDA in addition to the strong development community supporting CUDA.

3.3 Overview of CUDA

Compute Unified Device Architecture(CUDA) [4, 5, 29, 30] is a parallel programming model and software environment designed to develop scalable parallel applications on GPUs based on the C programming language. CUDA provides three key abstractions to the user to provide fine-grained data and task parallelism, along with coarse-grained data and task parallelism: a hierarchy of thread groups, shared memories, and barrier synchronization. This results in sub-problems that can be solved independently, along with pieces that can be solved cooperatively.

3.3.1 Memory Hierarchy

CUDA provides access to a number of different memory spaces. These memory spaces include local memory, shared memory, global memory, constant memory, texture memory, and registers. Each memory space provides a number of advantages and disadvantages. The size of each memory space for the GeForce GTX 260 GPU used in this investigation will also be listed.

Local memory provides each thread with 16KB of a uncached memory that is only accessible by each thread. Shared memory provides access to 16KB of fast memory that is accessible by all threads in a thread block. Global memory provides access to a large amount of uncached memory that can be accessed by all threads and can be used to transfer data between the host (CPU) and the device (GPU). Global memory is also the slowest memory, and therefore limiting the number of access to this memory is beneficial. Constant memory provides access to 64KB of read-only memory. Texture memory provides access to a large amount of cached read-only memory. Additionally the global, constant, and texture memory spaces are persistent across kernel launches by the same application. Registers provide very fast access to data, although there are only a very small number of registers available.

3.3.2 Runtime Components

CUDA provides a common runtime component containing language extensions and functions used by both host and device functions. CUDA offers a number of vector types usable by both host and device functions with up to four components. The dim3 type specifies the dimensions of grid blocks and thread blocks. Many C standard library math functions are currently supported for both single and double precision. When executed on the host, a given function uses the C runtime implementation.

CUDA provides a device runtime component containing functions used only on GPU devices. This includes a synchronization function, __syncthreads(), used to verify that all threads have reached this function before moving to the next instruction.

CUDA provides a host runtime component that can only be used by functions on the host to execute GPU functions. The host runtime component consists of two mutually exclusive APIs, a low-level driver API and a higher-level runtime API implemented on top of the CUDA driver API. The CUDA driver API provides a higher level of control through functions to configure and launch kernels. We focus on the driver API due to providing more control over a programs execution.

Initialization Functions

The CUDA driver API is initialized simply through calling the function cuInit() prior to any other driver functions.

CUDA device management provides functions to get properties of the GPU device such as cuDeviceComputeCapability() to get the device model and cuDeviceGetAttribute() to get properties of the GPU such as the maximum number of threads per block and maximum grid dimensions.

CUDA context management provides contexts similar to CPU processes. A CUDA context

encapsulates all resources and actions performed within CUDA, with each context having its own distinct 32-bit address space. A host thread has one device context current at a time. A context is created using cuCtxCreate() and is made current to the calling host thread. Each host thread has a stack of current contexts, with cuCtxCreate() pushing the newly created context onto the top of the stack.

CUDA module management provides access to GPU functions. Modules are dynamically loadable packages of device code and data output by nvcc, the CUDA compiler. The names for all symbols are maintained at module scope, allowing many modules to operate within the same CUDA context. CUDA includes functions to load a module using cuModuleLoad() and to get a handle to a function in a module using cuModuleGetFunction(). This function handle can then be used by the execution control functions to run the CUDA function on the GPU.

Execution Functions

CUDA provides execution control functions to allow programmers to execute functions on the GPU. These include functions to specify device parameters, specify function parameters, and to launch a function on the GPU.

CUDA provides the function cuFuncSetBlockShape() to set the number of threads per block for a given function and to specify how thread indices are assigned. The size of shared memory for each thread block is set using cuFuncSetSharedSize(). The cuParam*() family of functions specifies the parameters provided to the kernel, where * is replaced with i for an integer parameter, f for a floating point parameter, or v for arbitrary data. Functions are launched on the GPU using cuLaunchGrid(), which requires the dimensions of the grid as parameters.

CUDA provides function to allow programmers to allocate memory for variables and data structures, along with functions to allow the programmer to copy data from the host to the

device and vice versa. Linear device memory is allocated using the cuMemAlloc() functions. CUDA provides functions to copy data between the host and device and vice versa using cuMemcpyHtoD() and cuMemcpyDtoH(), along with other functions to copy data between devices and arrays.

CUDA provides functions to create streams and to synchronize the program through verifying that all streams have finished executing. Streams are created using cuStreamCreate(). The kernel is launched with the stream as a parameter to allow the device to execute a function while operating on a given stream. Stream calls are synchronized using cuStreamSynchronization() to make sure all streams have finished their work. Streams are destroyed using cuStreamDestroy(). Streams can be used with asynchronous functions to allow many streams execute in parallel.

CUDA provides asynchronous functions to return control of the program to the application before the device has finished its task. This allows programmers to asynchronously copy data between the host and device and asynchronously call the kernel. The functions cuMemcpyHtoDAsync() and cuMemcpyDtoHAsync() copy data asynchronously between the host and device, while cuLaunchGridAsync() asynchronously executes a function on the device.

3.3.3 Performance Guidelines

General Guidelines

There are three basic performance optimization guidelines for efficient CUDA programs running on GPUs. These include maximizing parallel execution, optimizing memory usage, and optimizing instruction usage.

Parallel execution is maximized through developing an algorithm that exposes as much data parallelism as possible. GPUs perform best when given very large amounts of data parallel arithmetically intense operations to execute, so creating algorithms that take advantage of this property results in efficient programs.

Memory usage is optimized through minimizing data transfers between the host and device due to the low bandwidth. Use of shared memory should be maximized due to its high bandwidth. Additionally it may be better to recompute some data instead of transferring it from the host to the device.

Instruction usage is optimized through reducing the number of arithmetic instructions with low throughput. Using intrinsic instead of regular functions or single-precision instead of double-precision can result in more speed when these changes do not affect the end result.

Instruction Performance

High instruction throughput can be maximized through minimizing the number of instructions with low throughput used, maximizing the use of available memory bandwidth for each memory category, and allowing the thread scheduler to overlap memory transactions with mathematical computations as much as possible. Therefore the program should have many arithmetic operations per memory operation and many active threads per processor.

Control flow instructions should be minimized, as they can cause threads of the same warp to follow different execution paths, increasing the number of operations. Controlling conditions should be designed to minimize the number of divergent warps. The compiler may also use branch prediction to prevent warps from diverging. This allows all instructions whose execution depends on the controlling condition to be executed, while other instructions are not actually executed.

Thread Block Size, Registers, and Shared Memory

The thread block size, number of registers per thread, and size of shared memory per thread block affect the GPU occupancy. Each of these settings should be chosen so that the GPU occupancy is 100% if possible. There are some cases where programs will execute faster with less than 100% occupancy due to performing better when using larger amounts of shared

memory or registers, or smaller thread block sizes. A variety of settings should be tested to find the optimal settings for each CUDA GPU program. NVIDIA provides the CUDA GPU Occupancy Calculator to assist with choosing these settings [5].

The number of threads per block should be chosen so that all computing resources are fully utilized. Therefore there should be at least as many blocks as there are processors in the device. Additionally having multiple active blocks per processor can allow one block to run while other blocks are waiting on a memory read or synchronization. The number of threads per block should also be chosen as a multiple of the warp size to prevent under-populated warps.

The number of registers per thread should be chosen so that all active threads can use the needed number of registers at the same time. If the number of registers per thread is large enough that not all threads can use the needed number of registers at the same time, then the number of active thread blocks is reduced and the GPU occupancy may also decrease.

The amount of shared memory per block should be at most half the total amount of shared memory available per processor so that more thread blocks can stream through the device. The amount of shared memory used per thread block should be small enough that all active thread blocks can use the needed amount of shared memory at once. If not all thread blocks can use the needed amount of shared memory at once, then the number of active blocks will decrease and the GPU occupancy may also decrease.

Data Transfers between Host and Device

The bandwidth between the device and the host is very low, therefore minimizing the number of data transfers between the host and device will result in faster execution. This can be completed through moving more code from the host to the device, which may be faster even if it results in lower parallelism. Additionally one big data transfer is much more efficient than many smaller data transfers.

3.4 Development of Test Version of STEM

The version of STEM I was originally given was written in Fortran 90, used MPI to run in parallel, and contained a large number of unused files. In order to develop a GPU version of STEM I needed to develop C versions of the code that would be executed on the GPU, remove common blocks, remove MPI calls, and remove unneeded files. This would result in a single processor forward test version of STEM that could be modified one step at a time to run on GPUs.

I began by developing a simplified version of the Fortran 90 version. Any files that were not used by the forward Rosenbrock integrator and explicit transport method were removed from STEM. Next the MPI calls were removed from all files along with all common blocks. These common blocks were replaced with Fortran 90 module files.

To further decrease the complexity of STEM, a checkpointing system was used to load data from ASCII files at the beginning of each simulation hour. This required the original STEM code to be modified to write data to a file at the beginning of each hour and the test version modified to read this data at the beginning of each hour. This allowed all of the previous I/O code to also be removed from STEM.

Currently CUDA only works for C, requiring the STEM chemistry and transport code to be rewritten in C to run on the GPU. For chemistry, this simply involved using KPP with the original STEM input file to generate a C model instead of a Fortran 90 model. For transport, this involved rewriting the Fortran 90 code in C. The Fortan 90 transport used a number of lapack routines which are not optimized for CUDA. To allow these routines to be optimized by hand for CUDA, C versions of these routines were found and copied to a new file containing linear algebra subroutines, with unused code within each subroutine being removed.

3.5 Literature Review

Using GPUs for scientific computing is a fairly new field, resulting in relatively little prior research related to running chemical transport models on GPUs. Most papers written up to this point focus on test problems containing small kernels with a small amount of data. These problems are able to achieve very large speedups due to having optimal conditions which are not found in real world applications. The data per thread is much larger for real world problems than many of these smaller test problems, resulting in many more challenges to achieve good performance. For example, an idealized transport simulation can use constant values for fluid velocity, while in real simulations different values for the wind field vector are used in each grid cell. This data is obtained through experimental measurements and needs to be moved to the GPU prior to the computations for a grid cell. This research provides a much more complete study of the challenges faced when developing large scale end-to-end applications to solve real world problems.

Previous work developing the KPP Rosenbrock chemical integrator for GPUs using CUDA has been completed by Michalakes et al. [27, 28], demonstrating approximately a two times speedup for KPP Rosenbrock chemistry on the GPU. Linford et al. [25] ran chemical transport models on the Cell Broadband Engine, achieving a superlinear speedup. Further research [26] showed that the Cell Broadband Engine performed similar to two nodes of the IBM BlueGene/P and eight Intel Xeon cores in a single chip.

There are a number of studies that have experimented with using CUDA for GPUs for large scale applications. Goddeke et al. [13, 14, 15], Elsen et al. [9] and Thibault et al. [38] produce effective speedups ranging from 10x to 20x for finite element Navier-Stokes solvers for more complex problems. Brandvik et al. [1] and Hagen et al. [16] also show 10x to 20x speedups for 3D Euler solvers. Stantchev et al. [37] achieved up to a 14x speedup for plasma turbulence modeling using the Hasegawa-Mima equation solver.

This research provides a much more complete study by optimizing a large scale end-to-end

application to run on a GPU using CUDA than these previous works. Many of the listed studies use single precision, use more ideal problems for their tests, or focus on a single problem instead of a full application, resulting in speedups beyond those produced for more complicated applications such as STEM.

Chapter 4

GPU Transport

This chapter will discuss the development of the STEM transport routines. This includes both implicit and explicit transport routines, discussion of how to run these methods on the GPU, and discussion on how these routines can be optimized for the GPU.

4.1 Overview of Transport Methods

STEM transport uses two horizontal transport routines along with one vertical transport routine. These three routines are called once prior to the chemistry routine for half of a time step, and again after the chemistry routine in reverse order for half of a time step.

The STEM transport solves the advection diffusion equation

$$y' + \nabla(uy) = \nabla(k\nabla y) + b \tag{4.1}$$

where $\nabla(uy)$ is the advection term, $\nabla(k\nabla y)$ is the diffusion term, and b is the boundary values.

This is written as the linear system

$$y' = A \cdot y + b(t). \tag{4.2}$$

4.1.1 Implicit Method

The original STEM transport method is the implicit Crank-Nicholson method

$$y^{n+1} = y^n + dt\left(\frac{A \cdot y^n + A \cdot y^{n+1}}{2}\right) + dt\left(\frac{b(t^n) + b(t^{n+1})}{2}\right), \tag{4.3}$$

where A is the Jacobian, y^n is the solution at step n, $b(t^n)$ is the free term at step n, and dt is the time step.

This can be simplified to

$$\left(I - \frac{dt}{2}A\right) y^{n+1} = \left(I + \frac{dt}{2}A\right) y^n + dt\left(\frac{b(t^n) + b(t^{n+1})}{2}\right). \tag{4.4}$$

This transport routine has already been implemented in STEM. Implementing this implicit method requires a number of linear algebra routines such as LU factorizations to run quickly. These methods are not easily parallelizable, resulting in some difficulties when attempting to optimize the transport code for GPUs, and limiting the potential speedup. Additionally there is more code prior to the for loop over species concentrations which may be more difficult to efficiently parallelize for large numbers of threads.

Implicit Transport Subroutine:

1. Compute Jacobian

2. Compute A = I - $\frac{dt}{2}$ Jac

3. Compute LU Factorization of A

4. Loop over chemical species

4.1 Compute Free Term B

4.2 Compute y = y^n + DT · B

4.3 Compute y = y + $\frac{DT}{2}$ · Jac · y

4.4 Solve $A \cdot y^{n+1}$ = y

5. End loop

4.1.2 Explicit Method

In order to test a method more suited for parallelization, we implement the Rk2a explicit transport method in addition to the implicit Crank-Nicholson method. The Rk2a method is as follows:

$$y_{(1)} = y^n + dt \cdot A \cdot y^n + dt \cdot b(t^n) \tag{4.5}$$

$$y_{(2)} = y_{(1)} + dt \cdot A \cdot y_{(1)} + dt \cdot b(t^{n+1}) \tag{4.6}$$

$$y^{n+1} = \frac{y^n + y_{(2)}}{2}, \tag{4.7}$$

where A is the Jacobian, y^n is the solution at step n, $b(t^n)$ is the free term at step n, and dt is the time step.

This method uses only highly parallel linear algebra routines, potentially providing better speedups than the implicit method.

4.1.3 Implementation of Explicit Method

The explicit transport method is implemented based on the code used for the implicit method. We use the same subroutine call, Jacobian evaluation, free term evaluation, and linear algebra subroutines. A Jacobian evaluation is followed by a loop over each chemical

species concentration. This loop copies the concentration values into Y and C1. The free term is then calculated. Next $y_{(1)}$ is calculated, using the dgbmv linear algebra routine to calculate $y_{(1)} = y + dt * Jac * y$, and then adding dt times the free term. Then $y_{(2)}$ is calculated, once again using dgbmv to calculate $y_{(2)} = y_{(1)} + dt * Jac * y_{(1)}$, and then adding dt times the free term. Then y^n and $y_{(2)}$ are added and divided by two in order to calculate y^{n+1}.

Explicit Transport Subroutine:

1. Compute Jacobian

2. Loop over chemical species

   2.1 Compute Free Term B

   2.2 Compute $y_1 = y^n$ + DT * Jac * y^n

   2.3 Compute $y_1 = y_1$ + DT * B

   2.4 Compute $y_2 = y_1$ + DT * Jac * y_1

   2.5 Compute $y_2 = y_2$ + DT * B

   2.6 Compute $y^{n+1} = \frac{y^n + y_2}{2}$

3. End loop

This subroutine contains little code outside of the main loop, and only calls a single linear algebra routine which can be parallelized well. Additionally this is a faster code due to the smaller number of calculations, while having a similar amount of accuracy. This subroutine can be more fully parallelized for GPUs, potentially providing better running times and speedups.

However, using only a single step to compute Z-Transport produces very inaccurate results, causing STEM to crash in a few iterations. Using more steps allows us to produce accurate results, however at least 35 steps are needed. This results in over a thirty times increase in

running time, making explicit Z-transport very inefficient. Therefore tests for the explicit X and Y transport will use the implicit Z-transport.

4.2 Development of GPU Transport

This section discusses the development of GPU STEM transport. Additional functionality is added one step at a time, starting with running a single grid cell on the GPU, then running multiple grid cells on the GPU, using the driver API instead of the runtime API, and finally using optimizations to make the code run as quickly as possible.

4.2.1 Single Grid Cell Transport

The STEM transport code was first modified to run a single grid cell at a time on the GPU using the runtime API. This required the development of a transport driver routine for each of the three transport routines. This driver used a global function type qualifier and allocated memory on the device, copied data from the host to the device, launched the kernel, and then copied the concentration data back to the host from the device. Device function type qualifiers were added to any subroutines called by driver subroutines. The main STEM driver was modified to call the global GPU functions.

Transport Driver Functions:

```
extern "C" void tranx_mf_(...)
extern "C" void trany_mf_(...)
extern "C" void tranz_mf_(...)
```

Global GPU Functions:

```
extern "C" __global__ void Advdiff_Fdhx_Mf(...)
extern "C" __global__ void Advdiff_Fdhy_Mf(...)
```

```
extern "C" __global__ void Advdiff_Fdz_Mf(...)
```

Driver Function Layout:

1. Allocate GPU Memory

2. Copy data from the host to the device

3. Launch GPU kernel

4. Copy concentration data from the device to the host

5. Free global GPU arrays

Overall the running time for the single grid cell version was very slow as expected. GPUs excel when allowing many threads to work together at once to run a program, therefore the next step is to modify the code to run multiple grid cells on the GPU at once.

4.2.2 Multiple Grid Cell Transport

The first step needed to allow multiple grid cells to run at the same time is to send all of the data from the host to the device at once. STEM stores the transport data in a number of three and four dimensional arrays. CUDA does not currently provide functionality for passing more than three dimensional arrays to the GPU. Therefore each array was copied into a one dimensional array, with the data for each grid cell being listed one after another within each array. This allows a single function to be called to pass all of the data for each array from the host to the device.

```
nsize = number of grid cells to loop over
for(n=0; n<nsize; n++) {
  Copy data from multi-dimension STEM arrays into 1-D arrays
}
cuLaunchGrid(...);
```

```
for(n=0; n<nsize; n++) {
  Copy data from 1-D concentration array into
  4-D STEM concentration array
}
```

Next the device functions needed to be modified to access the correct elements of each large array. The thread index for each grid cell is calculated using the CUDA block dimension and grid dimension variables. This allows each thread to access the data for a different grid cell. An if statement verifies that the thread index is not larger than the number of grid cells. This prevents threads from attempting to access non-existent data, since the number of total threads may be slightly larger than the number of grid cells in order to make the thread block size a multiple of the warp size.

```
tid = (blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x
  * blockDim.y + (threadIdx.y * blockDim.x) + threadIdx.x;
```

The last step is to modify the execution configuration to use multiple threads and multiple thread blocks. A number of different configurations were tested, with the fastest multiple grid cell transport times being similar to the CPU times. The next step is to develop a driver API version, which will allow data to be left on the GPU through multiple global GPU function calls, reducing the number of data transfers between the CPU and GPU and therefore reducing the running time.

4.2.3 Driver API Transport

Modifying the code to use the driver API instead of the runtime API requires many changes to each driver function, providing developers with more control of the programs execution. Many requirements that are implicitly handled by the runtime API need to be specified by the driver API. These requirements include initialization, context management, and module management. Code was added to the beginning of each global subroutine to initialize CUDA,

create a context, and load the needed module.

Initialize Driver API Transport:

1. Initialize CUDA

2. Get device handle

3. Create CUDA context for the device

4. Load module with GPU functions for the current context

Many CUDA functions needed to be changed to use the driver API instead of the runtime API, requiring minimal changes to the code. The execution configuration is only used by the runtime API, being replaced by CUDA module management functions to specify the GPU function to call, memory management function to specify parameters, and execution management functions to launch the kernel. First a handle to the GPU function is obtained and the block shape is set. Each parameter is then specified by listing the function handle, the location of the parameter in the parameter list, and the size of the parameter. Each parameter is must be aligned along 8 byte boundaries using __alignof(...).

Execute GPU kernel function:

1. Get function handle

2. Set block shape

3. Calculate alignment of each parameter

4. Set each parameter

5. Set size of all parameters

6. Launch GPU kernel

Since each subroutine is called once prior to the chemistry call, and in reverse order again after the chemistry call, initialization and closing routines along with four shuffling routines are added to the code as shown below. This allows us to develop a single version of each

transport subroutine and call each subroutine without copying the same data from the host to the device more than once. The initialization routines copy data used by all three subroutines from the host to the device, along with a flag to specify whether X or Z transport occurs first. The closing subroutine copies the main concentration array from the device to the host and frees the allocated device memory. Since each subroutine requires the data to be organized differently, data must be shuffled between each subroutine call. These subroutines shuffle data on the GPU including the primary concentration array and additional arrays used in two successive transport routines. The subroutine rxn calculates the rate constants and launches the GPU chemical integration kernel.

```
call init_transport(...)
call tranx_mf(...)
call shufflextoydrv()
call trany_mf(...)
call shuffleytozdrv()
call tranz_mf(...)
call close_transport(...)

call rxn(...)

call init_transport(...)
call tranz_mf(...)
call shuffleztoydrv()
call trany_mf(...)
call shuffleytoxdrv()
call tranx_mf(...)
call close_transport(...)
```

Modifying the code to use the driver API results in some slight decreases in running time

due to having less memory transfers between the host and the device. At this point the challenge is to further optimize the code through taking advantage of the different types of memory and allowing multiple threads to work together.

4.3 Transport Optimizations

This section discusses the different types of optimizations used to speed up the transport subroutines. This includes general optimizations such as storing data in different types of memory, along with a focus on using shared memory optimizations to allow multiple threads to work together.

STEM uses a grid of size 25 x 22 x 21, with each transport subroutines having at most 550 independent rows or columns of grid cells. This is a relatively small number of independent tasks for GPUs which perform best with thousands of independent tasks. Therefore one of the key optimizations for this code is using multiple threads to compute each grid cell, allowing us to use thousands of threads to compute each transport method.

Independent rows/columns:
ix = 25 rows in X horizontal direction
iy = 22 rows in Y horizontal direction
iz = 21 columns in Z vertical direction

X-transport: iy $\cdot$ iz = 462 independent rows
Y-transport: ix $\cdot$ iz = 525 independent rows
Z-transport: ix $\cdot$ iy = 550 independent columns

The implicit and explicit transport subroutines are very similar, with the key difference being that the explicit code does not compute a LU decomposition, and therefore can easily use many threads to execute all parts of the code. Similar optimizations are used by both the implicit and explicit transport subroutines. Detailed results demonstrating the performance

of each transport method are included in the results section.

4.3.1 General Optimizations

The simplest code modifications involve using fast math functions provided by CUDA. These less accurate functions run in less cycles, providing benefits when the code does not need the additional accuracy. This includes multiplying two small integers together, such as when calculating the thread index. These optimizations provide a very slight speedup.

```
tid = __mul24(__mul24((__mul24(by,gridDim.x) + bx),
  blockDim.x), blockDim.y) + __mul24(ty, blockDim.x) + tx;
```

GPUs provide access to a number of different types of memory. These memories include local, constant, global, and shared memory. Local memory provides the fastest access, but this memory is also the smallest. Constant memory provides faster access to data used by all threads that remains constant throughout the execution of the program. Global memory provides slow access to a large amount of memory which can be used to transfer data between the CPU and GPU. Shared memory provides very fast access to a small amount of memory, but requires many changes to allow threads in each thread block to work together to efficiently use this memory.

The transport subroutines use global memory to copy chemical concentrations from the host to the device. The concentration values are copied from global memory to an array in local memory prior to using these values in any computations. This array is then copied from local memory back to global memory after completing all computations. The remaining arrays passed from the host to the device are not modified by any subroutines, and are accessed only once or twice to calculate values stored in local memory or shared memory. Tests were performed placing these arrays in texture memory, however this did not provide any noticeable speedup. The arrays for the Jacobian, A, pivot, the product of the air and diffusion arrays, and dZ are placed in shared memory. The tested shared memory optimizations are

discussed in more detail in the next section.

Memory Layout for Implicit Transport Routines:

Variables:

```
Nspec = 66 \\ Number of chemical species
N = ix or iy or iz \\ Number of grid cells in row/column
```

Global arrays:

```
double Wind[ix*iy*iz];
double Dif[ix*iy*iz];
double Air[ix*iy*iz];
double Conc[ix*iy*iz*Nspec];
double Bdry[2*((ix*iy*iz)/N)*Nspec];
```

Implicit Local arrays:

```
double C1[N];
double C2[N];
```

Explicit Local arrays:

```
double C1[N];
double B[N];
double Y1[N];
double Y2[N];
```

Implicit Shared arrays:

```
__shared__ double Jac[5*N];
__shared__ double A[7*N];
__shared__ int ipiv[N];
__shared__ double AK[N];
```

Explicit Shared arrays:

```
__shared__ double Jac[5*N];
__shared__ double AK[N];
```

Z-Transport Only:

```
__shared__ double DZ[N];
```

Table 4.1: Comparison of the memory requirements per grid cell.

Transport Method	Global Memory	Local Memory	Shared Memory
Implicit X-Transport	14856	400	2700
Implicit Y-Transport	13200	352	2364
Implicit Z-Transport	12648	336	2420
Explicit X-Transport	14856	800	1200
Explicit Y-Transport	13200	704	1056

Table 4.1 shows the memory requirements per grid cell for X, Y, and Z implicit and explicit transport. This demonstrates that a significant amount of global memory is needed to compute each grid cell. Additionally there is a fairly small amount of shared and local memory needed. Explicit transport requires twice as much local memory as implicit transport, but also requires half as much shared memory.

4.3.2 Shared Memory Optimizations

Shared memory allows all threads in a thread block to access and modify the same data. This allows blocks of threads to operate on a smaller amount of data, resulting in more data fitting into faster memories and higher GPU occupancies. There are a number of minor drawbacks to shared memory. Threads must be synchronized to guarantee that writes are visible to all threads. Using too much shared memory also can reduce the number of thread blocks that can be fit into memory at once, limiting the GPU occupancy.

First we need to set variables specifying which data each thread needs to operate on. As shown below, tid contains the thread index, ranging from 0 to the maximum number of threads. The load index, ltid, specifies which data should be loaded from global memory. This is needed when more than 1 thread computes the data for a single grid cell. This value ranges from 0 to the maximum number of grid cells. The loop starting index, tstart,

specifies which index of a parallel loop each thread should compute. The shared memory index specifies which index in a shared memory array each thread should use. These variables allow each thread to load the correct data and to perform computations in parallel.

Number of threads per grid cell:

```
#define nt 66;
```

Number of grid cells computed per thread block:

```
#define sc 4
```

Number of Chemical Species:

```
Nspec = 66;
```

Thread Index:

```
tid = (blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x
  * blockDim.y + (threadIdx.y * blockDim.x) + threadIdx.x;
```

Load Index:

```
ltid = tid / nt;
```

Loop Starting Index:

```
tstart = tid % nt;
```

Shared Memory Index:

```
si = ((tid-(tid%nt))/nt) % sc;
```

Each transport subroutine performs a number of different computations and then loops over each of the different concentrations independently. This loop can be calculated in parallel, allowing us to use one thread per chemical species. This allows the inner loop to be calculated much faster, but requires the use of shared memory to allow the computations prior to this

loop to be parallelized. Since Nspec threads are now used to calculate one row/column of data, each of these Nspec threads must load the same data from global memory using ltid.

Original Inner Transport Loop:

```
for(i=0; i<Nspec; i++)
  Calculate Concentration i
```

Parallel Inner Transport Loop:

```
i = tstart;
Calculate Concentration i
```

Prior to this loop, the Jacobian, the A array, and the LU decomposition are computed for implicit transport, while only the Jacobian is computed for explicit transport. The next step is to compute these values in parallel. The Jacobian contains loops to zero the Jacobian array, calculate the product of the air and diffusion arrays, calculate the advection discretization, and calculate the diffusion. Each of these loops can be easily parallelized. The value of A is calculated using two loops, each of which can also be easily parallelized. The LU decomposition is then performed, which does not parallelize easily.

The first two Jacobian loops are parallelized by allowing each thread to operate on a different element of the array using tstart. The remaining two loops iterate over each grid cell in the row or column being operated on, performing independent calculations. These loops can be parallelized by allowing each thread to operate on a different iteration of the loop using tstart. The first and last grid cells require modified calculations due to being on boundaries. If statements are used to allow threads to operate on the correct values.

Calculate Air * K:

```
if(tstart<N)
  AK[tstart+ind] = Air[tstart]*K[tstart];
```

Initialize Jacobian:

```
for(j=tstart; j<5*ix; j=j+nt)
  Jac[j] = 0.0;
```

Advection Discretization:

```
if(tstart == 0) {
  Compute row 0
} else if(tstart>0 && tstart<N-1) {
  Compute row tstart
} else if(i==N-1) {
  Compute row N-1
}
```

Diffusion:

```
if(tstart == 0) {
  Compute row 0
} else if(tstart>0 && tstart<N-1) {
  Compute row tstart
} else if(i==N-1) {
  Compute row N-1
}
```

The A array is calculated using two separate loops which are parallelized similar to the Jacobian code. The LU decomposition provides little opportunity for parallelization. There are only a few small sections of code that can be parallelized, with the remaining code needing to be recomputed by each thread. This code runs slightly faster when only one thread computes the LU decomposition and the remaining threads wait for the computation to finish.

Calculate A:

```
for(n=tstart; n<(lda-kl)*N; n=n+nt)
  Calculate A
```

```
if(tstart<N)
  Add 1 to diagonal of A
```

LU Decomposition:

```
if(tstart == 0)
  dgbtf2(...);
```

These optimizations allow Nspec threads to work in parallel to complete each grid cell. For STEM, Nspec is only 66, which is a fairly small thread block. Since less than a fourth of the shared memory is currently in use, the code can be modified to compute up to four grid cells per thread block. Each shared memory array is increased in size by a multiple of 66, and the variable si specifies which section of the shared memory arrays each thread should operate on. This allows up to 264 threads per thread block to be used.

An additional optimization was attempted by allowing multiple threads to work on each iteration of the transport loop over chemical concentrations. This required the calculation of two index variables to determine which iteration of the inner loops to compute for each thread and which elements of the shared memory arrays to operate on. This also required optimization of the subroutine to calculate the free term and two linear algebra routines to run in parallel. The linear algebra routine to solve the linear system provides little opportunity for optimization and is computed faster when one thread computes the subroutine while the others wait. Additionally using extra threads to calculate each iteration of the for loop results in significantly more threads than needed to execute the code prior to the for loop, resulting in many threads being idle. This optimization results in a slightly slower running time for both the implicit and explicit transport methods, and therefore is not included in the final code. The code for explicit transport parallelizes slightly better due to all parts of the inner loop executing in parallel, however this code is still slower than using a single thread within each iteration of the loop. This is likely due to many threads being idle prior to the loop over chemical species, along with some threads being idle when computing parts

of the inner loops.

Multiple Threads within Implicit Transport Loop:

1. Compute Jacobian in parallel

2. Compute A in parallel

3. Compute LU Factorization of A with one thread

4. Calculate thread indices for each thread

5. Loop over chemical species

   5.1 Compute Free Term B in parallel

   5.2 Compute y = y^n + DT $\cdot$ B in parallel

   5.3 Compute y = y + $\frac{DT}{2}$ $\cdot$ Jac $\cdot$ y in parallel

   5.4 Solve $A \cdot y^{n+1}$ = y with one thread

6. End loop

Overall these optimizations produce a significant improvement in running time. The next optimization is to attempt to further improve the time spent on memory transfers.

4.4 Memory Transfer Optimizations

Using the driver API allows arrays to be left in global memory between kernel launches. Therefore we can further optimize the code through using asynchronous execution and through parallelizing the shuffling code. Since the transport code runs in less than 50 ms even on the CPU, memory transfer time results in a significant amount of the total running time, providing opportunities to further decrease the running time.

Asynchronous execution allows a GPU kernel to be launched, while the main program on

the CPU continues executing. Therefore each memory transfer function and the launch grid function can be replaced with asynchronous functions. A stream synchronization function is used to wait for all asynchronous functions to finish running. Asynchronous execution allows more CPU code to execute in between function calls, resulting in slightly faster execution.

Asynchronous Memcpy Host to Device:

```
cuMemcpyHtoDAsync(d_C, Conc, mem_size_Conc, stream);
```

Asynchronous Launch:

```
cuLaunchGridAsync(cuFunction, nt, 1, stream);
```

Asynchronous Memcpy Device to Host:

```
cuMemcpyDtoHAsync(Conc, d_C, mem_size_Conc, stream);
```

The shuffling subroutines consist of loops containing independent iterations. These loops can be parallelized by allowing each thread to compute different elements of each loop, similar to the parallelizations used within the transport code. This allows each thread to do independent work, with few threads being idle within the subroutine. These subroutines first shuffle data from the previous subroutine order back to the original order and store this data in a temporary array, and then shuffle the data from the original order to the order for the next subroutine and store this in the correct array.

Shuffle Subroutines:

```
extern "C" __global__ void ShuffleXtoY(...)
extern "C" __global__ void ShuffleYtoZ(...)
extern "C" __global__ void ShuffleZtoY(...)
extern "C" __global__ void ShuffleYtoX(...)
```

The asynchronous execution does not result in noticeably faster running times due to the limited amount of CPU code in between the asynchronous functions. The GPU shuffling

routines take the same amount of time as the CPU data shuffling routines. However this does decrease the memory transfer times by eliminating a significant number of memory transfers from the CPU to the GPU and from the GPU to the CPU.

4.5 Summary of Transport Optimizations

This section will briefly review the different optimization attempts for STEM transport on the GPU.

There are a number of optimizations that proved to be successful and were fully implemented for STEM transport. Experimenting with the number of registers and thread block size allowed us to find the best settings for the transport methods. Many settings gave similar good performances, however some settings gave much slower performance. Fast math functions provided by CUDA allowed some computations to be performed faster without sacrificing accuracy.

A number of different memory configurations were tested. Global memory allowed data to be transferred between the CPU and GPU, however performing computations on data in global memory was very slow. Local memory performed very quickly for per thread data. Texture memory showed similar performance to global memory for storing simulation data, allowing more data to be transferred from the CPU to the GPU if needed while providing similar performance.

Shared memory allowed multiple threads to work together to compute each row/column of data. The main transport loops over each chemical concentration were parallelized, allowing the number of threads computing each row/column of data to be set equal to the number of chemical species per grid cell. The smaller loops at the beginning of these methods to calculate the Jacobian and other data were also parallelized. Due to the small number of independent rows/columns of data, computing multiple rows/columns of data per thread block allowed more GPU threads to be used at once.

Using the driver API and shuffling subroutines allowed data to be left on the GPU between transport methods, reducing the number of memory transfers between the host and device.

Chapter 5

GPU Chemistry

This chapter discusses the development of STEM chemistry routines. This includes the mathematical background for both the Rosenbrock methods and QSSA methods, discussion on how to run these methods on the GPU, and discussion on how these routines can be optimized for the GPU. Previous work developing the KPP Rosenbrock chemical integrator for GPUs has been completed by Michalakes et al. [27, 28], demonstrating approximately a two times speedup for KPP Rosenbrock chemistry on the GPU. This investigation further explores methods and optimizations to quickly and accurately run chemical integrators on GPUs.

5.1 Overview of Chemical Integrators

The section provides the mathematical background for the Rosenbrock methods and QSSA methods used in this investigation.

5.1.1 Rosenbrock

An s-stage Rosenbrock method computes the solution of the chemical system using the following formulas:

$$y^{n+1} = y^n + \sum_{i=1}^{s} m_i k_i, \tag{5.1}$$

$$Err^{n+1} = \sum_{i=1}^{s} e_i k_i, \tag{5.2}$$

$$T_i = t^n + \alpha_i h, \tag{5.3}$$

$$Y_i = y^n + \sum_{j=1}^{i-1} a_{ij} k_j, \tag{5.4}$$

$$A = [\frac{1}{h\gamma} - J^T(t^n, y^n)], \tag{5.5}$$

$$A \cdot k_i = f(T_i, Y_i) + \sum_{j=1}^{i-1} \frac{c_{ij}}{h} k_j + h\gamma_i f_t(t^n, y^n). \tag{5.6}$$

where s is the number of stages, t^n is the discrete time moment, h is the time step, y^n is the numerical solution, f() is the function, J is the Jacobian, A is the system matrix, T_i is the internal stage time moment, Y_i is the internal stage solution, and α, a, b, c, e, and m are method coefficients.

We experiment with the stiffly accurate Rodas-3 and Rodas-4 methods. Rodas-3 uses four stages and three function calls, while Rodas-4 uses six stages and five function calls. Rodas-3 is order 3, while Rodas-4 is order 4. The only difference in the Rodas-3 and Rodas-4 integration code is the method coefficients, which are computed by different subroutines on the CPU and passed to the GPU. Rodas-3 also has a slightly smaller memory footprint due to storing the data for only 4 stages instead of 6 stages as in Rodas-4.

5.1.2 QSSA

The variable-step QSSA (Quasi-Steady-State-Approximation) method and fixed-step QSSA Exp2 method were implemented. Both QSSA methods use a split function evaluation where

$$y'_j = P_j(y) - D_j(y)y_j \tag{5.7}$$

where P(y) is the production term and D(y) the destruction term.

For the variable-step QSSA method (Jay et al. [23]), the basic approximation is

$$y^{n+1} = P_j(y)/D_j(y) - (P_j(y)/D_j(y) - y^n) \cdot e^{-hD_j(y)} \cdot e^{-hD_j(y)}, \tag{5.8}$$

where y^n is the solution at step n and h is the step size.

For very small absolute values of D_j we use the approximation

$$y^{n+1} = P_j(y)h(1 - e^{-0.5hD_j(y)} \cdot (1 - (2/3)e^{-0.5hD_j(y)})). \tag{5.9}$$

For species with a very long lifetime, i.e. with a small D_j, this can be simplified to the explicit Euler formula

$$y^{n+1} = y^n + h(P_j(y) - D_j(y)y^n). \tag{5.10}$$

For species with a very short lifetime, i.e. with a large positive D_j, this can be approximated as

$$y^{n+1} = P_j(y)/D_j(y). \tag{5.11}$$

These half-step approximations are computed twice per time step, with a separate split function evaluation prior to each approximation. The split function is evaluated for y^n,

followed by the half step approximation of V2 and the full step approximation of V1. The split function is computed using V2 followed by a second half step approximation to finish calculating y^{n+1}. The full step approximation V1 is used when calculating the error term. The error term is calculated as

$$Err_n = \sqrt{\frac{\sum_{i=0}^{n} \left(\frac{(V2_i - V1_i)}{(AbsTol + RelTol * V2_i)} \right)^2}{2}}, \tag{5.12}$$

where AbsTol is the absolute tolerance and RelTol is the relative tolerance.

The step size is then calculated as

$$h = 0.9 \cdot Err_n^{-0.35} \cdot Err_{n-1}^{0.2}. \tag{5.13}$$

Next we look at the fixed-step QSSA Exp2 method (Hockbruck et al. [20]). This method provides a simpler integration subroutine through using a fixed-step, eliminating the need to calculate the error term and step size. This is a second order exponential method based on the QSSA method. The basic approximation is

$$k_i = \varphi(\gamma h A) \left(f(u_i) + hA \sum_{j=1}^{i-1} \gamma_{ij} k_j \right), \text{i} = 1,...,\text{s}, \tag{5.14}$$

$$u_i = y^n + h \sum_{j=1}^{i-1} \alpha_{ij} k_j, \tag{5.15}$$

$$y^{n+1} = y^n + h \sum_{i=1}^{s} b_i k_i, \tag{5.16}$$

where

$$\varphi(z) = \frac{e^z - 1}{z} \tag{5.17}$$

$$A = f'(y^n) \tag{5.18}$$

For very small values of γhA, the simplified approximation of φ is

$$\varphi = 1 + \frac{z}{2} + \frac{1}{6}z^2 + \frac{5}{12}z^3. \tag{5.19}$$

The method coefficients are γ_i, γ_{ij}, α_{ij}, b_i, with $\gamma_{ij} = \alpha_{ij}$ for i $\leq$ j.

These half-step approximations are computed twice per time step, with a separate split function evaluation prior to each approximation. The split function is evaluated for y^n, followed by a half step approximation of V2. The split function is evaluated again using V2, followed by a half step approximation of y^{n+1}. This integrator uses a fixed-step and therefore does not need to calculate an error term in order to set the step size, further simplifying the integrator.

5.2 Development of GPU Chemistry

This section discusses the development of GPU STEM chemistry. Additional functionality is added one step at a time, starting with running a single grid cell on the GPU, running multiple grid cells on the GPU, using the driver API, and finally using optimizations to make the code run as fast as possible.

5.2.1 Single Grid Cell Chemistry

Developing a chemistry driver using the runtime API is the first step needed to run the chemical integrator on the GPU. This driver is similar to the transport driver through allocating memory on the device, copying data from the host to the device, launching the kernel, and then copying data from the device back to the host. The main integration subroutine needs the global function type qualifier, while the remaining chemistry subroutines need the addition of device function type qualifiers to allow the code to execute on the GPU. The main STEM driver is modified to call the global GPU subroutine.

Chemistry Driver Function:

```
extern "C" void integrate_(...)
```

Global GPU Functions:

```
extern "C" __global__ void Rosenbrock(...)
extern "C" __global__ void QSSA(...)
extern "C" __global__ void QSSA_EXP2(...)
```

Chemistry Driver Layout:

1. Allocate Device Memory
2. Copy host memory to device
3. Copy host constants to device
4. Bind textures
5. Launch GPU kernel
6. Copy concentrations from device to host
7. Unbind Textures
8. Free device memory

KPP produces fast unrolled function and Jacobian code, however this code requires long compile times and significantly increased register and local memory requirements for CUDA. The unrolled code uses more registers than available on the GPU for each thread, with the extra registers overflowing into local memory. Due to the high memory requirements for the Rosenbrock integrator, this prevents the code from compiling. KPP was modified to produce a number of constant arrays containing array indices and coefficient values such as IROW_STOICM and ICOL_STOICM. These constant arrays are then used by loops within the function and Jacobian code to perform the same calculations while using a smaller amount of registers and local memory.

Unrolled Function Code:

```
\\ Computation of equation rates
A[0] = RCT[0]*V[84];
...
A[234] = RCT[234]*V[11];
```

Rolled Function Code:

```
\\ Computation of equation rates
for(i=0; i<NREACT; i++)
  A[i] = RCONST[i];
for(i=0; i<NSTOICM_LEFT; i++) {
  sidx = IROW_STOICM_LEFT[i];
  ridx = ICOL_STOICM_LEFT[i];
  if(sidx < NVAR)
    A[ridx] *= pow(Y[sidx],
      (int)STOICM_LEFT[i]);
  else
    A[ridx] *= pow(FIX[sidx-NVAR],
      (int)STOICM_LEFT[i]);
}
```

```
\\ Aggregate function
Vdot[0]=A[43]+A[218]-A[221];
...
Vdot[87]=-A[78]+...+0.6*A[210];
```

```
\\ Aggregate function
for(i=0; i<NVAR; i++)
  Ydot[i] = 0;
for(i=0; i<NSTOICM; i++) {
  sidx = IROW_STOICM[i];
  if(sidx >= NVAR) continue;
  ridx = ICOL_STOICM[i];
  Ydot[sidx]+=STOICM[i]*A[ridx];
}
```

The running time for the single grid cell chemistry was very slow, as each iteration took over

a second to complete. This is expected, as GPUs need thousands of threads to achieve high performance. The next step is to modify the code to compute multiple grid cells at once.

5.2.2 Multiple Grid Cell Chemistry

The first step towards allowing multiple grid cells to execute on the GPU at once is to copy all of the data from the CPU to the GPU at once. This requires modifying the loop within rxn_eq.f which calls the chemical integrator. This subroutine originally contained three nested loop over the entire grid which copy data from the large STEM arrays to small 1-D arrays, call the chemical integrator, and copy the final concentrations back to the STEM concentration array. This is modified to use a single loop over the entire grid. This loop first copies the data from multi-dimension STEM arrays into 1-D arrays, calls the chemical integrator, copies the concentration data from the device to the host, and then copies the data from the 1-D concentration array to the 4-D STEM concentration array. The concentrations array, rate constants, and fixed species are each copied from the host to the device, while only the concentrations array is copied from the device to the host. A single call to the main chemical integration subroutine is used since all of the data can be stored on the GPU at once. If there is not enough room to store all of the data on the GPU at once, then a smaller number of grid cells can be copied to the GPU at once, and multiple calls to the chemical integrator can be used to integrate all of the grid cells.

Original rxn_eq.f loops:

```
do i=xstart,xend
  do j=ystart,yend
  Update photolysis
    do k=zstart,zend
      Calculate rate constants
      Copy rate constants
        to 1-D array
      Copy fixed species
        to 1-D array
      Copy concentration values
        to 1-D array

      call integrate(...)

      Copy concentration values
        to 4-D STEM array
    end do
  end do
end do
```

Modified rxn_eq.f loop:

```
do n=0,NTT-1
  Calculate indicies i, j, k
  Update photolysis
  Calculate rate constants
  Copy rate constants
    to 1-D array
  Copy fixed species
    to 1-D array
  Copy concentration values
    to 1-D array
end do

call integrate(...)

do n=0,NTT-1
  Calculate indicies i, j, k
  Copy concentration values
    to 4-D STEM array
end do
```

The device functions then need to be modified to access the correct elements of each array. The thread index for each grid cell is calculated using the CUDA block dimension and grid dimension variables, allowing each thread to access the data from a different grid cell. An if statement verifies that each thread index is not larger than the number of grid cells.

```
tid = (blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x
  * blockDim.y + (threadIdx.y * blockDim.x) + threadIdx.x;
```

The last step is to modify the execution configuration to use many large thread blocks. A number of different configurations were tested, with the fastest times being noticeably slower than the original CPU times. The next step is to develop a driver API version in order to have more control over the execution of the kernels on the GPU. Additionally the runtime API and driver API cannot be used together. This provides the potential to allow data to remain on the GPU in between calls to transport code and calls to the integration code. This is not possible at the moment due to significant sections of Fortran code being run on the CPU before and after the call to the main chemical integrator that modifies the concentrations array. If these sections of code were modified to run on the GPU, then data could be left on the GPU between calls to the transport functions and the chemistry function.

5.2.3 Driver API Chemistry

Modifying the chemistry code to use the driver API instead of the runtime API requires a number of changes to the driver function, similar to the changes used to modify the transport code to use the driver API. These include adding code for initialization, context management, and module management.

Modifying the chemistry code to use the driver API instead of the runtime API requires many changes to each driver function, similar to the changes needed to modify the transport code to use the driver API. Many requirements that are implicitly handled by the runtime API need to be specified for the driver API. Code is added to the beginning of each global subroutine to initialize CUDA, create a context, and load the needed module.

Initialize Driver API Chemistry:

1. Initialize CUDA

2. Get device handle

3. Create CUDA context for the device

4. Load module with GPU function for the current context

Many CUDA functions needed to be changed to use the driver API version instead of the runtime API version, requiring minimal changes to the code. The execution configuration is only used by the runtime API, being replaced by CUDA module management functions to specify the GPU function to call, memory management functions to specify function parameters, and execution management functions to launch the kernel. First a handle to the GPU function is obtained and the block shape is set. Each parameter is then specified by listing the function handle, the location of the parameter in the parameter list, and the size of the parameter.

Execute GPU kernel function:

1. Get function handle

2. Set block shape

3. Calculate alignment of each parameter

4. Set each parameter

5. Set size of all parameters

6. Launch GPU kernel

Modifying the code to use the driver API has no noticeable effects on the running time. The next challenge is to optimize the chemistry code through taking advantage of the different types of memory and features provided by CUDA for both the Rosenbrock code and the QSSA code.

5.3 Chemistry Optimizations

This section discusses the different types of optimizations used to speed up the chemistry subroutines. This primarily focuses on memory based optimizations for each chemical inte-

grator due to their ability to significantly decrease the overall running time.

5.3.1 Rosenbrock Method

Optimizing the memory layout used by the Rosenbrock method offers a significant amount of potential speedup, but requires a significant number of changes to condense the data into the fastest available memories. The original driver API version was forced to place the 974 element (7792 byte) Ghimj and Jacobian arrays into global memory. Due to the slow access times for global memory and the large size of these arrays, moving these arrays into local memory provides significant speedup.

Global memory is only used to copy data from the device to the host, with a single loop at the end of the subroutine copying the concentrations from local memory into global memory. We want minimal use of global memory due to the slow access times. Texture memory passes the chemical concentrations, rate constants, and fixed variables from the host to the device. Texture memory performs about as well as local memory, allowing large arrays that are not modified by the chemical integrator to be stored here. Constant memory holds all arrays used for the rolled function and Jacobian subroutines, method parameters, and integrator parameters. Constant memory allows fast access to constant data used by all threads. Global, texture, and constant memory provide more storage space than needed by the chemical integrators. The remaining arrays are placed in local memory, which contains 16KB of storage space per thread. This requires significant modification of the Rosenbrock data structures, as the arrays originally used by the Rosenbrock code require much more than 16KB of memory per grid cell.

Parameter Values:

Table 5.1: CPU memory requirements for Rosenbrock Rodas-4 and Rodas-3.

Integrator	Main Loop	Func. Eval.	Jac. Eval.	Linear Algebra	Total
Rodas-4	27360	1880	3232	704	33528
Rodas-3	25952	1880	3232	704	32120

```
NVAR = 88;                    Smax = 6;
NFIX = 5;                     LU_NONZERO = 974;
NREACT = 235;                 NSTOICM_LEFT = 404;
```

Original Arrays:

```
double Y[NVAR];               double Fcn[NVAR];
double FIX[NFIX];             double dFdT[NVAR];
double RCONST[NREACT];        double Jac0[LU_NONZERO];
double AbsTol[NVAR];          double Ghimj[LU_NONZERO];
double RelTol[NVAR];          double K[NVAR*Smax];
double Ynew[NVAR];            double Yerr[NVAR];
double Fcn0[NVAR];            int Pivot[NVAR];
```

Function Evaluation:

```
double A[NREACT];
```

Jacobian Evaluation:

```
double B[NSTOICM_LEFT];
```

Linear Algebra:

```
double W[NVAR];
```

Table 5.1 shows that the original STEM data structures require a total of 33528 bytes of data per grid cell for Rodas-4 and 32120 bytes of data per grid cell for Rodas-3. We want to reduce the amount of memory needed, either by condensing multiple arrays into a single array or removing arrays if possible. We can place some arrays in global, texture, or constant memory if needed.

The contents of FIX and RCONST were placed in texture memory, since these arrays are not modified by the chemical integrator. Additionally the concentration values are passed from the CPU to the GPU using texture memory, although they are copied into local memory at the beginning of the chemical integrator.

First we look to see which arrays can be removed. The function using the dFdT array is never called, allowing us to remove this array. The AbsTol and RelTol arrays contain the same value for each index, allowing these arrays to be removed and replaced with constant variables.

Next we look to see which arrays can be combined. The Ghimj array is first used within the prepare matrix subroutine, where the contents of Jac0 are copied into Ghimj. Jac0 is only used again if the step is rejected. Relatively few steps are rejected, so we can remove the Ghimj array and use Jac0 in its place. If a step is rejected, then the Jacobian will need to be recomputed. Similar changes can be used to reduce the memory requirements for the function evaluations. The function evaluation can be called with the K array instead of the Fcn and Fcn0 arrays, allowing the Fcn and Fcn0 arrays to be removed. This also requires the function to be recomputed if a step is rejected, however this is a small penalty due to the small number of rejected steps.

A new array called temp is created to be used in place of many of the other arrays. This is used in place of the arrays Yerr and Pivot, which are only used in small sections of code. This array is passed into the functions FunTemplate, JacTemplate, and KppDecomp where it is used in place of the local arrays A, B, and W. This size of this temporary array was originally NSTOICM_LEFT, the size of the largest array temp would replace, however the resulting

Table 5.2: Comparison of the CPU and GPU memory requirements for Rosenbrock Rodas-4 and Rodas-3.

Integrator	CPU Memory	Global Memory	Texture Memory	Local Memory	Total
Rodas-4	33528	704	1920	15952	18576
Rodas-3	32120	704	1920	14544	17168

memory footprint was still larger than 16KB. Therefore the size of the temp array was set equal to 316 and Ynew was passed into the Jacobian subroutine to store the remainder of the temporary data since Ynew is not used until after the Jacobian subroutine. These changes allow us to condense most of the data into local memory, making minimal use of global and texture memory. The resulting memory requirements are shown in table 5.2. This figure demonstrates that the memory requirements are almost reduced in half.

GPU Arrays:

Global Memory:

```
double YF[NVAR];
```

Texture Memory:

```
double FIX[NFIX];
double RCONST[NREACT];
```

Local Memory:

```
double Y[NVAR];
double Ynew[NVAR];
double K[NVAR*Smax];
double Jac0[LU_NONZERO];
double temp[NSTOICM_LEFT-NVAR];
```

A number of simple changes further optimized the code. Fast math functions provided by CUDA were used when possible. Unnecessary if statements were removed, such as through modifying the stage evaluation code to compute the first stage prior to a loop, instead of calculate the first stage differently from the remaining stages inside the loop.

Originally the rolled loops inside the function and Jacobian evaluation functions were used to reduce compile times. The unrolled CPU code out performed the rolled CPU code by

a significant margin, making unrolled loops a potential optimization. First the original unrolled loops were used with rate constants and fixed variables placed in global memory. These loops resulted in a very large register requirement, causing the registers to overflow into local memory and preventing the code from compiling. Next the rate constants and fixed variables were placed back in texture memory and the unrolled code was modified to use texture fetches. This produced slightly slower results than the rolled code. Therefore the Rosenbrock code kept the rolled loops.

The Rosenbrock chemical integrator was not modified to use shared memory due to the high memory requirements for each grid cell and the limited amount of parallelism within the integrator. Using shared memory would require each thread block to compute a single grid cell, limiting the number of grid cells that could be computed at once. Additionally the full shared memory would likely be needed to allow as much code as possible to run in parallel, limiting the number of threads per block and the GPU occupancy. Additionally sections of the code, such as the code to perform the LU decomposition, have limited amounts of potential parallelism, likely achieving the best performance when using only 1 thread.

The high memory requirements for the Rosenbrock integrator result in code that does not perform very well on GPUs. Therefore we developed and tested the QSSA integrators with a much smaller memory footprint due to their potential to perform much better on GPUs. The next section will discuss the attempts to optimize the QSSA method for the GPU.

5.3.2 QSSA Methods

Both the QSSA and QSSA Exp2 chemical integrators were tested with a variety of optimizations. Both integrators were tested with the same optimizations and had similar performance.

Optimizing the memory layout for the QSSA methods offers a significant amount of potential speedup. The QSSA method has a slightly smaller memory footprint than the QSSA Exp2

method, but the QSSA Exp2 method has simpler loops containing only one if statement within each of the two loops instead of the three if statements per loop in the QSSA method. Due to the much smaller memory footprint, we are also able to experiment with using shared memory.

Global memory copies the concentrations, rate constants, and fixed species from the host to the device. Loops at the beginning of the integration code copy this data into local memory and a loop at the end of the integration code copies the data back to global memory. Tests placing these arrays in texture memory showed no noticeable speedup. These variables can be moved back to texture memory if needed to reduce the amount of local memory used per grid cell. Method and integration parameters are placed in constant memory.

QSSA GPU Arrays:
Global Memory:

```
double YF[NVAR];
double RCT[NREACT];
double F[NFIX];
```

QSSA Local Memory:

```
double Y[NVAR];
double RCONST[NREACT];
double FIX[NFIX];
double P_VAR[NVAR];
double D_VAR[NVAR];
double V1[NVAR];
double V2[NVAR];
```

QSSA Exp2 Local Memory:

```
double Y[NVAR];
double RCONST[NREACT];
double FIX[NFIX];
double P_VAR[NVAR];
double D_VAR[NVAR];
double phi[NVAR];
double A[NVAR];
double V2[NVAR];
double K1[NVAR];
```

Table 5.3: GPU memory requirements for QSSA and QSSA Exp2.

Integrator	Global Memory	Local Memory	Total
QSSA	2624	5440	8064
QSSA Exp2	2624	6848	9472

Table 5.3 shows the memory layout for the GPU version of the QSSA and QSSA Exp2 methods. The QSSA Exp2 method requires two additional NVAR sized arrays, resulting in 1408 additional bytes of local memory.

The QSSA methods originally used unrolled loops due to the significantly smaller memory requirements needed to perform the split function evaluation. The QSSA code was able to fit all needed data into the registers and local memory. Tests using rolled loops within parts of the split function evaluation resulted in higher running times. Therefore the split function evaluation code was left unrolled.

A number of simple changes further optimized the code. Fast math functions provided by CUDA were used when possible. Some code was reorganized by removing unnecessary branches in order to further optimize the code.

Due to the much smaller memory footprint and more potential for parallelism within the integration subroutine, shared memory was used in an attempt to further optimize the code. The QSSA code contains a main while loop with two split function evaluations which can be modified to run in parallel and two parallelizable loops.

Main while loop:

```
while (T < TOUT) {

  Fun_SPLIT(Y, ...);
```

```
  for(i=0; i<NVAR; i++)
    Calculate V2[i]

  Fun_SPLIT(V2,...);

  for(i=0; i<NVAR; i++)
    Calculate Y[i]
}
```

First we need to set variables specifying which data each thread needs to operate on. The thread index tid ranges from 0 to the maximum number of threads. The load thread index, ltid, specifies which data to load from global memory. This is needed when more than 1 thread computes the data for a single grid cell. This value ranges from 0 to the maximum number of grid cells. The loop starting index, tstart, specifies which index of a parallel loop each thread should compute. The shared memory index specifies which index in a shared memory array each thread should use. These variables allow each thread to load the necessary data and to perform computations in parallel.

Thread Index:

```
tid = (blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x
  * blockDim.y + (threadIdx.y * blockDim.x) + threadIdx.x;
```

Load Thread Index:

```
ltid = tid/nt;
```

Thread Start Index:

```
tstart = tid % nt;
```

Shared Memory Index:

```
si = ((tid-(tid%nt))/nt) % sc;
```

The code was first modified to allow up to NVAR threads to work together on each loop. This required a load thread index variable to allow each of NVAR threads to use the same data for the concentrations, rate constants, and fixed variables. A thread start variable sets the starting index for each thread for each loop. All local arrays were moved to shared arrays, allowing up to five grid cells to be placed in shared memory at once. This allows both loops to be computed in parallel. Setting the number of threads equal to NVAR allows the for loops to be removed, as each thread will compute the correct iteration of the loop.

Parallel main while loop:

```
while(T < TOUT) {

  Fun_SPLIT(Y,...,tid);

  i=tstart;
  Calculate V2[i]

  Fun_SPLIT(V2,...,tid);

  i=tstart;
  Calculate Y[i]
}
```

Next the split function evaluation is rolled, allowing each line of unrolled code to be computed by a different thread. This requires creating a number of constant arrays containing array indices and coefficients, and then creating rolled loops to correctly compute the data. This allows multiple threads to compute the split function at once. The drawback is that some threads will do a large amount of work, while other threads do very little work due to the uneven workloads for computing each element of the split function.

```
for(i=tstart; i<NREACT; i=i+nt)
```

```
  Initialize A
for(i=tstart; i<NREACT; i=i+nt) {
  Set index variables
  while(...) {
    Calculate A
  }
}

\\ Calculate Production Term
for(i=tstart; i<88; i=i+nt)
  Initialize P_VAR
for(i=tstart; i<62; i=i+nt) {
  Set index variables
  while(...) {
    Calculate P_VAR
  }
}

\\ Calculate Destruction Term
for(i=tstart; i<88; i=i+nt)
  Initialize D_VAR
for(i=tstart; i<81; i=i+nt) {
  Set index variables
  while(...) {
    Calculate D_VAR
  }
}
```

Since only a small part of the shared memory is currently in use, we can use shared memory

to compute multiple grid cells at once. In order to compute the most grid cells at once, we must move the rate constants and fixed species to texture memory. This allows up to five grid cells to be computed at once, with up to NVAR threads per grid cell being used effectively. Each shared array is increased in size to hold the data for multiple grid cells at once. The variable si is used to specify which section of the shared memory arrays each thread should operate on. However using from 32(Warpsize) to 88(NVAR) threads per grid cell and 1 to 5 grid cells all produced significantly slower results than for the original GPU version of STEM using one thread to compute each grid cell. The limited number of threads per block, number of registers, and amount of shared memory resulted in slower running times. Additionally the function evaluations required unbalanced workloads, where some threads would perform only an assignment statement, while others would perform over fifty additions and multiplications.

We also developed a version of the QSSA Exp2 method using multiple kernels. This allowed optimal block sizes to be chosen for each kernel and shared memory to be used to quickly compute one small section of code at a time. This involved creating kernels for each loop and creating a kernel for each part of the split function evaluation. The while loop is placed inside the chemistry driver code and calls each of these kernels each iteration.

```
while(T < TOUT) {

  Launch calculate A kernel
  Launch calculate P_VAR kernel
  Launch calculate D_VAR kernel

  Launch calculate V2 kernel

  Launch calculate A kernel
  Launch calculate P_VAR kernel
  Launch calculate D_VAR kernel
```

```
  Launch calculate Y
}
```

This code runs significantly slower than the single kernel chemical integrators. While each kernel runs in parallel, some of the kernels do not have any effective method to take advantage of shared memory. The most significant problem is that all arrays used in multiple kernels must be stored in global memory. This results in a very large number of reads and writes to global memory in comparison to the number of reads and writes to global by the single kernel integrators. The slow access times for global memory cause this code to run very slowly. This code also suffers from having unbalanced workloads when computing the function evaluations. Therefore the single kernel integrators will continue to be used.

5.4 Summary of Chemistry Optimizations

This section will briefly review different optimization attempts for STEM chemistry on the GPU.

There are a number of successful optimizations that were fully implemented in STEM chemistry. Experimenting with the number of registers and thread block size allowed us to find the best settings for each chemical integrator. Many settings gave similar good performances, with some settings giving slightly better performance. Fast math functions provided by CUDA allowed some computations to be performed faster without sacrificing accuracy.

Reorganizing each algorithm to use as little memory as possible allowed more data to be stored in faster memories, reducing running times. Using rolled function and Jacobian evaluation functions reduced the number of registers and amount of local memory used, which was needed for some chemical integrators to compile. Other integrators performed better with the unrolled function evaluations if there was enough available memory.

Global memory transferred data between the CPU and GPU to reduce the number of reads and writes to global memory which resulted in significantly slower kernels. Texture memory held rate constants and fixed variables due to the large amounts of read-only memory. Local memory stored all other per thread data, allowing this data to be accessed faster. Constant memory stored constant data used by all threads due to the fast access times for read only data.

A number of shared memory optimizations were tested that did not produce faster code. Kernels were developed to use multiple threads to compute each grid cell for the QSSA Exp2 method. This was slower due to the limited number of threads per grid cell, limited number of registers, and limited amounts of shared memory. Another integrator was modified to call multiple fully optimized kernels. However this integrator ran slower due to greatly increasing the number of reads and writes to global memory. Both kernels suffered from unbalanced workloads when computing the function evaluations.

Chapter 6

STEM Results

This chapter presents the results for STEM transport and chemistry. Tests are performed to find the optimal settings for key simulation parameters for each method and to compare different methods. Each method will also be compared with the original single threaded CPU version and the OpenMP version. The fastest running times are used to produce the following results.

Many different attempts to optimize the code discussed in the previous sections that did not speed up the code are not explored further here. This section focuses on the fastest versions of each method.

6.1 Transport Results and Analysis

First we want to find the optimal settings for the implicit transport method. This includes varying the size of each thread block and the maximum number of registers per thread block for each of the three transport subroutines.

For the thread block tests, we use the maximum number of registers for each method(X-Transport=34, Y-Transport=34, and Z-Transport=41). The number of threads is chosen as

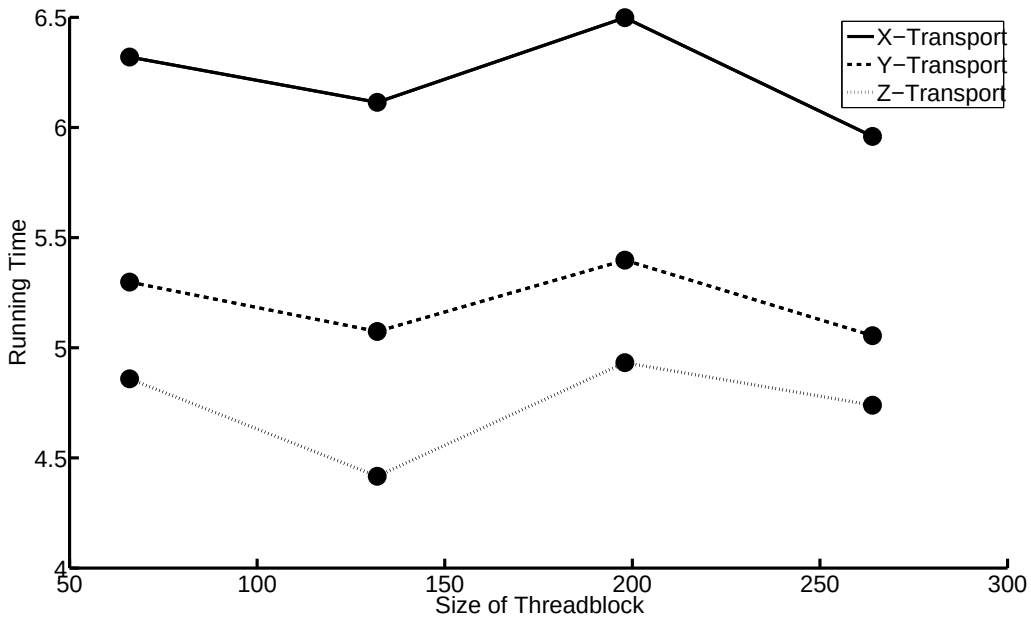


Figure 6.1: Comparison of the running time of the implicit transport methods on the GPU for different thread block sizes.

a multiple of 66, since up to 66 threads can be used to compute each grid cell. The data for up to four grid cells can be stored in shared memory at once, allowing up to 264 threads to be used. The register tests use 264 threads.

Figure 6.1 shows that using 132 and 264 threads produces the best results, with X and Y transport producing slightly better results for 264 threads and 132 threads producing slightly better results for Z transport. Figure 6.2 shows that similar results are produced for 28 or more registers per thread block, while there is a noticeable slowdown for less than 28 threads. Using less than 20 registers prevented the code from compiling. Therefore 264 threads and the maximum number of registers are used for future implicit transport tests.

These figures demonstrate that Z-Transport runs faster than Y-Transport, both of which are faster than X-Transport. As discussed at the beginning of section 4.3, the Z-Transport has 550 independent columns, Y-Transport has 525 independent rows, and X-Transport has

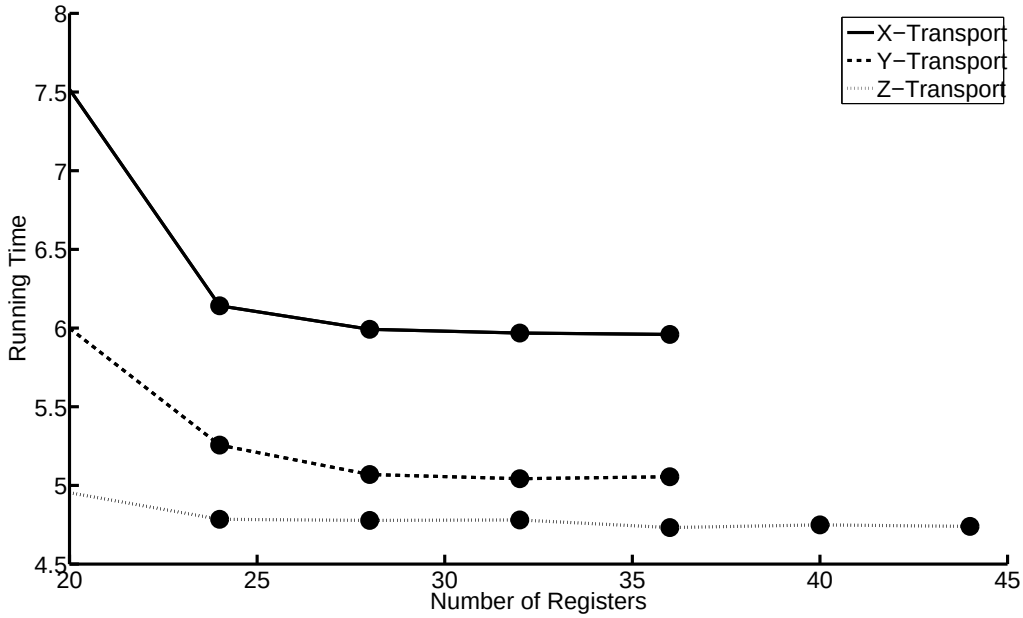


Figure 6.2: Comparison of the running time of the implicit transport methods on the GPU for different numbers of registers per thread block.

462 independent rows. Methods with more independent computations are able to occupy a smaller memory footprint per thread and take advantage of the many processors provided by GPUs, resulting in faster running times.

After finding the optimal GPU settings for each implicit transport method, we want to compare each GPU implicit transport routine with the original C version of the implicit transport routine.

Table 6.1 shows running times between 4.73 and 5.95 milliseconds and speedups ranging from 7.61 to 9.49 for the implicit transport methods. These timings do not take into account memory transfer times. Additionally the occupancy numbers show that the fastest explicit transport settings do not fully take advantage of the power of the GPUs, using 28% of the GPU at best.

Table 6.1: Comparison of the CPU and GPU running time (milliseconds) for 1 iteration of each implicit transport method.

Transport Direction	CPU Time	GPU Time	Occupancy	Speedup
X-Transport	45.25	5.95	28%	7.61
Y-Transport	47.91	5.05	28%	9.49
Z-Transport	41.01	4.73	28%	8.67

Table 6.2: Comparison of the CPU and GPU running time (milliseconds) for 1 iteration of each explicit transport method.

Transport Direction	CPU Time	GPU Time	Occupancy	Speedup
X-Transport	28.46	5.57	28%	5.11
Y-Transport	28.23	5.10	28%	5.54

Next we want to find the optimal settings for the explicit transport method. Similar tests are performed as above, although Z-transport is not included due to the significantly increased times needed to produce accurate results.

Figure 6.3 shows that using 264 threads produces the best results for X and Y transport, with 132 threads being slightly slower. Figure 6.4 shows that using 32 registers produces slightly better results than using less registers. Using less than 20 registers prevented the code from compiling. Therefore 264 threads and 32 registers are used for future explicit transport tests.

After finding the optimal GPU settings for each explicit transport method, we want to compare each GPU explicit transport routine with the original C version of the explicit transport routine.

Table 6.2 shows running times of 5.1088 and 5.5760 milliseconds and speedups of 5.11 and

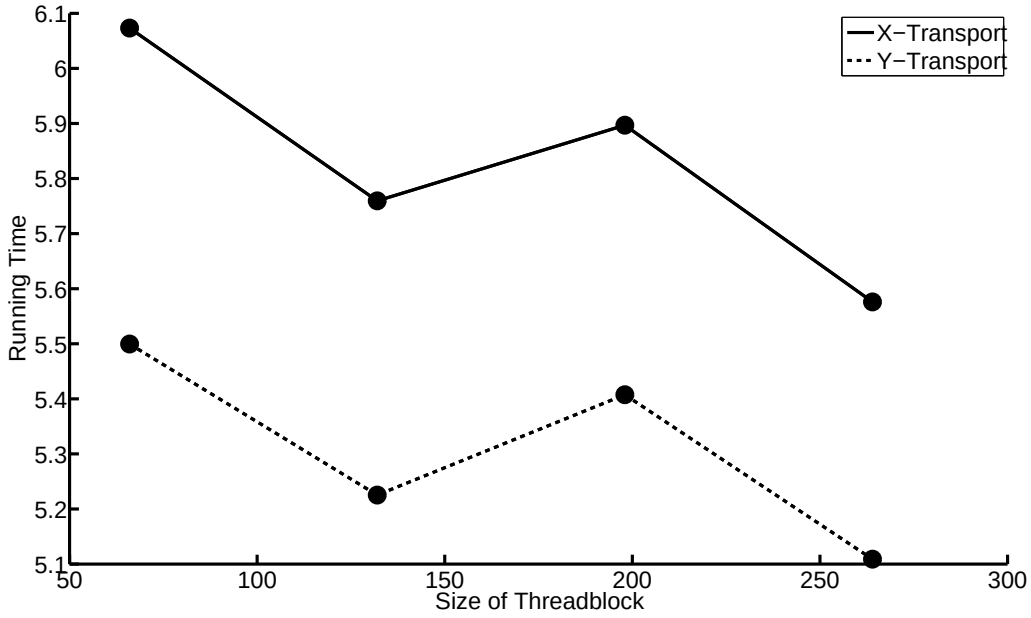


Figure 6.3: Comparison of the running time of the explicit transport methods on the GPU for different thread block sizes.

5.54 for the explicit transport methods. These timings do not take into account memory transfer times. Additionally the occupancy numbers show that the fastest explicit transport settings do not fully take advantage of the power of the GPUs, using 28% of the GPU at best.

Next we want to look at the memory transfer times. This includes the data transfers during initialization routines, prior to each transport routine, and during the closing routines.

Table 6.3 demonstrates that the memory transfer times are fairly minimal, with each set of memory transfers taking between 0.1919 and 2.0227 milliseconds. Overall the memory transfers take 13.8790 milliseconds, which is a significant amount of time in comparison to each transport subroutine.

Next we compare the overall running times for 1 time step for implicit transport and explicit

Table 6.3: Comparison of the GPU memory transfer times (milliseconds) for 1 full iteration.

Routine	Running Time
Initialize X	1.3691
Memcopy X Data	0.1919
Memcopy Y Data	1.8304
Memcopy Z Data	1.5093
Close Z	1.4078
Initialize Z	1.2433
Memcopy Z Data	1.5062
Memcopy Y Data	1.3317
Memcopy X Data	1.6394
Close X	2.0227
Total	13.8790

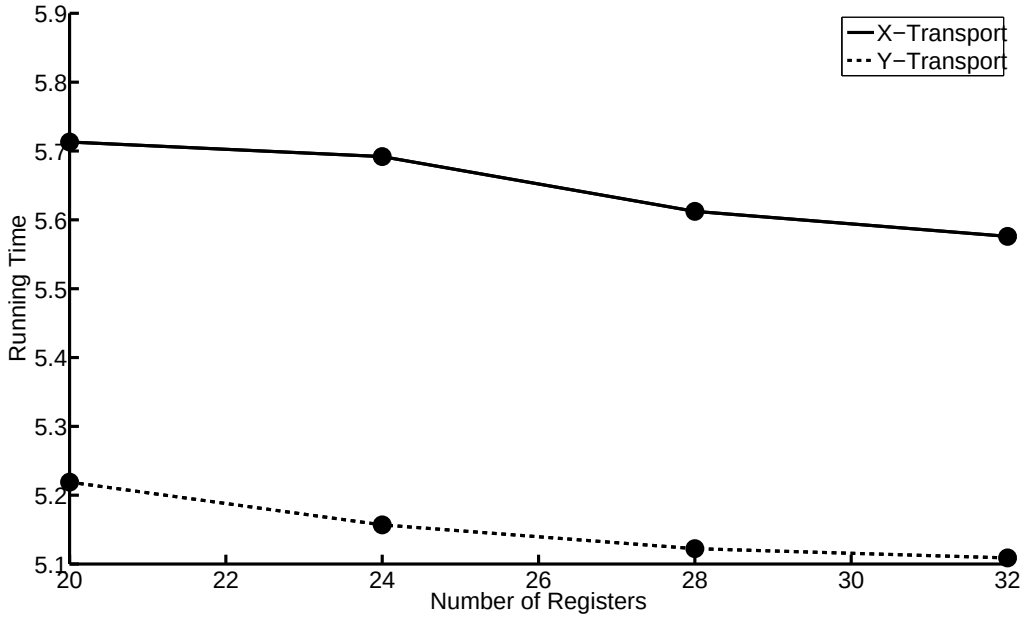


Figure 6.4: Comparison of the running time of the explicit transport methods on the GPU for different numbers of registers per thread block.

transport. These timings include all memory transfer times and the times for each of the transport methods. Note that each transport method loads some data from the CPU prior to executing the transport code, resulting in different running times for each transport routine due to loading different amounts of data.

Table 6.4 shows overall running times of 41.4114 seconds for implicit transport and 39.8735 for explicit transport. Due to the much longer running time for explicit Z transport, the implicit Z transport is also used by the explicit transport routine. This shows that about a third of the overall running time is due to memory transfers and about two thirds is due to the transport subroutines.

Once we have the running times for each transport method for a full iteration, we want to compare these running times to the CPU running times for a full iteration.

Table 6.4: Comparison of the GPU transport total running times (milliseconds) for 1 full iteration.

Transport Routine	Implicit Transport	Explicit Transport
Init X	1.3691	1.3691
Init Z	1.2433	1.2433
X-Transport x 2	12.3755	11.15
Y-Transport x 2	10.5224	10.21
Z-Transport x 2	12.4703	12.4703
Close Z	1.4078	1.4078
Close X	2.0227	2.0227
Total	41.4114	39.8735

Table 6.5: Comparison of the CPU and GPU transport total running times (ms) for 1 full iteration.

Transport Routine	CPU	GPU	Speedup
Implicit Transport	268.34	41.41	6.48
Explicit Transport	195.43	39.87	4.90

Table 6.6: Comparison of the transport running times(milliseconds) for GPU STEM and OpenMP STEM.

Method	CPU	GPU	OpenMP	GPU Speedup	OpenMP Speedup
Implicit X	45.25	5.95	17.54	7.61	2.57
Implicit Y	47.61	5.05	16.43	9.49	2.89
Implicit Z	41.01	4.73	18.74	8.67	2.19
Explicit X	28.46	5.57	12.03	5.11	2.36
Explicit Y	28.23	5.10	14.52	5.54	1.94

Table 6.5 shows that using GPUs results in a 6.48 speedup for implicit transport and a 4.9012 speedup for explicit transport.

Once we have analyzed the running times for GPU transport, we need to compare these running times against OpenMP running times. Table 6.6 shows a fairly small speedup for each transport method, producing at most a 2.89 times speedup. This is likely due to the very fast running times for the STEM transport methods, as OpenMP generally performs better on longer loops. This shows that GPUs produce a faster running time than OpenMP for these transport methods.

Overall, these transport simulation results demonstrate that GPUs can reduce the overall running time by a significant amount. However, many applications can use GPUs to achieve over a 100 times speedup.

The STEM transport subroutines face a number of problems when running on GPUs. First, there is a large amount of data needed per grid cell, requiring each thread to load and store a large amount of data. Most applications that see the best speedups use a fairly small amount of data per thread. This allows the data for a large number of threads to be stored in the fastest memories while maintaining 100% occupancy, reducing the data access times. STEM transport must choose between storing more data per thread in the fastest memories

and maintaining a higher occupancy.

Second, the STEM parameters do not match the ideal parameters needed to achieve maximum performance on GPUs. For example, it is best to use a multiple of 32 threads per thread block, while STEM achieves the best results when using a multiple of 66 threads due to each transport method containing a main loop over all 66 chemical species. This results in wasted cycles, which reduces the potential speedups.

Third, STEM transport does not provide many opportunities to use shared memory. Many of the fastest GPU programs are able to take advantage of shared memory to allow many threads to work together to compute data. STEM transport does provide some opportunities to use shared memory through allowing some data to be calculated by an entire thread block and then used by each thread, but many sections of code cannot use shared memory effectively.

6.2 Chemistry Results and Analysis

First we want to find the optimal settings for the Rosenbrock chemistry integrator for both Rodas-3 and Rodas-4. This includes varying the size of each thread block and the maximum number of registers. For the Rodas-3 and Rodas-4 thread block tests we set the maximum number of registers to 20. For the register tests, we set the thread block size to 160.

Figure 6.5 shows using 96 and 160 threads produces the best results. Using thread blocks with more than 160 threads produce significantly worse results than using 160 threads or less. Using 160 threads clearly produces the best results for Rodas-4, while Rodas-3 produces about the same results for both 96 and 160 threads.

Figure 6.6 shows that both Rodas-3 and Rodas-4 perform best with a smaller number of registers per thread. We also see that there are certain barriers that significantly reduce the running time once they are passed. This includes dropping from 48 to 44 registers for

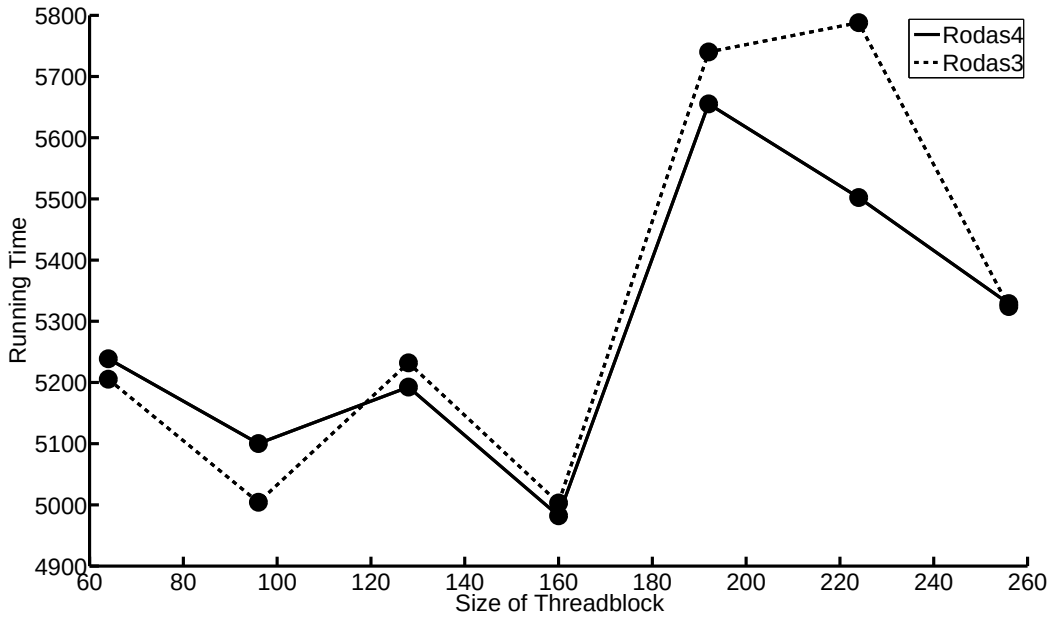


Figure 6.5: Plot comparing the running time of Rosenbrock methods on the GPU for different thread block sizes.

both integrators, dropping from 24 threads for Rodas-4, and dropping below 28 threads for Rodas-3. Using less than 20 registers prevented the code from compiling.

Next we want to perform the same tests for the QSSA methods to find the optimal settings for the QSSA and QSSA Exp2 chemical integrators. For the QSSA thread block tests we set the maximum number of registers to 80 and for QSSA Exp2 we set the maximum number of registers to 128. For the register tests, we set the thread block size to 64x181 for QSSA and 128x91 for QSSA Exp2.

Figure 6.7 shows that using 96 threads per thread block clearly produces the worst results, while other numbers of threads performed similarly. For QSSA, 64 threads performs the best, while 128 threads performs best for QSSA Exp2. Note that using more than 192 threads for the QSSA method or more than 128 threads for the QSSA Exp2 method would exit the chemical integration code early and not produce any results.

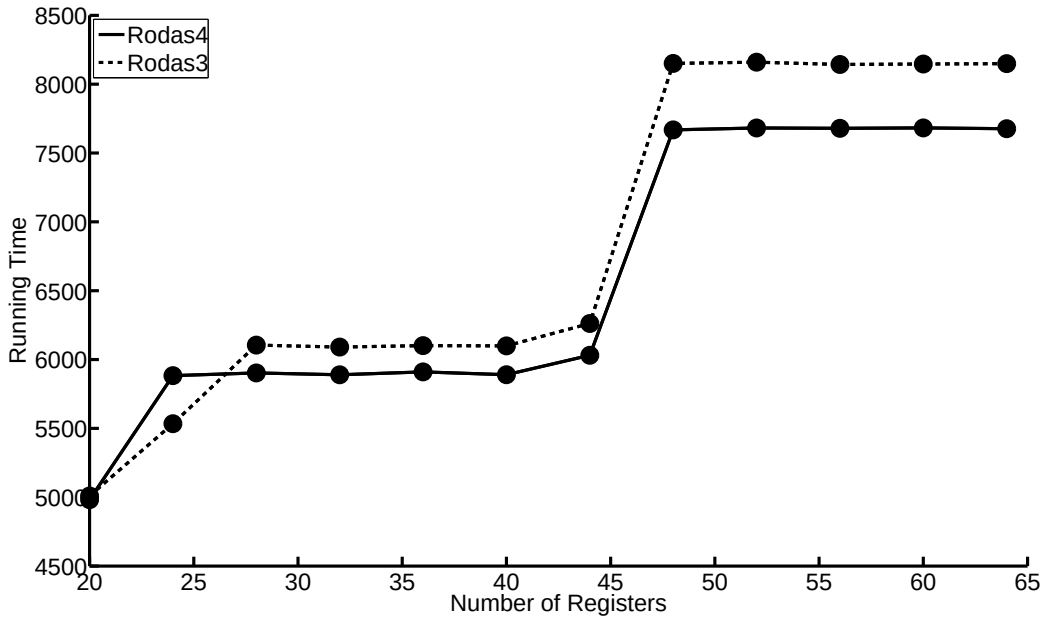


Figure 6.6: Plot comparing the running time of Rosenbrock methods on the GPU for different numbers of registers per thread block.

Figure 6.8 shows that using larger numbers of registers produced better results. 80 registers performed best for QSSA and 128 registers performed best for QSSA Exp2. Using less than 42 registers resulted in significantly slower running times. Using less than 20 registers prevented the code from compiling.

Modifications were made to the QSSA Exp2 method to use 16 registers. These changes involved simplifying some operations, such as using single precision divides instead of double precision divides. This allowed 100% occupancy to be achieved. However the resulting code was much slower than using 128 registers and a low occupancy.

After finding the optimal settings for each method, we want to compare the CPU and GPU versions of each method to see which has the best performance improvement. The CPU time, GPU time, occupancy, and speedup are listed in the table.

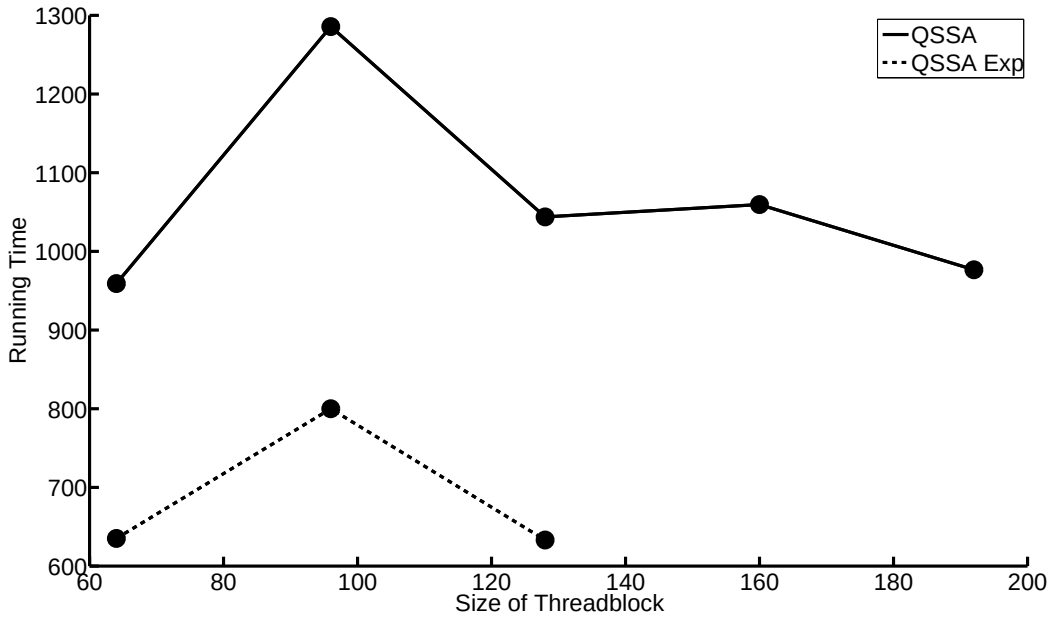


Figure 6.7: Plot comparing the running time of the QSSA methods on the GPU for different thread block sizes.

Table 6.7 shows that the Rosenbrock methods achieve little if any speedup per iteration, while the QSSA methods achieve speedups greater than 8. We see that the CPU times ranged from 4.78 seconds to 7.69 seconds, with the QSSA method having the slowest running time. We see the GPU times dropping to less than 1 second for the QSSA methods, while the Rosenbrock methods still take about 5 seconds. The Rodas3 method became slightly slower. The occupancy numbers show that the Rosenbrock methods are able to obtain a decent occupancy, however these methods are still very slow. The QSSA methods have a very low occupancy, however they achieved the best running times. This low occupancy occurs as a result of using a very large number of registers. This high number of registers is needed for the unrolled split function evaluation. Using less registers or using the rolled split function evaluation code results in longer running times. Therefore finding a more efficient method for calculating the split function may result in higher occupancies and much faster running

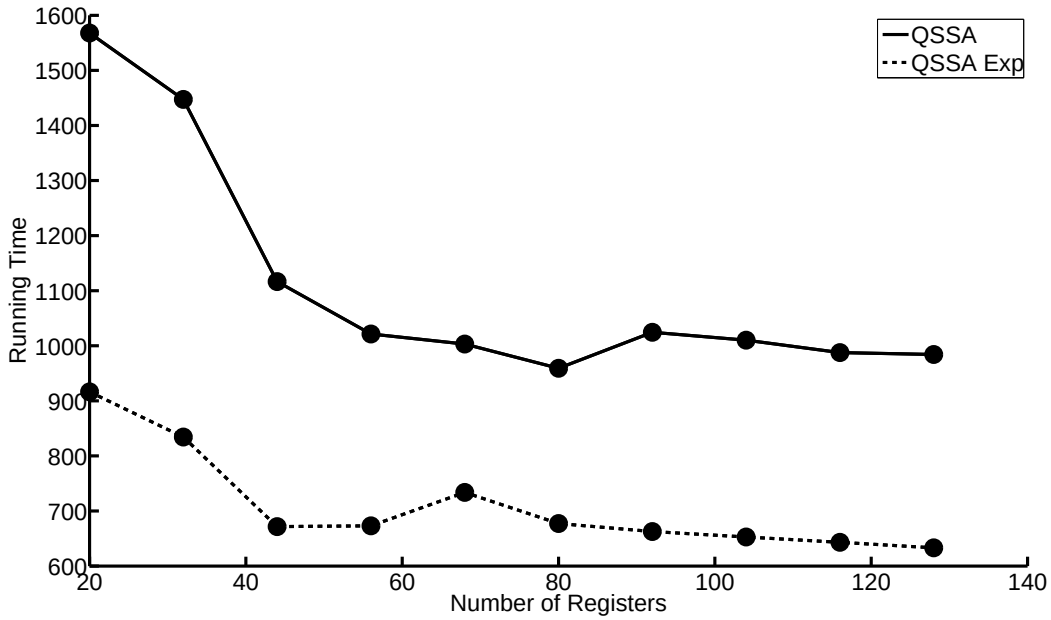


Figure 6.8: Plot comparing the running time of the QSSA methods on the GPU for different numbers of registers per thread block.

times.

The memory transfer time from the CPU to GPU prior to each chemistry subroutine is 13.71ms and the memory transfer time from the GPU to the CPU after each chemistry subroutine is 4.02ms. This results in an overhead of 17.73ms, which is minimal in comparison to the overall GPU running times for the chemical integrators.

Once we have analyzed the running times for GPU chemistry, we need to compare these running times against OpenMP running times. Table 6.8 shows near linear speedups for both the Rosenbrock methods and the QSSA methods, producing close to an 8 times speedup when using 8 processors. This shows a significant improvement in comparison to the GPU running times for the Rosenbrock methods. Using larger shared memory systems would likely produce much more than an 8 times speedup. This shows that OpenMP produces near

Table 6.7: Comparison of the CPU and GPU running time (seconds) for the Rosenbrock and QSSA methods for 1 iteration.

Integrator	CPU Time	GPU Time	Occupancy	Speedup
Rodas4	6.38	4.98	63%	1.28
Rodas3	4.78	5.00	63%	0.95
QSSA	7.69	0.96	19%	8.01
QSSA Exp	5.51	0.63	13%	8.74

Table 6.8: Comparison of the chemistry running times(seconds) for GPU STEM and OpenMP STEM.

Integrator	CPU	GPU	OpenMP	GPU Speedup	OpenMP Speedup
Rodas-4	6.38	4.98	0.81	1.28	7.87
Rodas-3	4.78	5.00	0.62	0.95	7.71
QSSA	7.69	0.96	0.98	8.01	7.84
QSSA Exp2	5.51	0.63	0.70	8.74	7.87

linear speedups for more complex chemical integrators such as the Rosenbrock integrators with a large memory footprint as well as for simpler chemical integrators such as the QSSA integrators with a small memory footprint.

Overall, these results demonstrate that the QSSA methods perform significantly better on the GPU than the Rosenbrock methods. This is due to the smaller memory footprint and simplified nature of the QSSA methods. The Rosenbrock methods require a significant amount of code modification to fit all of the needed data into local memory on the GPU, while the QSSA methods allow the data for multiple grid cells to fit into local memory on the GPU at once. Additionally the QSSA methods contains a simpler code structure, with two for loops and two function evaluations, while the Rosenbrock code requires multiple function

evaluations and a Jacobian evaluation in addition to a number of loops. This allows the QSSA methods to run much faster on the GPU.

Analyzing these results further suggests that the Rosenbrock method is memory bound due to performing best with a higher occupancy. The higher occupancy allows more blocks to run at once, hiding the memory access times. The QSSA method performs best when using a large number of registers and a lower occupancy. Using more registers allows each grid cell to complete much faster, suggesting that the QSSA method is computation bound.

6.3 Full STEM Results and Analysis

After looking at the individual performance of the GPU transport and chemistry codes, we want to see the overall performance of STEM for the GPU. For this experiment, we will test the fastest implicit and explicit transport methods with the Rosenbrock and QSSA chemistry integrators.

First we look at the overall running times for a 6 hour simulation containing both transport and chemistry.

Table 6.9 shows that the Rosenbrock simulations achieve between a 0.93 and 1.17 speedup, while the QSSA simulations achieve between a 1.75 and 1.87 speedup. These speedups are much lower than the speedups for the chemical integrators or for the transport routines. Therefore we need to see what percentage of the overall running time is GPU code and what percentage is CPU code.

We first look at the overall percentage of running time spent in chemistry code and in the transport code.

Table 6.10 shows that 50-53 percent of the running time for the Rosenbrock simulations is spent in the GPU chemical integration code, while 10-14 percent of the running time for the QSSA methods is spent in the chemical integration code. Table 6.11 shows that about

Table 6.9: Comparison of the overall STEM running time (seconds) on the CPU and GPU with implicit transport for a 6 hour simulation.

Integrator	Transport	CPU Time	GPU Time	Overall Speedup
Rodas4	Implicit	266.37	240.57	1.17
Rodas3	Implicit	226.98	226.12	1.00
QSSA	Implicit	302.69	161.44	1.87
QSSA Exp2	Implicit	279.19	159.24	1.75
Rodas4	Explicit	262.68	241.42	1.09
Rodas3	Explicit	223.22	238.86	0.93
QSSA	Explicit	300.17	159.79	1.88
QSSA Exp2	Explicit	281.47	159.03	1.77

Table 6.10: Comparison of the STEM chemistry running times (seconds) on the CPU and GPU for a 6 hour simulation.

Integrator	CPU Chemistry	GPU Chemistry	GPU Percent of Time
Rodas4	153.12	119.94	50%
Rodas3	114.72	120.00	53%
QSSA	184.56	23.46	14%
QSSA Exp	132.24	15.54	10%

Table 6.11: Comparison of the STEM transport running times (seconds) on the CPU and GPU for a 6 hour simulation.

Transport	CPU Transport	GPU Transport	GPU Percent of Time
Implicit	6.44	0.99	1%
Explicit	4.69	0.95	1%

Table 6.12: Comparison of setup times (seconds).

Code Section	1 Iteration	6 Hour
Integrator Setup/Close	5.30	127.20
Transport Setup/Close	0.10	2.4

1 percent of the overall running time is spent in the GPU transport code for both implicit and explicit transport. Therefore we see that about half of the overall running time for the Rosenbrock simulations is spent in single threaded CPU code, while about 85% of the overall running time for the QSSA simulations is spent in single threaded CPU code. Therefore in order to improve the overall STEM running time, we will need to parallelize the remaining CPU code to achieve a good performance.

Next we will look at the places a significant amount of CPU time is being spent.

Table 6.12 shows that a very large amount of time is spent in the integration setup and closing code. This includes code for calculating the rate constants and the effects of radiation using the Tropospheric Ultraviolet-Visible (TUV) radiation model [39]. The transport setup and closing code runs quickly, primarily copying data from the STEM data structures into STEM transport data structures.

These results could be significantly improved through developing a GPU version of all code that modifies the main concentration array for STEM. Currently only transport and chemistry run on the GPU, but there are a number of subroutines throughout the code that could benefit from being run on the GPU.

This research focuses primarily on studying the effects of running key atmospheric modeling processes such as transport and chemistry on GPUs. Future research will use the results from this thesis to develop GPU codes for larger commonly used models such as GEOS-Chem. Therefore fully parallelizing additional code within STEM will have a minimal benefit to the atmospheric modeling and high performance computing communities.

6.4 Accuracy of STEM

This section looks at the accuracy of the different chemical integrators. The Rosenbrock methods are commonly used due to their speed and accuracy, while the QSSA methods are rarely used due to their slow running times on a single processor and inaccurate results. We have seen that the QSSA methods parallelize much faster on the GPUs. Our next step is to compare their accuracy to the Rosenbrock methods. We use the Rosenbrock methods as a measure of accuracy due to their frequent use within the atmospheric modeling community through KPP and GEOS-Chem.

We run each chemical integrator for a 6 hour simulation with implicit transport. We then compare the results for O3. Note that when performing 6 hour simulations for the QSSA methods, the Tropospheric Ultraviolet and Visible (TUV) Radiation Model [39] subroutines originally produced error messages and stopped the program early. However, commenting out the error messages and the code to stop the program allowed the simulation to run to completion. First we compare Rosenbrock Rodas4 and Rodas3.

Figure 6.9 shows that the Rosenbrock methods produce about the same results. Therefore we will only use Rodas-4 in the remaining tests.

Next we compare the QSSA methods with each other.

Figure 6.10 shows that the QSSA methods produces some similar results, but that they also produces some results that are significantly different from each other.

Next we compare the Rosenbrock Rodas4 method to the QSSA and QSSA Exp2 methods.

Figures 6.11 and 6.12 show that the both QSSA methods produce noticeably different results than the Rosenbrock methods. We see that the QSSA methods produce significantly different values than the Rosenbrock methods in many places.

Next we compare visual plots of the Rosenbrock and QSSA simulations using explicit transport for the Northeast United States on July 20, 2004 at 6:00 PM GMT. We plot the

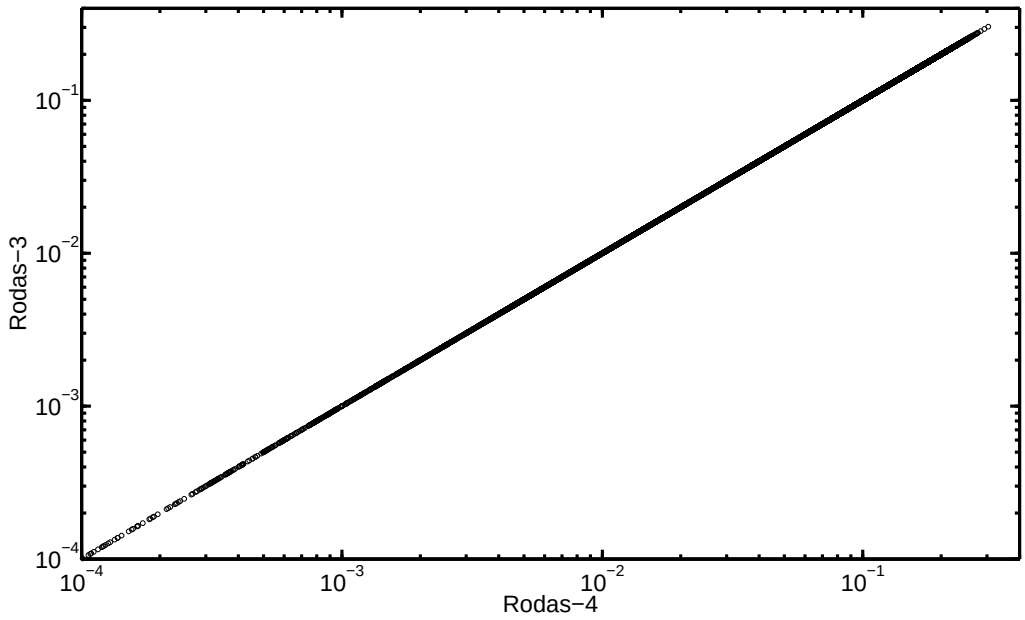


Figure 6.9: Plot comparing the values of O3 for the Rosenbrock Rodas4 and Rodas3 methods for a 6 hour simulation.

concentration results for O3 after a 6 hour simulation for an average of all 21 levels and the ground level in each grid cell.

The graphs in figures 6.13 to 6.18 display the concentration values of O3 on top of a map of the United States in order to provide readers with a more intuitive sense of the state of the atmosphere. The Rosenbrock Rodas4 graphs show higher O3 levels along the shores, especially in the north east. We also see significant differences between the Rosenbrock and both QSSA methods. This is due to the QSSA methods producing inaccurate values, including many zeros in places the Rosenbrock methods are producing nonzero values. The difference graphs in figures 6.19 and 6.20 show small but significant differences in many grid cells, with an area near the middle of the graph showing a large difference. We also see noticeable difference along the bottom part of the graph. This results in the Rosenbrock methods providing a much better simulation of the atmosphere and therefore producing

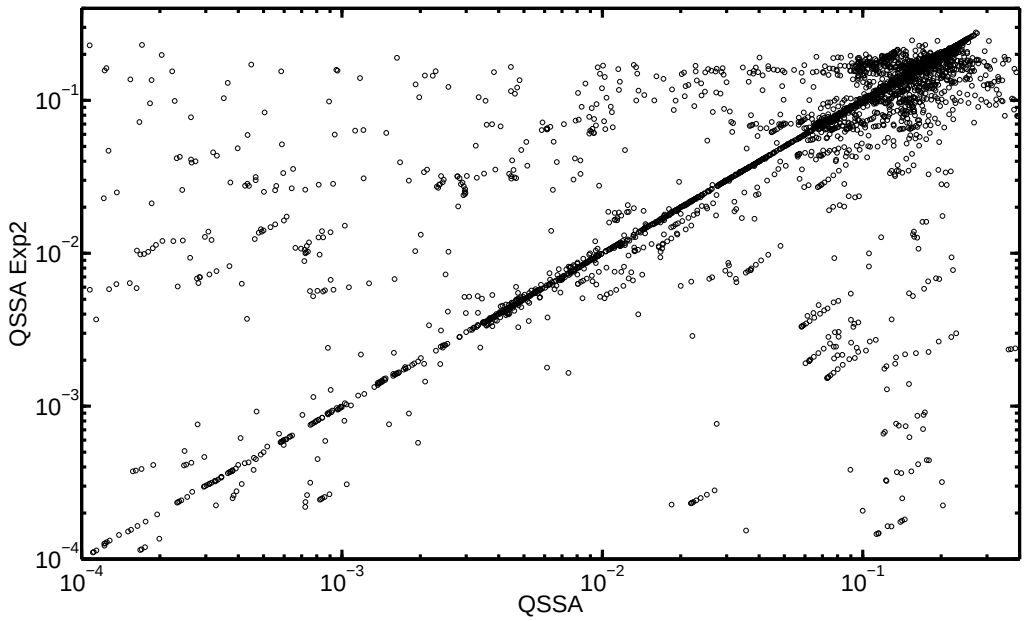


Figure 6.10: Plot comparing the values of O3 for the QSSA and QSSA Exp2 methods.

graphs that can be used to better understand the state of the atmosphere.

Overall, looking over the QSSA results shows significant differences not only for O3 but also for other chemical species. The above graphs clearly demonstrate that the QSSA methods are not always very accurate. However their improved performance on the GPU in comparison to the Rosenbrock methods suggests that in order to achieve the best performance on GPUs these methods should be reexplored to develop simpler methods with a small memory footprint that are also accurate.

6.5 Summary of Most Beneficial Optimizations

This research has discussed a variety of different optimizations to allow chemical transport models to execute code on GPUs. There were a number of optimizations that were very

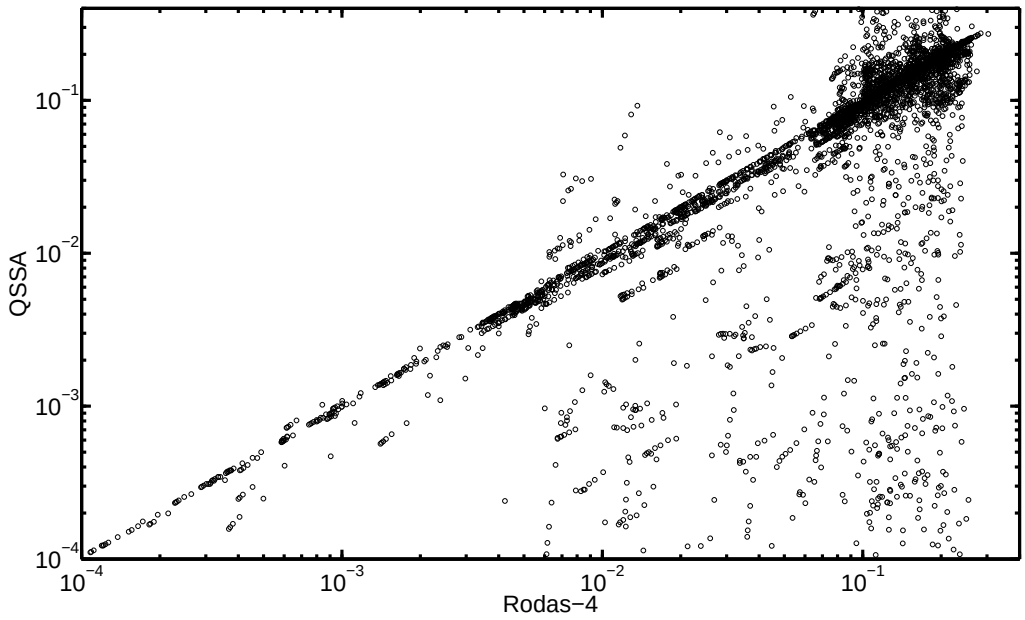


Figure 6.11: Plot comparing the values of O3 for the Rosenbrock Rodas4 and QSSA methods.

beneficial when developing GPU codes.

Experimenting with the number of registers and thread block size allowed us to find the best settings for each GPU function. Many settings gave similar good performances, however some settings gave better performance.

GPUs perform best when using smaller amounts of memory per thread block. Therefore programs should be modified to use minimal amounts of memory. Removing arrays that are not needed or combining multiple arrays into a single array can reduce the amount of memory needed. Modifying functions to use loops instead of unrolled code can reduce the number of registers needed in addition to reducing the amount of overflow into local memory.

GPUs perform best when taking advantage of the fastest memories available. Therefore global memory should be used primarily for data that needs to be transferred from the GPU to the CPU. Any data passed from the CPU to the GPU using global memory should be

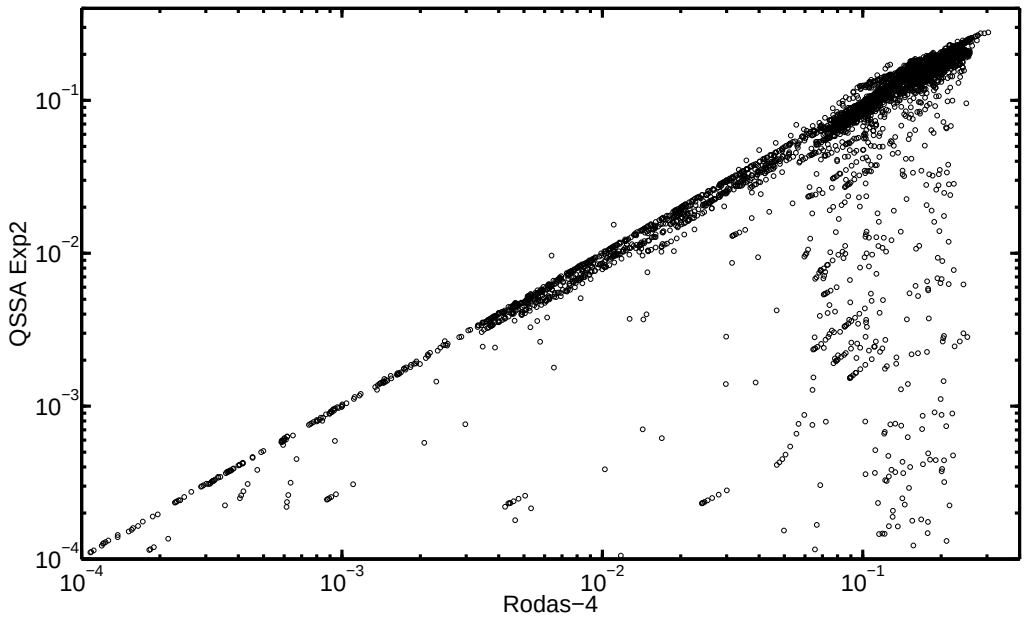


Figure 6.12: Plot comparing the values of O3 for the Rosenbrock Rodas4 and QSSA Exp2 methods.

copied into local memory if it needs to be accessed multiple times. Local memory allows per thread data to be accessed quickly. Therefore per thread data that needs to be modified by a GPU function should be stored in local memory. Constant memory allows fast access to read only data used by all threads. Texture memory provides fast access to read only data. Shared memory allows multiple threads to work on data together, potentially providing significant speedups.

GPUs perform best with a very large number of threads. Therefore any functions with a small number of independent tasks needs to be parallelized to allow multiple threads to complete each task. This can be done through using shared memory to allow loops to be computed in parallel. Additionally multiple tasks can be computed per thread block, further increasing the size of each thread block.

Algorithmic optimizations can provide significant speedups. GPUs perform best when each

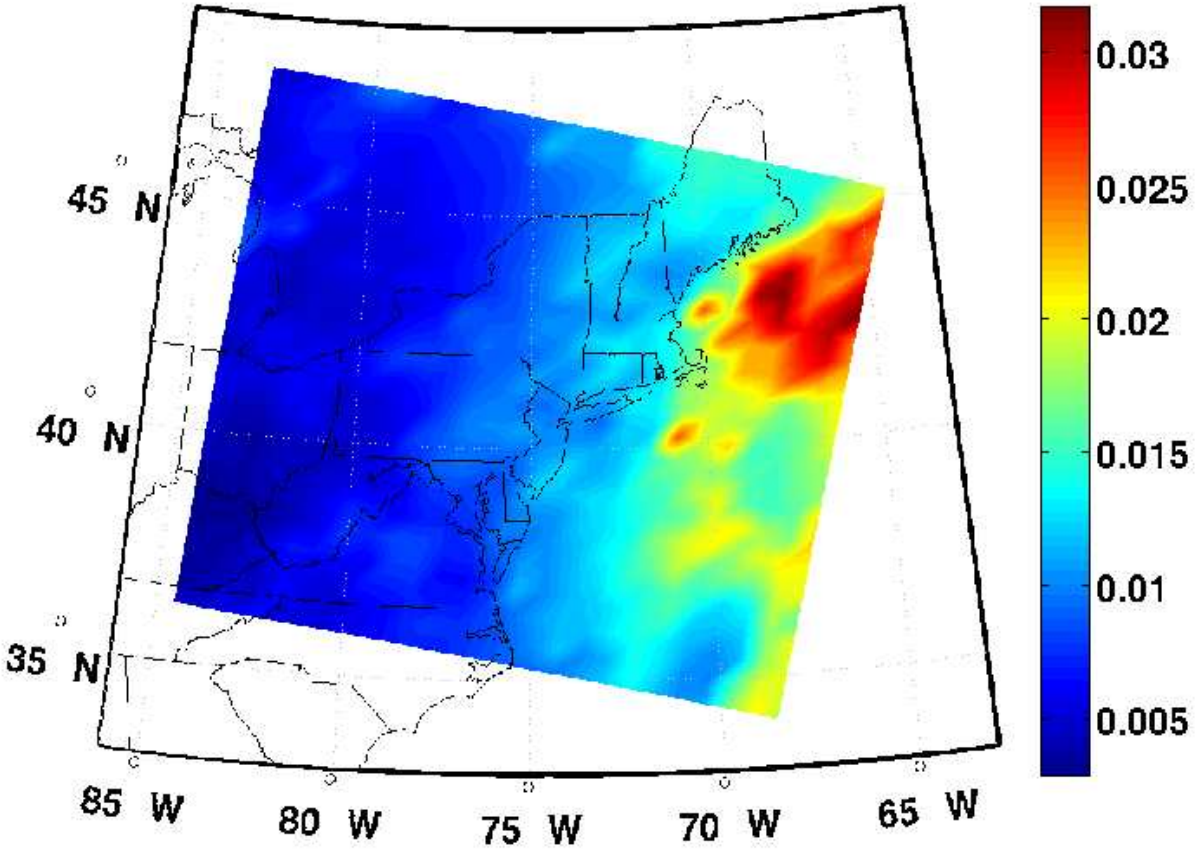


Figure 6.13: Average Rosenbrock Rodas4 produced O3 values for all levels (Scale 0 to 0.03).

thread loads a small amount of data and then performs the same work. Therefore we want to develop algorithms with a small memory footprint in order to reduce the amount of data that must be read and stored. We also want to develop simpler algorithms where each thread is doing the same computations on a balanced workload as much as possible.

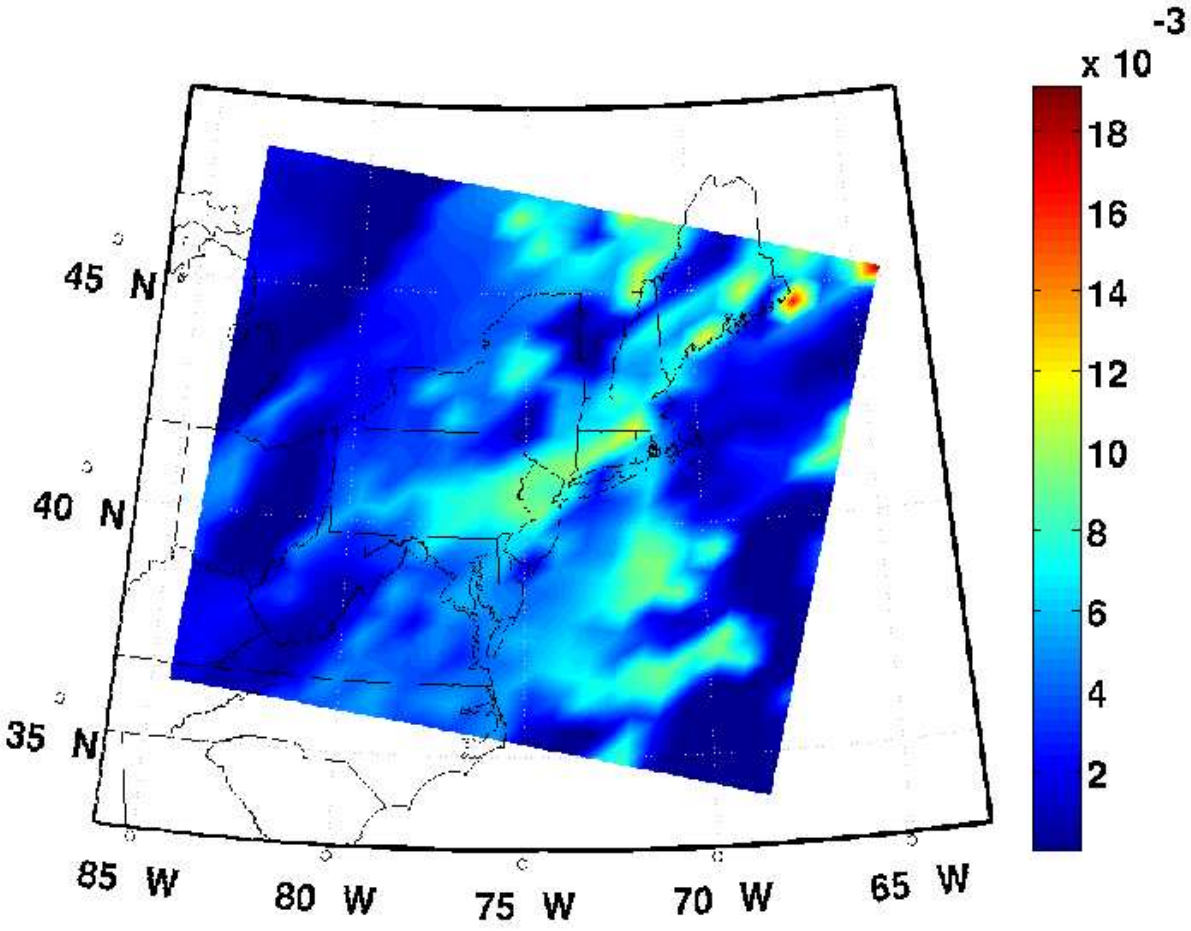


Figure 6.14: Average Rosenbrock Rodas4 produced O3 values for the ground level (Scale 0 to 18x10^{-3}).

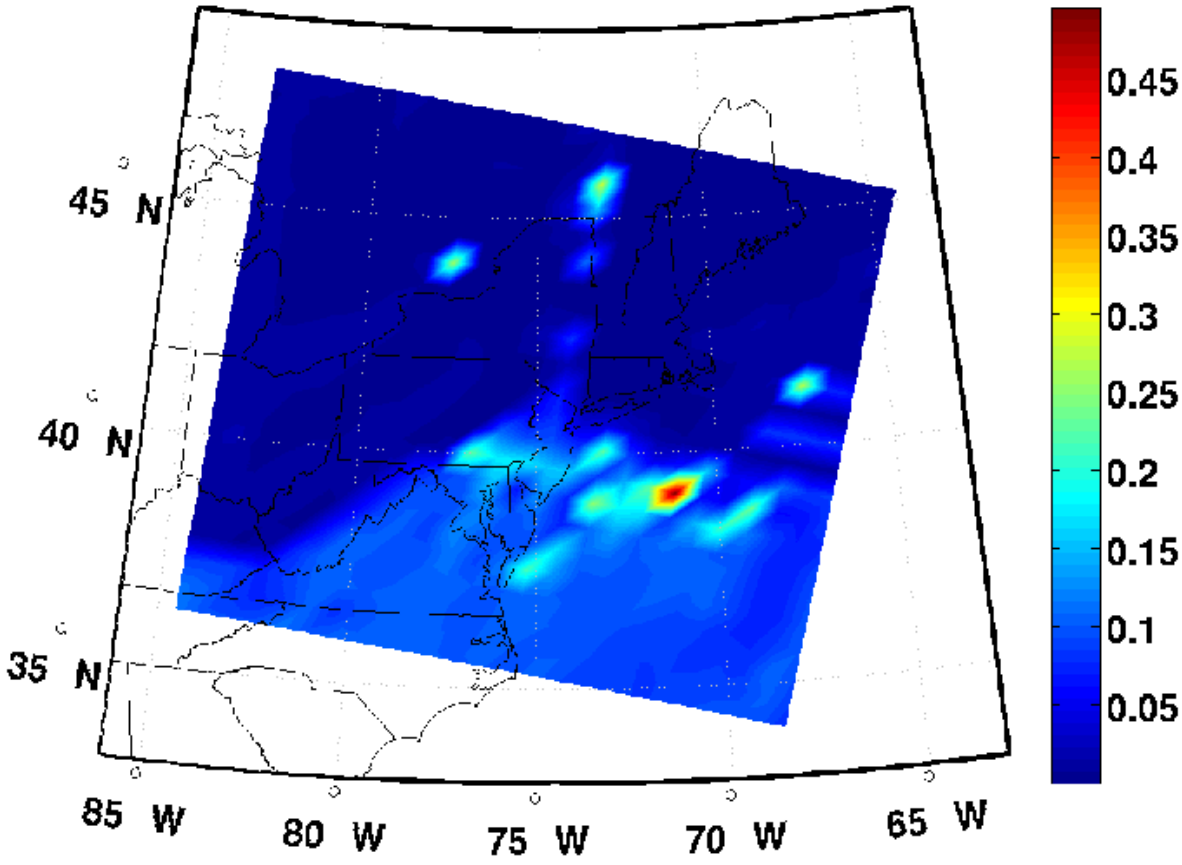


Figure 6.15: Average QSSA produced O3 values for all levels (Scale 0 to 0.45).

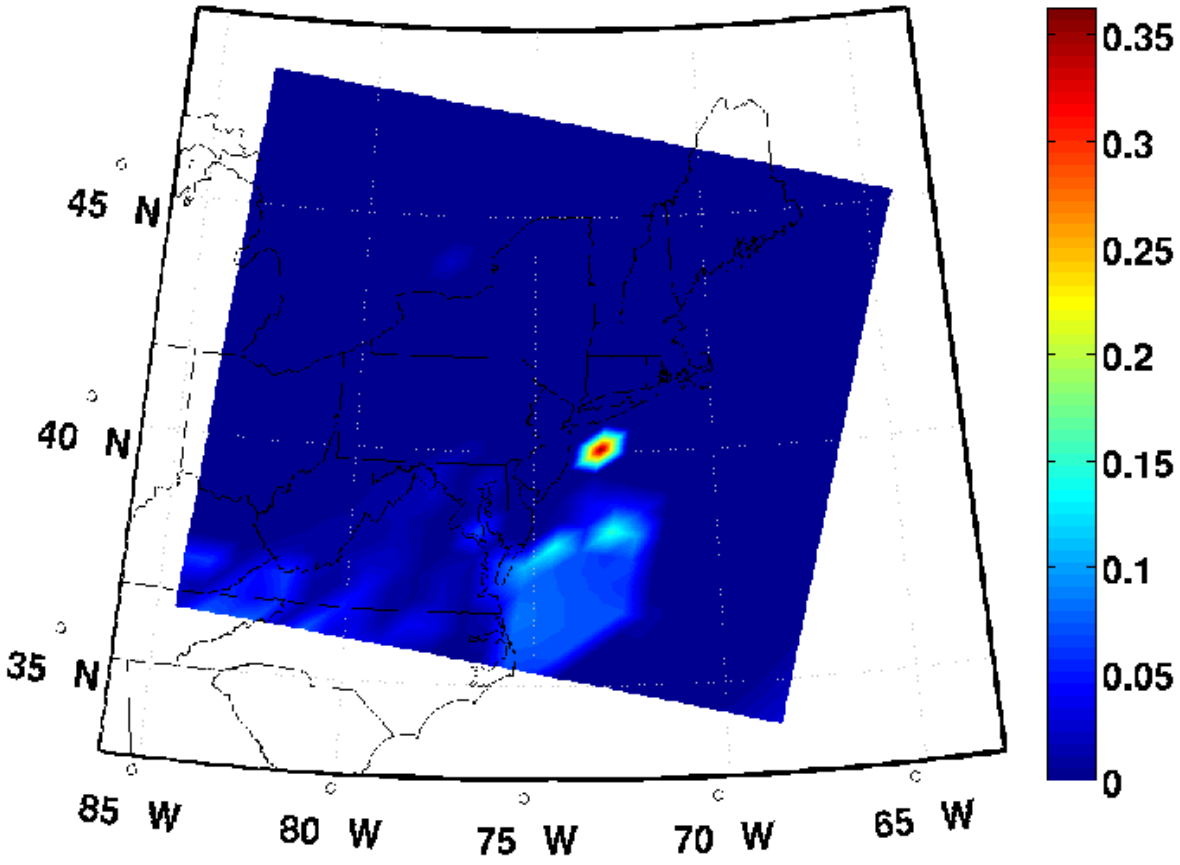


Figure 6.16: Average QSSA produced O3 values for the ground level (Scale 0 to 0.35).

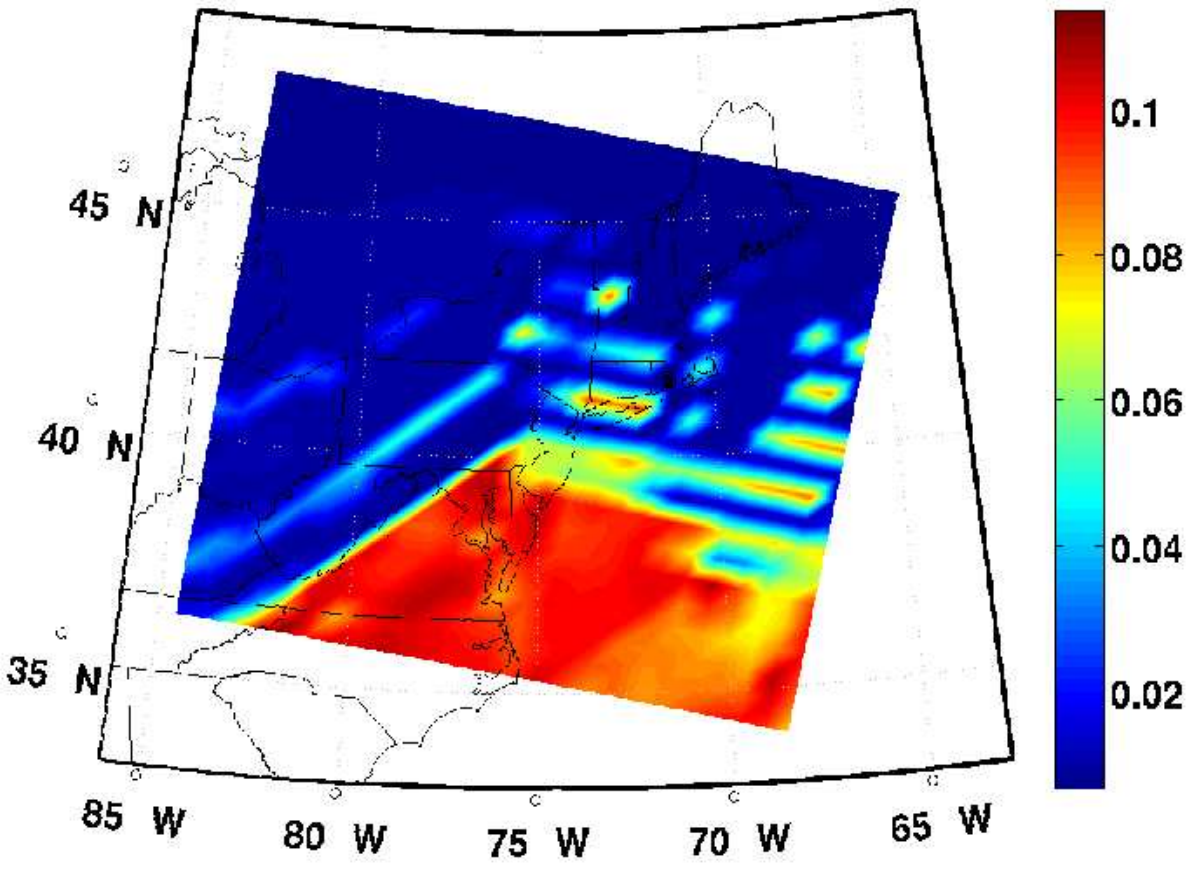


Figure 6.17: Average QSSA Exp2 produced O3 values for all levels (Scale 0 to 0.11).

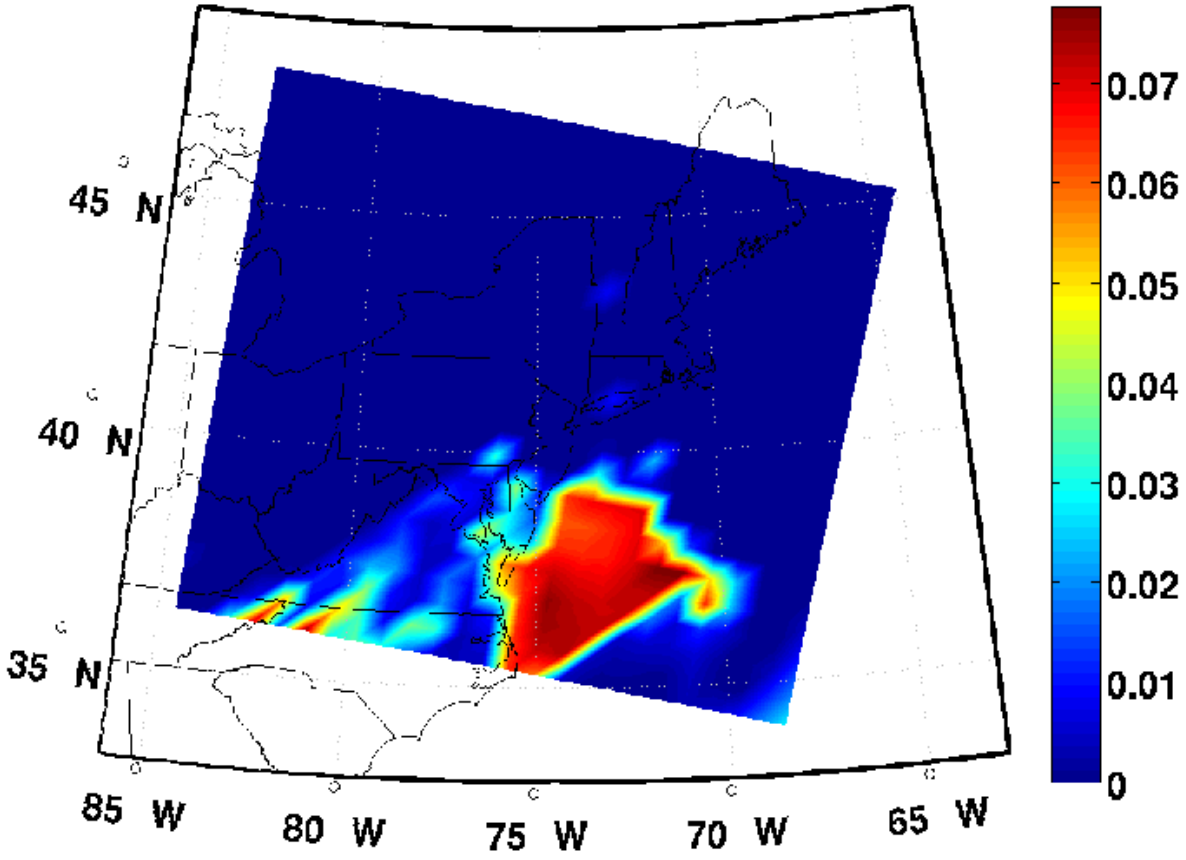


Figure 6.18: Average QSSA Exp2 produced O3 values for the ground level (Scale 0 to 0.07).

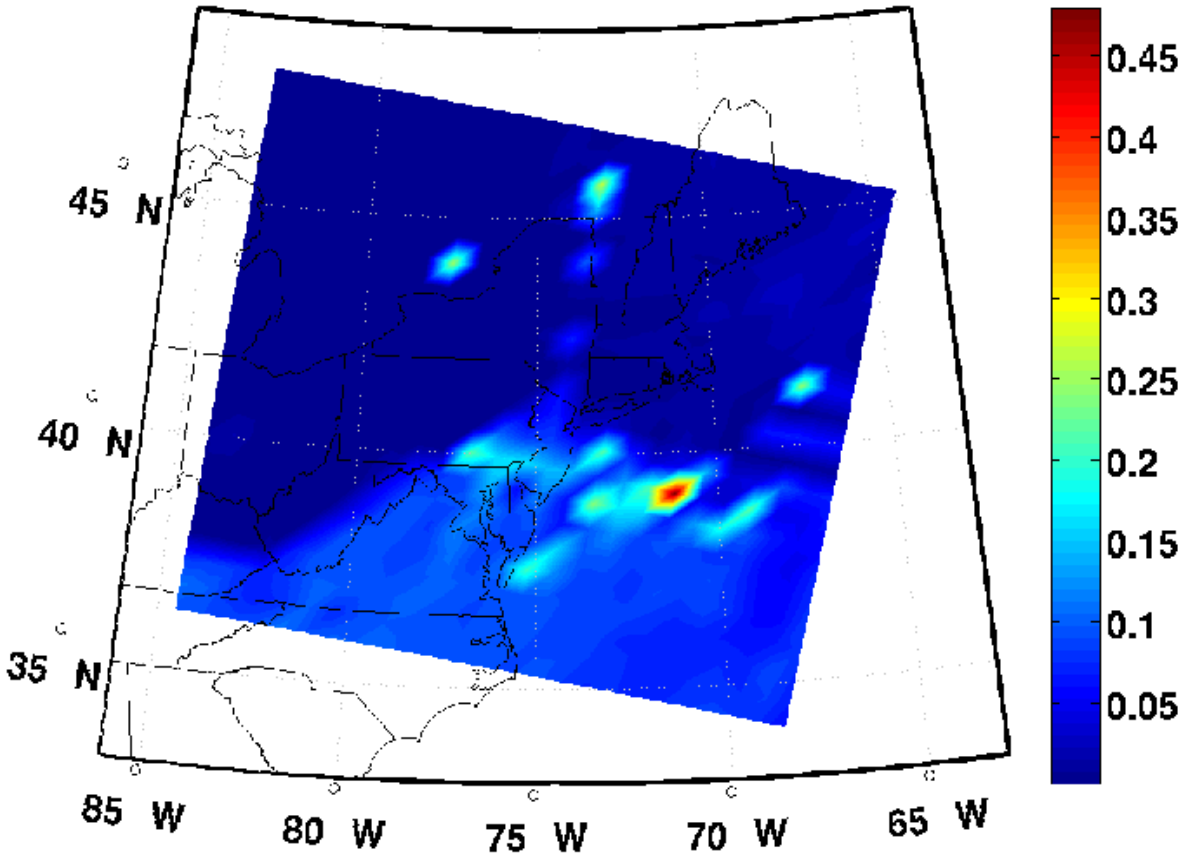


Figure 6.19: Difference between Rodas4 and QSSA produced O3 values for all levels (Scale 0 to 0.45).

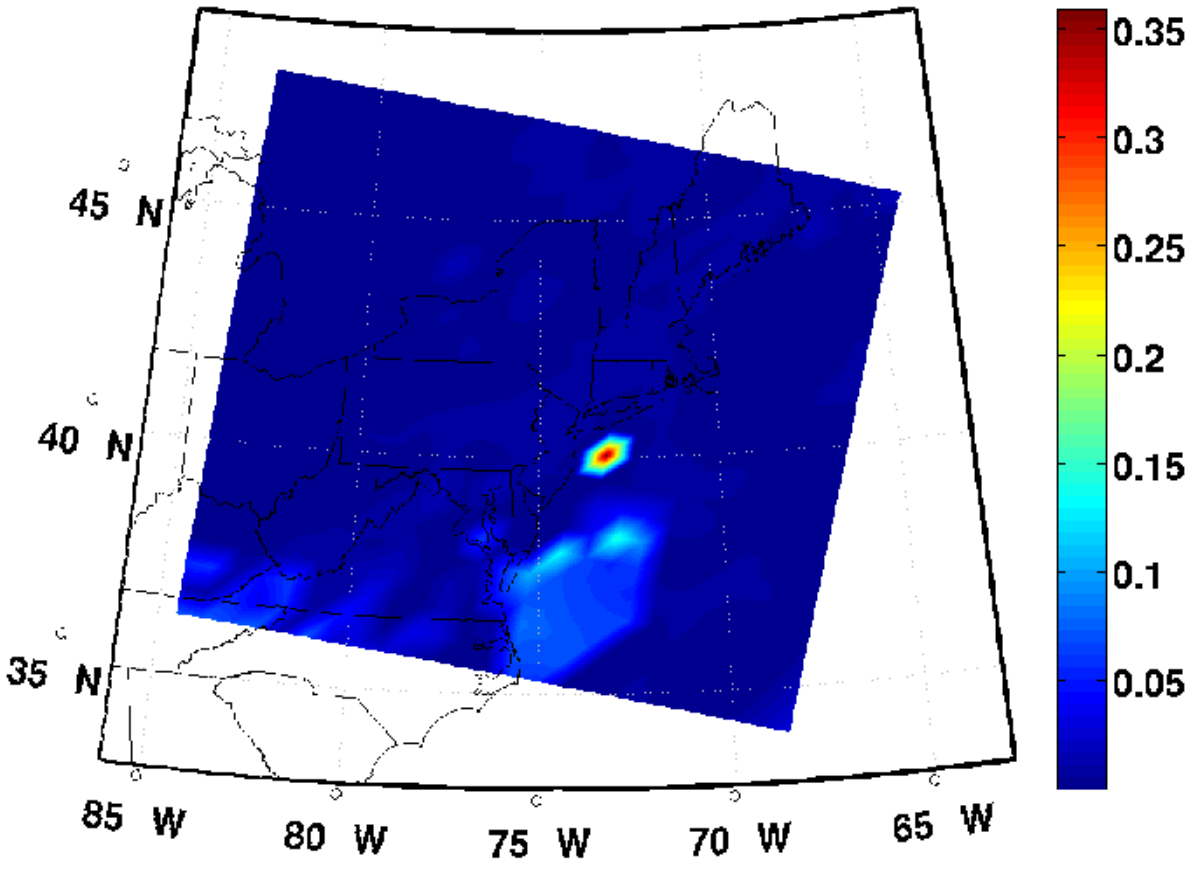


Figure 6.20: Difference between Rodas4 and QSSA produced O3 values for the ground level (Scale 0 to 0.35).

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This research has interfaced KPP chemistry with the GEOS-Chem chemical transport model, developed GPU versions of transport and chemistry code using STEM, and evaluated the speed and accuracy of each method.

Our results from interfacing KPP chemistry with the GEOS-Chem model demonstrate that the KPP solvers and the native GEOS-Chem solver (SMVGEARII) produce accurate results as demonstrated through visual examination as well as difference plots. The KPP Rodas3 and Rodas4 solvers achieve a similar level of accuracy at a lower computational expense than the SMVGEARII solver as demonstrated through the significant digits of accuracy plot. Additionally the tested KPP solvers have comparable scalability for parallel processing as the SMVGEARII solver.

Through developing transport and chemistry code for the GPU, we can conclude that GPUs provide significant potential speedup for chemical transport models, but that further GPU improvements are needed to allow these models to fully exploit the power of GPUs. The implicit and explicit transport routines were able to take advantage of shared memory and

a small memory requirement to achieve up to a 9.49 times speedup, but had a low GPU occupancy. The Rosenbrock chemical integrator achieved faster running times with a higher occupancy, but used a small number of registers and was unable to take advantage of shared memory, resulting in up to a 1.28 times speedup. The QSSA integrators were able to fit all data easily within local and shared memory to gain up to a 8.74 times speedup, but had a low occupancy.

Comparing the Rosenbrock methods with the QSSA methods demonstrates the key problem when developing GPU versions of chemical transport models, the tradeoff between placing more data in fast memories or achieving high occupancies. The Rosenbrock methods perform best when using as few registers as possible, resulting in higher occupancies. This allows memory transfer times to be hidden by executing more thread blocks. The QSSA methods perform best when using larger numbers of registers, resulting in very low occupancies. This allows each thread block to finish executing very quickly, although the GPU is not fully occupied.

When running the Rosenbrock and QSSA chemical integrators using OpenMP, we were able to achieve a near linear speedup, resulting in close to an 8 times speedup when using 8 processors. Using a larger shared memory system would likely produce much more than an 8 times speedup. This would likely allow the OpenMP Rosenbrock integrator to outperform the GPU QSSA integrators on larger shared memory systems. The implicit and explicit transport methods achieved very small speedups when using OpenMP. This is likely due to the very fast running times for the transport methods, as OpenMP performs best on longer loops such as those found in the chemical integrators. However, the transport running times are very small in comparison to the chemistry running times. Therefore using OpenMP for chemistry allow STEM to achieve higher speedups, even though the transport runs slightly slower. Therefore due to producing near linear speedups for STEM chemistry, we can conclude that OpenMP is a better option than GPUs for accelerating chemical transport models both quickly and accurately at this point in time. Future improvements to GPUs may allow them to produce larger speedups than OpenMP for chemical transport models.

This research has demonstrated that GPUs provide significant potential for speedups for chemical transport models, as the QSSA Exp2 integrator was able to achieve an 8.74 speedup while occupying only 13% of the GPU. However, to achieve an optimal speedup we will need more registers on GPUs so that enough thread blocks can execute at once to reach 100% occupancy. The transport methods demonstrated significant speedups through taking advantage of shared memory. However, these methods only achieved a 28% occupancy. Therefore we also need a larger amount of shared memory per thread block in order to allow enough thread blocks to execute at once to reach 100% occupancy. We will see the best speedups once we are able to fit a significant amount of data into the fastest memories while maintaining 100% occupancy on the GPU.

We have also seen that reducing algorithm complexity can allow codes to achieve higher speedups on the GPU. GPUs perform best when using a smaller memory footprint and executing the same code on each thread as much as possible. The simpler QSSA method provided larger speedups than the more complicated Rosenbrock methods, while the fixed step QSSA Exp2 method provided slightly larger speedups than the variable step QSSA method.

Looking at recently developed NVIDIA GPUs suggests that we will continue to see GPUs with more processors and more fast memory developed. Additionally features are being added to CUDA on a regular basis. This suggests that in the near future, we will see more powerful GPUs that are capable of accelerating chemical transport models to run much faster. Some companies are currently working on hybrids between CPUs and GPUs. NVIDIA is currently working on GT300 chips which will be a hybrid between a multi-core CPU and a GPU. This chip will have a multiple instruction multiple data design, improved double precision performance, and will likely have significantly more memory and registers. These hybrid processors may provide a better balance between access to fast memory and access to a larger number of processors at a reasonable price (Valich [40]).

7.2 Future Work

The next step is to implement KPP chemistry for larger chemical transport models such as GEOS-Chem for both the forward model and the adjoint model on the GPU. GEOS-Chem will use the same basic code as STEM for the chemistry, however the model is significantly larger, causing potential problems when trying to fit all data into memory on the GPU.

The effectiveness of QSSA methods in comparison to Rosenbrock methods has demonstrated that methods with a small memory footprint perform best on the GPU. However the accuracy plots demonstrate that QSSA methods can be very inaccurate at times. Therefore we need to develop more accurate methods with a small memory footprint to run on GPUs quickly and accurately.

Using GPUs for scientific computing is still a fairly new field. In the past, GPUs have primarily been used for displaying graphics, so there is significant room for improving GPU performance for scientific applications. Increasing memory sizes along with improved computing power will likely help chemical transport models fit all data into the faster memories and achieve a 100% occupancy. This work should be revisited in the future after more powerful GPUs such as NVIDIA's GT300 GPUs have been released, at which point we may be able to see significantly larger speedups, far beyond the speedups seen in this work.

Bibliography

[1] Brandvik, T. and Pullan, G.: Acceleration of a 3D Euler solver using commodity graphics hardware, 46th AIAA Aerospace Sciences Meeting and Exhibit, Jan. 2008.

[2] Carmichael, G.R., STEM-III: A Regional - Scale Analysis Tool, http://www.cgrer.uiowa.edu/people/carmichael/stem2_desc.html, 1999, Center for Global & Regional Environmental Research, June 2009.

[3] Carmichael, G.R., Chai, T., Sandu, A., Constantinescu, E.M., and Daescu, D.: Predicting Air Quality Improvements through Advanced Methods to Integrate Models and Measurements, J. Comp. Phys., 227, 3540-3571, 2008.

[4] CUDA Visual Profiler 1.1 Documentation (distributed with SDK 2.1). 2008.

[5] CUDA Zone, http://www.nvidia.com/object/cuda_home.html, nVidia, June 2009.

[6] Daescu, D., Sandu, A., and Carmichael, G.R.: Direct and Adjoint Sensitivity Analysis of Chemical Kinetic Systems with KPP: II - Validation and Numerical Experiments, Atmos. Environ., 37, 5097-5114, 2003.

[7] Damian, V., Sandu, A., Damian, M., Potra, F., and Carmichael, G.R.: The Kinetic PreProcessor KPP - A Software Environment for Solving Chemical Kinetics, Comp. and Chem. Eng., 26(11), 1567-1579.

[8] Eller, P., Singh, K., Sandu, A., Bowman, K., Henze, D. K., and Lee, M.: Implementation and evaluation of an array of chemical solvers in a global chemical transport model., Geosci. Model Dev. Discuss., 2, 185-207, 2009.

[9] Elsen, E., LeGresley, P., and Darve, E.: Large calculation of the flow over a hypersonic vehicle using a GPU, Journal of Computational Physics, 227(4):10148-10161, Dec. 2008.

[10] Errera, Q., Daerden, F., Charbrillat, S., Lambert, J.C., Lahoz, W.A., Viscardy, S., Bonjean, S., and Fonteyn, D.: 4D-Var Assimilation of MIPAS chemical observations: ozone and nitrogen dioxide analyses, Atmos. Chem. Phys. Discuss., 8, 8009-8057, 2008.

[11] Errera, Q., and Fonteyn, D: Four-dimensional variational chemical assimilation of CRISTA stratospheric measurements, J. Geos. Phys., 106(D11), 12253-12265, 2001.

[12] GEOS-Chem Model, http://www-as.harvard.edu/chemistry/trop/geos/, June 15, 2009, Harvard University Atmospheric Chemistry Modeling Group, June 2009.

[13] Goddeke, D., Buijssen, S.H.M., Wobker, H., and Turek, S: GPU Acceleration of an Unmodified Parallel Finite Element Navier-Stokes Solver, High Performance Computing and Simulation, 2009.

[14] Goddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Buijssen, S. H., Grajewski, M., and Turek, S.: Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. Parallel Computing, 33(1011):685699, 2007.

[15] Goddeke, D., Strzodka, R., and Turek, S.: Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations, International Journal of Parallel, Emergent and Distributed Systems, 22(4):221256, Aug. 2007.

[16] Hagen, T.R., Lie, K., and Natvig, J.: Solving the Euler equation on graphics processing units, ICCS 2006, volume 3994 of LNCS, 220-227, Springer, 2006.

[17] Hakami, A., Henze, D.K., Seinfeld, J.H., Singh, K., Sandu, A., Kim, S., Byun, D., and Li, Q.: The Adjoint of CMAQ, Environ. Sci. Technol., 41(22), 7807-7817, 2007.

[18] Henze, D.K., Hakami, A., and Seinfeld, J.H.: Development of the Adjoint of GEOS-Chem, Atmos. Chem. Phys., 7, 2413-2433, 2007.

[19] Henze, D. K., Seinfeld, J. H., Ng, N. L., Kroll, J. H., Fu, T.-M., Jacob, D. J., and Heald, C. L.: Global modeling of secondary organic aerosol formation from aromatic hydrocarbons: high- vs. low-yield pathways, Atmos. Chem. Phys., 8, 2405-2420, 2008.

[20] Hochbruck, M., Lubich, C., and Selhofer, H.: Exponential Integrators For Large Systems Of Differential Equations, SIAM J. Sci. Comput., 19, 1552-1574, 1998.

[21] Jacobson, M.Z.: Technical Note: Improvement of SMVGEAR II on Vector and Scalar Machines through Absolute Error Tolerance Control, Atmos. Environ., 32, 791-796, 1998.

[22] Jacobson, M.Z. and Turco, R.: SMVGEAR: A Sparse-Matrix, Vectorized Gear Code For Atmospheric Models, Atmos. Environ., 28, 273-284, 1994.

[23] Jay, L.O., Sandu, A., Potra, F.A., and Carmichael, G.R.: Improved QSSA methods for atmospheric chemistry integration, SIAM Journal on Scientific Computing, Vol. 18, Issue 1, p. 182-202, 1997.

[24] Kerkweg, A., Sander, R., Tost, H., Jockel, P., and Lelieveld, J.: Technical Note: Simulation of Detailed Aerosol Chemistry on the Global Scale using MECCA-AERO, Atmos. Chem. Phys. Discuss., 7, 3301-3331, 2007.

[25] Linford, J. and Sandu, A.: Optimizing large scale chemical transport models for multicore platforms, Proceedings of the 2008 Spring simulation multiconference, April 14-17, 2008, Ottawa, Canada.

[26] Linford, J. and Sandu, A.: Vector stream processing for effective application of heterogeneous parallelism, Proceedings of the 2009 ACM symposium on Applied Computing, March 08-12, 2009, Honolulu, Hawaii.

[27] Michalakes, J. and M. Vachharajani: GPU Acceleration of Numerical Weather Prediction, Parallel Processing Letters Vol. 18 No. 4., World Scientific., Dec. 531-548, 2008.

[28] Michalakes, J., Vachharajani, M., Linford, J., and Sandu, A.: GPU Acceleration of a Chemistry Kinetics Solver, http://www.mmm.ucar.edu/wrf/WG2/GPU/Chem.htm, May 29, 2009, National Center for Atmospheric Research, June 2009.

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

[30] NVIDIA CUDA Compute Unified Device Architecture: Reference Manual 2.0. 2008.

[31] Radhakrishnan, K. and Hindmarsh, A. C., "Description and Use of LSODE, the Livermore Solver for Ordinary Differential Equations," LLNL report UCRL-ID-113855, December 1993.

[32] Sandu, A., KPP - the Kinetic PreProcessor, http://people.cs.vt.edu/ãsandu/Software/Kpp/, June 2009.

[33] Sandu, A., Daescu, D., and Carmichael, G.R.: Direct and Adjoint Sensitivity Analysis of Chemical Kinetic Systems with KPP: I - Theory and Software Tools, Atmos. Environ., 37, 5083-5096, 2003.

[34] Sandu, A. and Sander, R.: Technical Note: Simulating Chemical Kintetic Systems in Fortran90 and Matlab with the Kinetic PreProcessor KPP-2.1, Atmos. Chem. Phys., 6, 1-9, 2006.

[35] Sandu, A., Verwer, J.G., Blom, J.G., Spee, E.J., Carmichael, G.R., and Potra, F.A.: Benchmarking stiff ODE solvers for atmospheric chemistry problems I: Implicit versus Explicit, Atmos. Environ., 31, 3151-3166, 1997a.

[36] Sandu, A., Verwer, J.G., Blom, J.G., Spee, E.J., Carmichael, G.R., and Potra, F.A.: Benchmarking stiff ODE solvers for atmospheric chemistry problems II: Rosenbrock methods, Atmos. Environ., 31, 3459-3472, 1997b.

[37] Stantchev, G., Juba, D., Dorland, W., and Varshney, A.: Using Graphics Processors for High-Performance Computation and Visualization of Plasma Turbulence, Computing in Science and Engineering, vol. 11, no. 2, pp. 52-59, Mar./Apr. 2009.

[38] Thibault, J., and Senocak, I.: CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows, 47th AIAA Aerospace Sciences Meeting, Orlando, FL, 2009.

[39] Tropospheric Ultraviolet and Visible (TUV) Radiation Model, http://cprm.acd.ucar.edu/Models/TUV/, National Center for Atmospheric Research, June 2009.

[40] Valich, Theo.: nVidia's GT300 specifications revealed - it's a cGPU!, http://www.brightsideofnews.com/news/2009/4/22/nvidias-gt300-specifications-revealed—its-a-cgpu!.aspx, April 22, 2009, Bright Side of News, June 2009.

Appendix A

GEOS-Chem KPP Test Setup

System: 8-core Intel Xeon x86_64 E5320 @ 1.86GHz

RAM: 8192MB

Cache: 4096KB per core

Operating System: Fedora Core 8, Kernel 2.6.26.8

Compiler: Intel Fortran Compiler fce 10.1.008

Compiler Options: -cpp -w -O2 -auto -noalign -convert big_endian -openmp -Dmultitask