Techniques for solving stiff chemical kinetics on GPUs

Christopher P. Stone
Computational Science and Engineering, LLC, Chicago, IL, 60622

Roger L. Davis†
University of California, Davis, Davis, CA, 95616

Balu Sekar‡
AFRL/RQTC, Wright-Patterson AFB, Dayton, OH, 45433

An implicit and explicit integration algorithm were ported to CUDA graphical processing units (GPUs) and used to concurrently solve sets of independent ordinary differential equations (ODEs) arising from finite-rate chemical kinetics. The GPU-enabled 4th-order accurate adaptive Runge-Kutta-Fehlberg (RKF45) ODE solver achieved a maximum speed-up of 20.2x over the baseline implicit 5th-order accurate DVODE CPU run-time for larger numbers of ODEs with comparable solution accuracy. The GPU implementation of the DVODE solver achieved a maximum speed-up of 7.7x over the baseline CPU run-time. The performance impact of mapping one thread to each ODE was compared to mapping an entire CUDA thread-block per ODE (i.e., multiple threads per ODE). The one-thread-per-ODE approach achieved greater overall speed-up compared to the one-block-per-ODE approach but only when the number of ODEs was large: 1,000 ODEs were needed just to break even with the scalar CPU version and over 50,000 ODEs to reach maximum parallel efficiency. The performance difference is most pronounced with the RKF45 algorithm. The peak performance with the one-thread-per-ODE method was nearly 2x faster than the one-block-per-ODE approach. The one-block-per-problem implementation of RKF45 and DVODE both achieved lower peak speed-ups but outperformed the scalar CPU performance with as few as 100 ODEs. The new GPU-enabled ODE solvers demonstrate a method to significantly reduce the computational cost of detailed finite-rate combustion simulations with turn-around cost savings exceeding an order of magnitude.

I. Introduction

The solution of finite-rate chemical kinetics problems in CFD combustion simulations can overwhelm the available computational resources. Finite-rate kinetics combustion simulations must solve $N_s$ additional species conservation equations on top of the traditional mass, momentum and energy conservation equations. Detailed chemical mechanisms can consist of 100’s of chemical species with 1000’s of elemental reactions leading to intractable storage and computational costs. The CFD algorithm can be simplified by solving the chemical mass action separately from the fluid transport. This approximation allows the chemical kinetics problem at each grid cell or vertex to be integrated independently over the specified time-step size ($\delta t$). At each element, the $N_s$ mass fractions and enthalpy (or temperature) are solved as a set of $N_s + 1$ stiff ODE’s. Even with the above simplification, the cost of solving the ODE systems can be prohibitive requiring massive parallel computing platforms. An alternative approach would be exploiting the available fine-grain parallelism using graphical processing units (GPUs).

GPUs are auxiliary devices installed in most workstations and are designed to offload the intensive computations involved in graphics and display rendering. These devices contain an immense amount of
raw computing power, more, in fact, than their CPU counterparts. Over the past several years, an intense effort has begun to exploit this available computing resource to solve computationally intensive, science and engineering problems. CPUs may contain, at most, 10’s of cores per package, high-end programmable GPUs commonly contain 100’s and can sustain substantially higher memory bandwidths. However, harnessing this power may require a redesign of many existing algorithms in order to efficiently fit into the massively parallel and multi-threaded environment of GPUs.

The current GPU application seeks to accelerate the chemical kinetics integration encountered within the counter-flow linear eddy model (CF-LEM). The CF-LEM is used to simulate the turbulent combustion found in opposed-flow configurations (e.g., augmenters). The combustion is modeled using finite-rate kinetics to capture complex reaction-diffusion processes such as extinction and re-ignition. The 3d turbulent fluid mechanics is modeled using a high-resolution 1d turbulent mixing model (i.e., the linear eddy model). In this fashion, the turbulent reaction zone is captured at direct numerical simulation (DNS) resolution. An example of a CF-LEM flame profile is shown in Fig. 1. There, the major fuel and oxidizer species are shown along with the temperature. Note that only the reaction zone is shown; the full CF-LEM domain is significantly larger. In this example, the combustion kinetics are modeled using a 19-species reduced $C_2H_4$ (ethene) mechanism. See Calhoon et al. for further details on the combustion model.

![Figure 1. An example LEM flame profile: only the reaction zone is shown and is resolved with 241 LEM cells.](image)

GPUs have recently been investigated for stiff chemical kinetics applications. Shi et al. studied the performance of the GPUs for detailed combustion kinetics. There, reaction mechanisms with over 1000 species were investigated. They reported speed-up only for mechanisms exceeding 300 species. Linford et al. investigated atmospheric chemistry problems on CUDA GPUs and Cell Broadband Engines (CBE). They found poor scalability on GPUs compared to CBE due to thread divergence and the large per-problem memory requirement. While similar, the present application differs significantly in several key aspects from these two previous investigations: mechanism size and the number of independent ODEs. As mentioned above, the chemistry at each LEM grid point can be integrated independently. The reaction mechanism of interest is also significantly smaller than those investigated by Shi et al. Therefore, a GPU method is needed that can rapidly integrate thousands of modestly-sized yet independent stiff ODEs. This readily exposes a baseline, coarse-grain level of parallelism that can be exploited.

The following sections provide details on (i) the mathematical basis of the combustion kinetics problem and the ODE integration algorithms, (ii) the parallel GPU implementation strategies and, finally, (iii) the performance of the GPU implementations.
II. Mathematical Model

The LEM combustion model integrates the $N_s$ species mass fractions ($y_k$) and temperature ($T$) equations in time ($t$) under the constant pressure assumption at every grid point. This can be expressed mathematically in the following set of coupled ODEs,

$$
\dot{y}_k = \frac{\dot{\omega}_k W_k}{\rho} 
$$

$$
\dot{T} = -\frac{1}{c_p} \sum_{i} h_i \dot{\omega}_k
$$

There $W_k$, $h_k$, and $\dot{\omega}_k$ are the molar mass, enthalpy and molar production rate for species $k$ and $c_p$ and $\rho$ are the specific heat at constant pressure and mass density of the system. The net molar production rate from combustion is a highly non-linear function of pressure $p$, $T$ and the species molar concentrations $c_i$. It is also the primary source of the ODE stiffness.

The reaction rate terms are computed from a reduced 19-species $C_2H_4$-Air (ethene) reaction mechanism with $N_k = 167$ reversible elemental reactions and $N_{qss} = 10$ quasi-steady-state (QSS) species (i.e., they are not solved in time). Each elemental reaction rate is calculated from a set of temperature and concentration-dependent factors. The temperature-dependent forward $k_f$ and reverse $k_b$ rate coefficients for each reaction are computed as

$$
k_f = A T^\alpha e^{-\frac{E_a}{RT}}
$$

$$
k_b = \frac{k_f}{e^{K_c}}
$$

where $E_a$ is the activation energy, $K_c$ is the equilibrium constant, $A$ is the pre-exponential rate constant and $\alpha$ is the temperature-dependency. Note that these constants are different for each elemental reaction. The equilibrium constant is computed as a function of the Gibbs energy ($\Delta g_k = \Delta h_k - T \Delta s_k$), the elemental reaction stoichiometric coefficients $\nu_i$, and the reference pressure $p_0$:

$$
K_c = \sum_{i} \nu_i (-\Delta G_i)/RT + \frac{p_0}{RT} \sum_{i} \nu_i
$$

where the summation is over the total number of products and reactants in the given elemental reaction.

The net molar rate-of-progress for each elemental reaction is computed from the above temperature-dependent rate coefficients and the concentrations as

$$
r_f = p_{FOCTB} \left[ k_f \prod_{i}^{N_p} c_i^{\nu_i} \right]
$$

$$
r_b = p_{FOCTB} \left[ k_b \prod_{i}^{N_r} c_i^{\nu_i} \right]
$$

There, $N_p$ and $N_r$ are the number of products and reactants for the given elemental reaction, $c_{TB}$ is the third-body concentration and $p_{FO}$ is the low-pressure factor (i.e., fall-off effect). Of the 167 reactions in the current mechanism, 24 involve third-body concentrations and 18 fall-off reactions factors. The third-body concentration effect for each relevant reaction is modeled as

$$
c_{TB} = \sum_{i}^{N_p} c_i + \sum_{i}^{N_{fall}} (A_i - 1) c_i
$$

The fall-off reaction rates account for low-pressure effects and are themselves functions of the third-body concentrations. Most of the current fall-off reactions follow the 4-parameter Troe form of the correction. These all require the calculation of an additional low-pressure forward Arrhenius rate coefficient $k_f^0$ along with a polynomial of temperature:
\[ p_f = \frac{k_f^0 c_{TB}}{k_f} \]

\[ f_c = (1 - P_4) e^{\frac{-p}{T}} + P_4 e^{\frac{-p}{T_5}} + e^{\frac{-p}{T_4}} \]

\[ p_{FO} = g(f_c) \frac{p_r}{1 + p_r} \]

The above sequence of rate calculations has neglected the \( N_{qss} \) quasi-steady-state species concentrations. These are computed as a complex non-linear combination of \( r_f \) and \( r_b \) only. The forward and reverse rates are then updated with the new QSS concentrations following the same form as Eq. 6.

Finally, the net molar production rate for species \( k \) is computed as the sum over all \( N_k \) elemental reactions

\[ \dot{\omega}_k = \sum_{i} \delta v_i r_i \]

where \( \delta v_i \) is the net stoichiometric coefficient for species \( k \) in reaction \( i \) with the net rate-of-progress \( r_i \) (i.e., \( r_i = r_{f_i} - r_{b_i} \)).

All temperature-dependent thermodynamic variables for the chemical species (e.g., \( c_p \), \( h \), and \( s \)) are computed from the NASA 7-term polynomial fit database, i.e.

\[ \frac{c_p}{R} = a_0 + a_1 T + a_2 T^2 + a_3 T^4 \]

\[ \frac{h}{R} = a_0 T + \frac{a_1 T^2}{2} + \frac{a_2 T^3}{3} + a_3 T^4 + a_4 T^5 + a_5 \]

\[ (h - s)/RT = a_0 (\log(T) - 1) + a_1 T + a_2 T^2 + a_3 T^3 + a_4 T^4 - \frac{a_5}{T} + a_6 \]

The parallel implementation of Eqs. 3 through 10 will strongly influence the computational performance of the ODE integration. Multiple sources of parallelism exist within the LEM application and the governing equations. Notably, each LEM cell represents an independent ODE that can be solved in parallel (i.e., coarse-grain parallelism). On a finer level, the forward and reverse elemental rate coefficients can be calculated in parallel for each individual ODE. The optimal balance between coarse- and fine-grain parallelism is strongly dependent upon the target hardware. Specific details on the implementation of the above RHS equations and the ODE solver implementations will be given later.

The mass fraction and temperature equations are easily combined into a single vector form suitable for solving with a generic ODE integration algorithm. The following form will be used throughout the remainder of this article:

\[ \dot{u}(t) = f(u(t), q) \]

\[ u(t_0) = u_0(q) \]

where \( f \) is the right-hand-side (RHS) vector function, \( u(t) \) is the vector of unknowns at some time \( t \), \( q \) is a vector of non-integrated parameters (e.g., pressure), \( \dot{u}(t) \) is the first derivative in time of \( u(t) \), \( u_0 \) is a vector of initial conditions, and \( t \) is the independent variable (i.e., time). The length of \( u \) is \( N_eq \) (i.e., \( N_s+1 \)) unknowns. In the current LEM combustion model context, the integration time in Eq. 11 is taken as the diffusion time-step size. That is, the ODEs at each grid point are independently integrated over the same time interval to maintain time-accuracy within the LEM model. Note that a different number of integration steps may be required for each different ODE problem. The ODE integration methods for Eq. 11 employed in this study are introduced below.

### III. Computational Approach

The DVODE package is commonly used to solve the stiff kinetics ODEs. This algorithm uses a 5th-order accurate implicit backwards difference formula (BDF) to integrate Eq. 11. DVODE is a legacy Fortran77 application and has no parallel computing capabilities. The CVODE ODE solver,\(^4\) part of the SUNDIALS non-linear solver package from Lawrence Livermore National Laboratory (LLNL), is an updated
and enhanced version of DVODE. CVODE is written in C and uses a modular object-oriented programming style. Both the C language and object-oriented design greatly simplify the implementation of CVODE in CUDA as detailed later. Note that CVODE provides all of the same functionality as DVODE and is, in fact, algorithmically equivalent to DVODE. That is, CVODE can be used as a one-to-one replacement for DVODE. With this in mind, CVODE, not DVODE, is used as the basis for the CUDA implementation.

In this study, the DVODE solver was used as the baseline for both solution accuracy and for serial CPU performance. The combustion kinetics is modeled using the reduced 19-species ethene mechanism. Thermochemistry properties are computed using the CANTERA computational library. CANTERA provides a large set of functions and databases useful for calculating thermodynamic state properties.

The dominant computational costs incurred at each ODE integration step with DVODE and CVODE are the RHS evaluation and the inversion of the Newton iteration matrix, a square \( N_{eq} \times N_{eq} \) matrix formed from the system Jacobian (i.e., \( \frac{\partial f}{\partial u} \)). The Jacobian isapproximated using finite differences as

\[
\frac{\partial f}{\partial u} \approx \frac{f(u + \epsilon) - f(u)}{\epsilon}
\]

where \( \epsilon \) is a vector of forward difference quotients dynamically determined each iteration. That is, the Jacobian approximation requires \( N_{eq} \) additional calls to the RHS function. The average total RHS cost per ODE (i.e., including the Jacobian estimation) is 83% of the run-time with the current ethene mechanism within DVODE.

A traditional CPU implementation of the above ODE equations would use a single threaded (i.e., sequential) algorithm and would implement the above ODE and right-hand-side (RHS) equations in a fashion that minimizes the number of operations. The minimum number of operations tends to equate to the minimum wall-clock time, the actual goal, with a sequential algorithm. Modern CPUs have large on-chip memory, high clock rates, and relatively large near-chip caches so this approach is generally optimal. Minimizing the total number of sequential operations does not necessarily equate to the minimum time on highly parallel hardware such as a GPU.

DVODE and CVODE contain many logical tests on convergence and refinement so as to minimize the number of necessary RHS evaluations and matrix computations. Many of these sequential optimizations come at the expense of extra storage requirements. This is optimal for CPUs with large amounts of on-chip memory and only a few cores. However, it can lead to poor parallel efficiency in a GPU environment where the computational cores are plentiful but the memory is scarce.

GPUs perform most efficiently when each thread executes the same instruction on different data. That is, they are optimized for single-instruction, multiple-data (SIMD) execution. Each system of ODEs in the kinetics application will have a unique set of initial conditions (ICs). Due to non-linearities in the kinetics, small perturbations in the ICs can cause each individual integration to progress differently through the implicit solvers (e.g., different rates of non-linear convergence and numbers of time-steps). This can lead to poor parallel efficiency and degraded performance in the GPU environment. An alternative strategy is to integrate the ODE problems using a high-order explicit algorithm such as an adaptive Runge-Kutta method. This approach is typically slower in a sequential environment compared to an implicit algorithm for stiff problems due to the necessarily smaller integration step sizes (i.e., more steps and more RHS evaluations). However, this type of algorithm requires minimal logic and may perform better in a parallel environment. Whether or not an explicit algorithm is numerically stable for the current application must be investigated.

### III.A. Explicit Runge-Kutta Method

An adaptive 4th-order accurate ODE solver has been created based on the Runge-Kutta-Fehlberg (RKF45) scheme. The RKF45 scheme has the useful property that both 4th and 5th-order accurate formulas can be constructed using the same six intermediate RHS evaluation points. The difference between these two schemes gives an order-of-magnitude estimate of the local truncation error (LTE). This, in turn, can be used to estimate the accuracy of a given solution. The integrated solution can then be monitored for acceptance and the step size \( h \) can be coarsened or refined based on the magnitude of the LTE. In this approach, a 4th-order accurate solution can be guaranteed.

The 4th and 5th-order accurate Runge-Kutta-Fehlberg formulas use the following six stages to advance the solution over the step size \( h \) from time \( n \) to \( n + 1 \):
\[ v^{n+1} = u^n + \frac{16}{256} k_1 + \frac{1438}{256} k_2 + \frac{2197}{256} k_3 + \frac{6656}{256} k_4 + \frac{9}{5} k_5 + \frac{2}{5} k_6 : 4^{th}\text{-order} \]
\[ u^{n+1} = u^n + \frac{16}{153} k_1 + \frac{12925}{256} k_2 + \frac{6656}{256} k_3 + \frac{1697}{56490} k_4 - \frac{1}{6} k_5 + \frac{9}{50} k_6 : 5^{th}\text{-order} \]  

where \( u \) is the unknown solution vector, \( \hat{u}_i \) and \( k_i \) are the intermediate solution and increments at stage \( i \), and \( v_i \) is the \( 4^{th}\)-order solution. The intermediate solution vectors are computed with the following coefficients and incremental times,

\[ \hat{u}_1 = h f(t, u^n) \]
\[ \hat{u}_2 = h f(t + \frac{1}{3} h, u^n + \frac{1}{2} k_1) \]
\[ \hat{u}_3 = h f(t + \frac{2}{3} h, u^n + \frac{3}{4} k_1 + \frac{6}{35} k_2) \]
\[ \hat{u}_4 = h f(t + \frac{8}{15} h, u^n + \frac{4381}{4680} k_1 - \frac{513}{1056} k_1 - \frac{27296}{4680} k_2 + \frac{2197}{1056} k_3 - \frac{845}{308} k_2 - \frac{1389}{308} k_3) \]
\[ \hat{u}_5 = h f(t + h, u^n + \frac{3680}{4680} k_1 - 8 k_2 + \frac{1389}{308} k_3 - \frac{845}{308} k_2 - \frac{1389}{308} k_3) \]
\[ \hat{u}_6 = h f(t + \frac{10}{11} h, u^n - \frac{8}{27} k_1 + \frac{2}{3} k_2 - \frac{4381}{274} k_3 + \frac{16}{7} k_4 - \frac{11}{27} k_5) \]

where \( f \) is the right-hand-side vector function.

The step size \( h \) can be adjusted during the course of the integration to insure the desired accuracy. A common approach with adaptive RK solvers is to advance the solution first from \( t = 0 \) to \( h \) in one full step and then again using two steps with step size \( h/2 \). The error estimation is then found from the difference between the \( h \) and \( h/2 \) solutions. The RKF45 method requires fewer RHS evaluations (approximately half) since the \( 4^{th}\) and \( 5^{th}\)-order accuracy schemes share the same RHS evaluation points. As a result, the overall cost of the RKF45 ODE solver is lower than other adaptive RK ODE solvers of comparable accuracy.

A solution acceptance criterion similar to that used by DVODE was implemented to ensure the RKF45 solutions match closely to the baseline ODE solver. Refinement is selected when the weighted root-mean-square (WRMS) truncation error is greater than 1. The WRMS error function is defined as

\[ WRMS = \frac{1}{N_{eq}} \left( \frac{|u^{n+1} - v^{n+1}|}{rtol |u^n| + atol} \right) \]

where \( rtol \) and \( atol \) are the user-specified relative and absolute tolerance parameters. Combining the relative and absolute tolerances allows the species and temperature to be compared more fairly even though their values differ by many orders of magnitude. When \( WRMS < 1 \), the solution is accepted, \( h \) is coarsened and the next integration may proceed. Otherwise, \( h \) is reduced and the integration step is repeated. In both cases, \( h \) is adjusted with the following formula:

\[ h_{new} = h \sqrt{\frac{1}{2} \frac{1}{WRMS}} \]  

The leading \( \frac{1}{2} \) factor in the adaption formula biases \( h \) towards a smaller value and avoids stagnation when \( WRMS \approx 1 \). We also limit the change in \( h \) to \( \frac{1}{2} < h_{new}/h < 4 \) to avoid large swings and potential overshoots. Finally, a ceiling is imposed upon \( h \) so that a minimum of 10 integration steps are taken.

The initial step size for RKF45 is estimated in a manner similar to DVODE as well. The initial step size \( h_0 \) is estimated such that the \( WRMS \) of the 2nd-order expansion term about \( u_0 \) is less than 1, i.e.,

\[ \frac{h_0^2}{2} \frac{\hat{u}(t)}{WRMS} \leq 1 \]

The first step in DVODE is limited to 1st-order accuracy so this estimation ensures the initial step does not produce excessive errors. This approach is likely overly conservative for RKF45 which is \( 4^{th}\)-order accurate for all time steps; however, this conservative approach can only benefit the solution accuracy.

**IV. GPU Implementations**

An efficient GPU implementation seeks to minimize the total run-time and often requires a different approach than would normally be used to produce an optimal sequential algorithm. For this work, the CVODE and RKF45 ODE solvers and the RHS reaction rate function were ported to run on NVIDIA CUDA GPUs. This section describes two different parallel strategies used to implement the ODE solvers and the
RHS function on these GPUs. However, before delving into the details of the parallel implementations, a brief overview of the CUDA GPU architecture and how it differs from common workstation-quality CPUs is necessary. The following overview is based on the NVIDIA Fermi M2050 CUDA GPU and the 8-core AMD Opteron 6134 Magny-cours CPU, both of which are used exclusively for benchmarking throughout this study.

IV.A. CUDA

The NVIDIA Fermi M2050 GPU has 14 streaming multi-processors (SMs) operating at 1.15 GHz. Each SM contains 32 scalar processors (SP) each capable of processing a single-precision floating-point operation per clock cycle; (two cycles are needed per double-precision operation). That is, each SM can execute 32 operations per cycle. By comparison, each of the AMD Operton 6134 Magny-cours’s 8 CPU cores operates at 2.3 GHz but is only capable of processing 4 single-precision operations per cycle using the SIMD processing unit. CPUs also have considerably larger on-chip memory (e.g., L1 and L2 on-chip caches). Therefore, the larger processor count (both SMs and SPs) of the CUDA hardware must be efficiently harnessed in order to overcome their significantly lower clock speeds and on-chip storage. Compounding this difficulty, any performance gains on the GPU must also overcome the overhead incurred when transmitting data from the host CPU to the GPU and vice-versa.

A CUDA code, much like a multi-threaded CPU application, defines the logical flow of a single thread of execution. CUDA threads are aggregated into thread-blocks where the user controls the number of threads per thread-block (e.g., 512 CUDA threads per thread-block). Each thread-block is mapped by the thread scheduler to an available SM. A key aspect of CUDA is that instructions are issued to thread warps, not individual threads. A warp is 32 threads and constitutes the finest level of practical parallelism. In this manner, each CUDA SM can be viewed as a SIMD processing unit capable of processing 32 elements per cycle. That is, all 32 threads within a warp can execute the same instruction on different data each clock cycle. Threads within the same warp are able to follow different logical paths in the code, that is, threads can diverge. Since only one instruction can be issued per warp per cycle, thread divergence can lead to extreme performance penalties. In the worst case scenario, each thread in a warp must execute a different instruction per cycle reducing the potential performance by a factor of 32x.

Two different parallel strategies were implemented that exploit the CUDA SIMD capabilities. In both methods, multiple ODEs are solved in concurrently on the GPU but they differ in how they map the ODEs to the available GPU hardware (i.e., the SMs). In the first method, one ODE is solved by one CUDA thread and is named the one-thread method. The second method solves each ODE using an entire thread-block and is named the one-block method. In the one-block method, all the threads within a thread-block work in concert to solve a single ODE. Also, the CUDA-based ODE solvers integrate the ODEs completely on the GPU from start to finish. An important design assumption in both methods is that the number of ODEs to be solved (Node) is large (i.e., Node \( \gg 1 \)) and that the number of unknowns per ODE is relatively small (i.e., Na is 19 in the current application). This differs considerably from the previous GPU combustion kinetics research of Shi et al.\(^2\) In their work, only one ODE system was solved at a time; Na was considerably larger ranging from 50 to well over 1000; and, lastly, the DVODE solver executed on the host CPU while only a select number of costly functions executed on the GPU.

In order for each CUDA SM to achieve its peak SIMD processing potential, data must be loaded efficiently. The aggregate peak bandwidth from GPU global-memory is considerably higher than commodity CPUs main-memory but this high bandwidth must be shared by 10’s of SMs and 100’s of SP concurrently. Recall that each SM can process 32 elements per cycle, 8x more than a common CPU, straining the available memory bandwidth. That is, each SM must load 8x more data to utilize all of the available computing hardware. CPUs hide slow main-memory operations with multiple levels of caching in per-core L1 and shared L2 caches of varying sizes and speeds. These data caches typically are hundreds of kilobytes (k) for L1 caches and several megabytes for L2 caches. CUDA has only rudimentary caching capabilities (512k L2 and between 16-48k L1 per SM) which can amplify the shared bandwidth issue. CUDA, however, provides several methods to overcome this bandwidth limitation: (i) coalesced reads/writes to global-memory, (ii) high-speed on-chip (i.e., local to each SM) shared-memory (shmem), and (iii) over-prescription of threads.

Coalesced reads and writes are highly optimized and are essential for efficient SIMD thread execution when operating on data stored in global-memory. A memory request is coalesced if all threads in a warp read from consecutive memory locations (i.e., contiguous elements in an array). Coalescing is the memory analog of a SIMD operation and the two must be used together to achieve peak performance. Coalescing is
best achieved with proper array formatting (i.e., dimension, rank and padding). Note that this optimization technique is not isolated to GPUs but is helpful in efficient SIMD execution on CPUs as well.

Over-prescription of threads is another critical mechanism used to hide the latency and bandwidth limits when using CUDA global-memory. CUDA Fermi allows up to 48 thread warps to run concurrently on each SM. An efficient hardware-based scheduler is used to swap out warps waiting for global-memory data with those that are ready to execute (i.e., each warps can efficiently swap context). For this reason, CUDA applications are encouraged to run 10,000’s of threads currently when using global-memory in order to hide the latency.

CUDA shmem is a small segment of high-speed memory local to each SM. All threads within the same thread-block share this small section of memory and, when combined with appropriate thread synchronization functions, can use it to execute fine-grain parallel algorithms. Note that CUDA shmem is distinct from global-memory. CUDA shmem is commonly used as a manual, user-defined high-speed data cache. That is, the user’s CUDA code (i) explicitly copies data from (slower) global-memory into (faster) shmem, (ii) performs many operations on the data in the shmem and (iii) copies the results stored in shmem back to global-memory to long-term storage keeping. For example, the $y_k$ array, $T$ and $p$ for a given ODE are needed frequently in the reaction rate calculations (i.e., the right-hand-side ODE function). This data is first copied from global-memory into shmem and then the RHS function is invoked using these shmem copies. The ODE state data is then local to the SM and can be fetched quickly reducing the global-memory overhead.

Over-prescription is not as critical in well-tuned shmem-based CUDA applications since data is local to the SM already. Efficient use of CUDA shmem is more nuanced than global-memory; however, good performance is generally obtained if the same rules for CUDA global-memory coalescence are followed. That is, each thread reads/writes to consecutive array elements. This guideline was used whenever possible. Note that 64k shmem on the CUDA Fermi GPUs is partitioned into two separate partitions 48k and 16k wide. Depending on the run-time configuration these can function as either an automatic L1 cache or the manual shmem as described above. For this reason, the user is limited to only 48k of manual shmem (with 16k of L1 cache) or vice-versa. The automatic L1 cache option should be enabled for all applications that do not use the manual shmem capabilities. The L1 cache can reduce latency and bandwidth demands in certain applications that use global-memory only.

### IV.B. Parallel Strategies

Now that the basic Fermi GPU hardware has been described, the differences in the two parallel implementations will now be addressed.

As mentioned above, one thread is assigned to each ODE in the one-thread implementation. To a large extent, each thread will execute the ODE solver algorithms and the RHS calculations (i.e., Eqs. 3-9) in sequential fashion. However, in the SIMD context, each thread warp can execute 32 different ODEs in parallel. Assuming the hardware scheduler affectively hides all latency, we expect to solve 448 ODEs concurrently (i.e., 14 SMs x 32 threads per warp) in this design. Of course, double-precision reduces this peak throughput (i.e., the number of ODEs that can be solved per unit time) by half and thread divergence will further degrade the potential parallel efficiency.

Each ODE requires a finite amount of storage the details of which shall be given in later sections. Depending on the CUDA-based ODE solver used, only on the order of 10 ODEs can be stored in the 48k available of CUDA shmem using the current 19-species ethene mechanism. This is obviously insufficient for the one-thread approach which requires at least 32 ODEs. Therefore, the one-thread approach does not use any manual shmem but global-memory exclusively. Also, the automatic 48k L1 cache configuration is enabled with the one-thread approach. Over-prescription is used to hide the global-memory latency and the data storage is designed for coalesced memory transactions. Note that effective over-prescription requires 10,000’s of ODEs to be solved concurrently.

The one-block approach attempts to exploit parallelism within each ODE instead of just solving many ODEs in parallel. CUDA supports up to 8 thread-blocks per SM which means there is sufficient shmem storage to hold the 8 ODEs per SM. Therefore, the one-block approach uses CUDA shmem extensively. Like the one-thread method, warp-level SIMD operations are still exploited but they are implemented in a very different manner. For example, Eq. 3 must be evaluated for all 167 forward reactions, i.e., there are 167 independent calculations. In the one-block approach with 32 threads per block, each thread will compute only 6 elements of the same $k_f$ array (i.e., ceiling(167/32)). In the one-thread method, each thread in a single warp computes all 167 elements of 32 different $k_f$ arrays concurrently.
A pseudo-code of the $k_f$ calculation using the one-thread and one-block approaches are shown in Algorithms 1 and 2, respectively. As can be seen in the one-thread approach pseudo-code, 32 different $k_f$ arrays are computed with a loop stride of 1. However, only one $k_f$ array is computed in the one-block version and the loop stride is set equal to the number of threads in the thread-block. Note that the $k_f$ array is defined as 2-d in the one-thread version but only 1-d in the one-block. This further demonstrates that multiple independent arrays, each of length 167, are computed by each thread in the one-thread approach but only 1 array is computed in the one-block method. The pseudo-codes also highlight the different memory layouts used to optimize the two different approaches. The inner dimension of $k_f$ (column-major format) equals the number of threads in the one-thread implementation. This ensures that the 32 array elements of $k_f$ saved each iteration are contiguous and can be coalesced into a single memory transaction. A similar scenario occurs in the one-block method: the 32 array elements saved each iteration are also contiguous ensuring fast storage to shared-memory.

**Algorithm 1** One-thread method: each thread executes the same loop on a different $k_f$ array.

```plaintext
nThreads := 32
nReactions := 167

dim kf[nThreads, nReactions]
dim A[nReactions]
dim Ea[nReactions]

tid := getThreadIndex()
i := 0
while i < nReactions do
    kf[tid, i] := A[i] \ast T^{\alpha[i]} \ast e^{-Ea[i]/R \ast T}
i := i + 1
end while
```

**Algorithm 2** One-block method: all threads operate on one $k_f$ array.

```
nThreads := 32
nReactions := 167

dim kf[nReactions]
dim A[nReactions]
dim Ea[nReactions]

tid := getThreadIndex()
i := tid
while i < nReactions do
    kf[i] := A[i] \ast T^{\alpha[i]} \ast e^{-Ea[i]/R \ast T}
i := i + nThreads
end while
```

Efficiently implementing Eq. 3 using either parallel strategy was straightforward since there was no data dependency. However, other equations such as 7 or 9 are more difficult since they require the summation over all species or all contributing elemental reactions, respectively. This type of array summation, generally termed a *reduction*, is required throughout the RHS and the ODE solvers themselves (e.g., vector dot-products and norms and matrix LU decompositions). The one-thread method requires no special attention since the parallelism comes from executing a sequential algorithm on multiple different arrays. There, a straightforward sequential implementation is sufficient. The difficulty only arises in the one-block approach. Here, we implement the array reduction operations (e.g., min, max, sum, etc.) using a parallel binary reduction algorithm. These algorithms are implemented using shared-memory and are capable of computing the sum of $N$ array elements in only $\log_2(N)$ steps as opposed to $N$ for a simple sequential summation.

The original ethene RHS function (Fortran) source code was fully in-lined. For example, the forward Arrhenius rate calculation (Eq. 3) used 167 different lines of code. There, certain combinations of coefficients were pre-computed (e.g., multiplied or exponentiated) and the results in-lined in the source code using fixed-precision syntax. This lead to very efficient CPU execution since it avoided many unnecessary transcendental function calls. However, this optimization strategy negates the SIMD nature of Eq. 3. That is, all 167 $k_f$ evaluations can use the same calculation with different coefficients. The original source code with pre-computed coefficients was factored back to a generic format to maximize SIMD execution. This allows efficient execution using either the one-thread or one-block strategy. However, the modified coefficients in the $k_f$ calculation incurred a level of round-off due to the finite precision of the original source code. The impact on the accuracy shall be reported later. Note that no other component of the CUDA implementation imposed a change in the numerical precision aside that incurred by different order of operations.

One component of the original ethene mechanism could not be re-factor to expose SIMD execution in the one-block method. Evaluating the QSS species concentrations involves a complex non-linear calculation that contains significant data dependency. This section of the RHS function is executed serially (i.e., by only one thread) in the one-block implementation. The serialization of the QSS calculation will degrade the overall parallel efficiency of the one-block method. Note that this does not impact the SIMD execution of
the one-thread method.

IV.C. Parallel ODE Solvers

Both the RKF45 and CVODE algorithms were implemented in CUDA following the one-thread and one-block strategies described above. Each thread in the one-thread CUDA implementations of RKF45 (CUDA-RKF45) and CVODE (CUDA-CVODE) integrates a separate ODE independently over its specified integration time. Note that each ODE problem may have different time integration limits. As discussed above, the one-thread approach takes a nominally sequential algorithm and solves it in a fully SIMD fashion. This approach is straightforward to implement from existing source code but the performance is entirely dependent upon the number of problems to be solved and the efficient coalescing of global memory operations.

Much of the original CVODE C source code from the SUNDIALS package was reused in the CUDA-CVODE implementation. All SUNDIALS solvers use an object-oriented approach for vector and matrix storage which greatly simplified the porting to CUDA C. The CUDA-RKF45 one-thread implementation relies almost entirely on the efficiency of the one-thread RHS function implementation. At the end of each explicit integration step, the WRMS truncation error is tested for suitability by each thread. If the solution is acceptable, the thread updates its solution vector and integration time. Thread divergence is incurred during the adaption test phase but should have only a minor performance impact compared to the cost of six RHS function evaluations each trial integration step. In both CUDA-RKF45 and CUDA-CVODE, a thread exits the integration loop once the final integration time is reached. Instructions are only issued per warp so all threads within a warp must complete their ODE integrations before proceeding to the next set of problems. Note that it is possible that a small number of ODEs may require significantly more integration steps causing most threads to stall leading to poor utilization of the CUDA processors. That is, integrating many independent ODEs constitutes an irregular workload.

A dynamic workload scheduler is implemented following the work of Tzeng et al.\(^8\) to optimize the throughput of the irregular workload in the one-thread approach. Work is scheduled in batches of 32 ODEs to match the warp size of the GPU. One thread-block with 512 threads per block is launched for each SM (i.e., 14 in total) and the threads persist until the queue is drained. A benefit of the warp-level queue is that explicit synchronization is required within each thread block. Note that all SMs are assigned at least one batch of 32 ODEs before additional work can be dequeued. This ensures that all parallel hardware is utilized and increases the throughput when the number of ODEs is small. A simple queue was implemented using a single global counter initialized to the number of ODEs to be solved. A free thread warp dequeues the next available batch of ODEs by (a) locking the global counter, (b) reading the batch number, (c) decrementing the counter and, finally, (d) releasing the global counter. These four steps were actually implemented by invoking the atomic decrement operation supported by CUDA. A global queue such as this can impose a significant bottleneck in the multi-threaded GPU environment;\(^8\) however, the high computational cost of the ODE integration easily dominates any minor overhead incurred.

The one-block CUDA-RKF45 implementation uses shared-memory for all vector storage. Explicit synchronization is enforced after each RHS function evaluation to avoid race conditions. After each integration step, the WRMS truncation error is computed using the parallel reduction algorithm. The optimal thread block size was found to be 64 for the CUDA-RKF45 one-block implementation.

CUDA-CVODE one-block follows the same approach as the one-block CUDA-RKF45 and RHS implementations. Again, the object-oriented structure of the CVODE vector and matrix operations simplifies the implementation effort. Each of the vector operations (e.g., dot product) is parallelized across the thread-block. Thread synchronization was critical in the one-block approach so explicit synchronization was enforced on entry and exit from each vector and matrix operation. It is important to note that the number of synchronization points in the CUDA-CVODE one-block implementation is significantly higher than that required for CUDA-RKF45 one-block. Parallel versions of the LU matrix factorization (with partial row pivoting) and back-solve routines were implemented for the CUDA-CVODE one-block algorithm. These two complex algorithms were implemented using only one thread warp and not the entire thread-block. Parallelism was only exploited in row or column (i.e., vector) operations so the maximum parallelism was only 20. As such, warp-level parallelism was sufficient. Further, the warp-level algorithm reduced the number of explicit synchronization points required during the matrix decomposition and back-solve and increased the overall performance.

The workload manager was modified to dispatch work to thread-blocks instead of to warps for the one-block algorithm. Currently, one ODE is dispatched to each thread-block per query instead of 32 per warp as
in the one-thread implementation. Each thread-block is assigned at least one ODE before additional work can be dequeued to maximize the utilization for small numbers of ODEs.

The CUDA versions of the ODE solvers each require a different amount of memory. The RHS function requires a fixed amount of array storage per-problem so each ODE solver must provide sufficient space for the RHS solution vector and scratch space. With double-precision, each RHS evaluation requires 3.2k. The one-block version requires additional shared-memory scratch space for the parallel reduction algorithm. This amounts to an extra 512 bytes for 64 threads per thread-block. Note that these storage requirements are only for array data and exclude local scalar variables.

The CUDA-RKF45 implementations require storage for 6 vectors of length \( N_{eq} \) (i.e., 5 intermediate solutions and 1 RHS vector) in addition to the RHS function storage for each ODE problem for a total of 4.6k of shared memory. Recall that each CUDA MP supports up to 48k of shared memory and 8 thread-blocks per MP. Therefore, there is sufficient shared memory storage for the maximum 8 blocks per SM.

The CUDA-CVODE implementations require significantly more space than the CUDA-RKF45 implementations. The vast majority of the extra storage is due to the Jacobian matrix with size \( N_{eq} \times N_{eq} \). The register usage is also at the maximum 63 per thread. CVODE (and DVODE) reuse previously generated Jacobian matrices if the solution vector has not deviated significantly. This requires that the Jacobian is copied and saved each instance doubling the already high storage cost. With this enabled, the CUDA-CVODE requires 11.2k per problem of which the two Jacobian matrices and its pivot array account for 57%. CUDA-CVODE one-block ideally uses shared memory exclusively. With the Jacobian-recycling strategy, there was sufficient shared-memory for only 4 blocks per SM. It was found that storing the saved Jacobian and Newton iteration matrix in global memory instead of shared-memory significantly increased the overall performance by doubling the number of possible thread blocks per SM to 8. The RHS function dominates the cost of the ODE solvers so any slowdown caused by the matrices in global memory appears to have had minimal impact.

The individual ODEs are solved in parallel within a single CUDA kernel. A CUDA kernel is initiated on the host device but executed on the GPU in parallel. Before the kernels can be executed, the initial conditions for the ODEs must be transferred from the host computer to the GPU device. And, the final integrated solution must then be transferred back to the host after the kernel is complete. The cost of these two transfers must be factored into the total GPU execution time since it represents a finite overhead.

V. Results and Discussion

The performance and accuracy of the new GPU-based RHS function and ODE solvers are reported here. All baseline benchmarks were run on an AFRL HPC DSRC Utility Server GPU node. Each GPU node has one NVIDIA Fermi M2050 GPU and two 8-core AMD Opteron 6134 Magny-Cours (2.3GHz) CPU’s. All source code was compiled with the GCC C++ compiler (v4.1.2), the PGI Fortran90 compiler (v11.10) and the NVIDIA CUDA C++ compiler (v4.1.28). It is important to note that all computations were conducted using double-precision exclusively. All reported wall-clock measurements were averaged over a minimum of 10 samples and the sample size was adjusted until the percent standard-deviation was less than 1%. Note that all baseline CPU benchmarks used only one CPU core (i.e., serial CPU execution) and all GPU benchmarks used only one GPU.

All benchmarks reported below used initial conditions (ICs) taken from a large CF-LEM database containing over 19 million individual samples. Recall that each grid-point in a CF-LEM domain constitutes an independent ODE system that must be integrated over a specified time-step size. For the CF-LEM application, the CF-LEM database provides the ICs (i.e., initial species concentration and temperature) along with the pressure and time-step for each ODE to be solved. A broad range of operating conditions (e.g., strain-rates and fuel-oxidizer ratios) is included in the database so that the ODE stiffness and integration times can vary widely for the individual ODEs.

V.A. RHS Function Performance and Accuracy

Parallel GPU implementations were required for both the RHS function and the ODE solvers themselves. In the current CF-LEM application, the computational cost of the ODE solvers is dominated by the cost of the RHS function evaluation. In this section, the parallel efficiency of the RHS function is examined independent of the ODE solvers.
The wall-clock time of the two CUDA RHS function implementations is shown in Fig. 2. The results are given as a function of the kernel size \( N \), i.e., number of ODEs evaluated in parallel within each kernel. The GPU speed-up is relative to the CANTERA library run-time executed sequentially on a single CPU core. In this test, the ODEs are not actually solved. Instead, the RHS function is called only once for each ODE. The RHS function has little conditional logic and is largely unaffected by variations in the ICs between the ODEs, i.e., there will be little thread divergence. Therefore, the following RHS performance results should be an indication of the maximum potential ODE solver speed-up since the RHS evaluation is the dominant cost and thread-divergence, a significant performance penalty, has been removed.

A maximum speed-up of 26.8x was obtained using the one-thread approach and 11.8x with the one-block algorithm. The one-block method is approximately 2x slower than the one-thread implementation for large \( N \). Recall that many RHS functions are evaluated in parallel so there is more parallelism for higher \( N \). The one-thread approach performs very efficiently under this scenario since thread divergence is minimal and there is sufficient parallelism (i.e., 10k's of threads) to hide the global-memory latency. However, for a small \( N \), the one-block strategy is far superior. For example, the one-block strategy is 4.3x faster than the one-thread method for \( N = 100 \). More importantly, the one-block algorithm is faster than the CANTERA CPU run-time at this size \( N \). The one-thread algorithm only becomes faster than the CPU baseline when \( N \) exceeds \( 10^5 \) whereas the one-block algorithm breaks even with as few as 100. The lower break-even point of the one-block approach must be balanced with the much faster potential throughput of one-thread when \( N \) is very large.

Calculations on the GPU must overcome the combined overhead of data transfer, memory allocation and kernel launch before the run-time savings can be realized. The combined overhead (i.e., all time but the kernel execution time) on the RHS function evaluation is also shown in Fig. 2. When compared to the one-thread run-time, the CUDA overhead accounts for 6% of the run-time for \( N = 1 \) and increases to over 17% for \( N = 10^6 \). The fact that the relative overhead increases with problem size is easily explained: the one-thread parallel efficiency increases with large \( N \) leading to a higher proportional overhead. Note that the same absolute overhead will be incurred by the full ODE solvers since the memory allocation and data transfer amounts are the same.

![Figure 2. Run-time scaling of baseline CANTERA and CUDA RHS functions. CUDA speed-up relative to the CANTERA CPU run-time is shown in the same scale with dashed lines and open symbols.](image)

The accuracy of the two CUDA RHS algorithms was also measured against the CANTERA baseline. The difference between the two CUDA RHS evaluations and the CANTERA library was computed over 50k randomly selected ODEs. The maximum difference over all the species and temperature was \( 3.7 \times 10^{-8} \)
measured in the $L_2$-norm. This difference is attributed entirely to modifications required to vectorize the forward Arrhenius rate coefficients as described previously. The impact of this numerical difference on the integrated solution will be shown shortly. Note that there is negligible difference between the one-thread and one-block versions. Any minor numerical difference between the two CUDA implementations is due to the one-block parallel reduction algorithm. The parallel reduction algorithm changes the order of operations which can change the final numerical value. However, the difference is only on the order of the double-precision round-off error, a value far below the measured numerical difference.

V.B. ODE Performance: CPU

The accuracy and stability of the new RKF45 ODE solver was assessed before it was accepted as a suitable alternative to the DVODE and CVODE algorithms. A serial CPU implementation of the RKF45 ODE solver was used to integrate 50k randomly selected ODEs from the CF-LEM database. The integrated solutions were compared to those from the DVODE baseline solver. The relative and absolute tolerances for both solvers were fixed at $10^{-11}$ and $10^{-13}$, respectively. The difference between the DVODE and RKF45 solutions for each component (i.e., species and temperature) is shown in Fig. 3. Both the $L_2$ and $L_\infty$ norms are shown and the values have been normalized by the square of the integrated DVODE values. The maximum normalized difference of any one component is on the order of $10^{-7}$ and the combined difference over all active species is $10^{-12}$. This small difference shows that the RKF45 algorithm is suitable for the current ethene combustion application. More importantly, the performance of the RKF45 algorithm can be directly compared to the baseline DVODE and GPU-based CVODE algorithms.

![Figure 3. Numerical difference between DVODE and RKF45 over 50k ODEs. Difference shown in the $L_2$ and $L_\infty$ norm.](image-url)

Figures 4 and 5 show the run-time and speed-up of the serial CPU and parallel GPU ODE solvers. As before, the speed-up is presented as a function of the number of ODEs solved per kernel $N$ and is relative to the serial CPU DVODE run-time. For the moment, focus only on the CPU DVODE and RKF45 performance. Some variation in cost with both solvers are observed for small $N$ due to differences in the ICs. These differences are largely averaged away after $10^3$ ODEs however. Beyond that point, both DVODE and RKF45 scale linearly with $N$. Overall, the DVODE run-time is approximately 50% faster despite taking 2x the number of integration steps on average. The cost saving in DVODE comes largely from the reduced number of RHS evaluations per integration step. For instance, DVODE required only 1.78 RHS evaluations per step compared to 6 for RKF45 for $N = 50k$. The number of failed integration steps in RKF45 was
minor at this point: only 5.6% of the integration steps failed forcing refinement. As noted previously, the sequential cost-saving measures taken by DVODE (e.g., Jacobian recycling) may prove counterproductive in the many-core GPU environment. However, the CUDA RKF45 implementations must overcome nearly a 50% performance penalty in order to break even with DVODE.

V.C. ODE Performance: GPU

The performance of the GPU implementations of the CVODE and RKF45 ODE solvers is now analyzed relative to the baseline DVODE and RKF45 solvers executed serially on the CPU. Referring again to Fig. 5, the CUDA-CVODE one-thread performance is seen to be many times slower than DVODE until \( N \) exceeds \( 10^3 \). After this break-even point, the speed-up with the CUDA-CVODE one-thread grows slowly, eventually reaching a steady 7.7x speed-up over the baseline DVODE CPU solver.

CUDA-RKF45 one-thread follows a similar scaling trend but is consistently 2.3x faster than CUDA-CVODE one-thread over the entire range of \( N \). The break-even point with the CUDA-RKF45 one-thread solver is between 100 and \( 10^3 \) ODEs; the speed-up is already 2.4x at only \( 10^3 \) ODEs. The maximum speed-up for the CUDA-RKF45 one-thread solver is 20.2x over the serial DVODE run-time. This 20.2x speed-up matches closely to the CUDA RHS speed-up previously reported suggesting that the RHS function is the limiting factor in the throughput using the CUDA-RKF45 one-thread method. Note that CUDA-RKF45 one-thread is 28.6x faster than the serial CPU implementation of RKF45.

Both one-thread versions of CUDA-CVODE and CUDA-RKF45 suffer from poor performance when \( N \) is small. This is consistent with the one-thread RHS-only results shown above in Fig. 2. The CUDA-RKF45 one-block break-even point is only slightly greater than 10 ODEs; far sooner than either one-thread ODE implementation and also sooner than was observed for the one-block RHS-only results. CUDA-RKF45 one-block quickly reaches a maximum speed-up of 10.7x speed-up relative to DVODE at \( N \approx 10^4 \). Again, the CUDA-RKF45 one-block maximum speed-up matches closely to the one-block RHS implementation (i.e., approximately 11x speed-up) clarifying that the RHS function is the limiting factor for both CUDA-RKF45 implementations. The CUDA-CVODE one-block performance is nearly identical to CUDA-RKF45 one-block for small \( N \) but achieves only a 7.3x speed-up for large \( N \).

The relative overhead cost can be inferred by referring back to Fig. 2. The absolute overhead for the ODE solvers is the same as the RHS-only performance test. Recall that the RHS function must be called at least 60 times (i.e., minimum of 10 time-steps) by the RKF45 ODE solver. There are similar lower limits for the CVODE as well. This effectively amortizes the overhead reported in Fig. 2 over many more RHS function evaluations and reduces the relative overhead. The CUDA-CVODE one-block overhead accounts for only 1.5% of the total run-time when \( N \) is less than 100 and quickly drops below 0.1% for large \( N \). Obviously, the data transfer and memory allocation overhead has little impact on the peak CUDA ODE solver performance.

The above benchmarks showed the performance of the various ODE solvers on a database of ICs taken from actual LEM simulations. In these simulations, 100’s of LEM cells are used to discretize the LEM computational domain. Many different LEM simulations are therefore solved concurrently when the number of ODEs (i.e., \( N \)) is much greater than \( 10^3 \). Recall that the LEM can be viewed as a 1-d DNS method and, therefore, the concentration profiles and temperature should vary smoothly throughout the domain. For example, Fig. 1 showed a non-premixed reaction zone simulation with 241 LEM cells. Since the profiles vary smoothly, neighboring LEM cells within each simulation can be expected to have similar, yet distinct, ICs and, presumably, final integrated ODE solutions. By extension, it is inferred then that their integration cost (e.g., number of steps and RHS evaluations) should be similar. In the one-thread parallel processing strategy, this neighboring cell similarity should translate into low thread-divergence since they are likely to be placed in the same thread warp. However, the contrary is also expected: dissimilar ICs within each warp may increase thread divergence and degrade performance. Note that this effect is only expected for the one-thread implementations; the one-block strategy was designed explicitly to minimize this effect. Two additional benchmarks were conducted to measure the impact of this LEM cell similarity on the CUDA ODE solver performance.

The same ODEs were solved as above in Fig. 5; however, the ordering of the ODEs was randomized (i.e., shuffled) in order to remove the implicit similarity between the LEM cells assigned to a given warp. Note that for each \( N \), the same ODEs were solved so that the sequential CPU cost is unchanged. The maximum speed-up for the CUDA-RKF45 one-thread algorithm dropped from 20.2 down to only 13.2 with the randomized database, a decrease of 35%. The CUDA-CVODE one-thread peak performance declined.
Figure 4. Total run-time (sec) of the baseline CPU DVODE and RKF45 solvers and their CUDA counterparts.

Figure 5. Speed-up of RKF45 and the CUDA ODE solvers relative to the baseline CPU DVODE run-time.
only 13%, from 7.7 down to 6.7 relative to the DVODE baseline. The CUDA-CVODE solver incurred a lower penalty by the randomization and the resulting increased thread-divergence. This implies that the thread-divergence within the CUDA-CVODE one-thread implementation is already higher than its RKF45 counterpart. Note that neither one-block implementation was effected by the randomization since each ODE is treated independently. That is, the similarity – or lack thereof – between the ICs only impacts the one-thread efficiency.

Randomizing the LEM database represented a worst-case scenario. The final test conducted examined the opposite best-case scenario: perfectly similarity between the ICs. For this test, the same ODE is solved repeatedly. This synthetic scenario should give the optimal throughput for the one-thread method since full SIMD vectorization is essentially guaranteed for all ODEs. In fact, the throughput should mimic the RHS-only tests above. A single IC was selected from the database that required a similar number of DVODE integration steps and RHS evaluations as the full database on average. In this scenario, the CUDA-CVODE one-thread achieved a maximum speed-up of 29.2x over the DVODE CPU baseline. The peak performance of CUDA-RKF45 one-thread was 20.4 relative to DVODE, significantly lower than observed above in comparison to the CUDA-CVODE performance. However, the RKF45 CPU run-time for this problem is 2.25x higher than the DVODE CPU, much higher than the 50% observed over the entire LEM database on average. The CUDA-RKF45 performance relative to the serial RKF45 CPU run-time shows a much different picture. CUDA-RKF45 one-thread achieves a peak speed-up of 46.2x relative to the RKF45 CPU run-time. Again, neither one-block implementation was affected.

VI. Conclusions

This research effort has investigated the performance and accuracy of solving stiff chemical kinetics ODEs on CUDA GPUs. The mathematical basis of the combustion kinetics ODE formulation and how it impacts a parallel implementation was reviewed. Two different GPU implementations of the RHS function (i.e., the reaction mechanism) and the two ODE solvers were reviewed in detail.

The two CUDA ODE solvers, CUDA-CVODE and CUDA-RKF45, both achieved significantly speed-up compared to the baseline serial DVODE CPU solver. The one-thread-per-ODE implementation of the CUDA-RKF45 ODE solver achieved over 14x speed-up compared to DVODE with only 10k ODEs. Speed-ups of 20.2x were observed when the number of ODEs exceeded for 50k. The one-thread version of CUDA-CVODE achieved lower performance speed-up, only 7.7x, compared to DVODE but exhibited a similar scaling trend. The one-thread-block-per-ODE CUDA implementations of both ODE solvers achieved lower maximum speed-ups compared to their one-thread counterparts. However, both one-thread-block-per-ODE implementations outperformed DVODE with as few as 100 ODEs and achieved peak performance with only $10^3$ ODEs, far sooner than the one-thread-per-ODE methods.

The typical LEM simulation uses hundreds of cells to resolve the reaction zone. The one-thread-block-per-ODE CUDA ODE solvers, either DVODE or RKF45, offer a potential speed-up of between 4 and 7x compared to the baseline serial DVODE if only one LEM domain is solved at a time. Far greater speed-up is possible if multiple LEM simulations are solved concurrently. In this manner, the total number of ODEs solved on the GPU can be increased offering speed-ups in excess of 20x compared to DVODE.

It is interesting to note that the slower RKF45 algorithm outperformed the CPU-optimized DVODE and CVODE algorithms when implemented in the massively parallel GPU environment. The simplified structure of the RKF45 algorithm provided greater parallel efficiency easily overcoming the lower sequential performance. The maximum throughput of the CUDA-RKF45 algorithm is approximately 2x faster than the CUDA-CVODE peak. Also, CUDA-RKF45 reached a speed-up of 28.6x compared to the RKF45 CPU implementation.

The best and worst-case performance with the one-threaded CUDA algorithms was also examined. The worst-case scenario was modeled by randomizing the original LEM database. Adjacent LEM cells tend to have similar ODE integration costs. Placing these neighboring ODE problems within the same warp was expected to minimize thread divergence and low processor utilization. This lead to a 35% performance drop for CUDA-RKF45 but only a 13% drop for CUDA-CVODE. The best-case scenario was estimated by forcing thread divergence to zero with a synthetic database. This was approximated by having all threads solve the same ODE problem. Comparing the original results to the best and worst-case scenarios, it is estimated that CUDA-RKF45 one-thread is achieving 62% of the peak performance while CUDA-CVODE one-thread only 26%. From this we can conclude that the CUDA-CVODE algorithm is highly susceptible to thread
divergence caused by differences in problems within a single warp.

In summary, the potential of GPUs to accelerate the stiff ODE integration was clearly demonstrated. The results presented here were taken from only one reduced $C_2H_4$ mechanism. However, these results should be applicable to a broad range of reaction mechanisms of similar size and complexity.

Acknowledgement of Support and Disclaimer

Funding for this work was provided by the DoD HPCMP’s User Productivity Enhancement, Technology Transfer, and Training (PETTTT) Program (Contract No: GS04T09DBC0017 through High Performance Technologies, Inc.) under project PP-CFD-KY02-115-P3 titled Implementing stiff chemical kinetic ODE solvers on GPUs using CUDA. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of U.S. Air Force Research Laboratory or the U.S. Government.

The authors would like to thank William Calhoon (CRAFT Tech, Inc.) for his assistance with the baseline DVODE solver and providing the LEM database.

References

5. code.google.com/p/cantera.