An investigation of GPU-based stiff chemical kinetics integration methods

Nicholas J. Curtis[a,*], Kyle E. Niemeyer[b], Chih-Jen Sung[a]

[a] *Department of Mechanical Engineering*
University of Connecticut, Storrs, CT 06269, USA
[b] *School of Mechanical, Industrial, and Manufacturing Engineering*
Oregon State University, Corvallis, OR 97331, USA

Abstract

A fifth-order implicit Runge–Kutta method and two fourth-order exponential integration methods equipped with Krylov subspace approximations were implemented for the GPU and paired with the analytical chemical kinetic Jacobian software `pyJac`. The performance of each algorithm was evaluated by integrating thermochemical state data sampled from stochastic partially stirred reactor simulations and compared with the commonly used CPU-based implicit integrator `CVODE`. We estimated that the implicit Runge–Kutta method running on a single Tesla C2075 GPU is equivalent to `CVODE` running on 12–38 Intel Xeon E5-4640 v2 CPU cores for integration of a single global integration time step of 10^{-6} s with hydrogen and methane kinetic models. In the stiffest case studied—the methane model with a global integration time step of 10^{-4} s—thread divergence and higher memory traffic significantly decreased GPU performance to the equivalent of `CVODE` running on approximately three CPU cores. The exponential integration algorithms performed more slowly than the implicit integrators on both the CPU and GPU. Thread divergence and memory traffic were identified as the main limiters of GPU integrator performance, and techniques to mitigate these issues were discussed. Use of a finite-difference Jacobian on the GPU—in place of the analytical Jacobian provided by `pyJac`—greatly decreased integrator performance due to thread divergence, resulting in maximum slowdowns of 7.11–240.96 ×; in comparison, the corresponding slowdowns on the CPU were just 1.39–2.61 ×, underscoring the importance of use of an analytical Jacobian for efficient GPU integration. Finally, future research directions for working towards enabling realistic chemistry in reactive-flow simulations via GPU/SIMT accelerated stiff chemical kinetics integration were identified.

Keywords: Chemical kinetics, Stiff chemistry, Integration algorithms, GPU, SIMT

*Corresponding author
Email address: `nicholas.curtis@uconn.edu` (Nicholas J. Curtis)

1. Introduction

The need for accurate chemical kinetic models in predictive reactive-flow simulations has driven the development of detailed oxidation models for hydrocarbon fuels relevant to transportation and energy generation applications. At the same time, growing understanding of hydrocarbon oxidation processes resulted in orders of magnitude increases in model size and complexity. Contemporary detailed chemical kinetic models relevant to jet fuel [1], diesel [2], gasoline [3], and biodiesel [4] surrogates consist of hundreds to thousands of species with potentially tens of thousands of reactions. Furthermore, kinetic models for large hydrocarbon fuels tend to exhibit high stiffness that requires implicit integration algorithms for practical solution.

Reactive-flow modeling codes commonly rely on high-order implicit integration techniques to solve the stiff governing equations posed by chemical kinetic models. The cost of these algorithms scales at best quadratically—and at worst cubically—with the number of species in a model [5], due to repeated evaluation and factorization of the chemical kinetic Jacobian matrix to solve the associated nonlinear algebraic equations through iterative solutions of linear systems of equations. Several recent studies [6–8] demonstrated that using even modestly sized chemical kinetic models can incur severe computation cost for realistic reactive-flow simulations. For example, a single high-resolution Large Eddy Simulation (LES) realization of a diesel spray—using up to 22 million grid cells with a 54-species *n*-dodecane model—for 2 ms after start of injection with the common implicit CVODE solver [9] took 48,000 CPU core hours and up to 20 days of wall clock time [8]. Lu and Law [5] extensively reviewed techniques for reducing the cost of using detailed chemical kinetic models; however, significant cost savings can be realized by using an analytical Jacobian formulation, rather than the typical evaluation via finite difference approximations. This analytical Jacobian approach eliminates numerous chemical source term evaluations, and for a sparse Jacobian (e.g., formulated in terms of species concentrations) the cost of evaluation can drop to a linear dependence on the number of species in the model [5].

In this work, our efforts to accelerate simulations with chemical kinetics focus on improving the integration strategy itself, by developing new algorithms for high-performance hardware accelerators, such as graphics processing units (GPUs) and similar single-instruction multiple-data/thread (SIMD/SIMT) devices, increasingly available on supercomputing clusters [10–12]. The ultimate goal of the combustion community is to enable use of detailed kinetic models in realistic reactive-flow simulations—potentially via use of GPU-accelerated chemical kinetics. However, a clear first step is to reduce the cost of realistic reactive-flow simulations with small-to-moderate sized model

to the point where they are practical for iterative design purposes.

1.1. SIMD/SIMT architecture

Central processing unit (CPU) clock speeds increased regularly over the past few decades—commonly known as Moore's Law—however, power consumption and heat dissipation issues slowed this trend recently. While multicore parallelism increased CPU performance somewhat, recently SIMD/SIMT-enabled processors have gained popularity in high-performance computing due to their greatly increased floating operation per second count. A SIMD instruction utilizes a vector processing unit to execute the same instruction on multiple pieces of data, e.g., performing multiple floating point multiplications concurrently. In contrast, a SIMT process achieves SIMD parallelism by having many threads execute the same instruction concurrently. Many different flavors of SIMD/SIMT processing exist:

- Modern CPUs have vector processing units capable of executing SIMD instructions (e.g., SSE, AVX2)
- GPUs feature hundreds to thousands of separate processing units, and utilize the SIMT model
- Intel's Xeon Phi co-processor has tens of (hyperthreaded) cores containing wide-vector units designed for SIMD execution, with each core capable of running multiple independent threads

Using the SIMD/SIMT parallelism model requires extra consideration to accelerate chemical kinetics integration.

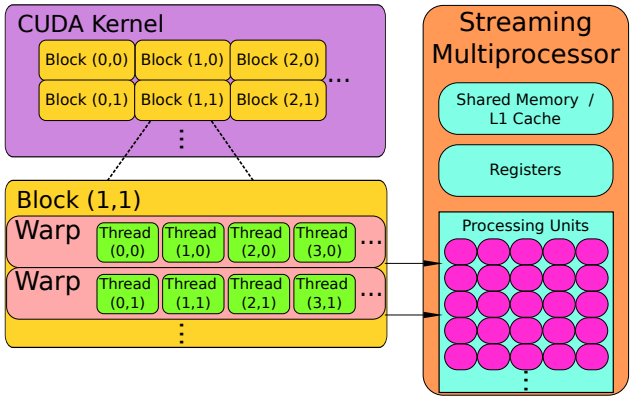


Figure 1: Example of the CUDA SIMT paradigm. Program calls (kernels) are split into a grid of blocks, which are in turn composed of threads. Threads are grouped in warps (note: warps are typically composed of 32 threads) and executed concurrently on streaming multiprocessors. Streaming multiprocessors have registers and L1 cache memory shared between all executing warps. Figure file is available under CC-BY [13].

This study used the NVIDIA CUDA framework [14, 15], hence the following discussion will use CUDA terminology; however, the concepts within are widely applicable to SIMT processing. The basic parallel function call on a GPU, termed a kernel, is broken up into a grid of thread blocks as seen in Fig. 1. A GPU consists of many streaming multiprocessors (SMs), each of which is assigned one or more thread blocks in the grid. The SMs further subdivide the blocks into groups of 32 threads called warps, which form the fundamental CUDA processing entity. The resources available on a SM (memory, processing units, registers, etc.) are split between the warps from all the assigned blocks. The threads in a warp are executed in parallel on CUDA cores (processing units), with multiple warps typically being executed concurrently on a SM. Thread divergence occurs when the threads in a warp follow different execution paths, e.g., due to if/then branching, and is a key performance concern for SIMT processing; in such cases the divergent execution paths must execute in serial. All threads in a warp are executed even if any thread in the warp is unfinished. When a divergent path is long and complicated or only a handful of threads in a warp require its execution, significant computational waste may occur as the other threads will be idle for long periods. A related concept of waste within a SIMD work unit is described by Stone and Niemeyer [16].

Furthermore, as compared with a typical CPU, GPUs possess relatively small memory caches and few registers per SM. These resources are further split between all the blocks/warps running on that SM (Fig. 1). Overuse of these resources can cause slow global memory accesses for data not stored locally in-cache or can even reduce the number of blocks assigned to each SM. The performance tradeoffs of various CUDA execution patterns are quite involved and beyond the scope of this work; for more details we refer the interested reader to several works that discussed these topics in depth [17–19]. Instead, we will briefly highlight key considerations for CUDA-based integration of chemical kinetic initial value problems (IVPs).

1.2. GPU-accelerated chemical kinetics

The extent of thread cooperation within a CUDA-based chemical kinetic IVP integration algorithm is a key point that shapes much of implementation. GPU-accelerated chemical kinetic solvers typically follow either a "per-thread" pattern [20–22], in which each individual GPU thread solves a single chemical kinetic IVP, or a "per-block" approach [21, 23], in which all the threads in a block cooperate to solve the ordinary differential equations (ODEs) that comprise a single chemical kinetic IVP. The greatest potential benefit of a per-thread approach is that a much larger number of IVPs can theoretically be solved concurrently; the number of blocks that can be executed concurrently

on each SM is usually around eight, whereas typical CUDA launch configurations in this work consist of 64 threads per block, or 512 sets of IVPs solved concurrently per SM. Unfortunately, the larger amount of parallelism offered by a per-thread approach does not come without drawbacks. A per-thread approach may also encounter more cache-misses, since the memory available per SM must now be split between many more sets of IVPs. This results in expensive global memory loads. The performance of a per-thread approach can also be greatly impacted by thread divergence, because different threads may follow different execution paths within the IVP integration algorithm itself [21, 22]. For example, in a per-thread-based solver each thread in a warp advances its IVP by one internal integration step concurrently, and here on a step failure the thread simply does not update the solution vector at the end of the internal time-step. If only a handful of threads in a warp require many more internal time-steps than the others, they will force the majority of threads to wait until all threads in the warp have completed the global integration step, wasting computational resources. Additionally, implicit integration algorithms—which typically have complex branching and evaluation paths—may suffer more from thread divergence when implemented on a per-thread basis than relatively simpler explicit integration techniques [21]. The impact of thread divergence on integrators is typically less severe when following a per-block strategy, since the execution path of each thread is planned by design of the algorithm. A per-block approach also offers significantly more local cache memory and available registers for solving an IVP, and thus memory access speed and cache size are less of a concern. However, in our experience, optimizing use of these resources requires significant manual tuning and makes it more difficult to generalize the developed algorithm between different chemical kinetic models—a key feature for potential non-academic applications. In addition, Stone and Davis [21] showed that a per-thread implicit integration algorithm outperforms the per-block implementation of the same algorithm in the best-case scenario (elimination of thread divergence by choice of identical initial conditions).

Various studies in recent years explored the use of high-performance SIMT devices to accelerate (turbulent) reactive-flow simulations. Spafford et al. [24] investigated a GPU implementation of a completely explicit—and thus well suited for SIMT-acceleration—direct numerical simulation code for compressible turbulent combustion. Using a Tesla C1060 GPU, an order of magnitude speedup was demonstrated for evaluation of species production rates compared to a sequential CPU implementation on a AMD-Operton processor; evaluating chemical source terms is much simpler than chemical kinetics integration on GPUs. Shi et al. [25] used a Tesla C2050 GPU to evaluate species rates and factorize the Jacobian for the integration of (single) independent kinetics systems,

showing order-of-magnitude or greater speedups for large kinetic models over a CPU-based code on a quad-core Intel i7 930 processor which used standard `CHEMKIN` [26] and LAPACK [27] libraries for the same operations; it was not clear how/if the CPU code was parallelized. Niemeyer et al. [20] implemented an explicit fourth-order Runge–Kutta integrator for a Tesla C2075 GPU, and found a speedup of nearly two orders of magnitude with a nonstiff hydrogen model when compared with a sequential CPU-code utilizing a single core of an Intel Xeon 2.66 GHz CPU. In a related work, Shi et al. [28] developed a GPU-based stabilized explicit solver on a Tesla C2050 and paired it with a CPU-based implicit solver using a single-core of a quad-core Intel i7 930 that handled integration of the most-stiff chemistry cells in a three-dimensional premixed diesel engine simulation; the hybrid solver was 11–46 $\times$ faster than the implicit CPU solver. Le et al. [29] implemented GPU versions of two high-order shock-capturing reactive-flow codes on a Tesla C2070, and found a 30–50$\times$ speedup over the baseline CPU version running on a single core of a Intel Xeon X5650. Stone and Davis [21] implemented the implicit VODE [30] solver on a Fermi M2050 GPU and achieved an order of magnitude speedup over the baseline CPU version running on a single core of a AMD Opteron 6134 Magny-Cours. They also showed that GPU-based VODE exhibits significant thread divergence, as expected due to its complicated program flow compared with an explicit integration scheme. Furthermore, Stone and Davis [21] found that a per-thread implementation outperforms a per-block version of the same algorithm for $\sim 10^4$ independent IVPs or more; the per-block implementation reached its maximum speedup for a smaller number of IVPs ($\sim 10^3$). Niemeyer and Sung [22] demonstrated an order-of-magnitude speedup for a GPU implementation of a stabilized explicit second-order Runge–Kutta–Chebyshev algorithm on a Tesla C2075 over a CPU implementation of VODE on a six-core Intel X5650 for moderately stiff chemical kinetics. They also investigated levels of thread divergence due to differing integrator time-step sizes, and found that it negatively impacts overall performance for dissimilar IVP initial conditions in a thread-block. Sewerin and Rigopoulos [23] implemented a three-stage/fifth-order implicit Runge–Kutta GPU method [31] on a per-block basis for high-end (Nvidia Quadro 6000) and consumer-grade (Nvidia Quadro 600) GPUs, as compared to a standard CPU (two-core, four-thread Intel i5-520M) and a scientific workstation (eight-core, 16-thread Intel Xeon E5-2687W) utilizing a message passing interface for parallelization; the high-end GPU was at best 1.8 $\times$ slower than the workstation CPU (16 threads), while the consumer level GPU was at best 5.5 $\times$ slower than the corresponding standard CPU (four threads).

While increasing numbers of studies have explored GPU-based chemical kinetics integration, there remains a clear need to find or develop integration algorithms simultaneously suited for the

SIMT parallelism of GPUs (along with similar accelerators) and capable of handling stiffness. In this work we will investigate GPU implementations of several semi-implicit and implicit integration techniques, as compared with their CPU counterparts and the baseline CPU `CVODE` [32] algorithm. Semi-implicit methods do not require solution of non-linear systems via Newton iteration—as required for implicit integration techniques—and instead solve sequences of linear systems [31]; accordingly these techniques are potentially better suited for SIMT acceleration due to an expected reduction of thread divergence (for a per-thread implementation) compared with implicit methods.

Several groups [33, 34] previously suggested so-called matrix-free methods as potential improvements to the expensive linear-system solver required in standard implicit methods. These methods do not require direct factorization of the Jacobian, but instead use an iterative process to approximate the action of the factorized Jacobian on a vector. Furthermore, Hochbruck and Lubich [35, 36] demonstrated that the action of the matrix exponential on a vector obtained using Krylov subspace approximation converges faster than corresponding Krylov methods for the solution of linear equations. Others explored these semi-implicit exponential methods for applications in stiff chemical systems [37, 38] and found them stable for time-step sizes greatly exceeding the typical stability bounds.

Since GPU-based semi-implicit exponential methods have not been demonstrated in the literature, we aim to conduct a systematic investigation to test and compare their performance to other common integration techniques. Finally, we will study the three-stage/fifth-order implicit Runge–Kutta algorithm [31] investigated by Sewerin and Rigopoulos [23] here to determine the impact of increasing stiffness on the algorithm and the performance benefits of using an analytical Jacobian matrix, such as that developed by Niemeyer et al. [39–41].

Recently, implicit methods improved using adaptive preconditioners have shown promise in reducing integration costs for large kinetic models, compared with implicit methods based on direct, dense linear algebra [42]. These require use of linear iterative methods in addition to the standard Newton iteration, and thus we expect increased levels of thread-divergence (and integrator performance penalties) for the per-thread approach used in this work. However, this area merits future study.

The rest of the paper is structured as follows. Section 2 lays out the methods and implementation details of the algorithms used here. Subsequently, Sec. 3 presents and discusses the performance of the algorithms run using a database of partially stirred reactor thermochemical states, with particular focus on the effects of thread divergence and memory traffic. Further, this work is a

starting point to reduce the cost of reactive-flow simulations with realistic chemistry via SIMT-accelerated chemical kinetics evaluation. Thus, we explore the potential impact of current state-of-the-art GPU-accelerated stiff chemical kinetic evaluation on large-scale reactive-flow simulations in Sec. 3, while identifying the most promising future directions for GPU/SIMT accelerated chemical kinetic integration in Sec. 4. The source code used in this work is freely available [43]. Appendix A discusses the validation and performance data, plotting scripts, and figures used in creation of this paper, as well as the supplementary material which includes unscaled plots of integrator runtimes and characterizations of the partially stirred reactor data for this work.

2. Methodology

In this section, we discuss details of the algorithms implemented for the GPU along with third-party software used. The generation of testing conditions will be discussed briefly, and the developed solvers will be verified for expected order of error.

2.1. Integration techniques

Method	CPU	GPU
`CVODE`	×	
`Radau-IIA`	×	×
`exp4`	×	×
`exprb43`	×	×

Table 1: The solvers used in this study, and platforms considered for each.

We investigated GPU implementations of three integration methods in this work, namely `Radau-IIA` [31], `exp4` [36], and `exprb43` [44], comparing them against equivalent CPU versions and a CPU-only implicit algorithm `CVODE` [9, 32]. Table 1 lists these solvers and their corresponding platforms. While we describe important details or changes made in this work, full descriptions of all algorithms may be found in the cited sources. The `pyJac` software [39–41] provided subroutines for both chemical source terms and the analytical constant-pressure, mass-fraction-based Jacobian matrix used by CPU- and GPU-based algorithms. We evaluated the relative performance impact of using a finite-difference Jacobian matrix (as compared with an analytical Jacobian) for both platforms with a first-order finite difference method based on that of `CVODE` [32]. `pyJac` also provided the

chemical source terms used by the finite-difference Jacobian in all cases. We direct readers to our previous work [40, 41] for verification and performance assessments of `pyJac` itself.

First, the `CVODE` solver [9, 32] (part of the `SUNDIALS` suite [45]) provided the baseline performance of a typical CPU-based (maximum of fifth-order) implicit integration technique. In addition, we developed CPU versions of the methods under investigation for direct comparison to the high-order implicit technique. These include the three-stage/fifth-order implicit Runge–Kutta algorithm [31] (`Radau-IIA`), the fourth-order exponential Rosenbrock-like method of Hochbruck et al. [36] (`exp4`), and the newer fourth-order exponential Rosenbrock method [44] (`exprb43`). For the exponential methods, we used the method of rational approximants [46] paired with the Carathédothy–Fejér method [47, 48] to approximate the action of the matrix exponential on a vector, as suggested by Bisetti [37]. This technique relied on the external `FFTW3` library [49, 50]. However, unlike the approach of Bisetti [37], we developed a custom routine based on the algorithm presented by Stewart [51] to perform LU decomposition of the Hessenberg matrix resulting from the Arnoldi iteration. Convergence of the Arnoldi iteration algorithm was computed using the second term of the exponential matrix/vector product infinite series, as suggested in several works [37, 52]. The exponential integrators used a rational approximant of type $(10, 10)$ as suggested by Bisetti [37]. To ensure high performance of CPU-based methods, the Intel `MKL` library version 11.3.2 handled linear algebra (i.e., BLAS/LAPACK) operations. Next, we developed GPU versions of the `Radau-IIA`, `exp4`, and `exprb43` methods. These follow the same descriptions as the CPU versions, but require specialized implementations of several BLAS and LAPACK methods, mostly related to LU factorization of the Jacobian or Hessenberg matrices. All GPU routines were developed using the NVIDIA CUDA framework [14, 15], and a block-size of 64 threads (8 blocks per SM) was found to be most efficient for all solvers. All solvers used adaptive time-stepping techniques; the `Radau-IIA` and `CVODE` integrators have built-in adaptive time-stepping, while the exponential methods, `exp4` and `exprb43`, used a standard adaptive time-stepping technique [31]. The adaptive time stepping procedures of all integrators used absolute and relative tolerances of 10^{-10} and 10^{-6}, respectively, throughout the work. Finally, the Jacobian was reused on a per-thread (per-IVP) basis according to the built-in rules for the implicit methods, and only recomputed on step failures for the exponential methods.

2.2. Testing conditions

In order to measure the performance of the integrators for realistic conditions, a database of thermochemical states covering a wide range of temperatures and species mass fractions was gen-

erated using a previously developed constant-pressure stochastic partially stirred reactor (PaSR) code [39, 41]. We selected two chemical kinetic models to span the range of model sizes typically used in high-fidelity simulations: the hydrogen model of Burke et al. [53] with 13 species and 27 reactions, and the GRI-Mech 3.0 model for methane with 53 species and 325 reactions [54]. The PaSR simulations were performed at the conditions listed in Table 2 for 10 residence times to reach a statistical steady state; Niemeyer et al. [41] describe the PaSR simulation process in greater detail, which follows approaches used by others [55–57]. The PaSR particles were initialized using the equilibrium state, and gradually move away from equilibrium conditions due to mixing, inflow, and outflow. In order to reduce the influence of equilibrium conditions on the solution runtime trends for small numbers of IVPs, the first 1000 datapoints were removed from each database; this corresponds to a single pairing time, τ_{pair}, the time interval at which selected particles in the reactor are randomly swapped with inflowing particles. At this point in the simulation, $\sim 80\,\%$ of the particles were at or near an equilibrium state, and by the 5000th datapoint only $\sim 20\,\%$ of the particles were near equilibrium. The hydrogen and GRI-Mech 3.0 databases consisted of 899,900 and 449,900 total conditions, respectively. Further characterization of the PaSR conditions used in this work can be found in Appendix A and our previous study [41].

Parameter	H_2/air	CH_4/air
ϕ	1.0	
T_{in}	400, 600, and 800 K	
p	1, 10, and 25 atm	
N_p	100	
τ_{res}	10 ms	5 ms
τ_{mix}	1 ms	1 ms
τ_{pair}	1 ms	1 ms

Table 2: PaSR parameters used for hydrogen/air and methane/air premixed combustion cases, where ϕ indicates equivalence ratio, T_{in} is the temperature of the inflowing particles, p is the pressure, N_p is the number of particles in the reactor, τ_{res} is the residence time, τ_{mix} is the mixing time, and τ_{pair} is the pairing time.

2.3. Solver verification

To investigate the correctness of the developed solvers, the first 10,000 conditions in the hydrogen database were integrated by each solver using a global time-step size of 10^{-6} s. The error for

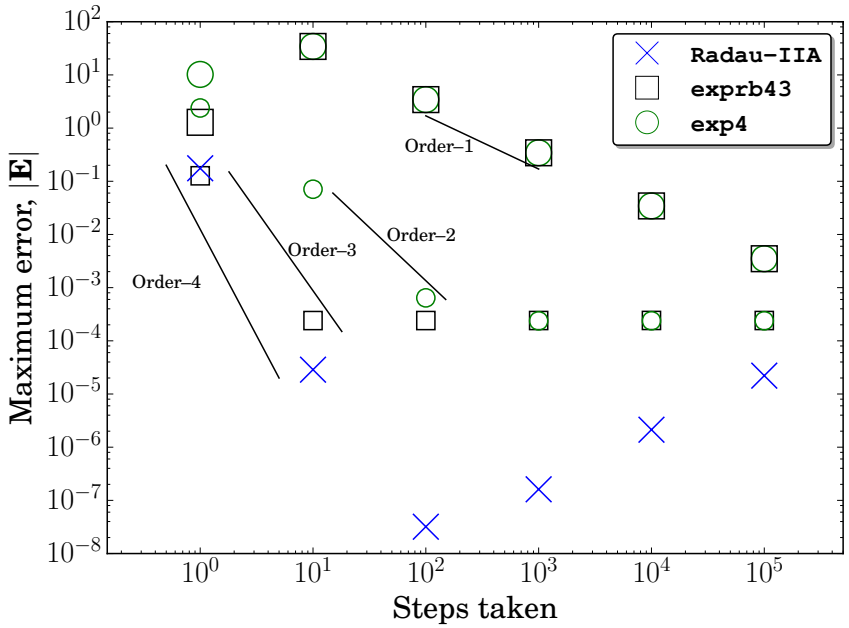


Figure 2: Maximum error of the three CPU solvers as a function of the total number of internal integration steps taken (corresponding to decreasing time-step size). Larger square and circle symbols indicate the use of Krylov subspace approximations with the exponential methods, while the smaller symbols indicate the use of "exact" Krylov subspaces. Data, plotting scripts, and figure file are available under CC-BY [13].

condition i was then determined using the weighted root-mean-square error

$$E_i(t) = \left\| \frac{y_i(t) - \hat{y}_i(t)}{\text{atol} + \hat{y}_i(t) \times \text{rtol}} \right\|_2 , \tag{1}$$

where the $y_i(t)$ is the solution obtained from the various solvers, atol/rtol are the absolute/relative tolerances, and $\hat{y}_i(t)$ is the "true" solution obtained via `CVODE` using the same global time-step of $\Delta t = 10^{-6}$ s and absolute/relative tolerances of 10^{-20} and 10^{-15}, respectively; note that the more stringent tolerances were used only to obtain the "true" solution. The maximum error over all conditions:

$$|\mathbf{E}| = \max_{i=1,\ldots,10{,}000} \{E_i(t)\} \tag{2}$$

was then used to measure the error of each solver. The error measurement used the same tolerances as for the performance testing (atol $= 10^{-10}$ and rtol $= 10^{-6}$, respectively). The constant internal time-step size was then varied from 10^{-6}–10^{-11} s—corresponding to 10^0–10^5 internal integration steps—to measure the convergence rates of the three solvers used in this study.

Figure 2 shows the convergence of error for the CPU solvers with decreasing internal time-step size, shown as increasing number of integration steps taken. The error of the `Radau-IIA` integrator

drops nearly four orders of magnitude when changing from a single internal time step of 10^{-6} s to ten internal time steps of 10^{-7} s each, i.e., fourth-order convergence. Increasing the number of integration steps—by further reducing the internal time-step size—past this point results in one further drop in error (of order ~ 3); however for more than 10^3 steps the overall error begins to climb due to accumulation of local error. Since the `Radau-IIA` solver is nominally fifth-order, it is unclear whether we are observing order reduction due to the stiffness of the problem, use of a numerically obtained "true" solution, or an accumulation of local error. Although a more accurate assessment of convergence order might be achieved through use of a stiff sample problem with an analytical solution—e.g., HIRES [31] or ROBER [58]—direct validation with the problem at hand was conducted here.

The exponential solvers utilizing an approximate Krylov subspace exhibit larger levels of error in general, with $|\mathbf{E}| \sim \mathcal{O}(1)$–$\mathcal{O}(10)$ for a single internal integration step of $\delta t = 10^{-6}$ s. As the time-step size is decreased, the convergence of the Arnoldi algorithm is affected by the internal integration time-step size (the matrix exponentials and error estimates are scaled by the internal time-step). To study the effect of the Arnoldi algorithm on error, Fig. 2 also presents the error convergence of the exponential integrators with the Krylov approximation error reduced far below the error of the overall method (for larger internal time-steps). Practically, this was accomplished by detecting when the nth Krylov subspace vector approaches zero, a condition known as the "happy breakdown" in literature [59]. At this limit, the approximate exponential matrix/vector product approaches the exact value and thus the Krylov approximation error is relatively small compared to the error of the overall method. It is clear the that error induced by the "exact" Krylov subspace is non-zero however, as both methods reach a minimum error around 10^2 steps and are unaffected by further step-size decreases, in contrast to the `Radau-IIA` solver which exhibits increasing error past this point due to local error accumulation. Figure 2 shows that the exponential methods achieve only first-order convergence to the true solution with the approximate Krylov subspace, but both methods converge at higher rates with the "exact" Krylov subspace. The nominal fourth-order convergence of the `exp4` algorithm is a classical nonstiff order, and thus order reduction is expected for stiff problems [37, 60]; the `exp4` solver reaches roughly second-order convergence with the "exact" Krylov subspace. The `exprb43` solver reaches third-order convergence with the "exact" Krylov subspace. Similar to the discussion on the `Radau-IIA` convergence order, it is difficult to determine whether order reduction has occurred due to problem stiffness, the use of a numerically obtained "true" solution, or some combination thereof. Furthermore, the error of Krylov subspace

approximation dominates the error measurement $|\mathbf{E}|$. From Fig. 2 we conclude that all three solvers produce reasonably accurate solutions as compared with `CVODE`. Additionally, although not shown, the GPU solvers produce identical results.

3. Results and discussion

We studied the performance of the three integrators by performing constant-pressure, homogeneous reactor simulations with two global integration time-step sizes, $\Delta t = 10^{-6}$ s and $\Delta t = 10^{-4}$ s, for each integrator. Initial conditions were taken from the PaSR databases described in Sec. 2.2. A larger global time step induces additional stiffness and allows evaluation of the performance of the developed solvers on the same chemical kinetic model with varying levels of stiffness. In reactive-flow simulations, the chemical integration time-step is typically determined by the flow time-scale and stability requirements determined by the Courant–Friedrichs–Lewy number. Typical global time-step values of reactive-flow simulations are not always clear in the literature, as adaptive time-stepping is often used, or the global time-step size is simply not reported; our own experience suggests global time-step sizes ranging from 10^{-7} s to 10^{-4} s. The global time-step size used in a given simulation depends highly on the problem and numerical methods, but large-eddy simulations usually require higher time resolution than Reynolds-averaged Navier–Stokes simulations [61]. Hence, the global time-step sizes we selected for study represent realistic values used in large-eddy [62, 63] and Reynolds-averaged Navier–Stokes [64, 65] simulations.

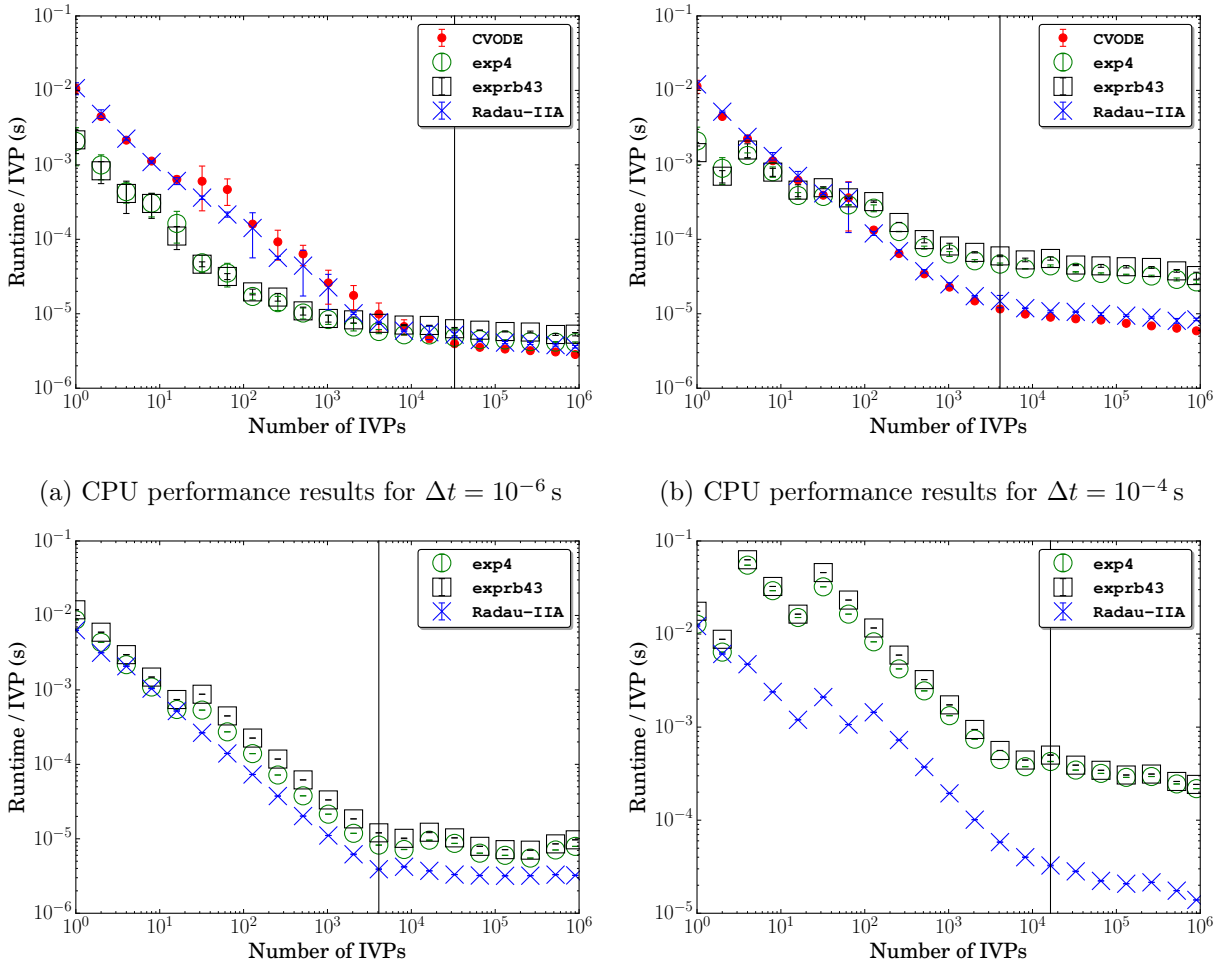
Runtimes are reported as the average over five runs, where each run started from the same set of PaSR conditions. All CPU integrators were compiled using `gcc 4.8.5` (with the compiler options "`-O3 -funroll-loops -mtune=native`") and executed in parallel via OpenMP on four ten-core 2.2 GHz Intel Xeon E5-4640 v2 CPUs with 20 MB of L3 cache memory, installed on an Ace Powerworks PW8027R-TRF+ with a Supermicro X9QR7-TF+/X9QRi-F+ baseboard. OpenMP was used to parallelize on a per-condition basis; i.e., each individual OpenMP thread was responsible for integrating a single chemical kinetic IVP, rather than cooperating with other OpenMP threads to solve the same. A six-core 2.67 GHz Intel Xeon X5650 CPU hosted the GPU integrators, which were compiled using `nvcc 7.5.17` (with compiler options "`-arch=sm_20 -O3 -maxrregcount 63 --ftz=false --prec-div=true --prec-sqrt=true --fmad=false`") and run on a single NVIDIA Tesla C2075 with 6 GB of global memory. Reported runtimes for the GPU-based algorithms include time needed for CPU–GPU data transfer before and after each global time step; in addition, the function `cudaSetDevice()` initialized the GPU before timing to avoid any device initialization delay.

The open-source `pyJac` software [39–41] produced CPU and GPU custom source-code functions for the chemical source terms and analytical Jacobian matrix evaluation. Finally, the L1/shared-memory cache was set to prefer a larger L1 cache using the `cudaDeviceSetCacheConfig()` function.

3.1. Runtime performance

For all cases in this section, the integrator runtimes are presented as the runtime per IVP solved, for two reasons. First, saturation of the available computational resources becomes visually apparent (transition from a nearly linear decrease to a flat trend), and second, it allows certain other performance trends (e.g., the effects of thread divergence) to be easily highlighted. The presentation of the performance data in raw form is also available in the supplementary material for completeness.

Figure 3 shows the runtimes of the CPU and GPU integrators for the hydrogen model. In Fig. 3a the runtimes per IVP for the CPU integrators for a single global time-step of $\Delta t = 10^{-6}\,\mathrm{s}$ decrease approximately linearly with the number of IVPs for small numbers of initial conditions (shown here on a log-log plot). For small numbers of IVPs, the exponential integrators are faster than the implicit integration techniques due to the modest stiffness of the hydrogen model; even with many near-equilibrium states removed from the beginning of the PaSR database, the model is not particularly stiff for this small time-step size. Larger numbers of IVPs begin to saturate the CPU resources, and the runtime per IVP levels off to a more constant value; vertical lines are shown in Fig. 3 where the relative change in runtime per IVP between successive data-points is first smaller than 15 % (based on the results for `CVODE`/`Radau-IIA` for the CPU/GPU respectively). Eventually, relatively more stiff conditions are encountered and the performance of the implicit integration techniques catches up and then surpasses that of the exponential integrators; `CVODE` is the most efficient solver on the CPU when solving more than 10^4 IVPs; however, `CVODE` is only $\sim$1.87$\times$ faster than the slowest solver (`exprb43`) on the whole database. Figure 3c shows the performances of the GPU integrators for the smaller global time-step size, which exhibit similar trends as the CPU solvers: a linearly decreasing solution cost that reaches a roughly constant value beyond 10^3–10^4 IVPs. Unlike for the CPU solvers, the GPU-based `Radau-IIA` performs faster than the exponential solvers for all numbers of IVPs. As will be seen in Sec. 3.3, both solver classes experience minimal thread divergence due to differing internal integration time-step sizes in this case. Therefore, we conclude that the relatively slower runtimes per IVP for the exponential algorithms on the GPU results from thread divergence in the Arnoldi iteration—caused by varying Krylov subspace sizes between threads.



(a) CPU performance results for $\Delta t = 10^{-6}\,\mathrm{s}$

(b) CPU performance results for $\Delta t = 10^{-4}\,\mathrm{s}$

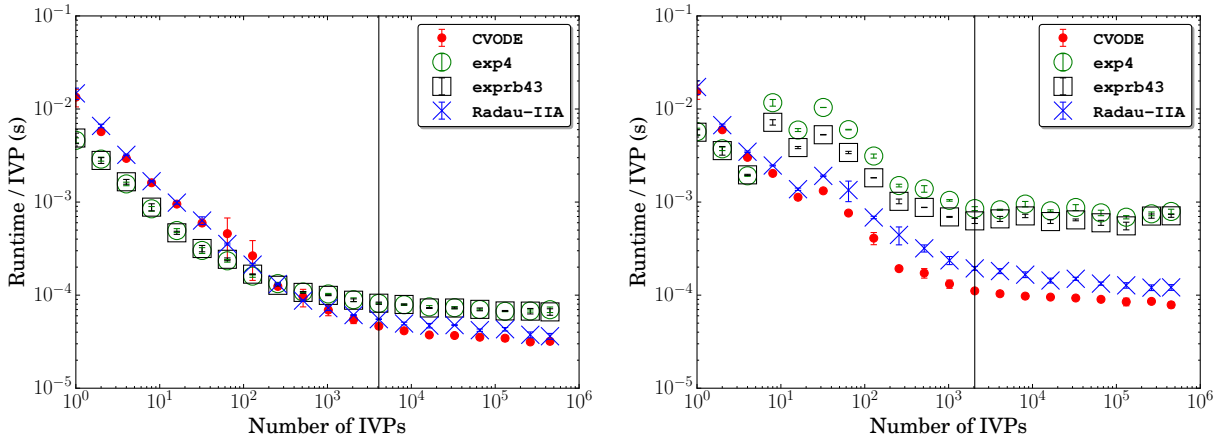
(c) GPU performance results for $\Delta t = 10^{-6}\,\mathrm{s}$

(d) GPU performance results for $\Delta t = 10^{-4}\,\mathrm{s}$

Figure 3: Average runtimes of the integrators on the CPU and GPU, scaled by the number of IVPs, for the hydrogen model at two different global time-step sizes. Estimation of where the runtime per IVP levels off to a constant value (based on the results for `CVODE`/`Radau-IIA` for the CPU/GPU, respectively) is marked with a vertical line for all cases. Error bars indicate standard deviation. Data, plotting scripts, and figure files are available under CC-BY [13].

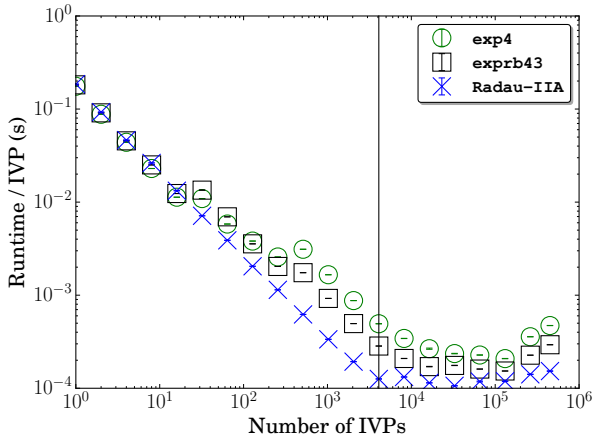
Figures 3b and 3d show the performance of the integration algorithms on both platforms for the hydrogen model with a single larger global time step ($\Delta t = 10^{-4}$ s). The performances of the CPU integration algorithms show similar trends to those of the smaller global time-step size case: decreasing cost per IVP before reaching a more constant performance for higher numbers of IVPs. The larger global time-step size induces additional stiffness, and the implicit solvers are more efficient for most numbers of IVPs; `CVODE` is again the most efficient CPU solver. Figure 3d shows the performance of the GPU solvers for the larger global time-step size. The exponential solvers exhibit significant spikes in computational cost when changing from 2–4 and 16–32 IVPs, with the latter mimicked somewhat by the implicit `Radau-IIA` solver. A jump in solution cost between 2–4 IVPs is also present for the CPU exponential integrators, indicating stiffness as the primary cause. On the other hand, between 16–32 IVPs the CPU exponential solvers exhibit only a very minor performance decrease, while the GPU-based `Radau-IIA` also shows a decrease in performance at the same point—a trend completely absent in the CPU `Radau-IIA` version. These factors indicate that thread divergence also plays a key role in the performance trend here, and will be investigated further in Sec. 3.3. As in case of the smaller global time-step size, the `Radua-IIA` solver is the most efficient GPU algorithm in all cases.

Figure 4 shows the runtime of the integrators for the GRI-Mech 3.0 model. Similar to the hydrogen case for the smaller global time-step size, the CPU exponential integrators are more efficient (Fig. 4a) for the near-equilibrium conditions at the beginning of the database. For larger numbers of conditions, the implicit integrators are more efficient, and `CVODE` again performs the fastest. Compared with the hydrogen model (Fig. 3a), the `CVODE` performs better than the exponential algorithms for the GRI-Mech 3.0 model with the small global time-step size (Fig. 4a), reaching a speedup of 2.18 × over `exp4` on the whole database; this results from the higher stiffness present in the model. This performance gap between the CPU implicit/exponential integrators increases for the larger global time-step size (Fig. 4b); `CVODE` is 10.1 × faster than `exp4` on the whole database. Comparing the performance of the CPU implicit solvers between the two kinetic models shows roughly an order-of-magnitude performance decrease for both global time-step sizes. This phenomena, due largely to the increase in model size, is also seen for the `Radau-IIA` GPU solver for the smaller global time-step size; the performance of which decreases by just over an order of magnitude. However, for the larger global time-step size, the GPU-based `Radau-IIA` solver performs roughly two orders-of-magnitude slower compared with the hydrogen case. As will be examined in Sec. 3.3, this dramatic decrease likely results from increased thread divergence in the `Radau-IIA`

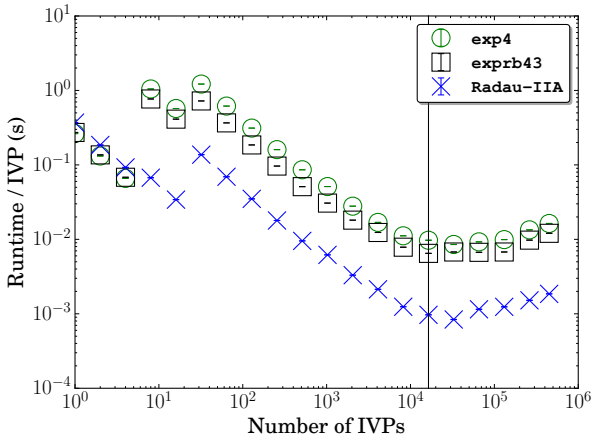


(a) CPU performance results for $\Delta t = 10^{-6}\,\mathrm{s}$

(b) CPU performance results for $\Delta t = 10^{-4}\,\mathrm{s}$



(c) GPU performance results for $\Delta t = 10^{-6}\,\mathrm{s}$



(d) GPU performance results for $\Delta t = 10^{-4}\,\mathrm{s}$

Figure 4: Average runtimes of the integrators, scaled by number of IVPs, on the CPU and GPU for the GRI-Mech 3.0 model at two different global time-step sizes. Estimation of where the runtime per IVP levels off to a constant value (based on the results for `CVODE`/`Radau-IIA` for the CPU/GPU respectively) is marked with a vertical line for all cases. Error bars indicate standard deviation. Data, plotting scripts, and figure files are available under CC-BY [13].

solver, as well as the increased memory traffic inherent in the larger model.

Unlike for the hydrogen model, the `exprb43` solver outperforms `exp4` with the GRI-Mech 3.0 model in almost all cases for the larger global time-step size for both the CPU and GPU. Although the `exprb43` and `exp4` algorithms each require three exponential matrix function approximations per step, a single internal time step of `exprb43` is more expensive due to the extra chemical source term evaluations, matrix multiplications, and higher-order exponential matrix function requirement. As such, the relatively simpler CPU `exp4` integrator outperforms the CPU `exprb43` integrator for the hydrogen model where there is relatively less stiffness. However, as previously discussed the `exp4` algorithm may experience order reduction for stiff problems, and the `exprb43` algorithm typically outperforms `exp4` on both the CPU and GPU in the larger global time-step GRI-Mech 3.0 case as a result.

3.2. CPU/GPU performance comparison

Comparing the performance of CPU- and GPU-based integrators in a meaningful way is challenging. First, the vastly different nature of the processing cores in each platform eliminates the possibility of comparing performance normalized by core count. In addition, the floating-point operation count is not readily available for chemical kinetics integration—unlike many GPU-accelerated applications where the number of operations required to solve the problem is known, e.g., as in linear-algebra operations or fast Fourier transforms—which precludes comparing performance on the basis of floating-point operations per second (FLOPS). Although the runtimes of the GPU integration algorithms can be directly compared with that of the CPU-based solvers (and often are), these figures do not provide much useful information. For instance, if a GPU algorithm performs $10\times$ faster than its equivalent on two six-core CPUs, how does this compare to two eight-core CPUs, etc.?

For researchers in numerical combustion, two issues stand out as particularly important for performance evaluation: runtime and cost. As established in Sec. 1, large-scale reactive-flow simulations with realistic chemical kinetic models are extremely computationally expensive, and remain outside of the capabilities of most in the field. With this in mind, we ask, for a given simulation, what is the effect on the overall runtime of adding more CPU cores compared with adding GPU accelerators? In addition, if a budget is allocated to expand available computational resources, how might these funds be best allocated? To answer these questions, we derived an estimate of the number of CPU cores required for equivalent performance on the GPU.

A nominal performance metric for both the CPU- and GPU-based integration algorithms must first be obtained. As the most efficient solvers in all cases with large numbers of IVPs are `CVODE` for the CPU and `Radau-IIA` for the GPU, these algorithms will be considered the performance benchmarks. Furthermore, most large-scale simulations consist of millions of cells (or more), and therefore we only consider the performance limit of each algorithm (i.e., the cost per IVP of each algorithm in the region where this cost reaches an approximately constant value). To this end, the previously discussed threshold—the first relative change in runtime per IVP between successive data-points smaller than 15 % (based on `CVODE`/`Radau-IIA` for the CPU/GPU accordingly)—is used, and marked as vertical lines on Figs. 3 and 4. The cost per IVP above and including these thresholds was averaged and forms our nominal performance measure. The CPU performance measure must also be normalized by the total number of cores used: 40. Table 3 presents the ratios of these performance measures, which give estimates for the number of CPU cores required to equal the GPU performance for the cases studied. The GPU is roughly equivalent to 12 or more CPU cores for all cases except GRI-Mech 3.0 with the larger global time-step size, and equivalent to at most 38 cores for the hydrogen case with the smaller global time-step size. With the increasing size of the chemical kinetic model, the equivalent CPU core count of the GPU `Radau-IIA` solver drops significantly. As will be discussed in Sec. 3.3, this drop in performance is primarily due to higher memory traffic requirements, however increased levels of thread divergence also play a role. Although this work represents the current state-of-the-art for implicit integration of stiff chemical kinetic IVPs on the GPU, it is clear that more effort is required to improve GPU performance for larger chemical kinetic models. Approaches to mitigate these issues will be discussed in the subsequent section.

Global time-step size	# equivalent CPU cores	
	Hydrogen	GRI-Mech 3.0
10^{-6} s	38	12
10^{-4} s	15	3

Table 3: The number of CPU cores (roughly) required for equivalent performance to a single GPU for the combinations of chemical kinetic models and global time-step sizes studied.

At the time of writing, the ten-core Intel Xeon E5-4640 v2 CPU used in this study was listed for a recommended customer price of $2725 [66], while a new Tesla C2075 GPU is available for ∼$1400 [67]. These prices are only rough estimates of the actual cost of these devices, since the

actual price for the Intel CPU may be significantly less in a configured server node, while the Tesla C2075 is no longer sold directly by NVIDIA—thus the prices are variable. Furthermore, the performance decrease using an older, cheaper CPU (e.g., the Intel Xeon X5650 used as host processor for the GPU simulations in this work) may not be that large. However, combined with the equivalent core counts in Table 3, this information suggests that the Tesla C2075 is a reasonable investment to supplement computing power for chemical-kinetic integration in large-eddy simulations.

3.3. Effects of thread divergence and memory traffic

Thread divergence and memory traffic are two performance concerns particularly important for chemical kinetics integration on GPU and SIMT platforms. Slowdown due to memory traffic for a GPU integration algorithm implemented on a per-thread basis primarily results from the small amount of on-chip memory available. Implicit integration algorithms, which typically require storage of the Jacobian matrix and/or factorized forms thereof, can quickly overwhelm the registers and L1 cache memory available to each thread and cause many slow global memory accesses. Reformulating the chemical kinetic equations to generate sparse Jacobian matrices [68] would greatly benefit GPU-based integration algorithms due to the reduced memory requirements, and in addition enable use of sparse multiplication/factorization algorithms (from which a CPU-based algorithm would also benefit); this is a planned improvement to the `pyJac` software [39, 41].[1] Further, the Tesla C2075 GPU used in this study was originally released nearly five years ago and is several generations old; the newer Tesla K40 is available for a similar price, $2950 [69], as the Xeon E5-4640 v2 CPU used in this study, and has 2× registers available per block [15] and 6.4× as many CUDA cores [70] as the Tesla C2075 used. Using a newer GPU model could significantly improve solver performance for larger models by relieving the scarcity of on-chip memory in a per-thread approach. Finally, a per-block approach may be required to efficiently integrate the largest models on the GPU, due to the much higher amount of cache memory allocated for each IVP solution.

The performance penalty due to thread divergence depends both on the cost of the divergent branches as well as the proportion of the warp that executes each branch. For example, if only one thread in a warp executes an expensive branch (e.g., a Jacobian update), the rest of the warp remains idle during that time, and the SM may become severely underutilized. To investigate the

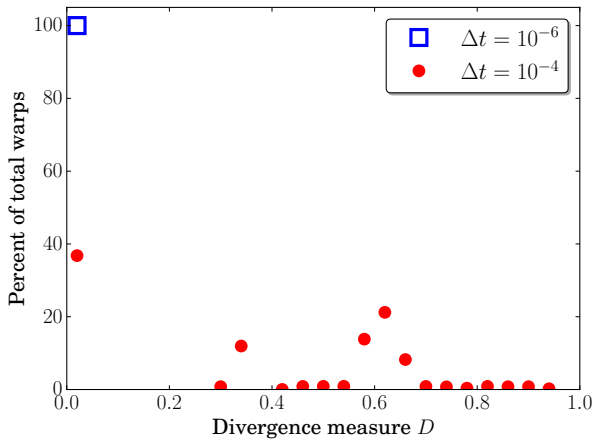
[1] Bisetti [37] demonstrated a method to exploit the underlying sparsity of a dense mass-fraction-based constant-pressure Jacobian matrix (used in this study) to accelerate Jacobian-vector multiplications; however, a reformulation is still more attractive as it enables sparse-LU factorization.

effects of thread divergence further, we adopted a modified version of the quantification of thread divergence of Niemeyer and Sung [22]:

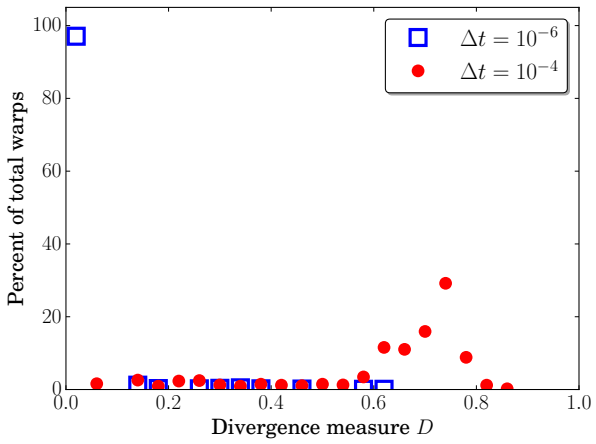
$$D = 1 - \frac{\sum_{i=1}^{32} d_i}{32 \times \max\limits_{i=1,\ldots,32} d_i} , \tag{3}$$

where d_i is the number of internal integrator time steps taken to reach the global time step by thread i in a warp (which consists of 32 threads). D represents the similarity of internal time step counts across threads in a warp—a significant source of thread divergence. If all threads in a warp use identical internal integration time steps and thus the warp experiences no thread divergence from this source, then $D = 0$; however, if a warp experiences an unbalanced number of internal integration time steps, then $D \to 1$. Differing internal time-step sizes are not the only source of thread divergence for the GPU integration algorithms. For instance, threads in a warp may use different Krylov subspace sizes for the exponential integrators or different numbers of Newton iterations for the `Radau-IIA` solver. Indeed, Sec. 3.1 notes that we suspect thread divergence from differing Krylov subspace sizes as the reason the exponential solvers are less efficient for small numbers of IVPs for the hydrogen model with the small global time-step size. However, these operations clearly cost less than an entire internal integration step (in which they are embedded) and thus we look only at the thread divergence of internal integration time steps. Thread divergence of such operations within an internal integration step could play an important role and will be investigated in our future work.

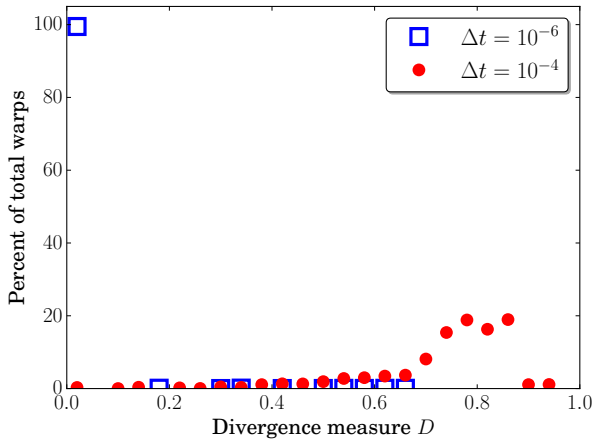
Figures 5a and 5b show the distribution of the divergence measure D for the `Radau-IIA` solver with both global time-step sizes and kinetic models when run on 262,144 IVPs, spread across 8192 warps. For both kinetic models with the smaller global time-step size, nearly 100 % of the warps had a divergence measure near zero. Increasing the global time-step size causes the number of warps with high levels of thread divergence (e.g. $D > 0.5$) to increase for both models. For the hydrogen model, over 40 % of warps were between $D = 0.55$ and $D = 0.65$, and the approximate equivalent CPU core-count (Table 3) dropped by 2.5 × between the small and large global time-step sizes. Further, over 75 % of warps were between $D = 0.6$ and $D = 0.8$ for the GRI-Mech 3.0 model for the larger global time-step size, and subsequently a higher drop in performance of 4 × occurred. This observation motivates future work aimed at developing strategies to reduce thread divergence. Potential solutions include adopting an IVP per-block approach [21], reordering IVPs to increase similarity of stiffness inside a warp, or synchronizing internal time-step sizes between threads in a warp. However, Figs. 5a and 5b do not explain the drop in equivalent core count between the hydrogen model and the GRI-Mech 3.0 model for the smaller global time-step size. The minimal



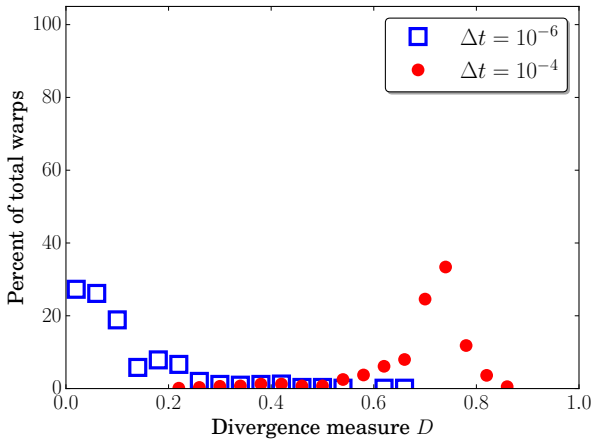
(a) `Radau-IIA` solver for hydrogen model



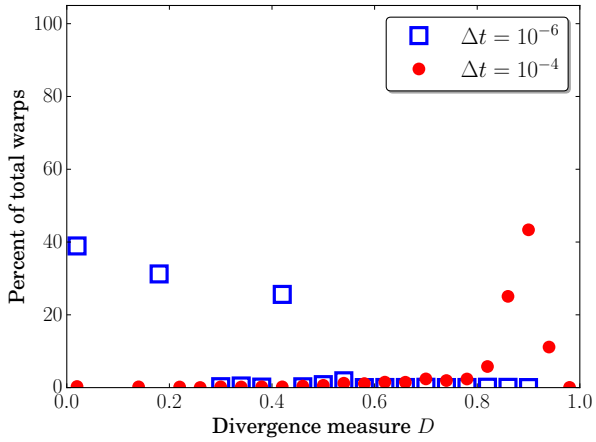
(b) `Radau-IIA` solver for GRI-Mech 3.0 model



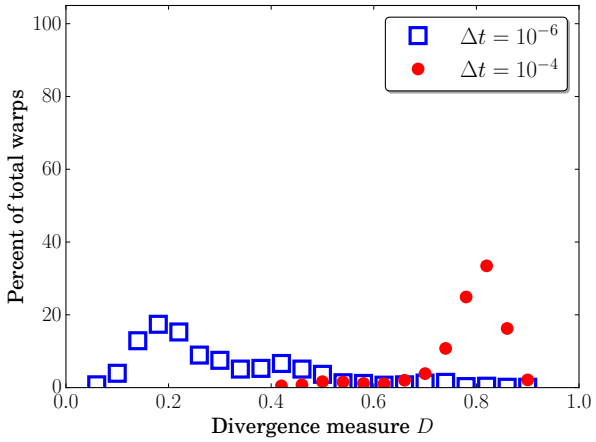
(c) `exprb43` solver for hydrogen model



(d) `exprb43` solver GRI-Mech 3.0 model



(e) `exp4` solver hydrogen model



(f) `exp4` solver GRI-Mech 3.0 model

Figure 5: Thread divergence estimate for the three solvers for both models and global time-step sizes. Data, plotting scripts, and figure files are available under CC-BY [13].

thread divergence of the `Radau-IIA` solver for both models at the smaller global time-step size suggests that this drop in performance is primarily caused by the increased memory traffic of the larger model, as well potential thread divergence inside the internal integration step; this further motivates development of a sparse version of the `pyJac` [39, 41] software.

Figures 5c and 5d show the divergence levels of the `exprb43` GPU solver. Similar to the `Radau-IIA` solver, nearly 100 % of warps for the `exprb43` solver have no thread divergence due to differing internal integration step sizes for the hydrogen model. The `exprb43` thread divergence levels increase somewhat for the GRI-Mech 3.0 model with the smaller time-step size; 27 % of warps still had a divergence measure of $D = 0$, but nearly 63 % of the warps had divergence measures between $D = 0.05$ and $D = 0.2$. With the larger time-step size, the `exprb43` solver experiences significantly more thread divergence for both models. The divergence measure distribution is fairly similar to that of the `Radau-IIA` solver for the GRI-Mech 3.0 model, but most warps experience a divergence measure of $D \sim 0.8$ for the hydrogen model (versus $D \sim 0.6$ for the `Radau-IIA` solver). The semi-implicit solvers deal with stiffness less efficiently, and end up using a greater range of internal time-step sizes between conditions of varying stiffness. This results in an increase in thread divergence levels due to differing internal time-step sizes.

The relatively worse stiffness handling of the `exp4` method is also apparent in Figs. 5e and 5f; in most cases, significantly more thread divergence is seen for `exp4` than for either of the other two solvers. The `exp4` algorithm is the only solver to show significant thread divergence even for the hydrogen model for the smaller global time-step size. Further, the `exp4` algorithm experiences more thread divergence than the `exprb43` for both models at the larger global time-step size.

3.4. Effect of using a finite-difference-based chemical kinetic Jacobian

While it is well established that using an analytical Jacobian matrix can significantly accelerate chemical kinetics integration on the CPU (e.g., [5, 68, 71]), relatively little study has been directed at use of a GPU-based analytical Jacobian. Dijkmans et al. [72] used a GPU-based analytical Jacobian code to accelerate various CPU-based chemical kinetics integration schemes, and our own previous works [40, 41] have detailed the performance of `pyJac`. However, to our knowledge no work using an analytical Jacobian for GPU-based chemical kinetics integration has been published. In this section, we explore the relative performance benefits of the analytical Jacobian compared with a first-order finite-difference Jacobian on both the CPU and GPU. The exponential methods require an exact Jacobian matrix (rather than an approximation as given by finite-difference methods), so

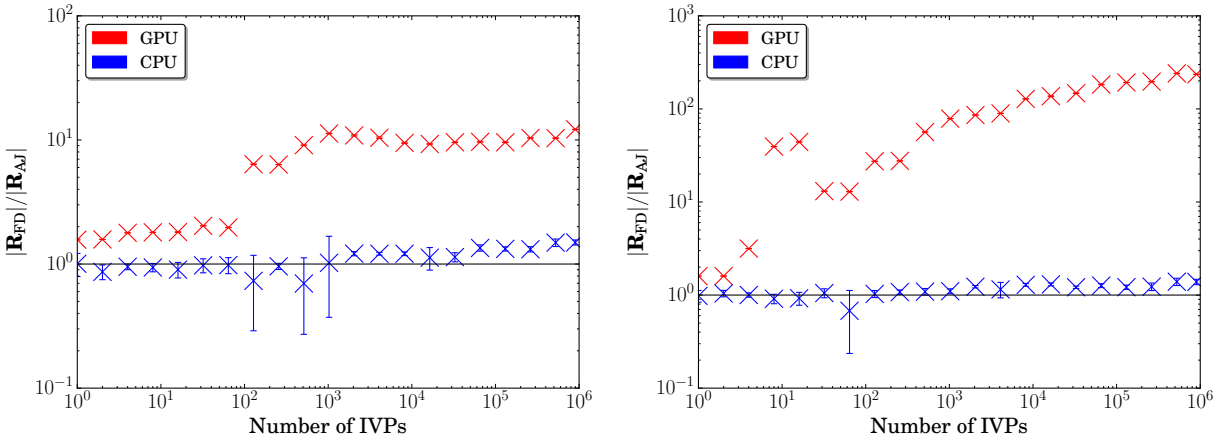
their performance was not considered in this section.

Figure 6 shows the speedup achieved on both the CPU and GPU for the `Radau-IIA` algorithm for various cases; the GRI-Mech 3.0 results for the larger global time-step size have been omitted due to long run times. For the hydrogen model (Figs. 6a and 6b), using the analytical Jacobian offers minimal performance benefit for the CPU-based integrators, reaching a maximum speedup of 1.49 × and 1.39 × for the small and large global time-step sizes, respectively. Our previous work [41] demonstrated that evaluation of the analytical Jacobian was 5.28 × faster on the CPU for the same chemical kinetic model; thus, the minor speedup seen here results from reuse of the Jacobian within the `Radau-IIA` solver, such that integration only requires a few Jacobian evaluations. In some cases the finite-difference Jacobian solver may be faster than the analytical Jacobian solver; although it is difficult to explain the exact cause of this phenomena, differences in the finite-difference Jacobian likely caused the integrator to follow a slightly different instruction path (e.g., with fewer Jacobian updates/chemical source term evaluations) changing the integration cost. However, for large numbers of conditions, the analytical-Jacobian-based CPU solver indeed performs faster than the finite-difference counterpart. In contrast, the analytical-Jacobian-based GPU solver performs significantly faster than the finite-difference GPU solver in all cases for the hydrogen model, reaching a maximum speedup of 12.16 × for the smaller global time-step size. As discussed in Sec. 3.3, significantly higher levels of thread divergence are expected for the larger global time-step size. Correspondingly, the maximum speedup of the GPU solver increases to 240.96 × for the larger global time-step size. Figure 6c shows that the speedup of the CPU and GPU solvers reach 2.61 × and 7.11 ×, respectively, for the larger GRI-Mech 3.0 model at the smaller global time-step size. It is clear that for a per-thread-based GPU integrator, using an analytical Jacobian is essential for efficient integration due to thread-divergence concerns.

4. Conclusions

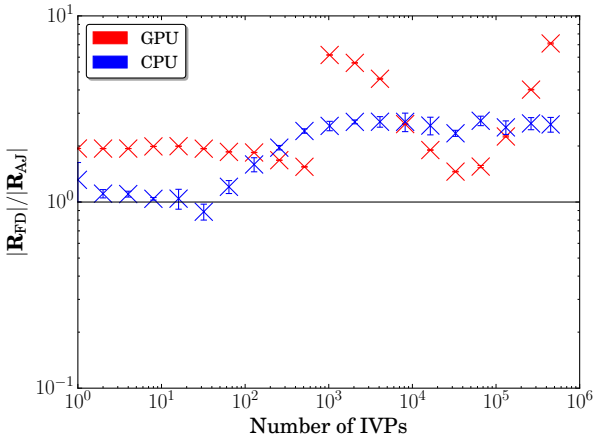
The large size and stiffness of chemical kinetic models for fuels traditionally requires the use of high-order implicit integrators for efficient solutions. Past work showed orders-of-magnitude speedups for solution of nonstiff to moderately stiff chemical kinetic systems using explicit solvers on GPUs [20, 22, 29]. In contrast, work on stiff chemical kinetics integration with implicit GPU solvers has been limited to specialized cases, or failed to surpass current CPU-based techniques.

This work demonstrated and compared the performances of CPU- and GPU-based integration methods capable of handling greater stiffness, including an implicit fifth-order Runge–Kutta al-



(a) Hydrogen model with $\Delta t = 1 \times 10^{-6}\,\mathrm{s}$

(b) Hydrogen model with $\Delta t = 1 \times 10^{-4}\,\mathrm{s}$



(c) GRI-Mech 3.0 model with $\Delta t = 1 \times 10^{-6}\,\mathrm{s}$

Figure 6: Ratio of the average finite-difference Jacobian based integrator runtime $|\mathbf{R}_{\mathrm{FD}}|$ to that of the analytical Jacobian runtime $|\mathbf{R}_{\mathrm{AJ}}|$ for the `Radau-IIA` (CPU/GPU) solvers. Error bars indicate standard deviation, and the horizontal lines show a ratio of one. Data, plotting scripts, and figure files are available under CC-BY [13].

gorithm and two fourth-order exponential integration algorithms, using chemical source term and analytical Jacobian subroutines provided by the `pyJac` software [39–41]. By comparing the performance of these algorithms using two chemical kinetic models, including hydrogen with 13 species and 54 reactions [53] and methane with 53 species and 325 reactions [54], and using two global time-step sizes (10^{-6} s and 10^{-4} s), we drew the following conclusions, q:

- For global time-step sizes relevant to large-eddy simulations (e.g., $\Delta t = 10^{-6}$ s), the GPU-based implicit Runge–Kutta method was roughly equivalent to the CPU-based implicit `CVODE` integrator running on 12–38 CPU cores.

- At larger global time-step sizes, the performances of all GPU-based integrators decreased significantly due to thread divergence.

- For a global time-step size relevant to Reynolds-averaged Navier–Stokes simulations (e.g., $\Delta t = 10^{-4}$ s), the GPU-based implicit Runga–Kutta solver performed equivalent to `CVODE` running on 15 cores for the hydrogen model, and just 3 cores for the GRI-Mech 3.0 model.

- The higher memory traffic required due to the size of the GRI-Mech 3.0 model significantly decreased GPU solver performance; a sparse analytical chemical kinetic Jacobian formulation must be developed to achieve high performance for still larger chemical kinetic models on the GPU.

- The exponential solvers were significantly less efficient than the implicit integrators on the CPU and GPU for all relevant cases.

- Using an analytical Jacobian matrix on the GPU is critical for efficient chemical kinetics integration due to thread divergence; speedups of 7.11–240.96 $\times$ over a finite-difference-approximation were reached on the GPU, far surpassing the corresponding CPU speedup of 1.39–2.61 $\times$.

Based on these results, we conclude that the exponential solvers poorly fit the SIMT acceleration paradigm due to high levels of thread divergence combined with the relatively high cost of integration steps due to Arnoldi iteration (as compared with other semi-implicit integration techniques). Instead, we recommend directing further focus on stiff semi-implicit solvers such as (non-exponential) Rosenbrock solvers, explored for the CPU by Stone and Bisetti [71], and inexact Jacobian W-methods [73, 74]. Further improvements to the analytical Jacobian code, e.g., by using

a chemical kinetic system based on species concentrations to increase Jacobian sparsity, are likely to further increase performance of the developed algorithms. Additionally, newer GPUs should be tested to examine the ability of larger cache sizes and more available registers to improve performance by reduction of slow global memory loads/stores; a per-block solution still may need to be adopted for efficient integration of larger chemical kinetic models. However, this work also showed that thread divergence poses a challenge to high performance of GPU-based integration techniques on a per-thread basis. Our future work will therefore include a more comprehensive study of thread divergence, as well as developing methods to mitigate or eliminate its negative performance impact. Finally, new integration techniques will be investigated and paired with work studying the selection of appropriate solvers based on estimated stiffness.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grants ACI-1534688 and ACI-1535065.

Appendix A.

The results for this paper were obtained using `accelerInt` v1.0-beta [43]. The most recent version of `accelerInt` can be found at its GitHub repository https://github.com/SLACKHA/accelerInt. All figures as well as the data and plotting scripts necessary to reproduce them, are available openly under the CC-BY license [13].

Supplementary material associated with this article includes unscaled plots of integrator runtimes and characterizations of the partially stirred reactor data for this work.

References

[1] C. V. Naik, K. V. Puduppakkam, A. Modak, E. Meeks, Y. L. Wang, Q. Feng, T. T. Tsotsis, Detailed chemical kinetic mechanism for surrogates of alternative jet fuels, Combust. Flame 158 (3) (2011) 434–445. doi:10.1016/j.combustflame.2010.09.016.

[2] S. M. Sarathy, C. K. Westbrook, M. Mehl, W. J. Pitz, C. Togbe, P. Dagaut, H. Wang, M. A. Oehlschlaeger, U. Niemann, K. Seshadri, P. S. Veloo, C. Ji, F. N. Egolfopoulos, T. Lu, Comprehensive chemical kinetic modeling of the oxidation of 2-methylalkanes from C_7 to C_{20}, Combust. Flame 158 (12) (2011) 2338–2357. doi:10.1016/j.combustflame.2011.05.007.

[3] M. Mehl, J.-Y. Chen, W. J. Pitz, S. M. Sarathy, C. K. Westbrook, An approach for formulating surrogates for gasoline with application toward a reduced surrogate mechanism for CFD engine modeling, Energy Fuels 25 (11) (2011) 5215–5223. doi:10.1021/ef201099y.

[4] O. Herbinet, W. J. Pitz, C. K. Westbrook, Detailed chemical kinetic mechanism for the oxidation of biodiesel fuels blend surrogate, Combust. Flame 157 (5) (2010) 893–908. doi:10.1016/j.combustflame.2009.10.013.

[5] T. Lu, C. K. Law, Toward accommodating realistic fuel chemistry in large-scale computations, Prog. Energy Comb. Sci. 35 (2) (2009) 192–215. doi:10.1016/j.pecs.2008.10.002.

[6] C. Huang, M. Yao, X. Lu, Z. Huang, Int. J. Therm. Sci. 48 (9) (2009) 1814 – 1822. doi:10.1016/j.ijthermalsci.2009.02.006.

[7] F. Bottone, A. Kronenburg, D. Gosman, A. Marquis, Flow, Turbul. Combust. 89 (4) (2012) 651–673. doi:10.1007/s10494-012-9415-y.

[8] A. A. Moiz, M. M. Ameen, S.-Y. Lee, S. Som, Combust. Flame 173 (2016) 123 – 131. doi:10.1016/j.combustflame.2016.08.005.

[9] A. C. Hindmarsh, R. Serban, `CVODE` v2.8.2, http://computation.llnl.gov/projects/sundials-suite-nonlinear-differential-algebraic-equation-solvers/download/cvode-2.8.2.tar.gz (Aug. 2015).

[10] NVIDIA, GPU Computing Clusters, http://www.nvidia.com/object/cuda_clusters.html.

[11] XSEDE, Resources Overview, https://www.xsede.org/resources/overview.

[12] Oak Ridge National Laboratory: Oak Ridge Leadership Computing Facility, Titan Cray XK7, https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/, accessed: 2017-01-30.

[13] N. J. Curtis, K. E. Niemeyer, C. J. Sung, Data, plotting scripts, and figures for "An investigation of GPU-based stiff chemical kinetics integration methods", Figshare, CC-BY license (2017). doi:10.6084/m9.figshare.4596847.

[14] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA, ACM Queue 6 (2) (2008) 40–53. doi:10.1145/1365490.1365500.

[15] NVIDIA, CUDA C programming guide, version 7.5, https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (Sep. 2015).

[16] C. P. Stone, K. E. Niemeyer, Accelerating finite-rate chemical kinetics with coprocessors: comparing vectorization methods on gpus, mics, and cpus, ArXiv e-printsarXiv:1608.05794.

[17] F. A. Cruz, S. K. Layton, L. A. Barba, How to obtain efficient GPU kernels: An illustration using FMM & FGT algorithms, Comput. Phys. Comm. 182 (10) (2011) 2084–2098. doi:10.1016/j.cpc.2011.05.002.

[18] A. R. Brodtkorb, T. R. Hagen, M. L. Sætra, Graphics processing unit (GPU) programming strategies and trends in GPU computing, J. Parallel Distrib. Comput. 73 (1) (2013) 4–13. doi:10.1016/j.jpdc.2012.04.003.

[19] K. E. Niemeyer, C. J. Sung, Recent progress and challenges in exploiting graphics processors in computational fluid dynamics, J. Supercomput. 67 (2) (2014) 528–564. doi:10.1007/s11227-013-1015-7.

[20] K. E. Niemeyer, C. J. Sung, C. G. Fotache, J. C. Lee, Turbulence-chemistry closure method using graphics processing units: a preliminary test, in: Fall 2011 Technical Meeting of the Eastern States Section of the Combustion Institute. doi:10.6084/m9.figshare.3384964.

[21] C. P. Stone, R. L. Davis, Techniques for solving stiff chemical kinetics on graphical processing units, J. Propul. Power 29 (4) (2013) 764–773. doi:10.2514/1.B34874.

[22] K. E. Niemeyer, C. J. Sung, Accelerating moderately stiff chemical kinetics in reactive-flow

simulations using GPUs, J. Comput. Phys. 256 (2014) 854–871. doi:10.1016/j.jcp.2013.09.025.

[23] F. Sewerin, S. Rigopoulos, A methodology for the integration of stiff chemical kinetics on GPUs, Combust. Flame 162 (4) (2015) 1375–1394. doi:10.1016/j.combustflame.2014.11.003.

[24] K. Spafford, J. Meredith, J. Vetter, J. Chen, R. Grout, R. Sankaran, Accelerating S3D: A GPGPU case study, in: Euro-Par 2009 Parallel Processing Workshops, LNCS 6043, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 122–131. doi:10.1007/978-3-642-14122-5_16.

[25] Y. Shi, W. H. Green, H.-W. Wong, O. O. Oluwole, Redesigning combustion modeling algorithms for the graphics processing unit (GPU): Chemical kinetic rate evaluation and ordinary differential equation integration, Combust. Flame 158 (5) (2011) 836–847. doi:10.1016/j.combustflame.2011.01.024.

[26] R. J. Kee, F. M. Rupley, J. A. Miller, Chemkin-II: A Fortran chemical kinetics package for the analysis of gas-phase chemical kinetics, Tech. rep., Sandia National Labs., Livermore, CA (1989).

[27] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users' Guide, 3rd Edition, SIAM, Philadelphia, PA, 1999.

[28] Y. Shi, W. H. Green, H.-W. Wong, O. O. Oluwole, Accelerating multi-dimensional combustion simulations using GPU and hybrid explicit/implicit ODE integration, Combust. Flame 159 (7) (2012) 2388–2397. doi:10.1016/j.combustflame.2012.02.016.

[29] H. P. Le, J.-L. Cambier, L. K. Cole, GPU-based flow simulation with detailed chemical kinetics, Comput. Phys. Comm. 184 (3) (2013) 596–606. doi:10.1016/j.cpc.2012.10.013.

[30] P. N. Brown, G. D. Byrne, A. C. Hindmarsh, VODE: a variable-coefficient ODE solver, SIAM J. Sci. Stat. Comput. 10 (5) (1989) 1038–1051. doi:10.1137/0910062.

[31] G. Wanner, E. Hairer, Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems, 2nd Edition, Springer-Verlag, Berlin, 1996. doi:10.1007/978-3-642-05221-7.

[32] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, C. S. Woodward, SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers, ACM T. Math. Softw. 31 (3) (2005) 363–396. doi:10.1145/1089014.1089020.

[33] F. Perini, E. Galligani, R. D. Reitz, A study of direct and Krylov iterative sparse solver techniques to approach linear scaling of the integration of chemical kinetics with detailed combustion mechanisms, Combust. Flame 161 (5) (2014) 1180–1195. doi:10.1016/j.combustflame.2013.11.017.

[34] M. J. McNenly, R. A. Whitesides, D. L. Flowers, Faster solvers for large kinetic mechanisms using adaptive preconditioners, Proc. Combust. Inst. 35 (1) (2015) 581–587. doi:10.1016/j.proci.2014.05.113.

[35] M. Hochbruck, C. Lubich, On Krylov subspace approximations to the matrix exponential operator, SIAM J. Numer. Anal. 34 (5) (1997) 1911–1925. doi:10.1137/S0036142995280572.

[36] M. Hochbruck, C. Lubich, H. Selhofer, Exponential integrators for large systems of differential equations, SIAM J. Sci. Comput. 19 (5) (1998) 1552–1574. doi:10.1137/S1064827595295337.

[37] F. Bisetti, Integration of large chemical kinetic mechanisms via exponential methods with Krylov approximations to Jacobian matrix functions, Combust. Theor. Model. 16 (3) (2012) 387–418. doi:10.1080/13647830.2011.631032.

[38] M. Falati, G. Hojjati, Integration of chemical stiff odes using exponential propagation method, J. Math. Chem. 49 (10) (2011) 2210–2230. doi:10.1007/s10910-011-9881-9.

[39] K. E. Niemeyer, N. J. Curtis, `pyJac` v1.0.2 (Jan. 2017). doi:10.5281/zenodo.251144.

[40] K. E. Niemeyer, N. J. Curtis, C. J. Sung, Initial investigation of `pyJac`: an analytical jacobian generator for chemical kinetics, in: Fall 2015 Meeting of the West. States Sect. Combust. Inst., 2015. doi:10.6084/m9.figshare.2075515.

[41] K. E. Niemeyer, N. J. Curtis, C. J. Sung, `pyJac`: analytical Jacobian generator for chemical kinetics, accepted to Comput. Phys. Comm. arXiv:1605.03262 [physics.comp-ph] (Jan. 2017).

[42] M. J. McNenly, R. A. Whitesides, D. L. Flowers, Adaptive preconditioning strategies for integrating large kinetic mechanisms, in: 8th US National Combustion Meeting, Park City, UT, 2013.

[43] N. J. Curtis, K. Niemeyer, `accelerInt` v1.0-beta (2017). doi:10.5281/zenodo.230256.

[44] M. Hochbruck, A. Ostermann, J. Schweitzer, Exponential Rosenbrock-type methods, SIAM J. Numer. Anal. 47 (1) (2009) 786–803. doi:10.1137/080717717.

[45] E. Banks, A. M. Collier, A. C. Hindmarsh, R. Serban, C. S. Woodward, `SUNDIALS` v2.6.2, http://computation.llnl.gov/projects/sundials-suite-nonlinear-differential-algebraic-equation-solvers/download/sundials-2.6.2.tar.gz (Aug. 2015).

[46] E. Gallopoulos, Y. Saad, Efficient solution of parabolic equations by Krylov approximation methods, SIAM J. Sci. Stat. Comp. 13 (5) (1992) 1236–1264. doi:10.1137/0913071.

[47] L. N. Trefethen, J. A. C. Weideman, T. Schmelzer, Talbot quadratures and rational approximations, BIT Numer. Math. 46 (3) (2006) 653–670. doi:10.1007/s10543-006-0077-9.

[48] K. E. Niemeyer, `cf_expm` v1.0 (Jan. 2016). doi:10.5281/zenodo.44291.

[49] M. Frigo, S. G. Johnson, The design and implementation of FFTW3, Proc. IEEE 93 (2) (2005) 216–231. doi:10.1109/JPROC.2004.840301.

[50] M. Frigo, S. G. Johnson, `FFTW` v3.3.4, http://www.fftw.org/ (Mar. 2014).

[51] G. W. Stewart, Matrix Algorithms: Volume 1: Basic Decompositions, SIAM, Philadelphia, 1998. doi:10.1137/1.9781611971408.

[52] Y. Saad, Analysis of some Krylov subspace approximations to the matrix exponential operator, SIAM J. on Numer. Anal. 29 (1) (1992) 209–228. doi:10.1137/0729014.

[53] M. P. Burke, M. Chaos, Y. Ju, F. L. Dryer, S. J. Klippenstein, Comprehensive H_2/O_2 kinetic model for high-pressure combustion, Int. J. Chem. Kinet. 44 (7) (2011) 444–474. doi:10.1002/kin.20603.

[54] G. P. Smith, D. M. Golden, M. Frenklach, N. W. Moriarty, B. Eiteneer, M. Goldenberg, C. T. Bowman, R. K. Hanson, S. Song, W. C. Gardiner, V. V. Lissianski, Z. Qin, GRI-Mech 3.0, http://www.me.berkeley.edu/gri_mech/ (1999).

[55] J.-Y. Chen, Stochastic modeling of partially stirred reactors, Combust. Sci. Technol. 122 (1–6) (1997) 63–94. doi:10.1080/00102209708935605.

[56] S. B. Pope, Computationally efficient implementation of combustion chemistry using in situ adaptive tabulation, Combust. Theor. Model. 1 (1) (1997) 41–63. doi:10.1080/713665229.

[57] Z. Ren, Y. Liu, T. Lu, L. Lu, O. O. Oluwole, G. M. Goldin, The use of dynamic adaptive chemistry and tabulation in reactive flow simulations, Combust. Flame 161 (1) (2014) 127–137. doi:10.1016/j.combustflame.2013.08.018.

[58] H. Robertson, The solution of a set of reaction rate equations, Numer. Anal.: An Introd. (1966) 178–182.

[59] B. N. Datta, Numerical Linear Algebra and Applications, SIAM, Philadelphia, PA, 2010.

[60] M. Hochbruck, A. Ostermann, Exponential integrators, Acta Numer. 19 (2010) 209–286. doi:10.1017/S0962492910000048.

[61] G. Iaccarino, A. Ooi, P. Durbin, M. Behnia, Reynolds averaged simulation of unsteady separated flow, Int. J. Heat Fluid Flow 24 (2) (2003) 147–156. doi:10.1016/S0142-727X(02)00210-2.

[62] H. Wang, S. B. Pope, Large eddy simulation/probability density function modeling of a turbulent jet flame, Proc. Combust. Inst. 33 (1) (2011) 1319–1330. doi:10.1016/j.proci.2010.08.004.

[63] G. Bulat, W. P. Jones, A. J. Marquis, Large eddy simulation of an industrial gas-turbine combustion chamber using the sub-grid PDF method, Proc. Combust. Inst. 34 (2) (2013) 3155–3164. doi:10.1016/j.proci.2012.07.031.

[64] J. A. Ramírez, C. Cortés, Comparison of different URANS schemes for the simulation of complex swirling flows, Numer. Heat Transf., Part B: Fundam. 58 (2) (2010) 98–120. doi:10.1080/10407790.2010.508440.

[65] E. Galloni, Analyses about parameters that affect cyclic variation in a spark ignition engine, Applied Thermal Engineering 29 (5–6) (2009) 1131–1137. doi:10.1016/j.applthermaleng.2008.06.001.

[66] Intel, Intel® Xeon® Processor E5-4640 v2 (20M Cache, 2.20 GHz), http://ark.intel.com/products/75288/Intel-Xeon-Processor-E5-4640-v2-20M-Cache-2_20-GHz, accessed: 06-06-2016.

[67] Amazon.com, Buying Choices: NVIDIA Tesla C2075 6GB GDDR5 PCIe Workstation Card, http://www.amazon.com/gp/offer-listing/B005OCMZ7A/ref=dp_olp_all_mbc?ie=UTF8&condition=all, accessed: 06-06-2016.

[68] D. A. Schwer, J. E. Tolsma, W. H. Green, P. I. Barton, On upgrading the numerics in combustion chemistry codes, Combust. Flame 128 (3) (2002) 270–291. doi:10.1016/S0010-2180(01)00352-2.

[69] Amazon.com, Tesla K40 Graphic Card - 1 GPUs - 745 MHz Core - 12 GB GDDR5 SDRAM , https://www.amazon.com/Tesla-K40-Graphic-Card-GDDR5/dp/B00KDRRTB8, accessed: 07-07-2016.

[70] NVIDIA, NVIDIA Tesla GPUs Datasheet, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities, accessed: 07-07-2016.

[71] C. P. Stone, F. Bisetti, Comparison of ODE solvers for chemical kinetics and reactive CFD applications, in: AIAA 52nd Aerospace Sciences Meeting (National Harbor, MD), 2014. doi:10.2514/6.2014-0822.

[72] T. Dijkmans, C. M. Schietekat, K. M. Van Geem, G. B. Marin, GPU based simulation of reactive mixtures with detailed chemistry in combination with tabulation and an analytical Jacobian, Comput. Chem. Eng. 71 (2014) 521–531. doi:10.1016/j.compchemeng.2014.09.016.

[73] T. Steihaug, A. Wolfbrandt, An attempt to avoid exact jacobian and nonlinear equations in the numerical solution of stiff differential equations, Math. of Comput. 33 (146) (1979) 521–534.

[74] B. A. Schmitt, R. Weiner, Parallel two-step w-methods with peer variables, SIAM J. on Numer. Anal. 42 (1) (2004) 265–282. doi:10.1137/S0036142902411057.