Title:
An investigation of GPU-based stiff chemical kinetics integration methods

Authors and Affiliations:

N. Curtis (860) 803-4660
C.J Sung (860) 486-3679
Department of Mechanical Engineering
The University of Connecticut
191 Auditorium Road, Unit 3139
Storrs, CT 06269-3139

K.E. Niemeyer (541) 737-5614
School of Mechanical, Industrial, and Manufacturing Engineering
Oregon State University

Corresponding Author:
N. Curtis
(860) 803-4660
Department of Mechanical Engineering
The University of Connecticut
191 Auditorium Road, Unit 3139
Storrs, CT 06269-3139
nicholas.curtis@uconn.edu

Colloquium:
REACTION KINETICS

Word Count:
Method 2
Formatted into two columns using elsarticle
With font sizes and spacing as specified in the Latex Author Instructions.
Six full pages $\times$ 900 words/page + 135 mm column $\times$ 2.2 words/mm = 5697 words.
Color Charges:

make grayscale

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 code `pyJac`. The performance of each was compared with the commonly used CPU-based implicit integrator `CVODE`. The implicit Runge–Kutta method performed over five times faster on a single GPU than `CVODE` on a single six-core CPU for an H_2/CO model, and up to 31.5 % faster than `CVODE` for the GRI-Mech 3.0 model. The exponential integration techniques performed less efficiently for all cases. The main limiter of GPU integrator performance was identified as thread divergence, and techniques to mitigate this issue were discussed. A novel shared memory caching strategy for the GPU was developed and shown to provide a 5–24 % performance increase to all GPU-based integrators. Additionally, an issue affecting storage of the chemical kinetic Jacobian in per-thread local memory was identified and work-around methods were discussed. Finally, future research directions were identified based on the current state-of-the-art of stiff chemical kinetics integration on the GPU.

Keywords: Chemical kinetics, Stiff chemistry, SIMD, GPU

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

1. Introduction

The need for accurate chemical kinetic models in predictive reacting-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 the hydrocarbon oxidation processes resulted in orders of magnitude increases in model size and complexity. For instance, a recently developed model for 2-methylalkanes, relevant for jet and diesel fuel surrogates, consists of over 7000 species and 30000 reactions [1] while a recent detailed gasoline surrogate mechanism contains over 1500 species and 6000 reactions [2]. Furthermore, kinetic models for large hydrocarbon fuels tend to exhibit chemical stiffness requiring implicit integration algorithms, for which the solution cost scales at best quadratically—and at worst cubically—with the number of species in a mechanism [3]. Lu and Law [3] extensively reviewed a number of techniques to accelerate chemical kinetics integration. In addition to methods discussed in their work, significant effort has been directed towards improvements of the integration techniques themselves.

Reactive-flow modeling codes commonly rely on high-order implicit integration techniques to solve the stiff governing equations posed by chemical kinetics models. These methods require repeated evaluation and factorization of the chemical kinetic Jacobian matrix in order to solve the associated non-linear algebraic equations through iterations of linear system solutions—the cost of which scales quadratically and cubically, respectively, with the number of species in a mechanism. However, significant cost savings in the Jacobian evaluation can be realized through the use of an analytic formulation, rather than the typical evaluation via finite difference approximations. This approach eliminates numerous chemical source term evaluations, and for a sparse Jacobian (e.g., formulated in terms of species concentrations) the cost of evaluation drops to a linear dependence on the number of species in the mechanism [3].

In this study, our efforts to accelerate chemical kinetics focus on improving the integration strategy itself, via development of new algorithms and use of high-performance hardware accelerators such as graphics processing units (GPUs) and other similar single instruction multiple data (SIMD) devices. 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 multi-core parallelism somewhat increased CPU performance, recently SIMD processors gained popularity as a low cost, low power consumption, and massively parallel high-performance computing alternative. GPUs were originally developed for graphics/video processing applications and consist of hundreds to thousands of separate cores, compared with the tens of cores found on typical CPUs. The SIMD parallelism model differs from a traditional CPU-based multi-threading model, with small per-core memory caches and acceleration resulting from executing the same instruction over multiple data. For more details, we refer the reader to several works that discussed these differences in depth [4–6].

A number of studies in recent years explored the use high-performance SIMD devices to accelerate (turbulent) reacting-flow simulations. Spafford et al. [7] investigated using GPUs to accelerate a turbulent combustion direct numerical simulation code, demonstrating a sub-order of magnitude speedup in evaluating the species production rates on the GPU. Shi et al. [8] used a GPU to evaluate species rates and factorize the Jacobian for the integration of independent kinetics systems, showing order-of-magnitude or greater speedups for large kinetic models. Niemeyer et al. [9] implemented an explicit fourth-order Runge–Kutta integrator for the GPU, and found a speedup of nearly two orders of magnitude with a non-stiff hydrogen mechanism. Shi et al. [10] implemented a stabilized explicit solver for the GPU and paired it with a CPU-based implicit solver that handled integration of the most-stiff chemistry cells in a three-dimensional premixed diesel engine simulation, demonstrating an overall speedup of 2–3×. Le et al. [11] implemented GPU versions of two high-order shock-capturing reacting flow codes, and found a 30–50× speedup over the baseline. Stone et al. [12] implemented the implicit VODE [13] solver for the GPU and achieved an order of magnitude speed-up over the baseline CPU version. Additionally, they showed that the implicit VODE algorithm exhibited significant thread divergence, as expected due to its complicated program flow (compared with an explicit integration scheme). Furthermore, Stone and Davis [12] showed that, for numbers of independent ODE systems greater than $\sim 10^3$, using individual GPU threads to handle independent chemical kinetic ODEs offered greater performance over using a block of GPU threads to cooperatively solve a single ODE [12]. Niemeyer and Sung [14] demonstrated an order-of-magnitude speedup for a GPU implementation of a stabilized explicit second-order Runge–Kutta–Chebyshev algorithm over a multi-core CPU implementation of VODE for moderately stiff chemical kinetics. They also investigated levels of thread

divergence due to differing integrator time-step sizes, and found it negatively impacts overall performance for dissimilar ODE initial conditions in a thread-block. Sewerin and Rigopoulos [15] implemented a three-stage/fifth-order implicit Runge–Kutta method [16] on a one-block per ODE basis on GPU, and found a 1.8× slowdown at best compared with the same on a single eight-core CPU.

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 SIMD parallelism of GPUs (along with similar hardware accelerators) and capable of handling stiffness. In this work we will investigate GPU implementations of several explicit and implicit integration techniques, as compared to their CPU counterparts and the baseline CPU `CVODE` [17] algorithm. Several previous works [18, 19] suggested so-called matrix-free methods—which 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—as potential improvements to the expensive linear-system solver required in standard implicit methods. Furthermore, Hochbruck and Lubich [20] 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 previously explored these so-called explicit exponential methods for applications in stiff chemical systems [21, 22] and found them stable for time-step sizes greatly exceeding the stability bound. In addition, their explicit nature makes them potentially much better suited for SIMD acceleration due to an expected reduction of thread divergence compared with implicit methods. Finally, the three-stage fifth-order implicit Runge–Kutta algorithm [16] investigated by Sewerin and Rigopoulos [15] will be studied to determine the impact of increasing chemical stiffness on the algorithm and the performance benefits of using an analytical Jacobian matrix, such as that developed by Niemeyer et al. [23, 24].

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 a shared memory caching algorithm will be outlined.

2.1. Integration techniques

We investigated GPU implementations of three integration methods in this work, comparing them against equivalent CPU versions and a CPU-only implicit algorithm. While we describe important details or changes here, full descriptions of all algorithms may be found in the cited sources. The `pyJac` software [23, 24] provided both chemical source term and analytical Jacobian subroutines for CPU- and GPU-based algorithms. For validation and performance assessments of `pyJac`, we direct readers to our previous work [24].

First, we used the `CVODE` solver [17] to provide the baseline performance of a typical CPU-based (maximum of fifth-order) implicit integration technique. In addition, we implemented 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 [16], called `Radau-IIA` here; the fourth-order exponential Rosenbrock-like method `exp4` of Hochbruck et al. [25]; and the newer fourth-order exponential Rosenbrock method [26], called `exprb43` here. For the exponential methods, we used the method of rational approximants [27] paired with the Carathédothy–Fejér method [28] to approximate the action of the matrix exponential on a vector, as suggested by Bisetti [21]. This technique relied on the external `FFTW3` library [29]. However, unlike the approach of Bisetti [21], we developed a custom routine based on the algorithm presented by Stewart [30] to perform the LU decomposition of the Hessenberg matrix resulting from the Arnoldi iteration. To ensure high performance of CPU-based methods, the Intel `MKL` library version 11.1.3 handled linear algebra (i.e., BLAS/LAPACK) operations. Next, we implemented GPU versions of the `Radau-IIA`, `exp4`, and `exprb43` methods; these follow the same descriptions as the CPU versions, but required specialized implementations of several BLAS and LAPACK methods, mostly related to LU factorization (or Hessenberg LU factorization for the exponential integrators). All GPU routines were developed using the NVIDIA CUDA framework [31, 32]. Finally, the adaptive time stepping procedures of all integrators used absolute and relative tolerances of 10^{-15} and 10^{-8} , respectively, throughout the work, and the exponential integrators used a rational approximant type of $(10, 10)$ as suggested by Bisetti [21].

2.2. Testing conditions

In order to validate and measure the performance of the integrators for realistic conditions, a database of thermochemical conditions covering a wide range of temperatures and species mass fractions was generated using a previously developed constant-pressure stochastic partially stirred reactor (PaSR) code [24]. We selected two chemical kinetic models to span the range of model sizes typically used in high-fidelity simulations: the H_2/CO model of Burke et al. [33] with 13 species and 27 reactions, and the GRI-Mech 3.0 model with 53 species and 325 reactions [34]. The PaSR simulations were performed at the conditions listed in Table 1 for ten residence times to reach a statistical steady state; Niemeyer et al. [24] described the PaSR simulation process in greater detail, which follows approaches used by others [35–38].

Parameter	H_2/air	CH_4/air
ϕ	1	
T_in	400, 600, and 800 K	
p_0	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 1: 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_0 is the pressure, N_p is the number of particles, τ_{res} is the residence time, τ_{mix} is the mixing time, and τ_{pair} is the pairing time.

2.3. Shared memory caching

One aspect of the GPU platform particularly important to high-performance algorithms are the unique memory access patterns; each streaming multiprocessor can access only a small amount of fast cache memory (typically 64 kB) and 32 bit registers (typically 32–64 kB). All other memory is stored globally on the device, with comparatively high latencies. Thus, it is critical to properly utilize the fast memory cache to achieve maximum performance on GPUs. When formulated on a per-block basis, GPU-based integration algorithms typically use this memory to store the current state vector and other commonly used values (e.g., the chemical Jacobian). However, on a per-thread basis the best use of this memory is less clear; this section will proposes one such approach for efficient use of this fast memory on a per-thread basis.

The largest issue with fast memory utilization on a per-thread basis is that it must be split between all threads in the blocks resident on a streaming multiprocessor. This fast memory is further split into a L1 cache, shared memory available for inter-thread communication within a block. Most versions of the CUDA compute standards allow the user to specify the relative allocation of the L1 cache and the shared memory pool. In our preliminary studies, a larger L1 cache proved to be the superior choice, to be expected for a per-thread approach where fast memory is at a premium, and memory accesses often resulted in much slower global memory read/writes. However, this leaves a significant portion of the cache (∼16 kB) as shared memory, with only around 2–4 double precision variables available per thread. The small amount of available memory poses challenges to use in the integration algorithm itself; however, evaluation of the chemical source terms and analytical Jacobian present opportunities for using the shared memory.

The presented algorithm describes a strategy to use this available shared memory to store commonly used species concentrations during evaluation of the reaction rates. Similar algorithms were developed for the species net production rates and Jacobian evaluation routines, using frequently updated species net production rates and frequently used reaction rates/species concentrations as the cached variables, respectively. As this algorithm is run during generation of the source code for the various subroutines, it introduces no computational or memory overhead related to determining the caching structure during runtime. Although this represents a relatively simple improvement, its use can achieve reasonable performance benefits, as will be seen in Sec. 3.2.

Algorithm: A procedure for memory caching during evaluation of reaction rates.

Initialize the cached species concentration set $C = \varnothing$
Let maximum set size be $C_{\max}$
for reaction r_i of reactions R **do**
  Let S_i be the set of participating species in r_i
  Let $P_{i,j}$ be the number of consecutive reactions starting from r_{i+1} in which each species $s_{i,j} \in S_i$ participates
  For each species $c_j \in C$ let L_j be the number of reactions since c_j directly participated in a reaction
  Priority sort the species $s_{i,j}$ in S_i in descending order by priority $P_{i,j}$ and store in S'_i
  for $s_{i,j}$ in S'_i **do**
    if $|C| < C_{\max}$ **then**
      Add $s_{i,j}$ to C
    else if $\max_j (L_j) \geq 2$ and $P_{i,j} > 1$ **then**
      Let $k = \text{argmax}_j (L_j)$
      Remove $s_{i,j}$ from S'_i and replace c_k in C with $s_{i,j}$
  for $s_{i,j}$ in S'_i **do**
    if $|S'_i| > 0$ and $|C| < C_{\max}$ **then**
      Add $s_{i,j}$ to C
  Compute ith reaction rate using values in C where appropriate.

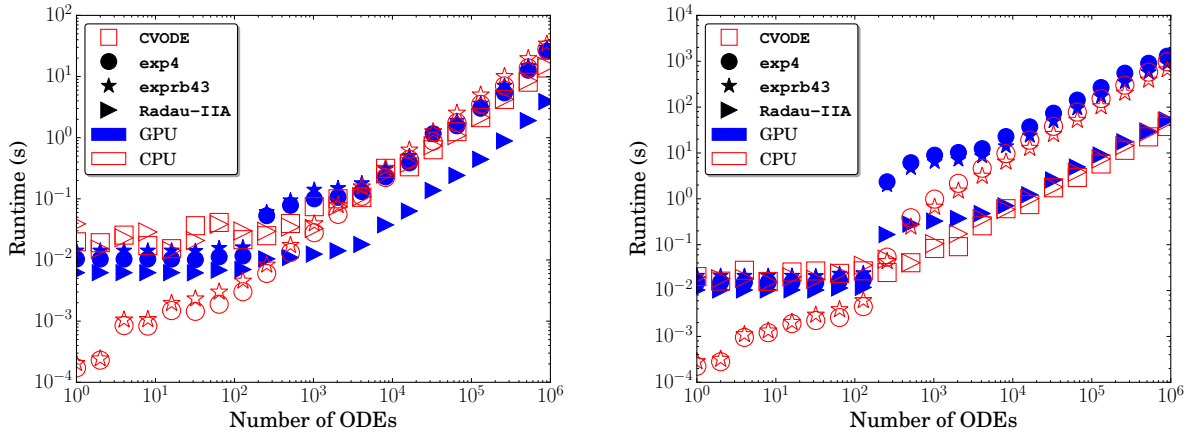
3. Results and discussion

To study the performance of the integrators, we tested each using the PaSR conditions described in Sec. 2.2 for two different global integration time-step sizes representative of those used in large eddy and Reynolds-averaged Navier–Stokes simulations: $\delta t = 10^{-6}$ s and $\delta t = 10^{-4}$ s, respectively. Furthermore, the use of a larger global time step induces additional stiffness for a given mechanism and enables evaluation of integrator performance at varying stiffness levels. Runtimes are reported as the average over five runs, where each run started from the same set of PaSR conditions. While the computed runtimes exhibited some variations, we chose to omit error bars from the plots for clarity; in general, the implicit methods exhibited variations within ±3 % and the exponential methods within ±1 %. All CPU integrators were compiled using `gcc 4.4.7` (with the compiler options "`-O3 -funroll-loops -mtune=native`"), and were executed in parallel via OpenMP on a six-core 2.67 GHz Intel Xeon X5650 with 256 kB of L2 cache memory per core and 12 MB of L3 cache memory. The same CPU served as host for the GPU integrators, which were compiled using `nvcc 7.0.27` (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 need to transfer data between the host CPU and GPU before and after each global time step; in addition, the function `cudaSetDevice()` was used to initialize the GPU before timing to hide any device initialization delay. The open-source `pyJac` software [23, 24] produced CPU and GPU custom source-code functions for the chemical source terms and analytical Jacobian matrix evaluation. In order to present unbiased speedup/slowdown numbers, in this section the best and worst performance will be computed for cases where the computational resources became fully utilized—i.e., when the runtime began to increase linearly with the number of ODEs; this occurred at 16,384 ODEs and above for both models. All of the GPU performance results shown in this section used the shared memory caching algorithm discussed in Sec. 2.3, the benefit of which will be explored in Sec. 3.2.

3.1. Performance

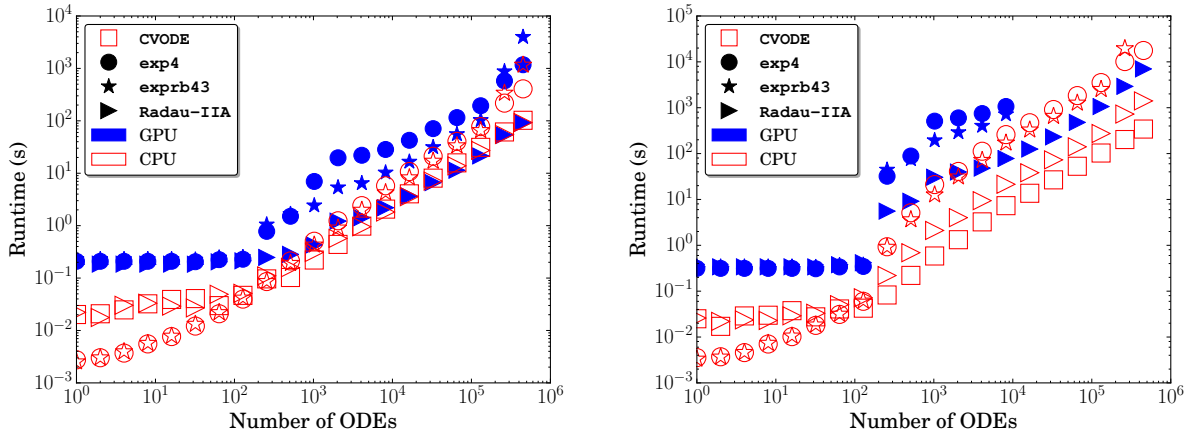
Figure 1 shows the runtimes of the CPU and GPU integrators for the H_2/CO model. The CPU `Radau-IIA` integrator was at best 14 % faster than `CVODE` (at 65,536 ODEs) and at worst 16 % slower (at 32,768 ODEs) for the smaller global time-step size. In the best case for the larger global time-step size, the CPU `Radau-IIA` solver was at best 3.8 % slower than `CVODE` (32,768 ODEs) and 35 % slower at worst (16,384 ODEs). For the smaller time-step size, the GPU `Radau-IIA` integrator was at best 5.31 × faster than `CVODE` (16,384

ODEs) and 3.45 × faster in the worst case (900,900 ODEs). Using the the larger time-step size negatively impacted the performance of the GPU `Radau-IIA` integrator—at best it ran 32 % slower (524,288 ODEs) than `CVODE`, and was 69 % slower in the worst case (65,536 ODEs). At the smaller time-step size the GPU `exprb43` integrator was at best 30 % faster than its CPU counterpart (262,144 ODEs) and was 13 % faster at worst (32,768 ODEs). The GPU `exp4` solver was 23 % faster the the CPU version at best (262,144 ODEs), and 32 % slower in the worst case (32,768 ODEs). Although the `exprb43` and `exp4` algorithms only require three exponential matrix function approximations, a single 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 CPU `exprb43` integrator is outperformed by the relatively more simple CPU `exp4` integrator at the smaller time-step size; by 22 % in the best case (900,900 ODEs) and 44 % in the worst case (65,536 ODEs). However, the nominal fourth-order convergence of the `exp4` algorithm is a classical non-stiff order, and thus order reduction is expected for stiff problems [21, 39]. Correspondingly, for the more stiff, larger time-step size case the CPU `exprb43` outperformed `exp4` by up to 33 % (65,536 ODEs) and 32 % in the worst case (524,288 ODEs).



(a) $\delta t = 1 \times 10^{-6}$ sec (b) $\delta t = 1 \times 10^{-4}$ sec

Figure 1: Average runtimes of the integrators for the H_2/CO model at two different global time-step sizes. CPU versions ran in parallel on six cores.



(a) $\delta t = 1 \times 10^{-6}$ sec (b) $\delta t = 1 \times 10^{-4}$ sec

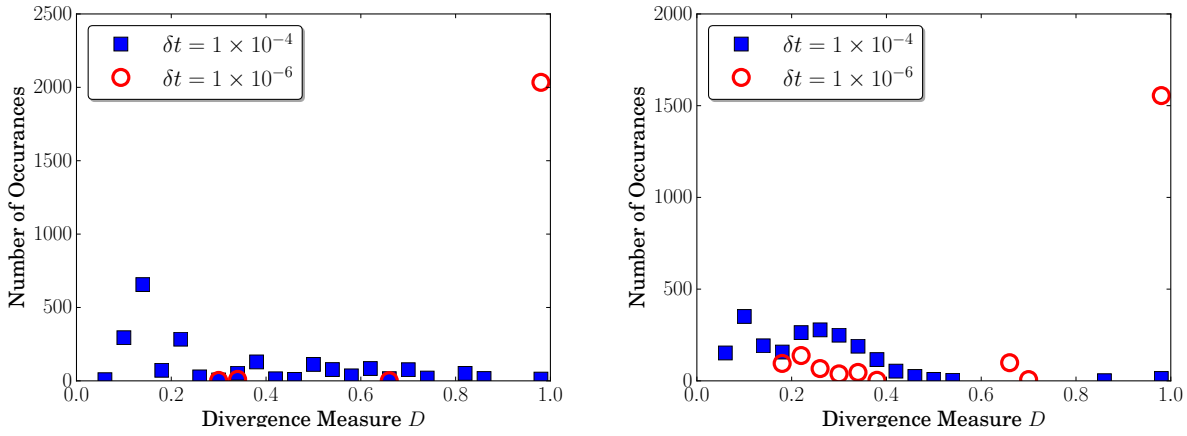
Figure 2: Average runtimes of the integrators for the GRI-Mech. 3.0 model at two different global time steps. CPU versions ran in parallel on six cores. Some of the longer-running exponential integration cases were omitted.

Figure 2 shows the runtime of the integrators for the GRI-Mech. 3.0 model. The GPU `Radau-IIA` integrator outperformed `CVODE` for the smaller time-step size; at best, it ran 31.5 % faster than `CVODE` (131,072 ODEs), and at worst was 10 % faster (16,384 ODEs). The larger time-step size again negatively impacted the performance of the GPU-based `Radau-IIA` integrator, dropping the performance to 8.63 × slower than `CVODE` at best (32,768 ODEs), and 20.48 × slower in the worst case (450,900 ODEs). For the smaller time-step size the CPU `Radau-IIA` integrator was 13 % faster than `CVODE` at best (131,072 ODEs) and 10 % faster at worst (450,900 ODEs). The larger time-step size—the stiffest case—was the only case where `CVODE` significantly outperformed CPU `Radau-IIA`, which was 2.66 × slower at best (65,536 ODEs) and 4.11 × slower at worst (450,900 ODEs). This difference may be due to the adaptive-order nature of the `CVODE` algorithm, or potentially due to its maturity and years of optimization. However, this effect is fairly minor and should not cause the order of magnitude decrease in relative performance between the GPU-based `Radau-IIA` integrator and `CVODE` observed when switching to the larger time step.

Two primary factors affect the performance of the GPU integration algorithms: chemical stiffness and thread divergence. To investigate this further, we adopted a similar quantification of thread divergence to that of Niemeyer and Sung [14]:

$$D = \frac{\sum_{i=1}^{32} d_i}{32 \max_{i \in 32} d_i} , \tag{1}$$

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). Thus, D represents the variation in number of internal time steps across all threads in a warp caused by differing internal integration time steps. If a warp has identical internal integration time steps and experiences no thread divergence, then $D = 1$; however, if a warp experiences an unbalanced number of internal integration time steps, then D will tend to zero.



(a) H_2/CO model (b) GRI model

Figure 3: Thread divergence measure of the `Radau-IIA` solver for both models and time-step sizes.

Figure 3 shows the distribution of the divergence measure D for the `Radau-IIA` solver with both time-step sizes and kinetic models when run on 65,536 ODEs, spread across 2048 warps. For both kinetic models, increasing time-step size resulted in a sharp increase in the number of warps with large thread divergence (i.e., $D < 0.5$). This implicates thread divergence, rather than chemical stiffness, as the main performance limiter for the GPU-based `Radau-IIA` integrator. Furthermore, increased thread divergence likely caused the relative performance difference of the GPU `Radau-IIA` solver between the H_2/CO and GRI models at the smaller time-step size. This observation motivates future work aimed at developing strategies to reduce thread divergence. Potential solutions include adoption of an ODE per-block approach [12], re-ordering of ODEs to increase similarity of chemical stiffness inside a warp, or time-step synchronization between threads in a warp.

Figure 2 also shows that the CPU exponential integrators are typically only 2–3× slower than `CVODE` for the smaller time-step size. As the exponential integrators must only factorize a much smaller Hessenberg

matrix (e.g., 5×5 for small integrator time-step sizes in stiff cases) rather than the entire Jacobian—as in the implicit integrators—significant savings in factorization can potentially be achieved with larger mechanism sizes. For larger integrator time-step sizes without severe chemical stiffness (e.g., near equilibrium) the size of the Hessenberg matrix to be factorized typically grows to 20×20. However, as stiffness increases, e.g., as for the larger time-step size, the stability characteristics of the implicit integrators make them more favorable. We hypothesize that the exponential integrators may be more efficient than the implicit integrators for moderately stiff systems (e.g., with small time-step sizes, or low tolerances) with larger chemical mechanisms. Prior work [14] showed that stabilized explicit integrators can provide significant speedups for moderately stiff systems. Although not shown in Fig. 3, the exponential integrators experienced significant thread divergence in all cases, and thus would also likely benefit from the thread divergence reduction techniques discussed above.

3.2. Effect of shared memory caching

The effectiveness of the shared memory caching scheme was evaluated in several cases by comparing the mean runtime of the integrators using rate and analytical Jacobian subroutines generated with and without the caching algorithm enabled. We observed a typical speedup of $\sim 5\,\%$ for the smaller global time-step size, and a $\sim 10\,\%$ speedup for the larger time-step size. This difference resulted from an increase in chemical source term and Jacobian evaluations required for the larger time-step size cases. Even larger speedup factors were observed in certain cases with the GRI model: a 24.4 % speedup with `exp4` for 131,072 ODEs and the 10^{-6} s time-step size, and a 13.2 % speedup with Radau-IIA for 32,768 ODEs and the 10^{-4} s time-step size. Thus, we recommend using this or a similar caching scheme in order to exploit shared memory in GPU integrators implemented on a per-thread basis.

3.3. Limitations of per-thread local memory for implicit GPU integration

Through the course of this work, we encountered a significant but under-discussed issue with the use of implicit and explicit exponential integrators on the GPU. One goal of GPU-based chemical kinetic integration is to enable use of larger detailed mechanisms in realistic reacting flow simulations. However, storage of the Jacobian on the GPU places severe restrictions on either the maximum allowable mechanism size or the total number of independent chemical-kinetic ODEs that can be solved concurrently. For instance, the `Radau-IIA` solver requires storage of $N_s \times N_s$ matrices for the Jacobian, LU factorization, and complex LU factorization, where N_s is the number of species in the mechanism. Similarly, the exponential integrators require storage of matrices for the Jacobian, the Hessenberg and vector subspace resulting from the Arnoldi process, and the corresponding Hessenberg exponential. While the Hessenberg, exponential Hessenberg , and vector subspaces may be smaller than the full Jacobian (e.g., by limiting the maximum Krylov subspace size), for the sake of simplicity in this analysis we assume full sizes. This implies that storage requirements per ODE solved concurrently scale as $\mathcal{O}\left(3 \times N_s^2\right)$ for the `Radau-IIA` solver and as $\mathcal{O}\left(4 \times N_s^2\right)$ for the exponential solvers.

Per-thread local memory, which resides in device global memory, offers one option for storing these matrices and other integrator variables. While this ensures coalesced memory access and simplifies indexing—both advantages from a programming standpoint—CUDA limits all threads to a maximum of 512 kB local memory, or 64,000 double precision floating point numbers. For example, even the USC Mech. II model for C_2H_4 [40] with 111 species caused the GPU to run out of local thread memory for our tests with all three GPU integrators.

Another option involves pre-allocating the Jacobian and associated matrices in global memory, and thus splitting the total global memory between all ODEs to be solved in a kernel launch. This work used a Tesla C2075 GPU, similar to that used by recent GPU-based chemical kinetic integration studies [8–12, 14]—with 6 GB of global memory and 10^6 independent ODEs, this works out to just 750 doubles available per concurrent ODE. Even assuming smaller problem sizes with 10^5 or 10^4 ODEs per kernel launch, each concurrent ODE is limited to just 7500 and 75,000 doubles, respectively. These limitations would severely constrain the allowable model sizes; for example, the `Radau-IIA` algorithm could handle models with approximately 15, 50, and 158 species for 10^6, 10^5, and 10^4 concurrent ODEs, respectively. Sewerin and Rigopoulos [15] discussed one approach to alleviate this issue: limiting the number of ODEs solved per kernel launch. However, for larger models, e.g., 250–500 species, this issue may negatively impact performance by forcing kernel launches small enough to under-utilize the GPU resources. Otherwise, mechanism reduction (a-priori or on the fly) may be employed to get around this complication.

4. Conclusions

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

This work demonstrated the performance of GPU-based integration methods, including an implicit fifth-order Runge–Kutta algorithm and two fourth-order exponential integration algorithms, using chemical source term and analytical Jacobian subroutines provided by the `pyJac` software [23]. For time-step sizes relevant to large eddy simulations, the GPU-based implicit Runge–Kutta method achieved a maximum speedup of over five compared with the CPU-based implicit `CVODE` integrator on a six-core CPU for a H_2/CO model, and outperformed to `CVODE` for the GRI-Mech 3.0 model by up to 31.5 %; the exponential integrators performed worse in all cases. For longer time-step sizes, the performance of all GPU solvers considered decreased significantly due to increased levels of thread-divergence. Furthermore, a shared memory caching algorithm was developed for the evaluation of chemical source terms and the analytical chemical Jacobian and resulted in modest performance benefits of 5–24 % speedup. In addition, an issue was identified with storage of the Jacobian and similar matrices in per-thread local memory.

When compared with the results of a similar integration technique by Sewerin and Rigopoulos [15], our results showed that using an analytical chemical Jacobian matrix greatly improves implicit integration speeds on the GPU. Further improvements to the analytical Jacobian code, e.g., by use of a chemical kinetic system based on species concentrations rather than mass fractions, are likely to further increase performance of the developed algorithms. However, this work showed clearly that thread divergence poses the largest challenge to high performance of GPU-based integration techniques on a per-thread basis. Therefore, our future work will focus on developing methods to mitigate and eliminate these effects. Finally, new integration techniques such as hybrid implicit/explicit solvers, and selection of appropriate solvers based on estimated chemical stiffness, will be investigated.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. 1534688 and 1535065.

[1] 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. F. Lu, Combust. Flame 158 (12) (2011) 2338–2357.

[2] M. Mehl, J.-Y. Chen, W. J. Pitz, S. M. Sarathy, C. K. Westbrook, Energy Fuels 25 (11) (2011) 5215–5223.

[3] T. Lu, C. K. Law, Prog. Energy Comb. Sci. 35 (2) (2009) 192–215.

[4] F. A. Cruz, S. K. Layton, L. A. Barba, Comput. Phys. Comm. 182 (10) (2011) 2084–2098.

[5] A. R. Brodtkorb, T. R. Hagen, M. L. Sætra, J. Parallel Distrib. Comput. 73 (1) (2013) 4–13.

[6] K. E. Niemeyer, C.-J. Sung, J Supercomput 67 (2) (2014) 528–564.

[7] 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.

[8] Y. Shi, W. H. G. Jr, H.-W. Wong, O. O. Oluwole, Combust. and Flame 158 (5) (2011) 836–847.

[9] K. E. Niemeyer, C.-J. Sung, C. G. Fotache, J. C. Lee, Turbulence-chemistry closure method using graphics processing units: a preliminary test, in: 7th Fall Technical Meeting of the Eastern States Section of the Combustion Institute, 2011.

[10] Y. Shi, W. H. Green, H.-W. Wong, O. O. Oluwole, Combust. and Flame 159 (7) (2012) 2388–2397.

[11] H. P. Le, J.-L. Cambier, L. K. Cole, Comput. Phys. Comm. 184 (3) (2013) 596–606.

[12] C. P. Stone, R. L. Davis, J. Propul. Power 29 (4) (2013) 764–773.

[13] P. N. Brown, G. D. Byrne, A. C. Hindmarsh, SIAM J. Sci. Stat. Comput. 10 (5) (1989) 1038–1051.

[14] K. E. Niemeyer, C.-J. Sung, J. Comput. Phys. 256 (2014) 854–871.

[15] F. Sewerin, S. Rigopoulos, Combust. Flame 162 (4) (2015) 1375–1394.

[16] G. Wanner, E. Hairer, Solving ordinary differential equations II, Vol. 1, Springer-Verlag, Berlin, 1991.

[17] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, C. S. Woodward, ACM T. Math. Software 31 (3) (2005) 363–396.

[18] F. Perini, E. Galligani, R. D. Reitz, Combust. Flame 161 (5) (2014) 1180–1195.

[19] M. J. McNenly, R. A. Whitesides, D. L. Flowers, Proc. Combust. Inst. 35 (1) (2015) 581–587.

[20] M. Hochbruck, C. Lubich, SIAM J. Numer. Anal. 34 (5) (1997) 1911–1925.

[21] F. Bisetti, Combust. Theor. Model. 16 (3) (2012) 387–418.

[22] M. Falati, G. Hojjati, J. Math. Chem. 49 (10) (2011) 2210–2230.

[23] K. E. Niemeyer, N. J. Curtis, C. J. Sung, `pyJac`, https://github.com/kyleniemeyer/pyJac (Aug. 2015).

[24] 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 Western States Section of the Combustion Institute, 2015.

[25] M. Hochbruck, C. Lubich, H. Selhofer, SIAM J. Sci. Comput. 19 (5) (1998) 1552–1574.

[26] M. Hochbruck, A. Ostermann, J. Schweitzer, SIAM J. Numer. Anal. 47 (1) (2009) 786–803.

[27] E. Gallopoulos, Y. Saad, SIAM J. Sci. Stat. Comp. 13 (5) (1992) 1236–1264.

[28] L. N. Trefethen, J. A. C. Weideman, T. Schmelzer, BIT Numer. Math. 46 (3) (2006) 653–670.

[29] M. Frigo, S. G. Johnson, Proc. IEEE 93 (2) (2005) 216–231.

[30] G. Stewart, Matrix Algorithms: Volume 1: Basic Decompositions, Soc. Ind. Appl. Math., 1998.

[31] J. Nickolls, I. Buck, M. Garland, K. Skadron, ACM Queue 6 (2) (2008) 40–53.

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

[33] M. P. Burke, M. Chaos, Y. Ju, F. L. Dryer, S. J. Klippenstein, Int. J. Chem. Kinet. 44 (7) (2011) 444–474.

[34] 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/.

[35] J.-Y. Chen, Combust. Sci. Technol. 122 (1–6) (1997) 63–94.

[36] S. B. Pope, Combust. Theor. Model. 1 (1) (1997) 41–63.

[37] Z. Ren, S. B. Pope, Combust. Flame 136 (1-2) (2004) 208–216.

[38] Z. Ren, Y. Liu, T. F. Lu, L. Lu, O. O. Oluwole, G. M. Goldin, Combust. Flame 161 (1) (2014) 127–137.

[39] M. Hochbruck, A. Ostermann, Acta Numer. 19 (2010) 209–286.

[40] H. Wang, X. You, A. V. Joshi, S. G. Davis, A. Laskin, F. Egolfopoulos, C. K. Law, http://ignis.usc.edu/USC_Mech_II.htm (May 2007).