Accelerating finite-rate chemical kinetics with coprocessors: comparing vectorization methods on GPUs, MICs, and CPUs

Christopher P. Stone[a,*], Andrew T. Alferman[b], Kyle E. Niemeyer[b]

[a] *Computational Science and Engineering, LLC*
Athens, GA 30605, USA
[b] *School of Mechanical, Industrial, and Manufacturing Engineering*
Oregon State University, Corvallis, OR 97331, USA

Abstract

Accurate and efficient methods for solving stiff ordinary differential equations (ODEs) are a critical component of turbulent combustion simulations with finite-rate chemistry. The ODEs governing the chemical kinetics at each mesh point are decoupled by operator-splitting allowing each to be solved concurrently. An efficient ODE solver must then take into account the available thread and instruction-level parallelism of the underlying hardware, especially on many-core coprocessors, as well as the numerical efficiency. A stiff Rosenbrock and a nonstiff Runge–Kutta ODE solver are both implemented using the single instruction, multiple thread (SIMT) and single instruction, multiple data (SIMD) paradigms within OpenCL. Both methods solve multiple ODEs concurrently within the same instruction stream. The performance of these parallel implementations was measured on three chemical kinetic models of increasing size across several multicore and many-core platforms. Two separate benchmarks were conducted to clearly determine any performance advantage offered by either method. The first measured the run-time of evaluating the right-hand-side source terms in parallel and the second benchmark integrated a series of constant-pressure, homogeneous reactors using the Rosenbrock and Runge–Kutta solvers. The right-hand-side evaluations with SIMD parallelism on the host multicore Xeon CPU and many-core Xeon Phi co-processor performed approximately three times faster than the baseline multithreaded C++ code. The SIMT parallel model on the host and Phi was 13 % to 35 % slower than the baseline while the SIMT model on the NVIDIA Kepler GPU provided approximately the same performance as the SIMD model on the Phi. The runtimes for both ODE solvers decreased significantly with the SIMD implementations on the host CPU (2.5–2.7 ×) and Xeon Phi coprocessor (4.7–4.9 ×) compared to the baseline parallel code. The SIMT implementations on the GPU ran 1.4–1.6 times faster than the baseline multithreaded CPU code; however, this was significantly slower than the SIMD versions on the host CPU or the Xeon Phi. The performance difference between the three platforms was attributed to thread divergence caused by the adaptive step-sizes within the ODE integrators.

Analysis showed that the wider vector width of the GPU incurs a higher level of divergence than the narrower Sandy Bridge or Xeon Phi. The significant performance improvement provided by the SIMD parallel strategy motivates further research into more ODE solver methods that are both SIMD-friendly and computationally efficient.

Keywords: Chemical kinetics, Integration algorithms, Stiff ODEs, SIMD, GPU

1. Introduction

Predicting turbulent combustion phenomena such as extinction and reignition with reactive computational fluid dynamics (CFD) simulations requires finite-rate chemical kinetics with detailed or reduced models. However, the computational costs of using these can overwhelm the available computer resources. High-fidelity combustion simulations with finite-rate kinetics must solve differential equations for the evolution of each species in the model in addition to the Navier–Stokes equations for momentum and energy. Detailed chemical kinetic models consist of hundreds (or more) of chemical species with thousands of elementary reactions, leading to intractable storage and computational costs. The computational cost is further increased by the stiffness of the ordinary differential equations governing chemical kinetics. For example, in H_2/air combustion, the time scales of induction (µs) and NO formation (ms) differ by a factor of 1000 [1]. This stiffness typically requires using implicit integration algorithms to solve the differential equations governing species evolution, the costs of which scale with the number of species cubed (in the worst case, associated with factorizing the Jacobian matrix) [2].

Operator splitting (e.g., Strang splitting) is commonly used to decouple the stiff chemical kinetics and nonstiff (or less stiff) convection-diffusion components of the conservation equations, as well as to reduce the size of the system of equations to be solved [3–10]. In this approach, the contribution of chemistry at each grid point is treated as an independent system of ordinary differential equations (ODEs) and integrated over the specified CFD time step. That is, a large partial differential equation (PDE) system is broken into a sequence of smaller ODE systems, one for each grid point. The species and the temperature equations are integrated in time using a constant-pressure or constant-volume assumption. The CFD time-step size must be relatively small to avoid large splitting errors caused by thermal expansion, diffusion, and convection. As a whole, solving all of the individual ODEs is far less expensive than a fully

*Corresponding author
Email addresses: chris.stone@computational-science.com (Christopher P. Stone), kyle.niemeyer@oregonstate.edu (Kyle E. Niemeyer)
URL: https://niemeyer-research-group.github.io (Kyle E. Niemeyer)

coupled PDE system. Yet even with this simplification, the computational cost of solving finite-rate kinetics via local initial value problems can consume 75–99 % of the runtime in CFD simulations [11–14].

Operator splitting provides a vast amount of parallelism since each ODE system can be solved concurrently. Several recent studies [15–22] investigated using graphics processing units (GPUs) (i.e., accelerators) to solve the ODEs in parallel. The kinetics ODEs are often solved with implicit backward differentiation formula (BDF) methods (e.g., VODE [23]). However, Stone and Davis [19] and Niemeyer and Sung [20] demonstrated that even moderately stiff ODEs can be efficiently integrated using explicit Runge–Kutta (RK) methods on GPUs by solving many ODE systems in parallel. For example, Stone and Davis reported a speedup of 23 × the baseline, single-core VODE CPU solver, while Niemeyer and Sung obtained a speedup of 57 × compared to a six-core OpenMP (CPU) VODE solver. The impressive RK performance on the GPU holds even when the single-core VODE solve is parallelized linearly across multiple cores (e.g., 16-core platform).

GPUs achieve high throughput rates by combining wide vector processing, high memory bandwidth, and fast thread context-switching to hide memory latency. NVIDIA CUDA-based GPUs implement the *single instruction, multiple thread* (SIMT) vector processing paradigm which allows up to 32 threads to execute the same operation concurrently on each processor. The high computational efficiency of the RK schemes reported above can be attributed to the fact that they have far fewer logical branches compared with the more elegant—and more complicated—BDF methods. This allows explicit RK methods to make more efficient use of the vector processing capabilities of the GPU, a major source of their performance. The high parallel efficiency of the RK methods can overcome their lower numerical efficiency under certain conditions.

Vector processing is a key performance feature of other many-core accelerator devices as well as most modern CPUs used in high-performance computing environments. For example, Intel Xeon Phi (MIC) accelerators have 512-bit *single instruction, multiple data* (SIMD) functional units within each core that can complete eight double-precision operations in parallel each cycle. Modern CPUs with 256-bit AVX (or AVX2) SIMD units can complete four double operations concurrently, and future Intel Xeon CPUs and Xeon Phi devices are expected to have similar 512-bit capabilities. Because of the high performance available from these HPC devices and the high cost of the ODE integration, adopting GPU-like, SIMD-friendly algorithms is desirable to achieve their full potential.

In this study, we compare two vectorization approaches for integrating the numerous ODE systems in parallel on modern multicore and many-core HPC platforms. Before presenting the parallel implementation, we first introduce two ODE integration algorithms well suited for SIMD parallel processing, and three common chemical kinetics models that will be used. We then present benchmark results using the models and discuss the performance using the various methods. Finally, we summarize our study and provide conclusions and some recommendations for future research.

2. ODE integration methods

ODE solvers seek to advance a set of N dependent variables $\boldsymbol{u}(t)$ through time t from an initial time t_i to a final time t_f through the action of the right-hand-side (RHS) function $\boldsymbol{f}(\boldsymbol{u})$. This can be expressed as

$$\frac{d\boldsymbol{u}}{dt} = \boldsymbol{f}(\boldsymbol{u}), \quad t_i \leq t \leq t_f \ . \tag{1}$$

Here, we have assumed that the ODE system is autonomous, i.e., $\boldsymbol{f}$ is not a function of t. A variety of techniques can be used to solve Eq. (1), but all methods advance $\boldsymbol{u}(t)$ to $\boldsymbol{u}(t+h)$, where h is an adjustable integration step size.

Integration algorithms are generally classified into two major categories: multistep and one-step methods. Multistep methods use past time steps (e.g., $\boldsymbol{u}(t-h), \ldots, \boldsymbol{u}(t-4h)$), while one-step methods start with only $\boldsymbol{u}(t)$. That is, one-step methods treat each integration step as a new integration problem. Both classes of methods adapt h in order to maintain the local truncation error (LTE) within a user-specified tolerance. Multistep BDF methods such as VODE [23] can also adjust the numerical order (p) of the method between time-steps to control the LTE. BDF methods start as first- or second-order and take several time-steps to reach their maximum order. One-step methods have a fixed order for all time steps. For further details on the taxonomy of ODE solvers and stability conditions, we refer readers to the book on numerical methods for stiff systems of ODEs by Hairer and Wanner [24].

The Runge–Kutta (RK) family of implicit and explicit methods are one-step methods widely used to solve both stiff and nonstiff ODE systems. A generic s-stage RK method for advancing the system $\boldsymbol{u}$ from time t_n to t_{n+1} is written as

$$\boldsymbol{u}_{n+1} = \boldsymbol{u}_n + \sum_{i=1}^{s} b_i \boldsymbol{k}_i \ , \tag{2}$$

where

$$\boldsymbol{k}_i = h\boldsymbol{f}\left(\boldsymbol{u}_n + \sum_{j=1}^{s} a_{ij}\boldsymbol{k}_j\right) \ , \tag{3}$$

and a_{ij} and b_i are the constant parameters that define the algorithm. Several types of RK methods exist, depending on the structure of the coefficient matrix a_{ij}. Explicit RK (ERK) methods are obtained when a_{ij} is strictly lower-triangular (i.e., $a_{ii} = 0$), fully implicit RK (FIRK) methods are obtained when a_{ij} is fully populated, and singly diagonally implicit RK (SDIRK) methods are a special case obtained when a_{ij} is lower triangular with $a_{ii} = \gamma$ (i.e., with a constant diagonal coefficient).

Examining Eq. (2), we see that the FIRK and SDIRK methods are implicit, i.e., $\boldsymbol{k}_i$ depends upon itself. This characteristic results in a system of non–linear equations commonly solved with the iterative Newton–Raphson method. The Newton iterative solver is a major expense for both implicit RK methods since

they must compute (or approximate) and factorize the Jacobian matrix $\boldsymbol{J}$ of the ODE system, i.e., $\boldsymbol{J} \equiv \partial \boldsymbol{f} / \partial \boldsymbol{u}$.

ERK methods are efficient for nonstiff problems but are only conditionally stable. As such, they are generally inefficient for stiff problems because the step-size h is limited by stability and not by the desired accuracy. ERK methods do not require the calculation of the Jacobian matrix (and the associated cost of solving the linear matrix systems) since they are fully explicit. This reduces the storage requirements and computational costs for each step considerably compared to implicit methods. The lower cost per-step of ERK may, at times, overcome the larger number of steps often required by explicit methods relative to implicit methods. Implicit methods can typically take step sizes on the order of the CFD application's time step.

In this study, we used the five-stage, fourth-order accurate embedded Runge–Kutta–Fehlberg (RKF45) ERK solver. Appendix B contains the RKF parameters $\boldsymbol{a}$, $\boldsymbol{b}$, $\hat{\boldsymbol{b}}$, and $\boldsymbol{c}$. The embedded fourth-order method ($p = 4$) is solved simultaneously with the fifth-order method; the difference between these two solutions is used to estimate the LTE and adapt h to meet the specified accuracy.

The Rosenbrock (ROS) family of one-step methods have much in common with RK methods. ROS can be described as solving a linearized version of Eq. (2). This leads to the following s-stage ROS scheme [24]

$$\boldsymbol{k}_i = h\boldsymbol{f}\left(\boldsymbol{y}_n + \sum_{j=1}^{i-1} \alpha_{ij}\boldsymbol{k}_j\right) + h\boldsymbol{J}\sum_{j=1}^{i} \gamma_{ij}\boldsymbol{k}_j \ , \quad i = 1, \ldots, s \tag{4}$$

$$\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + \sum_{i=1}^{s} b_i \boldsymbol{k}_i \ , \tag{5}$$

where α_{ij}, γ_{ij}, and b_i are the unique method coefficients. ROS methods are usually designed so that α_{ij} is lower triangular and each stage can be solved sequentially; in addition, $\gamma_{ii} = \gamma \ \forall \ i$. A direct implementation of Eq. (5) requires at each stage i the solution of a linear system with the matrix $I - h\gamma_{ij}\boldsymbol{J}$ for $\boldsymbol{k}_i$, which involves N^2 multiplications for $(\gamma_{ii}h)\boldsymbol{J}$, as well as the matrix-vector multiplication $J\sum \gamma_{ij}\boldsymbol{k}_j$. Transforming Eq. (5) eliminates these expensive operations:

$$\left(\frac{1}{h\gamma_{ii}}\boldsymbol{I} - \boldsymbol{J}\right)\boldsymbol{u}_i = \boldsymbol{f}\left(\boldsymbol{y}_n + \sum_{j=1}^{i-1} a_{ij}\boldsymbol{u}_j\right) + \sum_{j=1}^{i-1}\left(\frac{c_{ij}}{h}\right)\boldsymbol{u}_j \ , \quad i = 1, \ldots, s \tag{6}$$

$$\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + \sum_{i=1}^{s} m_j \boldsymbol{u}_j \ , \tag{7}$$

where

$$\boldsymbol{u}_i = \sum_{j=1}^{i} \gamma_{ij} \boldsymbol{k}_j \ , \quad i = 1, \ldots, s \ , \tag{8}$$

$$\boldsymbol{a} = \boldsymbol{\alpha} \boldsymbol{\Gamma}^{-1} \ , \tag{9}$$

$$\boldsymbol{c} = \gamma \boldsymbol{I} - \boldsymbol{\Gamma}^{-1} \ , \tag{10}$$

$$\boldsymbol{m} = \boldsymbol{b} \boldsymbol{\Gamma}^{-1} \ , \tag{11}$$

and $\boldsymbol{\Gamma} = (\gamma_{ij})$. We implemented the four-stage, fourth-order ROS4 scheme of Hairer and Wanner [24]—corresponding to their L-stable fourth-order Rosenbrock method, where $\gamma = 0.572816$—also available in the FATODE package [25]. Appendix C contains the ROS4 parameters $\boldsymbol{a}$, $\boldsymbol{c}$, $\boldsymbol{m}$, α_i, and γ_i needed to reimplement the method, although Fortran implementations are provided by Hairer and Wanner [24, 26] and Zhang and Sandu [25, 27].

The major distinction between the Rosenbrock and implicit RK methods lies in the role of the Jacobian matrix. $\boldsymbol{J}$ is only used to converge the nonlinear systems in the fully implicit schemes and is not part of the final solution. As such, $\boldsymbol{J}$ can be approximated or reused over many steps so long as the Newton iteration converges economically. Conversely, $\boldsymbol{J}$ appears explicitly in Eq. (5) and must be computed at each step in ROS methods. This requirement increases the computational cost of Rosenbrock methods if the construction and factorization of the Jacobian is costly.

The non-iterative nature of the ROS methods has several advantages from a parallel processing point of view. As discussed earlier, ERK methods perform favorably on GPUs primarily due to their simplicity and low level of divergence relative to the more complicated BDF methods. This allows their high vector parallel efficiency to overcome their lower numerical efficiency. Unlike ERK methods, ROS methods are L-stable and can handle stiff ODEs. Since they do not require any iterative solution, they can be implemented efficiently in a SIMD environment much like ERK methods. We implemented the fourth-order accurate ROS method (ROS4) to permit direct comparisons with the RKF45 method. For further details on the Rosenbrock method used here, see Hairer and Wanner [24] or Zhang and Sandu [25].

3. Chemical kinetics model

In this section, we introduce the model chemical kinetics problem that will be used throughout the performance benchmarks.

The following ODE system governs the time evolution of N_{sp} chemical species and energy for a constant-pressure, gas-phase combustion process at each grid

point or cell:

$$\dot{Y}_k = \frac{\dot{\omega}_k W_k}{\rho} \text{ and} \tag{12}$$

$$\dot{T} = -\frac{1}{c_p} \sum_{k=1}^{N_{\text{sp}}} h_k \dot{\omega}_k \ , \tag{13}$$

where Y_k, W_k, h_k, and $\dot{\omega}_k$ are the mass fraction, molar mass, enthalpy, and molar production rate for species k; and T, ρ, and c_p are the mixture temperature, density, and specific heat at constant pressure ($c_p = \sum_{k=1}^{N_{\text{sp}}} Y_k c_{p,k}$, where $c_{p,k}$ is the constant-pressure specific heat in mass units). Equations (12) and (13) are closed by the equation of state for an ideal gas, $p = \rho \overline{R} T$, where p is the thermodynamic pressure and $\overline{R}$ is gas constant of the mixture. A constant-volume process, i.e., where ρ is constant, can be modeled by replacing c_p with c_v and h_k with u_k in Eq. (13).

The net molar production rate terms ($\dot{\omega}$) are nonlinear functions of pressure p, T, and the species molar concentrations $[X_k]$. They are also the source of the stiffness in the ODE system and their calculation is generally the most computationally intensive component of the integration. They are expressed as

$$\dot{\omega}_k = \sum_{i}^{N_{\text{reac},k}} c_i \left(\nu''_{ki} - \nu'_{ki}\right) \left(k_{f,i} \prod_{j}^{N_{\text{sp},i}} [X_j]^{\nu'_{ji}} - k_{r,i} \prod_{j}^{N_{\text{sp},i}} [X_j]^{\nu''_{ji}} \right) \ , \tag{14}$$

where $N_{\text{reac},k}$ is the number of reactions involving species k, c_i is a parameter accounting for any third-body and/or pressure effects, ν'_{ki} and ν''_{ki} are the reactant and product stoichiometric coefficients for species k in reaction i, $N_{\text{sp},i}$ is the number of reactants and products in reaction i, and k_f and k_r are the forward and reverse reaction rate coefficients. Details regarding the third-body and pressure fall-off effects embodied in c_i are given by Niemeyer et al. [28].

The N_{reac} forward rate constants are given in Arrhenius form as

$$k_{f,i} = A_i \, T^{\beta_i} \, exp\left(-\frac{E_i}{\Re T}\right) \ , \tag{15}$$

where $\Re$ is the universal gas constant. If reaction i is irreversible, $k_{r,i}$ is zero. Explicit reverse Arrhenius rate coefficients can be given. Otherwise, they are computed as a function of the equilibrium constant $K_{c,i}$

$$k_{r,i} = \frac{k_{f,i}}{K_{c,i}} \ , \tag{16}$$

where $K_{c,i}$ is computed as

$$K_{c,i} = \left(\frac{p_0}{\Re T}\right)^{\sum_j^{N_{\text{sp}_i}} \nu_{ji}} K_{p,i} \tag{17}$$

$$K_{p,i} = \exp\left[\sum_{j}^{N_{\text{sp}_i}} \left(\nu''_{ji} - \nu'_{ji}\right) \left(\frac{S_j}{\Re} - \frac{H_j}{\Re T}\right)\right] \ , \tag{18}$$

where p_0 is the standard pressure at one atmosphere (in the appropriate units), and S_j and H_j are the standard-state entropy and enthalpy of species j in molar units. Temperature-dependent thermodynamic properties (e.g., $c_{p,j}$, H_j, S_j) are computed from polynomial fits using the following formulas:

$$\begin{aligned}
\frac{c_{p,j}}{R_k} &= \sum_{k=1}^{4} A_k T^{k-1} \\
\frac{H_j}{\mathfrak{R}} &= \sum_{k=1}^{5} A_k T^{k} + \frac{A_6}{T} \\
\frac{S_j}{\mathfrak{R}} &= A_1 \, \log(T) + \sum_{k=2}^{5} A_k T^{k-1} - \frac{A_6}{T} + A_7 \, .
\end{aligned} \tag{19}$$

The polynomial coefficients A_k are taken from the NASA seven-term polynomial [29] database. Two or more sets of coefficients are typically used, with each valid over a specified temperature range.

4. Parallel integrator implementations

As noted earlier, we wish to solve thousands of ODE systems concurrently on multicore and many-core devices. Before presenting the parallel implementation strategies, it is necessary to define terminology that spans the various HPC architectures. These devices offer two distinct levels of parallelism: multiple processing elements (e.g., multiple cores) and SIMD vector processing within each processing element. Vector instructions are issued by the processing elements and executed in SIMD parallel fashion across multiple data streams. In this paradigm, logical flow is controlled at the processing-element level and fine-grain data parallelism is implemented within each processing element. For this discussion a lane represents a single slot within the SIMD unit, logical threads issue vector instructions, one or more threads occupy a single processing element (e.g., *hyperthreading*), and threads may execute separate logical tasks.

The numbers of species (N_{sp}) in the chemical kinetic models of interest range between $O(10)$ and $O(100)$ and the number of reactions (N_{reac}) scales linearly (i.e., $N_{\text{reac}} \approx 5N_{\text{sp}}$ [2]). The number of independent ODE systems (N_{ode}) range from $O(10^4)$–$O(10^6)$ *per device* [30] in 3-D combustion simulations. Two SIMD parallel processing strategies can be designed based on these expected values.

In the first approach, a single thread solves multiple ODE systems together by mapping each ODE to a separate SIMD lane. For example, each lane evaluates Eqs. (12)–(19) with a unique set of y_k and T values at a given time. Multiple sets of ODEs can be integrated by separate threads on other PEs. Since vectorization is applied across only the breadth of the set of ODEs, we refer to this approach as *shallow* vectorization. This technique is analogous to the GPU/CUDA-specific *per-thread* approach demonstrated by Stone and Davis [19] and Niemeyer and Sung [20].

Conversely, in the second approach, a thread integrates a single ODE system but evaluates Eqs. (12)–(19) using the data parallel vector operations. For example, all N_{reac} forward rate coefficients given by Eq. (15) can be evaluated in parallel. More complex terms such as Eq. (14) require parallel reductions for each species k. In this approach, vectorization is applied through the depth of each ODE system. This method is therefore termed *deep* vectorization. This technique is analogous to the *per-block* (or *per-thread-block*) CUDA-specific method demonstrated by Stone and Davis [19]. As before, multiple ODEs are still evaluated concurrently across all of the available processing elements. For example, each OpenMP thread would solve a separate ODE system.

Stone and Davis [19] demonstrated these two vectorization techniques for a single chemical kinetic model [31, 32] with 19 species, 167 elemental reactions, and ten quasi-steady-state intermediate species on an NVIDIA C2050 (Fermi) GPU. They used CUDA implementations of the RKF45 and VODE BDF solvers to integrate hundreds of thousands of stiff ODE systems. They reported a 20.2 × speedup for CUDA-RKF45 but only 7.7 × with CUDA-VODE relative to the single CPU core VODE (CPU-VODE) runtime with the shallow vectorization approach. When normalized by the RKF45 solver on the CPU, the CUDA-RKF45 was 28.6 × faster with shallow vectorization. With the deep vectorization method, they reported speedups of only 10.7 × and 7.3 × with CUDA-RKF45 and CUDA-VODE, respectively, relative to CPU-VODE. They attributed the lower speedup of both solvers with deep vectorization to the small model, e.g., 19 species is much smaller than the effective SIMD width (32 lanes) on the CUDA device. They also attributed the superior vector efficiency of the RKF45 solver to its simplicity relative to the VODE. Niemeyer and Sung [20] similarly reported a speedup of 57 × using a CUDA GPU with shallow vectorization compared with execution on a six-core CPU. They used a stabilized Runge–Kutta–Chebyshev (RKC) ODE solver with a moderately stiff chemical kinetic model with 53 species and 634 irreversible reactions.

Both of these prior studies used CUDA GPUs. CUDA offers a particularly straightforward approach to implement the shallow vectorization paradigm. In the CUDA development environment, explicit SIMD instructions are not necessary. Instead, the runtime gathers sets of CUDA threads into *warps* and maps these warps to individual streaming multiprocessors. Threads within the same warp all execute the same instruction in lockstep following the SIMT paradigm. That is, CUDA threads map to lanes within the individual streaming-multiprocessor vector units. This effectively permits a serial implementation to be replicated across all vector lanes and all processors. As Stone and Davis [19] noted, implementing deep vectorization is much more complicated and requires explicit synchronization and communication among CUDA threads within the same *thread-block*, i.e., a collaborating team of warps.

This paradigm is largely inverted on modern multicore CPU environments including the Intel Xeon Phi, where a single CPU thread occupies the entire processing element. The CPU thread issues explicit SIMD instructions to enact operations across the parallel lanes of the vector units. Kroshko and Spiteri [33] demonstrated this approach in their SIMD implementation of a RODAS Rosen-

brock solver. There, they reported a speed-up of 1.89 × (i.e., 94 % parallel efficiency) when solving multiple systems of stiff IVPs on a cell broadband engine.

Explicit SIMD programs have historically been platform-dependent and their implementation has been quite difficult. Instead, most developers rely upon vectorizing compilers to identify parallel loops and automatically generate SIMD code for each target platform. Due to the structure of the ODE solver implementations and libraries, this approach generally results in an application following the deep vectorization paradigm. That is, each ODE system is solved by a single CPU thread and the compiler vectorizes loops with fixed (i.e., known) length. For example, the loop evaluating the forward reaction rate coefficients (Eq. (15)) for all N_{reac} can be vectorized. Implementing shallow vectorization on the Xeon Phi or the host CPU cores requires explicit SIMD programming, a more challenging parallel programming style. As demonstrated in the above citations, this approach appears to hold much promise for HPC platforms and may warrant the added implementation complexity.

For this study, the RKF45 and ROS4 ODE solvers were implemented using shallow vectorization with the OpenCL [34] language. OpenCL provides SIMD datatypes of varying lengths, e.g., two to sixteen doubles per SIMD superword, as well as traditional scalar datatypes. Scalar datatypes are suitable when relying upon compiler-generated vectorization or for SIMT (GPU) environments. OpenCL is also platform independent which allows performance studies across multiple platforms.

We did not explicitly implement deep vectorization in OpenCL. Instead, deep vectorization was realized through *guided* compiler vectorization of the original C/C++ implementation. Compiler vectorization directives were added to species and reaction loops within the chemical kinetics RHS function and matrix factorization and other loops within the ODE solvers to facilitate vectorization where necessary and appropriate.

As noted in Section 2, the RK and ROS algorithms are quite similar and largely share the same logical flow. Algorithm 1 represents both solvers using traditional scalar datatypes. The function `OneStep` advances $\boldsymbol{u}$ from t to $t + h$ using Eq. (2) or (5) for RK or ROS solvers, respectively. This function returns a trial solution $\boldsymbol{u}^*(t + h)$ and an approximation of the LTE. `AdjustStepSize` implements step-size size adaption based on the LTE approximation.

In contrast, Algorithm 2 shows the equivalent SIMD implementation. Several SIMD functions are introduced there and their meanings are:

Gather Read multiple scalar values from arbitrary locations into a single SIMD word.

Scatter Write the SIMD vector elements to arbitrary scalar locations.

Broadcast Replicate a scalar value across all SIMD vector elements or lanes.

Select(mask,a,b) Merge the SIMD words $\boldsymbol{a}$ and $\boldsymbol{b}$ based on the SIMD logical $\boldsymbol{mask}$. That is, for each *lane* k within the SIMD word, return a[k] if mask[k] evaluates True; otherwise, return b[k].

isLess Logically evaluate if the vector elements are less than a given value and return a logical SIMD mask.

isGreaterEqual Logically evaluate if the vector elements are greater than or equal to a given scalar value and return a logical SIMD mask.

Any Return True if any vector elements evaluate True; False otherwise. This is a SIMD *reduction* operation.

All Return True if all vector elements evaluate True; False otherwise.

The SIMD implementation in Algorithm 2 is equally valid for scalar datatypes (i.e., a SIMD word width of one is a scalar datatype), and the scalar and SIMD results are numerically equivalent (to within double datatype precision).

While complex and computationally intensive, the chemical kinetics rate calculations (Eq. (12)–(18)) require no SIMD reduction operations or collective logic tests. The exception is Eq. (19) since the temperature of each SIMD vector element (i.e., a *lane*) could lie within different polynomial fit ranges. That is, different polynomial coefficients are needed for different lanes. For two temperature ranges, we compute both polynomial equations for the desired thermodynamic quantity (e.g., C_p). A SIMD masked **Select** operation then selects the correct polynomial fit depending upon the lane mask. The number of valid temperature ranges is arbitrary [29], though two is common in practice. More than two temperature ranges would require a more complex SIMD algorithm.

Algorithm 1 Scalar RK and ROS solver algorithms to integrate N_{ode} independent ODE systems: Advance the initial state y_0 from t_i to t_f using step-size adaption to control the leading truncation error (LTE).

for $k \leftarrow 1, N_{\text{ode}}$ **do**
  $\boldsymbol{u}(t) \leftarrow \boldsymbol{y}_0[k]$
  $t \leftarrow t_i$
  $h \leftarrow$ `EstimateH0`$(\boldsymbol{u}(t),$ `MaxError`$)$
  while $t < t_i$ **do**
    $\boldsymbol{u}^*(t+h), err \leftarrow$ `OneStep`$(\mathbf{u}(t))$
    if $err <$ `MaxError` **then**
      $\boldsymbol{u}(t+h) \leftarrow \boldsymbol{u}^*(t+h)$
      $t \leftarrow t+h$
    end if
    $h \leftarrow$ `AdjustStepSize`$(err,\ h)$
  end while
  $\boldsymbol{y}[k] \leftarrow \boldsymbol{u}$
end for

A major distinction of the SIMD solver implementation is that the time-step iteration loop continues for all ODE systems grouped into the same SIMD data stream until all reach t_f. This can lead to inefficiency if the number of iterations varies significantly. Both Stone and Davis [19] and Niemeyer and Sung [20] found

Algorithm 2 SIMD RK and ROS solver algorithms to integrate N_{ode} ODE systems: All values are SIMD datatypes with W vector elements per word.

1: **for** $k \leftarrow 1, N_{\text{ode}}$ in increments of W **do**
2: $\quad \boldsymbol{u}(t) \leftarrow \textbf{Gather}(\boldsymbol{y}_0[k : k+W])$
3: $\quad t \leftarrow \textbf{Broadcast}(t_i)$
4: $\quad h \leftarrow \texttt{EstimateH0}(\boldsymbol{u}(t), \texttt{MaxError})$
5: $\quad$ **repeat**
6: $\quad\quad \boldsymbol{u}^*(t+h), err \leftarrow \texttt{OneStep}(\boldsymbol{u}(t))$
7: $\quad\quad mask \leftarrow \textbf{isLess}(err, \texttt{MaxError})$
8: $\quad\quad$ **if** $\textbf{Any}(mask)$ **then**
9: $\quad\quad\quad \boldsymbol{u}(t+h) \leftarrow \textbf{Select}(mask, \boldsymbol{u}^*(t+h), \boldsymbol{u}(t))$
10: $\quad\quad\quad t \leftarrow \textbf{Select}(mask, t+h, t)$
11: $\quad\quad$ **end if**
12: $\quad\quad h \leftarrow \texttt{AdjustStepSize}(err, h)$
13: $\quad\quad finished \leftarrow \textbf{isGreaterOrEqual}(t, t_f)$
14: $\quad$ **until** $\textbf{All}(finished)$
15: $\quad \boldsymbol{y}[k : k+W] \leftarrow \textbf{Scatter}(\boldsymbol{u})$
16: **end for**

that this phenomena can significantly degrade performance if the initial states of the ODE systems within the same SIMD stream differ widely.

Both Algorithms 1 and 2 were implemented using OpenCL scalar and SIMD datatypes, respectively. All computations were performed exclusively in double precision. OpenCL supports SIMD datatypes with 2, 4, 8, and 16 vector elements per word. This permits between 2 and 16 ODEs to be solved concurrently within each invocation of the ODE solvers. All SIMD functions discussed above are provided suitable gather/scatter operations. The scalar and SIMD algorithms were implemented separately despite their similarity due to incompatibility between the OpenCL scalar and SIMD functions and the lack of operator-overloading features.[1]

5. Results

We performed a series of benchmarks across three fundamentally different platforms and three different chemical kinetic models to assess the performance of the ODE solvers and the chemical kinetics rate evaluations within the SIMD context.

The platforms include a NVIDIA Kepler K20m GPU, an Intel Xeon Phi SE10P (MIC) coprocessor with 61 cores, and an Intel Sandy Bridge E5-2680 CPU with eight cores and two CPUs per compute node (for 16 total cores).

[1] At the time of this study, the OpenCL standard (1.2) supported SIMD vector loads/stores to contiguous memory locations but not strided variants. In addition, OpenCL did not support C++ within device code, which precluded operator-overloading or template functions.

The MIC and CPU OpenCL routines were compiled using the Intel OpenCL 1.2 SDK (v1.2.0.76921); Kepler routines used the NVIDIA OpenCL 1.1 driver (v331.67). The host driver application was built using the Intel C++ compiler (v14.0.1). Multithreaded (OpenMP) C++ implementations of the ODE solvers and RHS function evaluations were also compiled with the Intel C++ compiler for performance comparison.

Benchmarks on the MIC and GPU accelerators do not include communication time. The focus of this study is computational throughput on these devices and on the host devices, not specifically the acceleration over the host offered by these accelerators. All GPU benchmarks used 512 threads and 32 thread blocks per SMX. This gave the highest performance on the GPU for the three kinetic models. Also, all data is stored in *global* GPU memory; no *shared* or *constant* memory was used for these benchmarks due to their size restrictions.

Table 1: Details of the chemical kinetic models considered here.

Name	Fuel	N_{sp}	N_{reac}	Reference
H_2/CO	H_2	14	38	[35]
GRI Mech 3.0	CH_4	53	325	[36]
USC Mech II	C_2H_4	111	784	[37]

Table 1 shows details of the three chemical kinetic models considered in this study: the H_2/CO model of Davis et al. [35], GRI Mech 3.0 [36], and USC Mech Version II [37]. Most reactions are reversible and all three models contain both third-body and pressure-dependent reactions; specific details can be found in their associated references. As noted earlier, all thermodynamic polynomial curve fits (Eq. (19)) for these models use two temperature ranges.

The chemical kinetic models were interpreted using the `create_rate_subs` software [38], and the necessary species and reaction rate information saved to a binary database in turn read by the source term functions. Jacobian matrices (needed for the ROS integrator) are constructed using first-order forward finite differences, following the approach used in CVODE [23]. Evaluating each Jacobian thus requires $N_{\text{sp}} + 1$ RHS function evaluations, in addition to one RHS evaluation per stage.

5.1. Performance of RHS evaluation

The first set of benchmarks studied the throughput of RHS function evaluations. Figure 1 shows the average runtimes for one million RHS evaluations with the GRI Mech 3.0 model on the host CPU, MIC coprocessor, and the Kepler GPU using the SIMD and thread-parallel OpenCL implementations. These benchmarks show the best performing configurations for each device. For the SIMD host and MIC benchmarks, the most efficient word length was twice the native size, i.e., eight-wide on the host.

The unique thermochemical input states for each RHS evaluation were generated by setting a uniform composition for all species but linearly varying the

temperature over 500–1500 K to cross the polynomial temperature ranges. This requires evaluation of all temperature–dependent branch statements, such as the polynomial curve fits for specific heat, which reduces the SIMD (or SIMT) parallel efficiency. Even with this forced divergence, the explicit SIMD implementations improve the runtime by a factor of 3.1 on the host CPU and 3.3 on the MIC. The OpenCL thread-parallel runtimes are considerably slower than with the OpenCL SIMD method and are slower than the OpenMP baseline. The OpenMP baseline on the host used 16 threads and 240 threads[2] on the MIC. The thread-parallel implementation on the Kepler GPU gives favorable performance and is 2.3 times faster than the baseline host runtime. This indicates that the thread-parallel approach on the SIMD platforms (i.e., the host CPU and MIC) is inefficient for this type of application. However, the thread-parallel SIMT method is most efficient on the GPU.

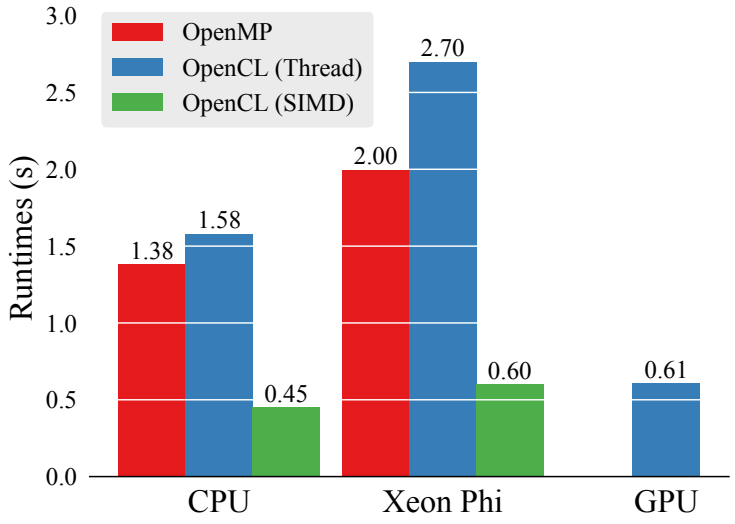


Figure 1: Wall-clock runtimes (seconds) for one million RHS evaluations of the GRI Mech 3.0 model on the host CPU, MIC, and Kepler GPU using the OpenCL SIMD (CL-SIMD) and thread-parallel (CL-Thread) implementations. OpenMP uses sixteen threads with compiler auto-vectorization. Data, plotting scripts, and figure file are available [39].

Figure 2 shows the runtimes (from Figure 1) normalized by the host runtime with OpenMP and auto-vectorization. This gives the relative performance compared to the host baseline. The GPU and MIC (with CL-SIMD) both give nearly identical speedup (2.3 ×) over the baseline. These accelerators perform

[2]Each MIC core can support four hardware threads, and one core is reserved for the MIC operating system.

well compared with the baseline; however, the significant host improvement from CL-SIMD means that the host is still faster by nearly 25 %.

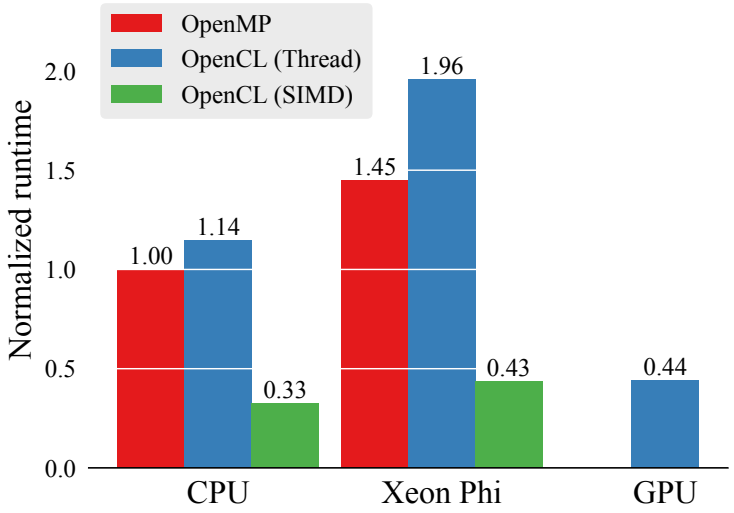


Figure 2: Normalized runtimes (speedup) for one million RHS evaluations of the GRI Mech 3.0 model on the host CPU, MIC, and Kepler GPU using the OpenCL SIMD (CL-SIMD) and thread-parallel (CL-Thread) implementations. Runtimes are normalized by the OpenMP runtime on the host CPU using compiler auto-vectorization. Data, plotting scripts, and figure file are available [39].

Figure 3 shows RHS runtimes for all three models studied using the SIMD method for the CPU and MIC and the thread-parallel method for the GPU (i.e., the fastest method for each device). For the larger two models, the relative performance between the host and the accelerators is nearly the same. However, the GPU and host perform equivalently for the smaller H_2/CO model.

These results show that the SIMD method is quite effective for evaluating the RHS function. The RHS function is complex; however, as noted earlier, only the thermodynamic polynomials are able to diverge across SIMD lanes.

5.2. Performance of ODE time integration

Let us now shift to applying this approach to time integration of the ODE systems described above. The kinetic reaction rate evaluation function is the primary computational expense since it is used both for the RHS function and for generating the system Jacobian with finite-differences. Unlike the previous experiment, the potential for lane divergence increases when mapping an ODE system to each lane.

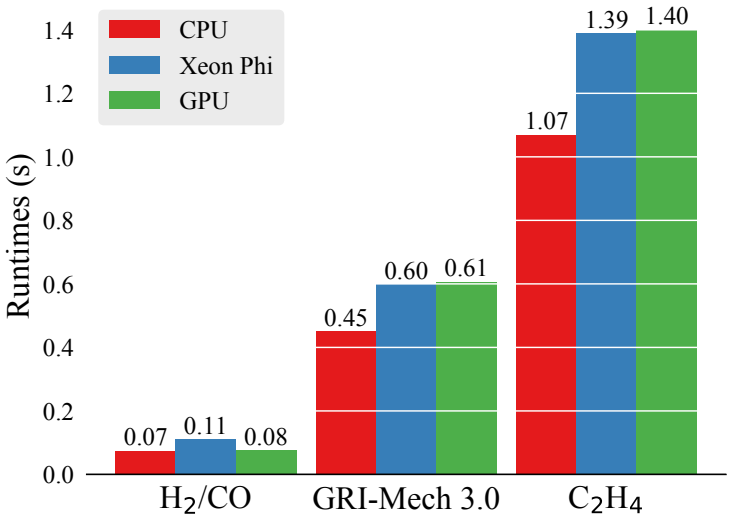


Figure 3: Wall-clock runtimes (seconds) for one million RHS evaluations for all three models in Table 1 using the SIMD method (CL-SIMD) for the CPU and MIC and the thread-parallel method (CL-Thread) for the GPU. Data, plotting scripts, and figure file are available [39].

The RKF45 and ROS4 solvers are single-step methods with a fixed cost per-step. The number of steps needed to solve the ODE system may vary between systems, which causes divergence of severity depending highly on the problem.

The parallel ODE solvers were applied to state data obtained from a stationary one-dimensional premixed methane/air flame simulation. The initial flame profile was computed using the GRI Mech 3.0 model [36] and Cantera's `FreeFlame`, a simulation tool for modeling freely propagating flat flames [40], and then interpolated onto a uniform mesh with 1601 points. The unburned temperature at the left boundary is 300 K and the equivalence ratio of the fresh reactants is 0.67. The resulting state data are available openly [41]. This spatial resolution (i.e., approximately 25 mesh points across the thermal flame thickness [42]) compares with that needed for a direct numerical simulation of complex phenomena such as flame-turbulence interaction. To mimic an operator-splitting framework, the ODE systems at each mesh point are integrated independently over a fixed time of 1 µs. This time interval represents a feasible convective time-step size for semi-implicit [14] CFD methods.

Figure 4 shows the temperature profile normal to the flame, where the thin reaction zone is evident. The number of *attempted* integrator steps (i.e., accepted and rejected steps) needed for the RKF45 and ROS4 ODE solvers are shown as well. Both ODE solvers use the same absolute (10^{-8}) and relative (10^{-11}) tolerances, the same initial time-step size (h_0) estimation, and the same

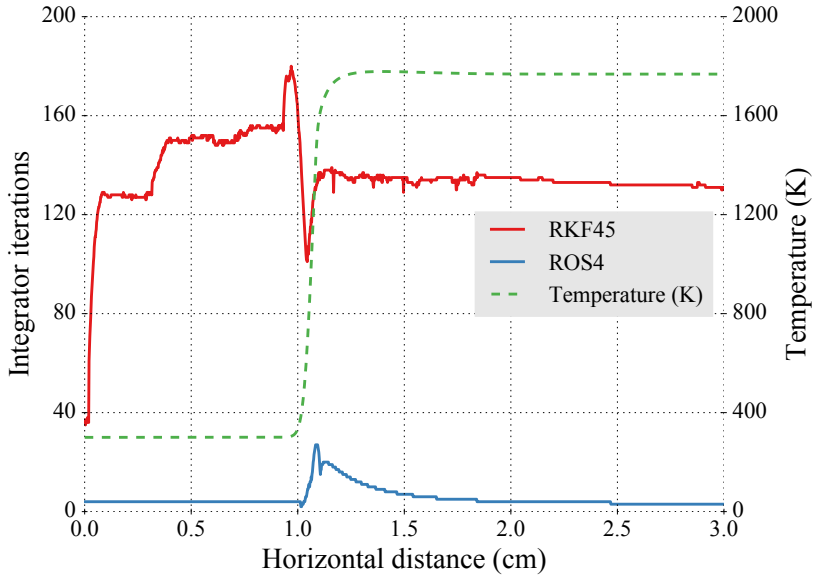


Figure 4: Temperature profile normal to the flame and the number of integrator steps needed for RKF45 and ROS4 ODE solvers. Data, plotting scripts, and figure file are available [39].

h-adaption algorithm. The differences in stability characteristics of the two ODE solvers cause the differences in number of steps. The L-stable ROS4 solver can quickly solve the largely non-reactive zones upstream and downstream of the flame with only a minimal, and largely constant, number of integrator steps. Only in the thin flame itself does the solver need more than this minimum. The number of steps needed by RKF45 fluctuates but is, in general, higher regardless of local conditions. This results from the stiff conditions of the kinetics problem. In this scenario, stability limits h, rather than local error as in the case of the stiff ROS4 solver. The cost per step is not equal between RKF45 and ROS4 since the latter must also construct and factorize the Jacobian matrix. For the GRI Mech 3.0 model, ROS4 requires $10\times$ more RHS function evaluations per step. The RHS ratio is a good estimation of the overall per-step cost ratio of the two methods.

To mimic the cost of a multidimensional reactive CFD simulation, the one-dimensional domain shown in Figure 4 is replicated 10, 40, 100, and 200 times vertically to give approximately 16×10^3, 48×10^3, 160×10^3, and 320×10^3 points. These sizes were selected to approximate mesh sizes that may be encountered on a per-core (e.g., 16×10^3) and per-device (e.g., 320×10^3) basis.

Figures 5 and 6 show the RKF45 and ROS4 runtimes, respectively, for the OpenMP and CL-SIMD methods on the host and MIC accelerator for the model problem sizes. The GPU runtimes with CL-thread are also shown for comparison. The fastest SIMD runtimes are shown based on the word size. The RHS

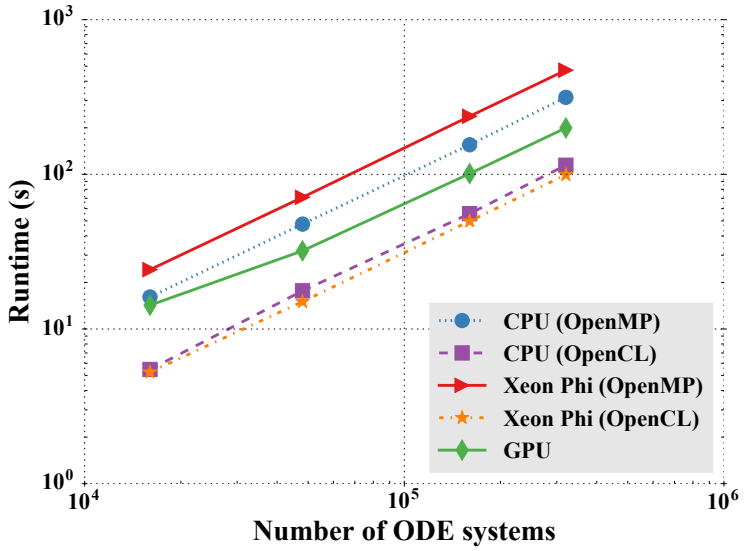


Figure 5: Comparison of runtimes of the RKF45 solver for the model problem, between OpenMP and CL-SIMD on the host CPU and Xeon Phi, and the GPU. Data, plotting scripts, and figure file are available [39].

runtimes all used twice the native word size. Here, the fastest host performance was found with 16-wide SIMD words (i.e., four times the native word size) while the MIC was fastest using the native size, eight-wide.

The high cost of the ODE integration with RKF45 is evident even for the smallest problem size. There, tens of seconds are needed for the reaction integrations on the host using the baseline OpenMP method. (This cost is incurred at least once per global CFD time-step and many thousands of steps may be needed.) The ROS4 solver is more efficient and consistently 2.5 $\times$ faster on the host on the model problem. The ROS4 to RKF45 runtime ratio is identical on the MIC accelerator using OpenMP.

The runtimes scale linearly with the number of ODE systems solved on the host and MIC accelerator using OpenMP. This is not unexpected as the number of ODE systems solved concurrently on these devices using OpenMP is small relative to the total problem size; that is, the parallelism is small relative to the total problem size.

The ODE systems require different numbers of iterations (see Figure 4) and this leads to variability in the computational cost. Dynamic loop scheduling, with granularity of one, was used with OpenMP to account for the variable costs. A strategy was implemented in all OpenCL versions of the integrators to mimic this type of dynamic scheduling. Here, a simple queue was created using a global counter incremented atomically (i.e., lock-free) by each parallel

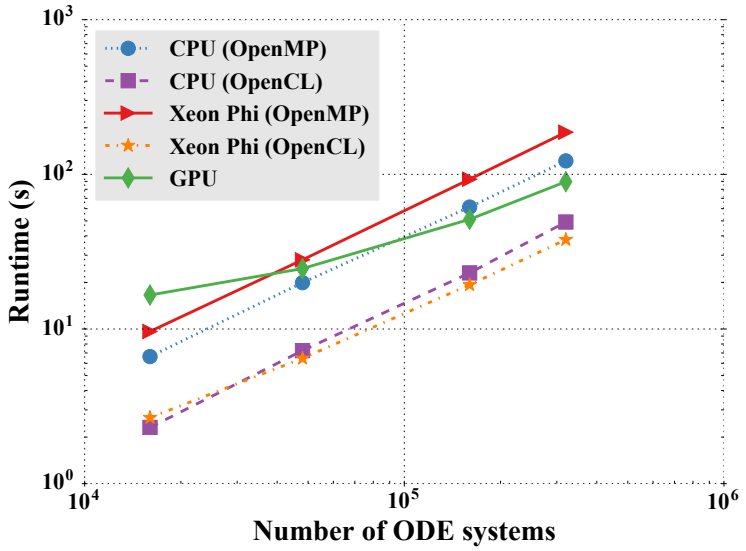


Figure 6: Comparison of runtimes of the ROS4 solver for the model problem, between OpenMP and CL-SIMD on the host CPU and Xeon Phi, and the GPU. Data, plotting scripts, and figure file are available [39].

instruction stream. Instead of fetching one ODE from the queue as in the OpenMP implementation, the OpenCL version fetches the SIMD (or SIMT) parallel width, i.e., 4, 8, 16, and 32 ODEs depending upon the platform. This leads to coarse-grained dynamic scheduling and does not address variable costs within each parallel stream. This impact will be discussed subsequently.

The relative performance differences between the devices and data parallelism methods are more clearly seen by examining the throughput instead of runtime. The throughputs, defined as the number of ODE systems solved per second, for the RKF45 and ROS4 methods are shown in Figures 7 and 8, respectively. For the RKF45 method, the throughput is nearly constant for 48,000 ODE systems and higher. The SIMD method on the host gives a speedup of 2.7–2.9 $\times$ over the baseline host OpenMP run-time. On the MIC accelerator, the speedup is up to 4.8 $\times$ over OpenMP on the MIC. That is, the SIMD speedup on the MIC is approximately double that observed on the host, which matches the ratio of the native SIMD word widths on the two devices.

Figure 8 clearly shows the superior throughput of the ROS method. A more pronounced dependency upon the number of ODE systems is observed, particularly with the MIC SIMD method and the GPU method. This is driven by the thousands of ODE systems needed to saturate the device with both of these methods, while only tens or hundreds are needed with the other methods. The ratio of the RKF45 and ROS peak throughputs differs across the platforms:

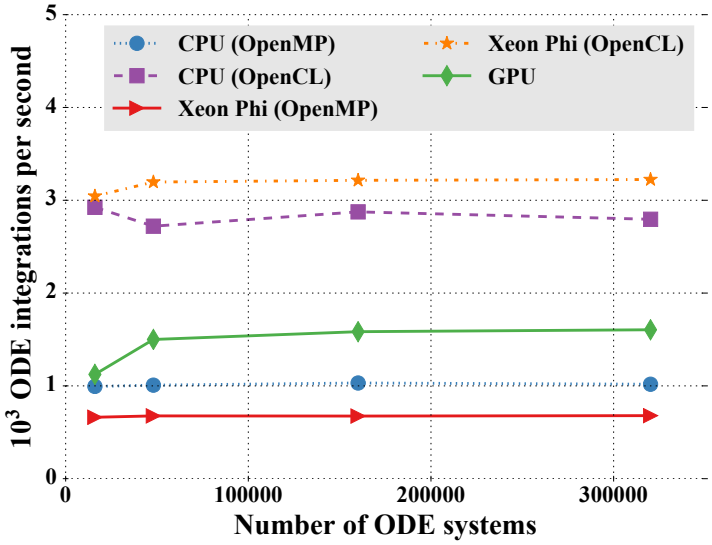


Figure 7: Comparison of computational throughput, measured in 10^3 ODE integrations per second, of the RKF45 solver for the model problem, between OpenMP and CL-SIMD on the host CPU and Xeon Phi, and the GPU. Data, plotting scripts, and figure file are available [39].

the lowest at 2.2 with the GPU, and the highest of 2.6 with the host and MIC SIMD.

Of note is the lower GPU performance compared with that observed for the RHS function evaluations. The maximum GPU throughputs are only 1.6 × (RKF45) and 1.4 × (ROS4) higher than the baseline host OpenMP throughput, yet throughput was 2.3 × higher for the RHS function evaluations with GRI Mech 3.0 (see Figure 2). The host SIMD methods are also lower but to a lesser extent, i.e., 2.7 and 2.5 × compared to 3.1 seen previously with the RHS evaluations. Conversely, the speedup with the MIC SIMD methods (3.2 × for both) are higher relative to the RHS function evaluation benchmark (2.3 ×).

A possible cause of this lower performance on the GPU is variability in the number of integrator iterations needed between neighboring ODE systems. As noted above, there is significant variability in the number of RKF45 iterations between neighbor mesh points. A unique GPU thread solves each ODE system, which means that the realized cost will be the maximum number of iterations needed by any thread within the same *thread warp*. The ODE systems in Figure 4 are mapped linearly to the GPU threads within each warp. Recall that the only variability between RHS function evaluations was the temperature polynomials, a relatively minor cost. Performance degradation caused by differing numbers of integration iterations has been reported before by Stone and Davis [19] and Niemeyer and Sung [20].

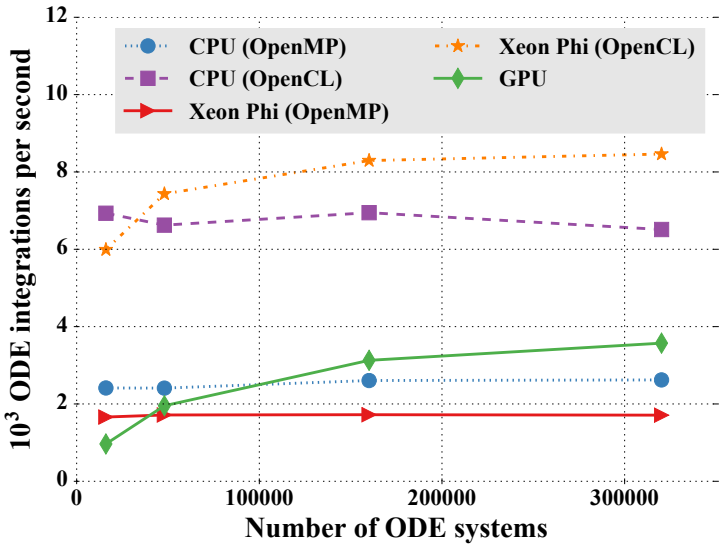


Figure 8: Comparison of computational throughput, measured in 10^3 ODE integrations per second, of the ROS4 solver for the model problem, between OpenMP and CL-SIMD on the host CPU and Xeon Phi, and the GPU.

Data, plotting scripts, and figure file are available [39].

Problem-to-problem variability should also impact the SIMD implementations. We define an inefficiency metric to better assess this performance impact of the ODE variability: the *waste* within each SIMD work unit (i.e., a SIMD word or SIMT warp) represents the number of excess integrator steps taken. The cost per integrator step is constant for the RKF45 and ROS4 methods so this is a logical quantity to measure. The waste within SIMD work unit j can be expressed in a normalized form as

$$W_j = 1 - \frac{\sum_i^P N_i}{P \times \max_i^P N_i} \,, \tag{20}$$

where N_i is the number of integrator iterations needed for ODE i and P is the parallel width (i.e., 4, 8, 16, or 32 depending upon the device). Equation (20) extends the CUDA-specific *warp* divergence metric proposed by Niemeyer and Sung [20] to any SIMD (or SIMT) platform with vector length P.

Figures 9 and 10 show the cumulative probability distribution of W for the model problem for the four relevant SIMD widths for RKF45 and ROS4, respectively. The impact of wider SIMD parallelism is evident for both solvers. That is, as the SIMD width is increased, the proportion of wasted computation increases. For RKF45, we see that approximately 92 % of the work units have less than 1 % waste with a width of four, but this drops to only 63 % for a width of 32. We observe a similar behavior with ROS4, though with reduced magnitudes. This is consistent with Figure 4 where the variation between adjacent ODE systems (i.e., mesh points) is less.

Figure 11 shows the relative throughput of the SIMD versions of the ODE solvers on the host and MIC using increasing SIMD word sizes. There, the throughput on each device is normalized by the throughput with the native SIMD word size. Specifying an SIMD word larger than the native size (e.g., `double8` on the host) should result in multiple SIMD operations in sequence. The analysis above predicts that the performance, especially with the RKF45 solver, should degrade with wider word size. However, using wider words significantly improves performance on the host. In fact, the highest performance on the host with both RKF45 and ROS4 is obtained using `double16`, four times the native word size, and the performance consistently improves using wider SIMD words. On the MIC, the wider word size degrades performance of the RKF45 solver by approximately 10 %. On the other hand, the wider word size using the ROS4 solver improves the MIC performance by approximately 10 %. The performance metrics presented here indicate that while problem-to-problem variation leads to increased computational waste, this does not necessarily translate into reduced computational throughput on the host and MIC devices.

6. Conclusions

In this paper, we presented and discussed the parallel performance of thread- and data-parallel methods applied to chemical kinetics integrations. Benchmarks were conducted using multithreading and SIMD parallel methods on a

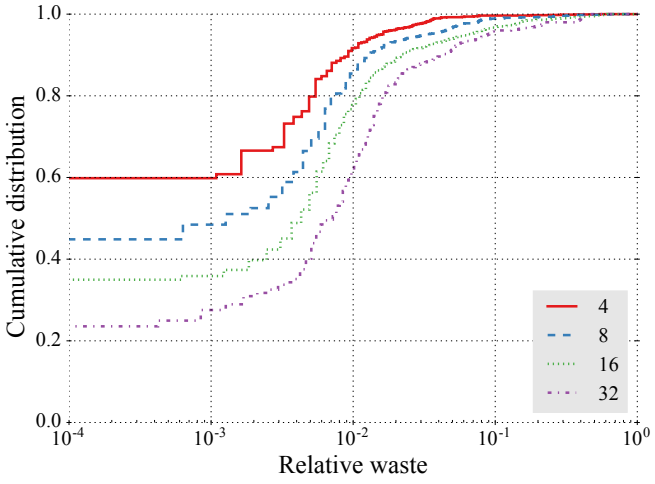


Figure 9: Cumulative population of relative waste in parallel SIMD work, given by Eq. (20), for the RKF45 solver on the model problem with SIMD parallel widths 4, 8, 16, and 32. Relative waste calculated from sample of 1601 points. Data, plotting scripts, and figure file are available [39].

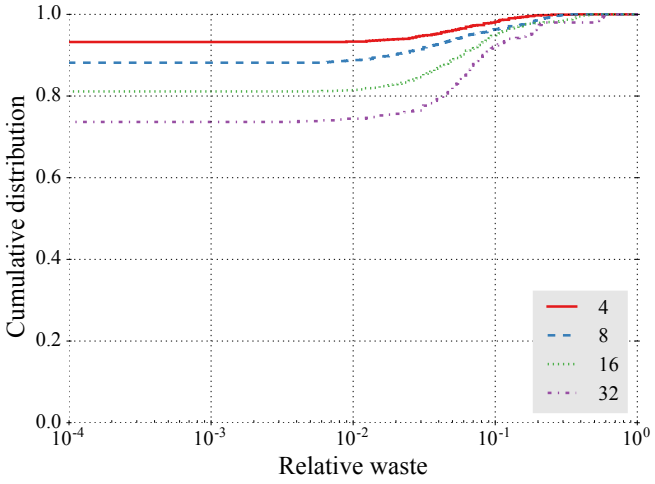
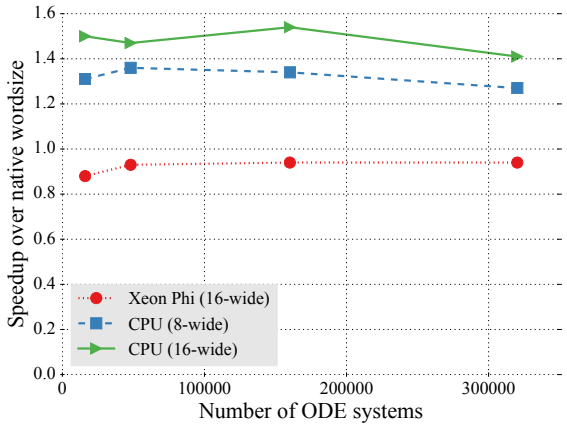
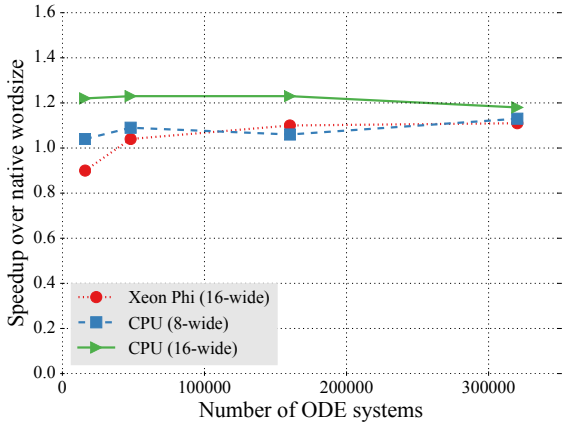


Figure 10: Cumulative population of relative waste in parallel SIMD work, given by Eq. (20), for the ROS4 solver on the model problem with SIMD parallel widths 4, 8, 16, and 32. Relative waste calculated from sample of 1601 points. Data, plotting scripts, and figure file are available [39].



(a) RKF45



(b) ROS4

Figure 11: Speedup of integrators using SIMD word sizes of 16-wide on the Xeon Phi, and 8- and 16-wide on the CPU, which represent two, two, and four times the native word sizes, respectively. Data, plotting scripts, and figure file are available [39].

multicore CPU system and on two coprocessors (or accelerators): an Intel Xeon Phi (or MIC) and an Nvidia Kepler K20m GPU. We implemented both multi-threading and data-parallel models using OpenCL to allow a study of the same code base across all three platforms. Two vectorization models were examined within OpenCL: (1) one thread maps to each parallel task and sets of cooperating threads execute instructions in a SIMT paradigm, and (2) the same instruction stream concurrently computes explicit SIMD vector datatypes and multiple parallel tasks.

All benchmarks were compared with multicore runs on the host CPU and the MIC using OpenMP. We were particularly interested in the performance difference between OpenMP with automated compiler vectorization compared with manual SIMD programming on the host and MIC platforms.

The first benchmark series studied the performance of evaluating many instances of the RHS function for three chemical kinetic models of increasing size and complexity. The SIMT implementations on the host and MIC devices both perform slower (13 % and 35 %, respectively) than their baseline OpenMP implementations with the common GRI Mech 3.0 model [36]. However, the explicit SIMD model provides a speedup of approximately $3\times$ over the baseline OpenMP on both of these platforms. The SIMT model on the GPU performs far better than the SIMT model on the host and MIC devices, and matches the performance of the MIC device with SIMD programming. Thus, while both SIMD and SIMT models are possible on CPU and the CPU-like MIC using OpenCL, SIMD methods provide considerably higher performance. Furthermore, SIMT is necessary on the GPU platform meaning that two separate programming models are needed to reach peak performance across the three HPC devices.

Studies with all three chemical kinetic models produced similar results, and performance showed no significant dependence upon model size. For this reason, we used only the GRI Mech 3.0 model for subsequent benchmarks.

The second benchmark series studied the performance of integrating many independent constant-pressure ODE systems with the GRI Mech 3.0 model for methane oxidation. This model problem mimicked what is commonly encountered when simulating chemical kinetics phenomena within an operator-splitting framework. The ODE systems were integrated using the nonstiff RKF45 ODE solver and stiff Rosenbrock ROS4 solver. Both methods have the same fourth-order theoretical accuracy and use the same step-size adapation and initial step-size estimation algorithms.

The ROS4 solver consistently performs 2.2–2.6 times faster than the RKF45 solver on the model problem on the various platforms. This matches analysis that shows RKF45 needs approximately 25 times more iterations than ROS4, while ROS4 costs approximately ten times more than RKF45 per iteration (based on the ratio of RHS function evaluations). Finite-difference Jacobian matrices were used with ROS4 for this study for simplicity, which increased the number of RHS evaluations per-iteration by $N_{\text{sp}} + 1$—an increase of nine times for GRI Mech 3.0. Analytical Jacobian matrices for the model constant-pressure problem have been derived [28, 43] and can reduce the cost per-iteration of the Rosenbrock family of solvers. (Stone and Bisetti [44] showed a 2.9 times speedup

with analytical Jacobian matrices for GRI Mech 3.0 with ROS4.)

The SIMD implementation of the two solvers shows a significant performance acceleration compared with the baseline OpenMP implementation. In general, the SIMD method improves performance by 2.5–2.8 times on the host and 4.7–4.9 times on the MIC. The higher MIC SIMD acceleration is consistent with the ratios of the SIMD word widths.

The GPU ODE integrators do not perform as efficiently as the RHS function evaluation. The GPU integrator only offers a 1.4–1.6 times speedup over the host baseline throughput. We attributed the lower performance to thread divergence caused by ODE systems requiring different numbers of integrator steps in each SIMT parallel work unit. We quantified this impact with a SIMD waste metric that shows that the wasted number of integrator steps increases with increasing vector width (32 for the GPU). The ROS4 integrator exhibits a lower occurrence of wasted work, which can help explain why the ROS integrator performed better than the RKF45 integrator on the GPU and MIC devices. However, the performance on the host and MIC often *improves* with increasing SIMD word size. This indicates that the improved computational performance due to wider word sizes (e.g., instruction parallelism, cache efficiency) can actually overcome increased integrator waste. Nevertheless, reducing the wasted number of integrator steps could improve the performance on all devices and should be investigated in future studies. For example, Murray [45] demonstrated a strategy of mitigating inefficiency due to variable-length RK integration tasks (i.e., variation in steps) on GPUs by assigning multiple tasks to each GPU thread. This strategy may be extendable to SIMD platforms and should be investigated in future studies to determine the impact of variable task lengths.

Overall, this paper shows that the explicit SIMD methods offer a promising strategy for more efficiently using MIC and host CPU systems within the context of chemical kinetics applications. Further research is needed to improve the SIMD-friendly ROS methods, e.g., with analytical Jacobian matrices. The SIMD performance advantages demonstrated here may warrant investigation into more numerically efficient, but less SIMD efficient, ODE solver methods such as implicit Runge–Kutta integrators.

Acknowledgements

This material is based upon work supported, in part, by the National Science Foundation under grant ACI-1535065. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575. Code development and performance measurements were conducted on the Stampede system at the Texas Advanced Computer Center (TACC) through resource allocation TG-ASC130025.

Appendix A. Availability of material

The integrators used to perform this study are available openly via the `accelerInt` software package [46]. The most recent version of `accelerInt` can

be found at its GitHub repository: https://github.com/SLACKHA/accelerInt. All figures, and the data and plotting scripts necessary to reproduce them, are available openly under the CC-BY license [39].

Appendix B. RKF parameters

Table B.2 shows the method parameters in a modified Butcher tableau, where a_{ij} are the coefficients, $b_j/\hat{b}_j$ are the weights of the embedded fourth-order and fifth-order methods, respectively, and c_i are the nodes (not used here, since the ODE systems are autonomous).

c_i	a_{ij}					
0						
$\frac{1}{4}$	$\frac{1}{4}$					
$\frac{3}{8}$	$\frac{3}{32}$	$\frac{9}{32}$				
$\frac{12}{13}$	$\frac{1932}{2197}$	$-\frac{7200}{2197}$	$\frac{7296}{2197}$			
1	$\frac{439}{216}$	-8	$\frac{3680}{513}$	$-\frac{845}{4104}$		
$\frac{1}{2}$	$-\frac{8}{27}$	2	$-\frac{3544}{2565}$	$\frac{1859}{4104}$	$-\frac{11}{40}$	
b_j	$\frac{25}{216}$	0	$\frac{1408}{2565}$	$\frac{2197}{4104}$	$-\frac{1}{5}$	0
$\hat{b}_j$	$\frac{16}{135}$	0	$\frac{6656}{12825}$	$\frac{28561}{56430}$	$-\frac{9}{50}$	$\frac{2}{55}$

Table B.2: Coefficients for the five-stage, fourth-order Runge–Kutta–Fehlberg method, adopted from Hairer et al. [47] and displayed in a modified Butcher tableau.

Appendix C. ROS4 parameters

Table C.3 contains the ROS4 parameters, including the strictly lower-triangular matrices $\boldsymbol{a}$ and $\boldsymbol{c}$, and the vectors $\boldsymbol{m}$, $\boldsymbol{b}$, α_i, and γ_i. In addition, it shows the vector $\boldsymbol{E}$, the difference in $\boldsymbol{b}$ coefficients for method orders three and four used for error estimation.

References

[1] K. Radhakrishnan, Comparison of numerical techniques for integration of stiff ordinary differential equations arising in combustion chemistry, NASA Technical Paper 2372 (Oct. 1984).
URL http://ntrs.nasa.gov/search.jsp?R=19850001758

[2] 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.

$a_{21} = 2.0$	$\alpha_2 = 1.14564$
$a_{31} = 1.867943637803922$	$\alpha_3 = 0.6552168638155900$
$a_{32} = 0.2344449711399156$	$\alpha_4 = \alpha_3$
$a_{41} = a_{31}$	$\gamma_1 = \gamma = 0.57282$
$a_{42} = a_{32}$	$\gamma_2 = -1.769193891319233$
$c_{21} = -7.137615036412310$	$\gamma_3 = 0.7592633437920482$
$c_{31} = 2.580708087951457$	$\gamma_4 = -0.1049021087100450$
$c_{32} = 0.6515950076447975$	$E_1 = -0.2815431932141155$
$c_{41} = -2.137148994382534$	$E_2 = -0.07276199124938920$
$c_{42} = -0.3214669691237626$	$E_3 = -0.1082196201495311$
$c_{43} = -0.6949742501781779$	$E_4 = -1.093502252409163$
$b_1 = 2.255570073418735$	$b_3 = 0.4353179431840180$
$b_2 = 0.2870493262186792$	$b_4 = 1.093502252409163$

Table C.3: Parameters for ROS4, adopted from Hairer and Wanner [26].

[3] J. Kim, S. Y. Cho, Computation accuracy and efficiency of the time-splitting method in solving atmospheric transport/chemistry equations, Atmos. Environ. 31 (15) (1997) 2215–2224.

[4] O. M. Knio, H. N. Najm, P. S. Wyckoff, A semi-implicit numerical scheme for reacting flow II. stiff, operator-split formulation, J. Comput. Phys. 154 (1999) 428–467. doi:10.1006/jcph.1999.6322.

[5] D. Lanser, J. G. Verwer, Analysis of operator splitting for advection–diffusion–reaction problems from air pollution modelling, J. Comput. Appl. Math. 111 (1999) 201–216.

[6] M. S. Day, J. B. Bell, Numerical simulation of laminar reacting flows with complex chemistry, Combust. Theor. Model. 4 (4) (2000) 535–556. doi:10.1088/1364-7830/4/4/309.

[7] E. S. Oran, J. P. Boris, Numerical Simulation of Reactive Flow, 2nd Edition, Cambridge University Press, 2001.

[8] M. Singer, S. Pope, H. Najm, Operator-splitting with ISAT to model reacting flow with detailed chemistry, Combustion Theory and Modelling 10 (2) (2006) 199–217. doi:10.1080/13647830500307501.

[9] Z. Ren, S. B. Pope, Second-order splitting schemes for a class of reactive systems, J. Comput. Phys. 227 (17) (2008) 8165–8176. doi:10.1016/j.jcp.2008.05.019.

[10] R. L. Speth, W. H. Green, W. H. Green, S. MacNamara, G. Strang, Balanced splitting and rebalanced splitting, SIAM J. Numer. Anal. 51 (6) (2013) 3084–3105. doi:10.1137/120878641.

[11] S. R. Tonse, N. W. Moriarty, M. Frenklach, N. J. Brown, Computational economy improvements in PRISM, Int. J. Chem. Kinet. 35 (9) (2003) 438–452. doi:10.1002/kin.10140.

[12] L. Liang, S.-C. Kong, C. Jung, R. D. Reitz, Development of a semi-implicit solver for detailed chemistry in internal combustion engine simulations, J. Eng. Gas. Turb. Power 129 (1) (2007) 271–278. doi:10.1115/1.2204979.

[13] Y. Shi, L. Liang, H.-W. Ge, R. D. Reitz, Acceleration of the chemistry solver for modeling DI engine combustion using dynamic adaptive chemistry (DAC) schemes, Combust. Theor. Model. 14 (1) (2010) 69–89. doi:10.1080/13647830903548834.

[14] A. Cuoci, A. Frassoldati, T. Faravelli, E. Ranzi, Numerical modeling of laminar flames with detailed kinetics based on the operator-splitting method, Energy and Fuels 27 (12) (2013) 7730–7753. doi:10.1021/ef4016334.

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

[16] 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, Storrs, CT, 2011. doi:10.6084/m9.figshare.3384964.

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

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

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

[20] K. E. Niemeyer, C. J. Sung, Accelerating moderately stiff chemical kinetics in reactive-flow simulations using GPUs, J. Computational Physics 256 (2014) 854–871. doi:10.1016/j.jcp.2013.09.025.

[21] 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.

[22] N. J. Curtis, K. E. Niemeyer, C. J. Sung, An investigation of GPU-based stiff chemical kinetics integration methods (2017). doi:10.1016/j.combustflame.2017.02.005.

[23] A. Hindmarsh, P. Brown, K. Grant, R. Serban, D. Shumaker, C. Woodward, SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers, ACM Transactions on Mathematical Software 31 (3) (2005) 363–396. doi:10.1145/1089014.1089020.

[24] E. Hairer, G. Wanner, Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems, 2nd Edition, Springer, 1996.

[25] H. Zhang, A. Sandu, FATODE: A library for forward, adjoint, and tangent linear integration of stiff systems, SIAM J. Sci. Comput. 36 (5) (2014) C504–C523. doi:10.1137/130912335.

[26] E. Hairer, G. Wanner, `ROS4`, http://www.unige.ch/~hairer/prog/stiff/Oldies/ros4.f, accessed: 2016-08-17 (Nov. 1992).

[27] H. Zhang, A. Sandu, FATODE v1.2, http://people.cs.vt.edu/~asandu/Software/FATODE/index.html (Apr. 2013).

[28] K. E. Niemeyer, N. J. Curtis, C. J. Sung, `pyJac`: analytical Jacobian generator for chemical kinetics, Computer Physics Communications 215 (2017) 188–203. doi:10.1016/j.cpc.2017.02.004.

[29] B. McBride, S. Gordon, M. Reno, Coefficients for calculating thermodynamic and transport properties of individual species, TM 4513, NASA (Oct. 1993).

[30] D. Unat, C. Chan, W. Zhang, S. Williams, J. Bachan, J. Bell, J. Shalf, Exasat: An exascale co-design tool for performance modeling, International Journal of High Performance Computing Applications 29 (2) (2015) 209–232. doi:10.1177/1094342014568690.

[31] T. Lu, C. Law, A directed relation graph method for mechanism reduction, Proceedings of the Combustion Institute 30 (1) (2005) 1333–1341.

[32] D. Lignell, J. Chen, P. Smith, T. Lu, C. Law, The effort of flame structure on soot formation and transport in turbulent non-premixed flames using direct numerical simulation, Combustion and Flame 151 (1–2) (2007) 2–28.

[33] A. Kroshko, R. Spiteri, Efficient SIMD solution of multiple systems of stiff IVPs, J. of Computational Science 4 (2013) 377–385. doi:10.1016/j.jocs.2012.08.017.

[34] J. E. Stone, D. Gohara, G. Shi, OpenCL: A parallel programming standard for heterogeneous computing systems, IEEE Des. Test 12 (3) (2010) 66–73. doi:10.1109/MCSE.2010.69.

[35] S. Davis, V. Joshi, Ameya, H. Wang, F. Egolfopoulos, An optimized kinetic model of H_2/CO combustion, Proceedings of the Combustion Institute 30 (1) (2005) 1283–1292. doi:10.1016/j.proci.2004.08.252.

[36] 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).

[37] H. Wang, X. You, A. V. Joshi, S. G. Davis, A. Laskin, F. Egolfopoulos, C. K. Law, USC Mech Version II. High-temperature combustion reaction model of H_2/CO/C_1–C_4 compounds, http://ignis.usc.edu/USC_Mech_II.htm (May 2007).

[38] K. E. Niemeyer, `create_rate_subs` v1.0, Zenodo. https://doi.org/10.5281/zenodo.44336 (Jan. 2016).

[39] C. P. Stone, A. T. Alferman, K. E. Niemeyer, Data, plotting scripts, and figures for "Accelerating finite-rate chemical kinetics with coprocessors: comparing vectorization methods on GPUs, MICs, and CPUs", Figshare (2017). doi:10.6084/m9.figshare.5353183.

[40] D. G. Goodwin, H. K. Moffat, R. L. Speth, Cantera: An object-oriented software toolkit for chemical kinetics, thermodynamics, and transport processes, http://www.cantera.org, version 2.2.1 (2016).

[41] C. P. Stone, A. T. Alferman, K. E. Niemeyer, Methane premixed-air flame data used in "Accelerating finite-rate chemical kinetics with coprocessors: comparing vectorization methods on GPUs, MICs, and CPUs", Figshare (2017). doi:10.6084/m9.figshare.5350435.

[42] A. Aspdena, M. Dayb, J. Bell, Turbulence-chemistry interaction in lean premixed hydrogen combustion, Proceedings of the Combustion Institute 35 (2015) 1321–1329. doi:10.1016/j.proci.2014.08.012.

[43] C. Safta, H. N. Najm, O. M. Knio, TChem - a software toolkit for the analysis of complex kinetic models, Tech. Rep. SAND2011-3282, Sandia National Laboratories (May 2011). doi:10.2172/1113874.

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

[45] L. Murray, GPU acceleration of Runge–Kutta integrators, IEEE Transactions on Parallel and Distributed Systems 23 (1) (2012) 94–101. doi:10.1109/TPDS.2011.61.

[46] N. J. Curtis, K. E. Niemeyer, C. P. Stone, `accelerInt` v1.1-beta (2017). doi:10.5281/zenodo.842845.

[47] E. Hairer, S. P. Nørsett, G. Wanner, Solving Ordinary Differential Equations I: Nonstiff Problems, 2nd Edition, Vol. 8 of Springer Series in Computational Mathematics, Springer, Berlin, Heidelberg, 1993. doi:10.1007/978-3-540-78862-1.