Performance Analysis and Acceleration of Explicit Integration for Large Kinetic Networks using Batched GPU Computations

Azzam Haidar*, Benjamin Brock‡¶, Stanimire Tomov*, Michael Guidry†¶‖,
Jay Jay Billings§¶, Ahmad Abdelfatah*, Daniel Shyles†, Jack Dongarra*¶**
{haidar,tomov,ahmad,dongarra}@icl.utk.edu,
{brock,guidry}@utk.edu, jayjaybillings@gmail.com, dshyles@vols.utk.edu
*Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA
†Department of Physics and Astronomy, University of Tennessee, Knoxville, TN, USA
‡Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN, USA
§The Bredesen Center for Interdisciplinary Research and Graduate Education, University of Tennessee, Knoxville, TN, USA
¶Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA
‖Physics Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA
**University of Manchester, Manchester, UK

***Abstract*—We demonstrate the systematic implementation of recently-developed fast explicit kinetic integration algorithms on modern graphics processing unit (GPU) accelerators. We take as representative test cases Type Ia supernova explosions with extremely stiff thermonuclear reaction networks having 150-365 isotopic species and 1600-4400 reactions, assumed coupled to hydrodynamics using operator splitting. In such examples we demonstrate the capability to integrate independent thermonuclear networks from ∼300–600 hydro zones (assumed to be deployed on CPU cores) in parallel on a single GPU in the same wall clock time that standard implicit methods can integrate the network for a single zone. This two or more orders of magnitude increase in efficiency for solving systems of realistic thermonuclear networks coupled to fluid dynamics implies that important coupled, multiphysics problems in various scientific and technical disciplines that were intractable, or could be simulated only with highly schematic kinetic networks, are now computationally feasible. As examples of such applications we present the computational techniques developed for our ongoing deployment of these new methods (for Type Ia supernova explosions in astrophysics and for simulation of the complex atmospheric chemistry entering into weather and climate problems) on modern GPU accelerators. We show that similarly to many other scientific applications, ranging from national security to medical advances, the computation can be split into many independent computational tasks, each of relatively small-size. As the size of each individual task does not provide sufficient parallelism for the underlying hardware, especially accelerators, these tasks must be computed concurrently as a single routine, that we call `batched routine`, in order to saturate the hardware with enough work.**

***Keywords*-batched computation, high-performance, GPUs, multi-physics, explicit integration**

I. Introduction

Many important physical processes can be modeled by the coupled evolution of a reaction network (kinetic network) and fluid dynamics. A representative example is provided by astrophysical thermonuclear reaction networks, where a proper description of the overall problem typically requires multidimensional hydrodynamics coupled to the network across a spatial grid involving many independent zones. Within each zone of the simulation the hydrodynamical (hydro) evolution controls the temperature and density, while the network influences the hydrodynamical evolution through energy release and composition changes. Many other scientifically-interesting problems employ kinetic networks. Representative examples include the networks of chemical reactions required to model atmospheric chemistry, chemical evolution networks in contracting molecular clouds during star formation, plasma-surface interactions in magnetically confined fusion devices, fuel depletion in fission power reactors, and chemical burning networks in combustion chemistry. Realistic atmospheric simulations, combustion of larger hydrocarbon molecules, studies of soot formation, core-collapse supernovae, and thermonuclear supernovae all can involve hundreds to thousands of reactive species undergoing thousands to tens of thousands of reaction couplings [1], [2]. The corresponding reaction networks are large. Current techniques based on implicit numerical integration typically are not fast enough to allow coupling of realistic reaction networks to the full dynamics of such problems and even the most realistic simulations have employed highly schematic reaction networks.

As a representative example, the present situation in Type Ia supernova simulations is illustrated in Figure 1. A physically-realistic network is displayed in Figure 1(a), a minimal physically-correct network for coupling to the hydrodynamical simulation is displayed in Figure 1(b), and the current state of the art for Type Ia simulations employing multidimensional hydrodynamics is displayed in Figure 1(c). The species omitted in reducing the 365-isotope network in Figure 1(a) to the 150-isotope network in Figure 1(b) are populated sufficiently weakly that they play little role in energy release and do not

have significant influence on the coupling of the network to hydrodynamics. Thus, the 150-isotope network is a minimal network for coupling to the hydrodynamics; any network smaller than this will omit non-trivial parts of the realistic coupling between kinetics and fluid dynamics. The disparity between the minimal realistic network in Figure 1(b) and the current state of the art in Figure 1(c) is a consequence of insufficient computational power to couple a realistic kinetic network in real time to the fluid dynamics using current technology.

There are two general approaches that we might take to address the preceding issues. The first is to seek faster algorithms for solution of the typically large and stiff system of differential equations that describe the kinetic evolution. The second is to take advantage of advances in computational architectures to solve the chosen algorithm more rapidly. In previous work [3]–[7], we described a new algebraically-stabilized explicit approach to solving kinetics equations that extends earlier work by Mott [8] and is capable of taking stable integration timesteps comparable to those of standard implicit methods, even for extremely stiff systems of equations. Since the methods are explicit, they do not involve matrix inversions and thus scale linearly with network size. Because of this much more favorable scaling and competitive integration step size, we demonstrated that such algorithms are capable of performing numerical solution of extremely stiff kinetic networks containing several hundred species about 36 times faster than the best implicit codes [3]–[7].

One implication of new algorithms is that they may give new perspectives on optimization. Standard implicit methods spend most of their time on linear algebra operations for larger networks because they must be solved iteratively, which requires inversion of large matrices. Thus, the optimization strategy is clear for implicit algorithms: do the linear algebra faster. Conversely, the new explicit methods do not involve matrix inversions, so optimizing them involves different strategies. These may offer unique opportunities for implementation on newer architectures such as GPU or many-core accelerators coupled to standard CPUs. In this paper we take a first step in addressing these issues by deploying the explicit integration methods described in our previous work [3]–[7] on coupled CPU-GPU systems. We show that using GPUs to exploit the parallelism inherent in the explicit kinetic algorithm for networks of realistic size makes it possible find the solution for a single network on the GPU faster than on the CPU and the solutions for many networks simultaneously on the GPU versus serially on the CPU.

Solving many networks simultaneously on the GPU can be done very efficiently, as we show, due to the increased parallelism of the computation. To fully benefit from the parallelism, the processing for all networks must be grouped into a single routine, that we refer to as **batched routine**. The purpose of batched routines is to solve a set of independent problems in parallel. Such configuration arises not just at the astrophysics application at hand using explicit solvers, but in many other real applications, including astrophysics

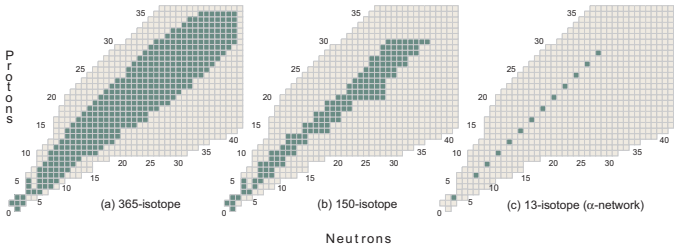

Fig. 1. Typical thermonuclear networks that might be used for simulating Type Ia supernova explosions [3]. (a) A network containing the 365 isotopes that are populated with measurable intensity in the explosion. (b) A subset representing the 150 isotopes populated with sufficient intensity to influence the hydrodynamics. This is the minimal realistic network for coupling to hydrodynamical simulations of the Type Ia explosion. (c) The largest network that has typically been coupled to multidimensional hydrodynamics in a Type Ia simulation.

with implicit solvers [9], quantum chemistry [10], metabolic networks [11], CFD and resulting PDEs through direct and multifrontal solvers [12], high-order FEM schemes for hydrodynamics [13], direct-iterative preconditioned solvers [14], image [15] and signal processing [16]. If a single computational task in the batch is large enough to allow efficient use of the entire device, there is no benefit of using batched computation; it is preferred to execute the set of independent tasks in serial fashion as a sequence of tasks, to better enforce locality of data and increase the cache reuse. However, this is typically not the case in many real applications, including the ones investigated here. Although the entire computation is extremely large, the separate tasks are so small, that the amount of work needed to perform the computation cannot saturate the device, either CPU or GPU, and thus there is a need for batched routines. In general, the tasks do not necessarily have the same size, which further complicates the development of efficient computing techniques for these types of workloads.

II. Algorithmic Advancments and Novel Approach

A. Formulation and Novel Approach

We shall assume that the coupling of reaction networks is done using operator splitting, where the hydrodynamical solver is evolved for a numerical timestep holding network parameters constant, and then the network is evolved over the time corresponding to the hydrodynamical timestep holding the new hydrodynamical variables constant (see Section III below). The general task for the kinetic network then is to solve efficiently N coupled ordinary differential equations subject to initial conditions that have been determined in the current hydrodynamical timestep as shown in Figure 2.

In this expression, the $\mathrm{y}_{i,(i=1\ldots N)}$ describe the dependent variables (typically measures of abundance), the fluxes between species i and j are denoted by $(f_j)_i$. The sum for each variable i is over all species j coupled to i by a non-zero flux $(f_j)_i$, and for later convenience we have decomposed the flux into a component F_i^+ that increases the abundance of y_i and a component F_i^- that depletes it. For an N-species network there will be N such equations in the population variables y_i,

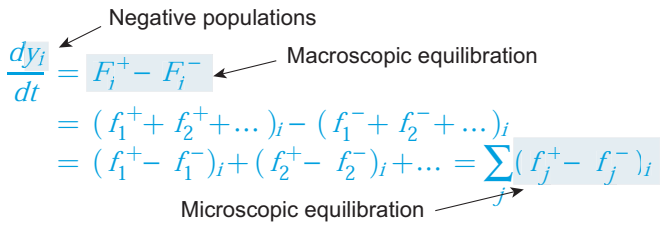

Fig. 2. Sources of stiffness in explicit integration of a set of coupled differential equations. *Negative probabilities* correspond to populations (which cannot be negative) being driven negative by numerical error because of a too-large timestep. This turns damped exponentials into growing exponentials and destabilizes the network. *Macroscopic equilibration* occurs when F_i^+ becomes approximately equal to F_i^-, so that one is taking numerically the tiny difference of two very large numbers. This too will destabilize the network if the explicit timestep is too large. *Microscopic equilibration* occurs when forward–reverse terms at the reaction level become almost equal. This again implies numerically taking the tiny difference of very large numbers, and will destabilize the network if the timestep is too large.

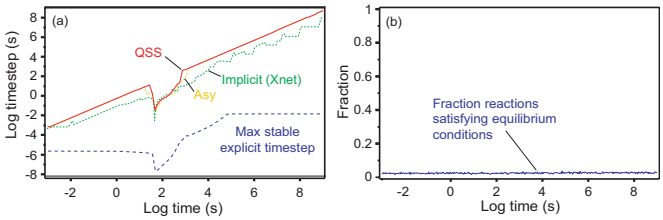

Fig. 3. (a) The integration timestepping for the nova simulation of figure **??** for a standard implicit method (in this and other discussion in this paper we use the implicit backward Euler code Xnet [17] for comparison), and for two implementations of the algebraically-stabilized explicit method (labeled Asy and QSS) [3]. (b) Fraction of reactions satisfying the microscopic equilibration condition in the calculation.

generally coupled to each other because of the dependence of the fluxes on the different y_j. The variables y_i are typically proportional to a number density η_i for the species i. For the specific astrophysical examples that follow we shall replace the generic population variables y_i with the mass fraction X_i, which satisfies

$$X_i = \frac{A_i}{\rho N_A}\eta_i, \text{ where } \sum_i X_i = 1, \qquad (1)$$

N_A is Avogadros number, ρ is the total mass density, and A_i is the atomic mass number for the species i.

Two broad classes of numerical integration may be defined. In *explicit* numerical integration, to advance the solution from time t_n to t_{n+1} only information already available to the calculation at t_n is required. In *implicit* numerical integration, to advance the solution from t_n to t_{n+1} requires information at t_{n+1}, which is of course unknown. Thus implicit integration requires an iterative solution. Such solutions are expensive for large sets of equations because they involve matrix inversions.

Why then would one want to use implicit methods? The answer is *stability*, in particular for the integration of *stiff equations.* Systems of equations exhibit *stiffness* if multiple timescales differing by many orders of magnitude are an essential feature. Since most processes of physical importance involve changes occurring on more than one timescale, stiffness is a common feature in computer modeling of realistic physical systems. It is commonly understood that stiff systems cannot be integrated efficiently with explicit methods because the largest stable timestep is much too short, so the traditional view is that stiff systems require implicit numerical methods for their solution. Thus, in the standard view explicit methods are inherently simple, but potentially unstable, particularly for stiff systems, while implicit methods are inherently complicated but usually stable for stiff systems. We now wish to describe new explicit methods that have the potential to combine the desirable attributes of both implicit and traditional explicit methods: These new *algebraically-stabilized explicit methods* will be shown to have the simplicity of an explicit method but with stability rivaling that of implicit methods [3]–[7].

B. Algebraically-Stabilized Explicit Integration

The key to stabilizing explicit integration is to understand that there are three basic sources of stiffness for a typical reaction network. We have termed these as:

1) Negative populations
2) Macroscopic equilibration
3) Microscopic equilibration

and they are illustrated in Figure 2. We have developed a systematic set of algebraic constraints within the context of explicit numerical integration that remove these sources of stiffness and permit the algebraically-stabilized explicit method to take stable and accurate timesteps that are competitive with that of standard implicit methods. Details of this novel approach may be found in [3]–[7]. Since the stabilized explicit method can execute each timestep faster for large networks (no matrix inversions), the resulting algorithm is intrinsically faster than implicit algorithms.

In figure 3 we illustrate the new method in action for a nova simulation. Because the timestepping for the algebraically-stabilized explicit integrations (labeled Asy and QSS for two somewhat different implementations) is competitive with that of the implicit-code reference, the algebraically-stabilized explicit method exhibited a speed $\sim 5-10$ times faster than that of the implicit code for this problem. In Refs. [3]–[7] we have documented a number of such tests of the new explicit approach and the intrinsic speedup on a serial CPU implementation of the algorithm over state of the art implicit methods is summarized in figure 4. [1]

III. Designing the Explicit Asymptotic Algorithm Using Operator Split on a CPU-GPU System

The prototype implementation that we shall describe here assumes an operator split formulation of fluid dynamics coupled

[1] We use a standard implicit code running on a CPU as the benchmark because this is the current state of the art for large-scale astrophysical simulations. It remains to be seen whether implicit codes can be made substantially more efficient with GPU acceleration. As of this writing the implicit code we are using for comparison [17] runs at most several times faster on a GPU than a CPU for a 150-isotope network [18].)

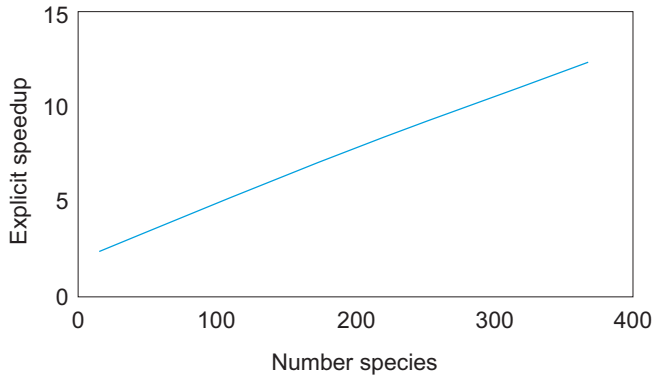

Fig. 4. The approximate intrinsic speedup factor as a function of network size for the new explicit methods relative to standard implicit methods. Adapted from results presented in [3]–[6].

to a large kinetic network, with the computation of the fluid dynamics implemented on the CPUs and the computation of the kinetic networks implemented on the GPUs. This framework describes qualitatively a large number of potential scientific applications in a variety of fields, but to be definite we shall emphasize astrophysical thermonuclear networks coupled to hydrodynamical simulations in explosive burning scenarios. Our reference example will correspond to a 150-isotope network containing 1604 reactions (the minimal realistic network of Figure 1(b)), integrated at a constant temperature of 7.10^9K and constant density of 10^8 g cm3, but we shall display calculations with as many as 365 network species and 4300 reactions. The hydro integrator takes an adaptive timestep Δt_{hydro} while the kinetic network is dormant. Then the updated hydrodynamical variables (temperature, density, ...) are held constant while the kinetic network is integrated over the interval Δt_{hydro} using adaptive network timesteps Δt_{net}. The updated abundance variables and the energy released by the kinetic network are then passed from the kinetic network to the hydro integrator, which uses these and the equation of state to set the initial conditions for the next hydro integration timestep, and so on. The hydro timestep Δt_{hydro} and the network timestep Δt_{net} are different timescales $\Delta t_{hydro} \gg \Delta t_{net}$ and should not be confused in the following discussion. For our purposes, the hydro integration may be viewed as a black box to which the kinetic network is coupled by two-way transfer of information, and the only role of the current hydro timescale is to set the interval Δt_{hydro} over which the kinetic network is to be integrated using adaptive timesteps Δt_{net}.

The schematic implementation for a single network is shown in Figure 5. In Figure 5 we illustrate the basic structure of our example calculation. We shall refer to the code running on the CPU as the host and the code running on the GPU as the kernel in the following discussion. In a setup step that is executed once for each overall problem, library parameters required to calculate the temperature and density-dependent reaction rates are copied to the GPU and are resident on the GPU for the duration of the calculation. At the end of each

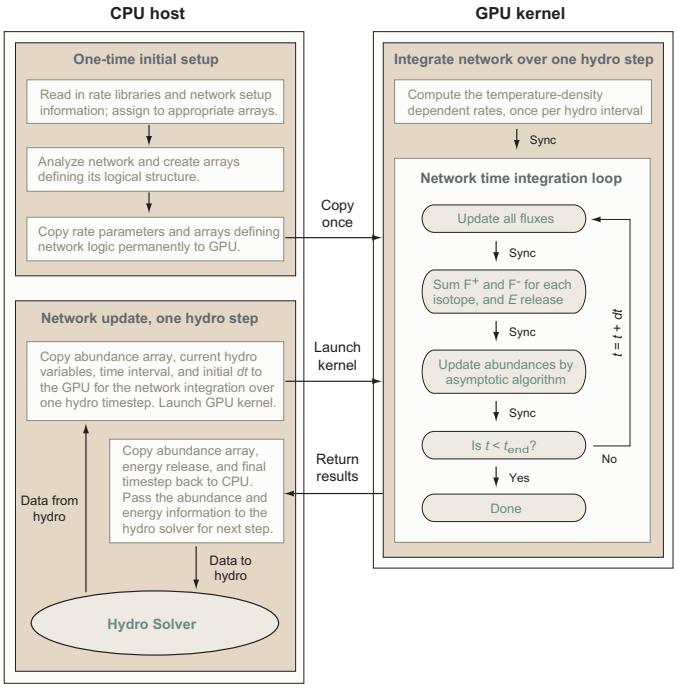

Fig. 5. Schematic flow for the example calculation presented here for a single kinetic net- work. The kinetic network is assumed to be coupled to the hydrodynamical integration by operator splitting. The kinetic network integration over a time interval corresponding to one hydro timestep is executed entirely on the GPU. Once the problem is set up, the only communication between CPU and GPU is to copy data from the last hydro timestep to the GPU, launch the kernel, and then copy the network integration results back to the CPU. The points labeled Sync in the GPU kernel are points where the algorithm requires that all threads be synchronized before proceeding because subsequent operations require the completed results of those threads.

hydrodynamical integration step a fully coupled CPU code would hold initial conditions for the kinetic network, which consists of the current temperature and density, and the current abundances for all species in the kinetic network. In a realistic simulation these values would be supplied to the CPU code by the hydrodynamical integration, but the source is irrelevant for our present tests and in our simulation we simply read in a trial set of initial conditions for a hydro timestep. One full kinetic network integration over a time interval corresponding to one hydrodynamical timestep then consists of the following steps.

1) Copy from the CPU to the GPU (1) a vector of current network abundances, (2) the temperature, density, and duration of the current hydro timestep Δt_{hydro}, and (3) a trial initial network timestep. For the representative 150-isotope network this corresponds to copying a total of 154 floating point numbers from the CPU to the GPU.
2) Launch a GPU kernel to integrate the kinetic network over the time interval corresponding to the hydro timestep.
3) When the integration on the GPU is complete, copy back to the CPU values of (1) the updated species abundances at the end of the kinetic network integration, (2) the integrated energy release over the kinetic network

integration, and (3) a final kinetic network integration timestep for use in setting the initial trial timestep in the next network integration. For the representative 150-isotope network this corresponds to copying a total of 152 floating point numbers from the GPU to the CPU.

In this scheme, during the kinetic network integration corresponding to one hydrodynamical timestep (see Figure 5) the network integration is done entirely on the GPU and the only communication between the CPU and GPU is at the beginning and end of the network integration. For the examples discussed here, the required data transfer between the CPU and GPU over each hydro integration step is thus 1 kB at the beginning and a similar amount at the end of each network integration. Thus the communication time is negligible fraction of the total network integration time. To our knowledge, this work constitutes nearly optimal implementation scenario that by far exceeds the state-of-the-art implementations currently available.

IV. Performance Analysis and Optimization Techniques for a Single Network

In this paper we will focus on the Kinetic network implementation and optimisation since it is the most expensive phase and the one that we are going to implement on GPU. In this section we will start our discussion by studying the performance behavior of one single network, proposing different optimization technique and algorithmic design and then we will study the case of many network simulations.

A. The Algorithm

In order to analyze and understand the kinetic simulation, we will start by describing the algorithm steps. In Algorithm 1, we did describe the different steps of the integration. The integration consists of a main loop until convergence and each iteration follows four major steps. Lets denote by s the number of spices. The first step of the algorithm, consists of populating the components F_i^+ and the components F_i^- that increases and deplete the abundance of each spice respectively. In order to perform this step efficiently, we propose to store all the F_i^+ and the F_i^- consecutively in two vectors F^+ and F^-. In our example here, the size of F^+ and F^- is in the range of 2720 elements for each. The computation is described in step 1 of Algorithm 1 and represented in Figure 6. It can be well implemented on GPU where each thread compute one element of F^+ and F^- in a coalescent order. However, it involves mapping of the indices to read the global flux F, but since F is of size "s" and can be cached, the overhead of this mapping is marginal. Then, –step 2–, for each spice, we have to compute the effect of the increasing and depleting abundance. In other term, it means that computing the sum of all the components $\sum F_{ij}^+$ and $\sum F_{ij}^-$ of Equation **??** that correspond to spices i. This step can be viewed as one tensor contraction computation that involves level 1 BLAS routine for norm computation. In our case, its implementation is not straightforward as the sum involves performing "$2 \times s$" summation of chunks of elements of F^+ and F^- of different sizes. It is described in step (2) of Algorithm 1 and illustrated in Figure 7, where every color corresponds to a chunk of elements related to spice i. The third and the fourth steps are straight forward. Step 3 update the flux F of size s by the stabilization formula that depends on the resulting summation of step 2 while step 4 checks the convergence criteria and prepares the parameters of the next iteration in case of non convergence. A simple

Algorithm 1 The kinetic integration algorithm.

copy data from CPU
while convergence **do**
  1-populate the F^+ and F^-
  for $i \in \{1, 2, \#coponents\}$ **do**
    $\mathsf{F}^+(i) = \mathsf{FFac}^+(i) \times \mathsf{Flux}(\mathsf{map}^+[i])$
    $\mathsf{F}^-(i) = \mathsf{FFac}^-(i) \times \mathsf{Flux}(\mathsf{map}^-[i])$
  end for
  2-compute the contribution of each $\sum F^+$ and $\sum F^-$
  for $k \in \{1, 2, \#spices\}$ **do**
    $\mathsf{sum}^+(k) = \sum F^+_{rk+}$
    $\mathsf{sum}^-(k) = \sum F^-_{rk-}$
    where rk^* is the indices of the contribution of spice k for F^+ and F^-
  end for
  3-update the Mass fraction X, the abundance Y and the flux F based on the resulting sum^+ and sum^-
  for $k \in \{1, 2, \#spices\}$ **do**
    $\mathsf{D}(k) = \mathsf{sum}^+ - \mathsf{sum}^-$
    $\mathsf{Y}(k) = function(\mathsf{D}, \mathsf{Y}, \mathsf{sum}^+, \mathsf{sum}^-)$
    $\mathsf{X}(k) = \mathsf{M}(k) * \mathsf{Y}(k)$
    $\mathsf{F}(k) = \mathsf{R}(k) * \mathsf{Y}[u] * \mathsf{Y}[v] * \mathsf{Y}[t]$
    where u, v, t are mapping indices for the abundance reactions of spice k,
    where R is the rate parameter, M is the Mass data, X is the Mass fraction, Y is the vector of abundance indices.
  end for
  4-check convergence and compute parameters of next iteration
end while
send data to CPU

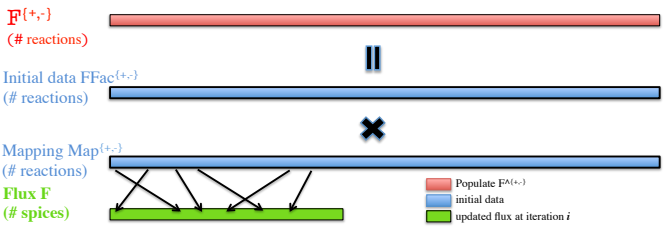

Fig. 6. Step 1 of Algorithm 1, populate the F^+ and F^-.

analyses of the steps of Algorithm 1 shows that it can be viewed as a set of tensor contraction summation and Level 1 BLAS operation on small sizes vectors, thus, a perfect memory-bound algorithm. Since our main focus is to describe a general

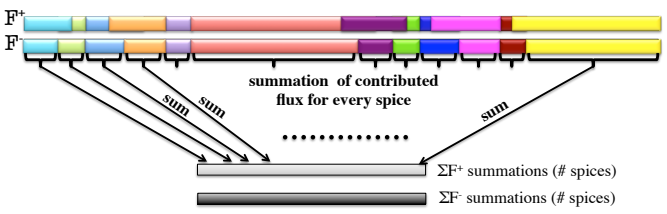

Fig. 7. Step 2 of Algorithm 1, compute the contribution of each spice.

framework of designing and optimizing a GPU kernel for batched computation on GPUs, the proposed analyses and optimisation techniques presented below hold for any memory-bound and tensor contraction algorithm.

B. Performance model

To evaluate the efficiency of our algorithms we derive theoretical bounds for the maximum achievable performance $P_{max} = Flop/T_{min}$, where $Flop$ is the flops needed and T_{min} is the fastest time to solution. Lets denote by s the number of spices and by $r+$ and r^- the total number of contributions (e.g., the increasing and the depleting abundance reactions) for all the spices, respectively, in other term the length of the flux vectors F^+ and F^-. We have $Flop \approx 2r^+ + 2r^- + 24s$. Since r^+ and r^- are within 5% of difference, let us consider the size of either to be r for simplicity, then $Flop \approx 4r + 24s$. Assuming that all the data fit into the fast cache, which might not be the case, but, just to derive the possible performance upper bound:

$$\begin{aligned} T_{min} = min(&T_{Read(F^+,Fac^+,Map^+,F^-,Fac^-,Map^-,F,Y,R,M)} \\ &+ T_{Compute} \\ &+ T_{Write(F^+,F^-,sumF^+,sumF^-,Y,F,X,D)}) \end{aligned} \tag{2}$$

Note that we have to read/write $6r + 10s$ double precision elements and $2r$ integer elements, or $52r + 80s$ Bytes. Thus, if the maximum achievable bandwidth is β (in Bytes/second), The minimal time can be expressed as:

$$T_{min} = (52r + 80s)/\beta \tag{3}$$

Since the computation is of order of $\mathcal{O}(1)$, and its execution time can be considered as marginal, this minimal time is theoretically achievable if the computation totally overlaps the data transfers and does not disrupt the maximum rate β of read/write to the GPU memory. The achievable bandwidth can be obtained by benchmarks, e.g., using NVIDIA's `bandwidthTest`. For example, on a K40 GPU with ECC on the peak is about 200 GB/s when all SMX of the GPU are loaded by more than one Thread-blocks each. Figure 8 illustrates the achievable bandwidth in function of the number of Thread-blocks which is the metric that we will use in our study.

C. Methodology and Performance Analysis

A recommended way of writing efficient memory bound GPU kernels is to use the GPU's shared memory – load it with

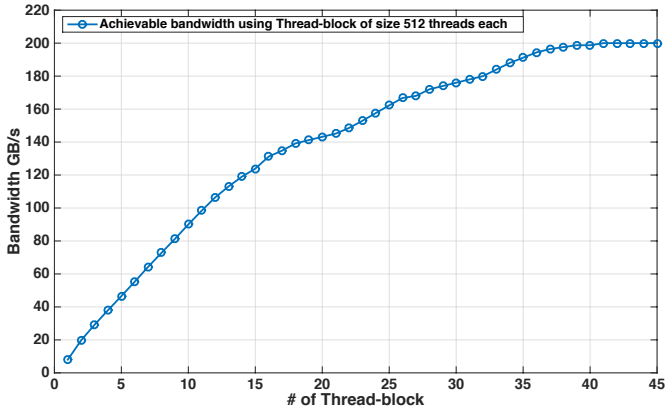

Fig. 8. The achievable bandwidth when varying the number of Thread-blocks running on the K40c GPU.

data and reuse that data in computations as much as possible. The advantage behind this idea is to do the maximum amount of computation before writing the result back to the main memory. However, the implementation of such technique may be complicated for the small problems considered as it depends on the hardware, the precision, and the algorithm. Moreover, our experience showed that this procedure might provide good performance for a single GPU Thread-block but is not that appealing for batched computation (where many kernels need to be executed simultaneously on the GPU) for two main reasons. First, the current size of the shared memory is 48 KB per streaming multiprocessor (SMX) for the newest Nvidia K40 (Kepler) GPUs, which is a low limit for the amount of the problem data that can fit at once. Second, completely saturating the shared memory per SMX can decrease the performance of memory bound routines, since only one thread-block will be mapped to that SMX at a time. Indeed, due to a low computational rate, the memory-bound algorithm will result in low occupancy, and subsequently poor core utilization.

To obtain a near optimal performance, we conduct an extensive study over the performance counters using the Nvidia profiler tools [19]. Our analysis concludes that in order to achieve an efficient execution for such memory bound computation, we need to maximize the occupancy and minimize the data traffic while respecting the underlying hierarchical memory design. Unfortunately, todays compilers cannot introduce highly sophisticated shared memory/register based loop transformations and, consequently, this kind of optimization effort should be studied and implemented by the developer. This includes techniques like reordering the data so that it can be easily vectorized, reducing the number of instructions so that the unit spends less time in decoding them, and try using a predefined loop boundary strategy to allow unrolling techniques.

We developed a first simple implementation denoted by "*version 1*" of Algorithm 1. Some collection of the hardware counter readings are shown in Figures 13-16. As observed, the reading/writing throughput data of such implementation is

very low and since such algorithm is memory bound (mostly requiring reading of a lot of data), one can expect very low performance behavior from such implementation. This version requires about 12.5 seconds to solve the integration of one kinetic network and is far from being close to the lower limit as shown in Figure 9. We performed a set of extensive experimentation and found that the third step (the summation step) is the most time consuming (about 95% of the total time). This summation involves vectors of variable length. In our example, we represent the length of each of these summation for both the increasing and the depleting flux ($\sum F_k^+$ and the $\sum F_k^-$) in Figure 10. Most of the summation are of size less than 40 except few of them where the size is about 400. Our experiments also showed that 73% of the summation time is drained in the large vector (number of elements larger than 40) and about 27% in all the rest. There is many reasons for this expensive cost. First, this variable size summation involves irregular data accesses which generates a lot of memory bank conflict and threads divergence. Second, The variation of the length of each summation makes it hard for the compiler to perform unrolling and optimisation. Third, the summation is kind of a sequential process where all the threads have to wait the results which is synchronizing as well. This is verified by the performance counters measured in Figures 13-16. The read throughput is about 1.5 GB/s compared to about 8.1 GB/s that one Thread-block can achieve (see Figure 8). This irregular access generate huge amount on integer instructions that the Nvidia profiler wasn't able to measure. The profiler returned overflow for the number of integer instructions. Most of the time the threads are stalling for either execution dependency or data request (see Figure 15).

A possible optimisation for the summation step is to implement a tree reduction to perform the summation. We developed a tree reduction routine to compute the summation. It is similar to the one proposed by Nvidia [20] and also used in Magma [21] for the norm and dot product computation and it has been proved to be the most efficient technique for such calculations on vector of length larger than a warp at least. This involves copying the elements of the corresponding vector into shared memory, then perform a tree reduction as described in Figure 12. We also included all possible optimisations such as, decreasing the amount of instruction by unrolling the last warp meaning unrolling the last 6 iterations of the inner most loop, as well as implementing a template interface where the blocksize is predefined at compile time which allows the compiler to perform a complete unrolling of the loop. However, we should mention that the tree reduction is powerful when the length of the vector is large enough (at least larger than 64) and slower than the sequential summation otherwise. We proposed our "*version 2*" implementation, where we used the tree reduction summation whenever possible and kept the sequential summation for small vectors length. This later turned to be twice faster than the previous implementation (see Figure 9), but reach about 30% of the optimal theoretical limit described in Equation (3) and drawn in Figure 9. Version 2 requires about 5.1 seconds to perform the integration over 32182 iterations while the optimal time computed from Equation (3) is equal to 1.3 seconds. The performance metrics collection resulting from version 2 are illustrated in Figure 13-16. It is clear from Figure 13 that the resulting throughput efficiency of this version is about 3.3 GB/s which is about half what one Thread-block can achieve (see Figure 8). The percentage of the stalling reasons is better than version 1 but the amount of data to write remains the same (see Figure 16). This is not surprising since it follows the same algorithmic order as version 1, the only difference is that it use the reduction tree summation for the large vectors which work on shared memory only and thus do not decreases any data writes instructions. The drawback of this version is that it optimizes the summation of the large vectors, but many threads remain idle during this step. It do not resolve the issue of branching as well. We did several attempts to parallelize the sequential summation over different many warps but the resulting implementation was slower.

The technique to minimize bank conflict and to achieve higher load throughput is to minimize branching, using all available threads to load coalescent data in a consecutive order and to predefine fixed loop bound if possible to allow compiler unroll some portion of the code, in particular, the expensive section (e.g., step 3). Hence, loop unrolling, loop peeling, and section interchange can be useful techniques to achieve our goal. Nevertheless, since the size of the integration data is larger than the available shared memory, we might expect to never reach the optimal timing of Equation (3). Indeed, technique like locality and reusing data can be very helpful.

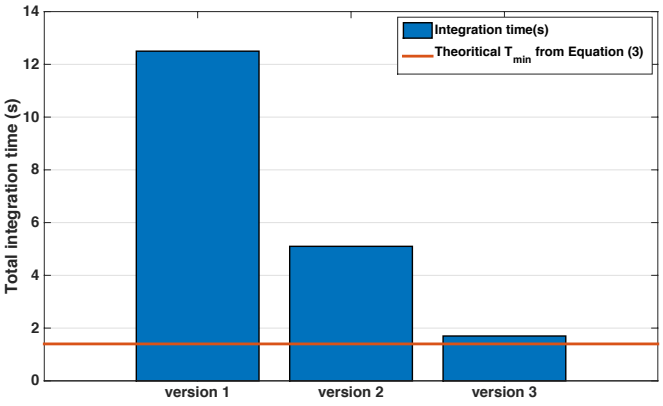

Fig. 9. The elapsed time to perform the computation of the whole kinetic integration process.

Thus, in our third design, we propose to reorder the data storage as well as the computation "when possible" in such a way to increase data locality and expose more parallelism for vectorization. We propose to remap the storage of the flux components data F^+, F^-, $FFac^+$, $FFac^-$, Map^+ and Map^- to a $2D$ matrix format (m,nb) as illustrated in Figure 12, and where the size nb is predefined fixed at compile time. This way the computation of both F^* and $\sum F^*$ ("$*$" corresponds to either "$+$" or "$-$") can be executed efficiently in parallel using all the threads of the Thread-block and where a warp

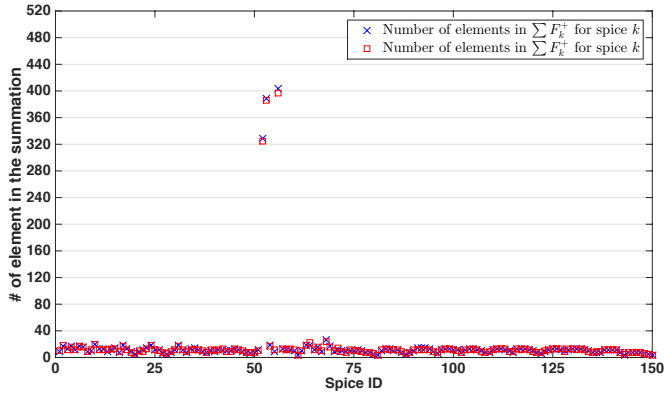

Fig. 10. The number of elements in the $\sum F_k^+$ and the $\sum F_k^-$ summation for spice k.

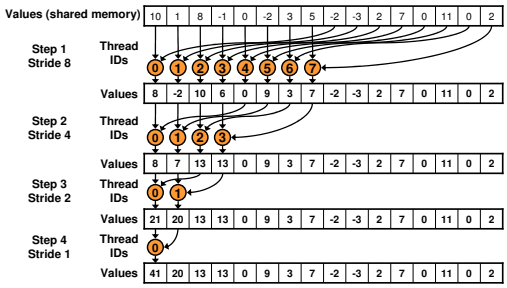

Fig. 11. The parallel summation techniques based on reduction tree (courtesy from [20]).

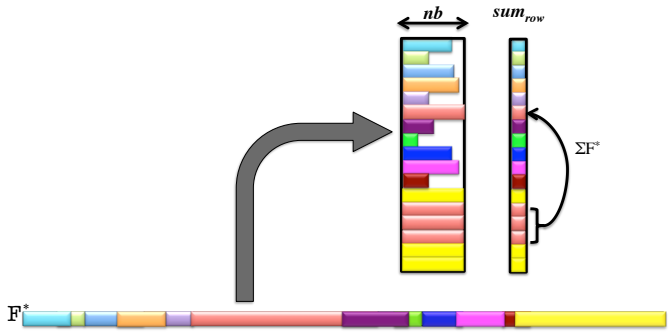

Fig. 12. Reshaping data structure and reorder the computations for continuous access and improved vectorization as well as for exhibiting more parallelism.

will be working of 32 consecutive data elements. Moreover, the loop is ordered by the index of the columns "j" and since nb is predefined fixed, the compiler is able to unroll the loop over j allowing predicted data access pattern which increases the load throughput. Within this proposition, the summation process has to be split over two phases. First, phase (a), the rowwise summation "sum_{row}", meaning summation over the columns of F^+ and F^-. Second, phase (b) the spices that its reactions components are larger than nb will own more than one row of F^* and thus a fast summation over its corresponding sum_{row} is needed to get $\sum F^*$ and finalize step 3. For example,

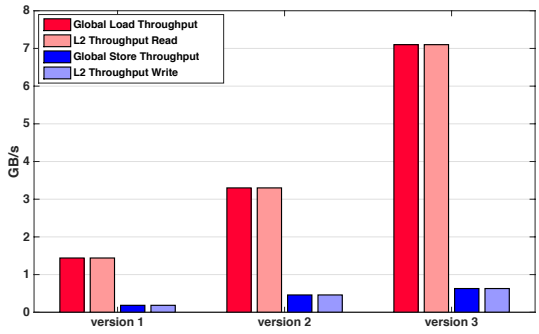

Fig. 13. Performance counters measurement: global memory load/store efficiency

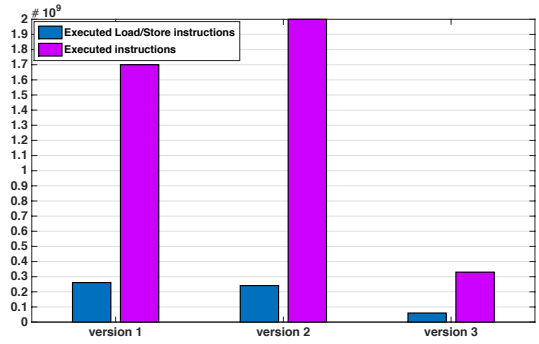

Fig. 14. Performance counters measurement: executed load/store instructions and executed total instructions

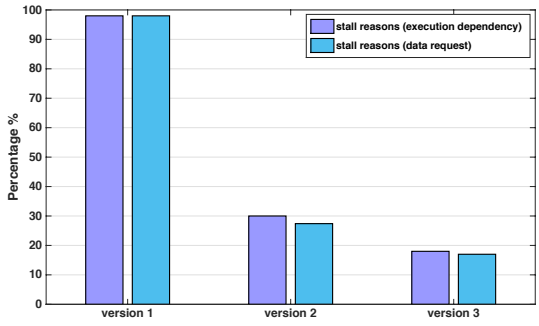

Fig. 15. Performance counters measurement: stalling percentage and reasons

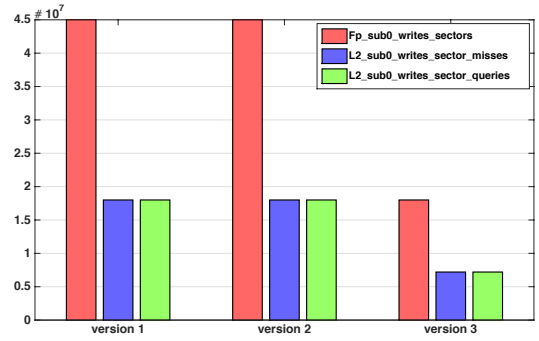

Fig. 16. Performance counters measurement: number of write requests from L2 to DRAM (red), number of write requests from L1 to L2 (green), which is equal to the number of write misses in L2 (blue)

the chunk of elements colored in "red" in Figure 12 is split over 4 rows in the new data structure. Thus, its final sum is the summation of its corresponding sum_{row} that have been computed in phase (a). In our example nb was equal to 20 where most of the contributions are within one row. Note that in order to increase data reuse, we can fuse step 2 and step 3 on a warp fashion style. During the computation of every element of F^*, its value can be directly accumulated into a register or a shared memory variable "v". Once a row of F^* is computed, its summation is also computed into v. As consequence, we expect that this version will exhibit better locality, less instructions and higher throughput. Figures 13-16 shows that this version exhibit very high read throughput, close to the peak bandwidth, and requires 7 times less instructions than the previous two proposed versions. Moreover, the percentage of time the threads are idles is about half of what version 1 expose and the amount of data to be written to the main memory is also about 2.3 $\times$ less than the other versions (see Figure 16). We also propose to merge the F^+ and F^- data structure into one matrix, such a way to expose more parallelism which increases the occupancy of the kernel and minimize the amount of time threads could be "idles". This version run close to the optimal bound as shown in Figure 9. This GPU implementation is about 6 $\times$ faster than its CPU counter part. Note that the CPU implementation of the explicit integration approach presented in [3]–[6] was also about 6 $\times$ faster than the state-of-the art implicit method for the 150-isotope example (see Figure 4).

V. Performance Analysis for Multiple Network

A factor of 36 speedup (6$\times$ from the explicit approach and 6$\times$ from the GPU acceleration) over current state of the art for thermonuclear networks coupled to hydrodynamics is impressive, but in reality the GPU running a single kinetic network is highly under-utilized: There is only one Thread-block performing the computation of one network, and even within this Threads-block, only 512 Threads are used, which means that only one SMX of the 15 SMX of the K40c is used and not fully loaded. In typical applications the CPUs of a compute node will host multiple fluid dynamics zones and each zone has an independent network reflecting the conditions in that zone. This motivate us to investigate launching many networks running in parallel on the GPU, as illustrated schematically in Figure 17.

There are two possible design strategies to execute concurent networks on a GPU: either using CUDA streams or using batched computations [22]. The batched computation recently attracted increased research interest due to its feature in exploiting all the SMX of a GPU, providing effecient high-performance computation for problems of very small size which are similar to our test cases. We investigate and developed prototypes using both design strategies. For the cuda-streamed design, we deploy a number of OpenMP threads, with one asynchronous kernel launched by each thread on a different cuda-stream while for the batched design it is only one kernel that internally parallelize the networks over many Thread-blocks. For the cuda-stream design, we observe that the time

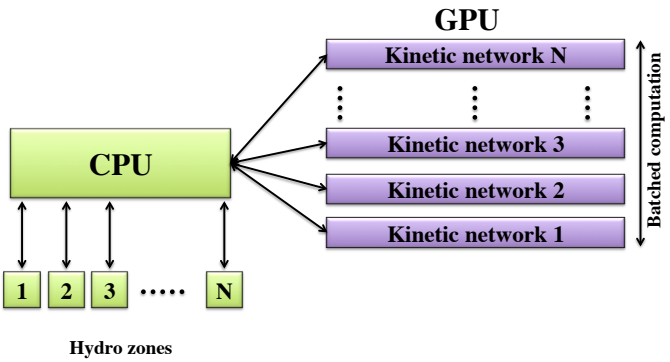

Fig. 17. Integrating multiple kinetic networks in parallel on a GPU by using batched computation.

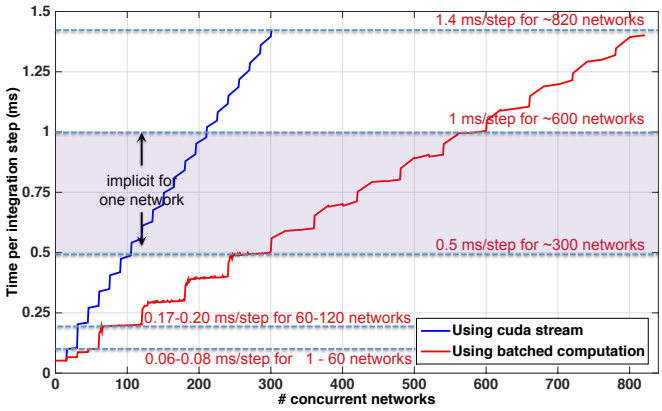

Fig. 18. Massively parallel integration of many 150-isotope networks on a Tesla K40c GPU. The approximate range of integration times for integrating a single network on a CPU with current implicit codes is indicated by the horizontal band. In this example the GPU has 15 available streaming multiprocessors and we have used batched methods to launch 4 networks per multiprocessor. This accounts for the step structure with width $4 \times 15 = 60$ networks.

to solve upto 15 networks is roughly the same as solving one network and the time to solve 15 to 30 network is about twice the time of one network and so on. The period of 15 concurrent networks in the steps reflects the availability of 15 streaming multiprocessors on the Kepler K40c GPU microarchitecture.

Timing results for the launch of many representative 150-isotopes networks running in parallel using the batched computation are displayed in Figure 18. We see that the time to run n networks scales almost perfectly. The batched design is advantageous, it follows the same stair shape observed for cuda-stream, but with a period of 60. The batched computation takes advantage of all the 15 SMX of the GPU but also is able to schedule about 3-4 Thread-block per SMX. As a result our batched design was about 3-4 times faster than the cuda-stream design as shown in Figure 18. and we are able to solve about 60 networks in the same time as one network. The slight rise in execution time on any given period which presumably reflects a small increase in CPU overhead associated with the

preparation of increasing numbers of networks concurrently, since the timing includes CPU processing and copying overhead as well as kernel execution time.

The results implied by Figure 18 have large implications for simulations in a variety of scientific fields. They demonstrate that not only can a single realistic network be run fast enough now to couple to fluid dynamics, but in fact many such networks can execute in a short enough time to make the simulation feasible. Here we see that the new algorithms are capable of running $\sim 300 - 600$ realistic (150-isotope) networks in the same length of time that a standard implicit code can run one such network on a CPU.

VI. Conclusion

In summary, our new algebraically-stabilized explicit algorithms are intrinsically faster than standard implicit algorithms by factors of 5-10 for networks with several hundred species, primarily because they require neither matrix inversions nor interations. For a single network, GPU acceleration increases this to a factor of 20-40 for networks with several hundred species. The GPU is capable of running 250-500 networks in parallel in about the same length of time that a standard implicit code can run one such network on a CPU (this is also true presently within a factor of two or so for an implicit code accelerated with a GPU). These potential orders of magnitude increases in computational efficiency imply that much more realistic kinetic networks coupled to fluid dynamics are now feasible in a broad range of large problems in astrophysics and other disciplines. Presently we are investigating applications of these new methods to large-scale computer simulation for Type Ia supernovae, neutrino transport in core-collapse supernovae, real-time atmospheric forecasting, climate science, and materials science.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CSR 1514286, NVIDIA, the Department of Energy, and in part by the Russian Scientific Foundation, Agreement N14-11-00190.

References

[1] O. E. S and B. J. P, "Numerical simulation of reactive flow," 2005.

[2] W. R. Hix and B. S. Meyer, "Thermonuclear kinetics in astrophysics," *Nuclear Physics A*, vol. 777, pp. 188 – 207, 2006, special Isseu on Nuclear Astrophysics. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0375947404011005

[3] M. Guidry, "Algebraic stabilization of explicit numerical integration for extremely stiff reaction networks," *Journal of Computational Physics*, vol. 231, no. 16, pp. 5266 – 5288, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999112002070

[4] M. W. Guidry, R. Budiardja, E. Feger, J. J. Billings, W. R. Hix, O. E. B. Messer, K. J. Roche, E. McMahon, and M. He, "Explicit integration of extremely stiff reaction networks: asymptotic methods," *Computational Science & Discovery*, vol. 6, no. 1, p. 015001, 2013. [Online]. Available: http://stacks.iop.org/1749-4699/6/i=1/a=015001

[5] M. W. Guidry and J. A. Harris, "Explicit integration of extremely stiff reaction networks: quasi-steady-state methods," *Computational Science & Discovery*, vol. 6, no. 1, p. 015002, 2013. [Online]. Available: http://stacks.iop.org/1749-4699/6/i=1/a=015002

[6] M. W. Guidry, J. J. Billings, and W. R. Hix, "Explicit integration of extremely stiff reaction networks: partial equilibrium methods," *Computational Science & Discovery*, vol. 6, no. 1, p. 015003, 2013. [Online]. Available: http://stacks.iop.org/1749-4699/6/i=1/a=015003

[7] B. Brock, A. Belt, J. J. Billings, and M. Guidry, "Explicit integration with gpu acceleration for large kinetic networks," *Journal of Computational Physics*, vol. 302, pp. 591 – 602, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999115006063

[8] M. D. R, "New quasi-steady-state and partial-equilibrium methods for integrating chemically reacting systems," 1999.

[9] O. Messer, J. Harris, S. Parete-Koon, and M. Chertkow, "Multicore and accelerator development for a leadership-class stellar astrophysics code," in *Proceedings of "PARA 2012: State-of-the-Art in Scientific and Parallel Computing."*, 2012.

[10] A. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Luc, M. Nooijene, R. Pitzerf, J. Ramanujamg, P. Sadayappanc, and A. Sibiryakovc, "Automatic code generation for many-body electronic structure methods: the tensor contraction engine," *Molecular Physics*, vol. 104, no. 2, pp. 211–228, 2006.

[11] J. L. Khodayari A., A.R. Zomorrodi and C. Maranas, "A kinetic model of escherichia coli core metabolism satisfying multiple sets of mutant flux data," *Metabolic engineering*, vol. 25C, pp. 50–62, 2014.

[12] S. N. Yeralan, T. A. Davis, and S. Ranka, "Sparse mulitfrontal QR on the GPU," University of Florida Technical Report, Tech. Rep., 2013. [Online]. Available: http://faculty.cse.tamu.edu/davis/publications_files/qrgpu_paper.pdf

[13] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra, "A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU," in *IEEE 28th International Parallel Distributed Processing Symposium (IPDPS)*, 2014.

[14] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 135–158, Feb. 2004. [Online]. Available: http://dx.doi.org/10.1177/1094342004041296

[15] J. Molero, E. Garzón, I. García, E. Quintana-Ortí, and A. Plaza, "Poster: A batched Cholesky solver for local RX anomaly detection on GPUs," 2013, PUMPS.

[16] M. Anderson, D. Sheffield, and K. Keutzer, "A predictive model for solving small linear algebra problems in gpu registers," in *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, 2012.

[17] W. Hix and F.-K. Thielemann, "Computational methods for nucleosynthesis and nuclear energy generation," *Journal of Computational and Applied Mathematics*, vol. 109, no. 12, pp. 321 – 351, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0377042799001636

[18] J. A. Harris, "private communication."

[19] A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb, "Parallel performance measurement of heterogeneous parallel systems with gpus," in *Proc. of ICPP'11*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 176–185.

[20] "Optimizing parallel reduction in cuda," http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf, 2007.

[21] "Matrix algebra on GPU and multicore architectures (MAGMA), MAGMA Release 1.6.0," 2014, available at http://icl.cs.utk.edu/magma/.

[22] A. Haidar, T. Dong, S. Tomov, P. Luszczek, and J. Dongarra, "Framework for Batched and GPU-resident Factorization Algorithms to Block Householder Transformations," in *ISC High Performance*, Springer. Frankfurt, Germany: Springer, 07-2015 2015.