# 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*†, Daniel Shyles*, Jack Dongarra*†‡
*University of Tennessee, Knoxville, TN, USA
†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 that solve efficiently $N$ coupled ordinary differential equations (subject to initial conditions) on modern GPUs. We take representative test cases (Type Ia supernova explosions) and demonstrate two or more orders of magnitude increase in efficiency for solving such systems (of realistic thermonuclear networks coupled to fluid dynamics). This 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 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 for 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.**

## I. Introduction

Many important physical processes can be modeled by the coupled evolution of a reaction (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. 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.

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], [4], [5], [6], [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 time-steps 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× faster than the best implicit codes [3], [4], [5], [6], [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], [4], [5], [6], [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 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 Advancements and Novel Approach

### A. Formulation and Novel Approach

The prototype implementation that we describe here is based on an operator split formulation of fluid dynamics coupled to a large kinetic network, where the hydrodynamical solver is evolved for a numerical timestep $\Delta t_{hydro}$ holding network parameters constant, and then the network is integrated over the interval $\Delta t_{hydro}$ using adaptive network timesteps $\Delta t_{net}$ holding the new hydrodynamical variables constant. The computation of the fluid dynamics is implemented on the CPUs while the one of the kinetic networks is 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 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 1.

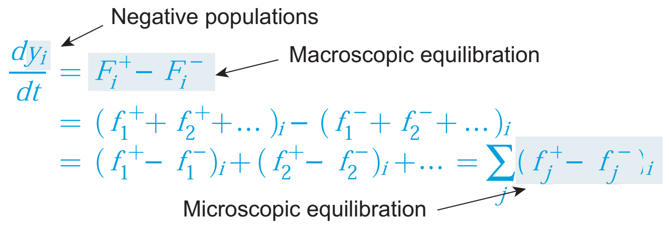
Negative populations
Macroscopic equilibration
Microscopic equilibration
$\frac{dy_i}{dt} = F_i^+ - F_i^-$
$= (f_1^+ + f_2^+ + \ldots)_i - (f_1^- + f_2^- + \ldots)_i$
$= (f_1^+ - f_1^-)_i + (f_2^+ - f_2^-)_i + \ldots = \sum_j (f_j^+ - f_j^-)_i$

Fig. 1. 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.

In this expression, the $y_{i,(i=1\ldots N)}$ 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 $y_i$, generally coupled to each other because of the dependence of the fluxes on the different $y_j$.

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.

### 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 1. 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. 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. We have documented the details of this novel approach and documented a set of experiments that shows a speed of $\sim 6$ times faster than that of the state of the art implicit code for this problem [3], [4], [5], [6], [7].

## III. Performance Analysis and Optimization Techniques for a Single Network

In this paper we focus on developing high-performance GPU algorithms, implementations, and optimization techniques for the Kinetic network, which is the the most expensive phase of the simulations of interest. We start by studying the performance behavior of a single network, proposing different optimization techniques and algorithmic designs, and then we study the case of many network simulations.

### A. The Algorithm

In order to analyze and understand the computational challenges of the kinetic simulation, we concentrate on the main steps of the integration process, described in Algorithm 1. The integration consists of a main loop until convergence. Each iteration follows four major steps. In our example, the average number of iterations is 32,182. We denote by $s$ the number of species. The first step of the algorithm populates the components $F_i^+$ and $F_i^-$ that increase and deplete, respectively, the abundance of each species. 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 the sizes of $F^+$ and $F^-$ are in the range of 2,720 elements. The computation is described in step 1 of Algorithm 1. An efficient implementation is to assign to each thread the computation of one or more elements of $F^+$ or $F^-$ in a coalescent order. Next, in step 2, we compute the effect of the increasing and depleting abundance for each species. In other words, for every species $k$, we compute $\sum_j f_{jk}^+$ and $\sum_j f_{jk}^-$. This step can be viewed as one tensor contraction computation that involves Level 1 BLAS routine. The implementation is not straightforward, as it consists of "$2 \times s$" summations of chunks of elements of $F^+$ and $F^-$ that are of different length. The third and the fourth steps are straightforward. Step 3 updates the flux $F$ of size $s$ by the stabilization formula that depends on the result of the summation from step 2, while step 4 checks the convergence criteria, and prepares the parameters of the next iteration in case of non-convergence.

In conclusion, our analyses shows that computationally the algorithm can be viewed as a set of tensor contractions and Level 1 BLAS operations on small size vectors. Thus, this is a memory-bound algorithm. Since our main focus is to describe a general framework of designing and optimizing a GPU kernel for batched computations on GPUs, the proposed analyses and optimization techniques presented below hold for any memory-bound and tensor contraction algorithm.

### B. Methodology and Performance Analysis

A recommended way of writing efficient memory bound GPU kernels is to use the GPU's whole shared memory, load it with data, and reuse that data in the computations as much as possible. The goal of 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 sensitive and not portable as it depends on the hardware, the precision, and the algorithm. Moreover, our experience showed that this procedure might provide good performance when only one Thread-block (TB) is running but is not that appealing for batched computation (where many TBs 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 Nvidia K40 (Kepler) GPUs, which is low for the amount of data that we want to load and use. Second, completely saturating the shared memory of a SMX by one TB can decrease the performance of the memory bound routines, since only that TB will be mapped to the SMX (no shared memory is available to host another TB) while our bandwidth benchmark showed that an SMX requires to host more than one TB in order to achieve the maximal bandwidth that can be retrieved from the hardware. Indeed, due to its low computational rate, the memory-bound algorithm will result in low occupancy, and subsequently poor core utilization.

**Algorithm 1** The kinetic integration algorithm.

```
copy data from CPU
while convergence do
  // 1. Populate $F^+$ and $F^-$
  for $i \in \{1 .. \#components\}$ 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 $k \in \{1 .. \#species\}$ do
    $\mathsf{sum}^+(k) = \sum_j f_{jk}^+$
    $\mathsf{sum}^-(k) = \sum_j f_{jk}^-$
    where $jk$ are the indices of the fluxes $f_{jk}^*$ from species
      $j$ to species $k$ in $F^+$ and $F^-$
  end for
  // 3. Update the Mass fraction $X$, the abundance $Y$ and
  //   the flux $F$ based on the resulting $\mathsf{sum}^+$ and $\mathsf{sum}^-$
  for $k \in \{1 .. \#species\}$ do
    $\mathsf{D}(k) = \mathsf{sum}^+(k) - \mathsf{sum}^-(k)$
    $\mathsf{Y}(k) = function(\mathsf{D}, \mathsf{Y}, \mathsf{sum}^+, \mathsf{sum}^-)$
    $\mathsf{X}(k) = \mathsf{M}(k) * \mathsf{Y}(k)$
    $\mathsf{F}(k) = \mathsf{R}(k) * \mathsf{Y}[u] * \mathsf{Y}[v] * \mathsf{Y}[t]$
    where $u, v, t$ are mapping indices for the abundance
    reactions of species $k$, $\mathsf{R}$ is the rate parameter, $\mathsf{M}$
    is the Mass data, $\mathsf{X}$ is the Mass fraction, and $\mathsf{Y}$ is
    the vector of abundance.
  end for
  // 4. Check convergence and compute parameters of
  //   next iteration
  ...
end while
send data to CPU
```

To obtain a near optimal performance, we conducted an extensive study over the performance counters using the Nvidia profiler tools [17]. Our analysis concluded 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 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 targeting the use of predefined loop boundary strategies in order to enable loop unrolling techniques.

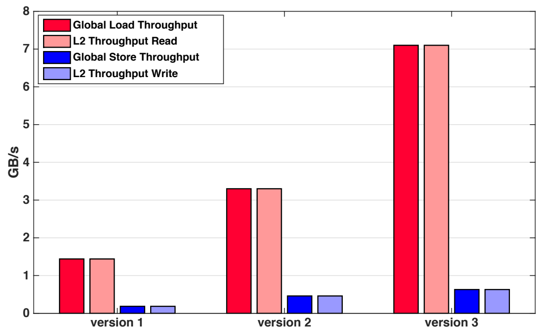

Fig. 2. Performance counters measurement: global memory load/store efficiency

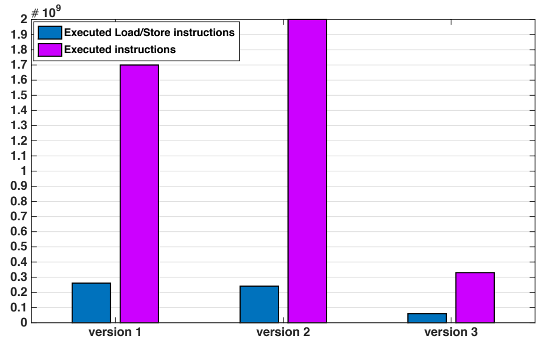

Fig. 3. Performance counters measurement: executed load/store instructions and executed total instructions

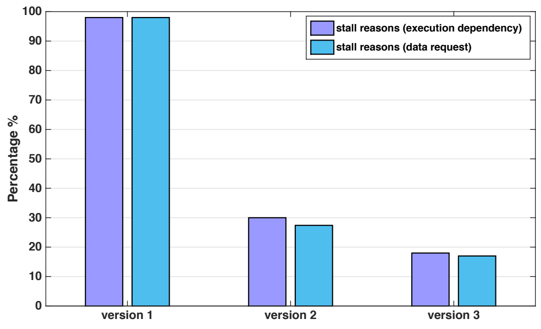

Fig. 4. Performance counters measurement: stalling percentage and reasons

First, we developed a reference implementation of Algorithm 1, denoted by "*version 1*". A collection of the hardware counter readings is shown in Figures 2 to 5. The data

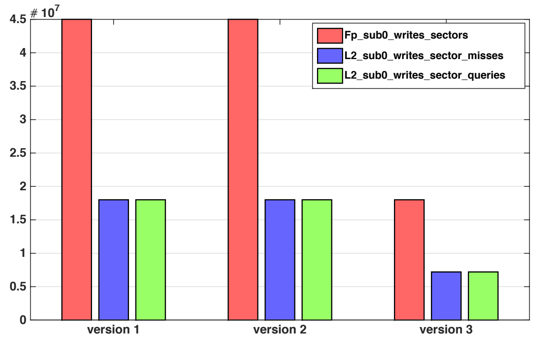

Fig. 5. Performance counters measurement: number of write requests from L2 to DRAM (red), number of write requests from L1 to L2 (green), which is equal to the number of write misses in L2 (blue)

read/write throughput of such implementation is very low and since the algorithm is memory bound (mostly reading data), one can expect a very low performance. This version requires about 12.5 seconds to solve the integration of one kinetic network and is far from the theoretical lower limit, which we derive to be about 1.3 seconds (see Figure 6). We performed a set of extensive experiments 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. For our example, we analyzed the length of each of these summations and found that most of them are of size less than 40, except for a few of them, where the size is about 400. Our experiments also showed that 73% of the summation time is spent in computations involving large vectors (number of elements larger than 40) and about 27% in all the rest. There are several reasons for step 3 being slow. First, the variable size summation involves irregular data accesses which generate many memory-bank conflicts and high thread divergence. Second, the variation of the length of each summation makes it hard for the compiler to perform unrolling and optimizations. Third, parallel variable-size summations involve synchronizations, thread idle time, and load imbalance. This is verified by the performance counters illustrated in Figures 2-5. The read throughput obtained from *version 1* is about 1.5 GB/s, which is very low compared to about 8.1 GB/s that one TB can achieve based on a bandwidth benchmark that we performed. This irregular access generates huge amount of integer instructions that the Nvidia profiler was not able to measure. The profiler returned overflow for the number of integer instructions. We found that for most of the time, the threads are stalling for either execution dependency or data requests (see Figure 4).

A possible optimization for the summation step is to implement it as a tree reduction. We developed a tree reduction routine to compute the summation. It is similar to the one proposed by Nvidia [18], which has been proven to be the most efficient technique for such calculations on vector of length larger than two warps. The computation involves copying the elements of the corresponding vector into shared memory, and then performing a tree reduction. We also included a number of optimizations, including, decreasing the amount of instructions 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. This allows the compiler to perform a complete unrolling of the loop. However, we should mention that our study showed that the tree reduction is powerful when the length of the vector is larger than 64, and slower than the sequential summation otherwise. Therefore, we developed a "*version 2*" implementation, where we used the tree reduction summation for large sizes, and kept the sequential summation for small vector lengths. This version turned to be twice faster than the previous implementation, but is about 4 times more expensive than the optimal theoretical limit, as shown in Figure 6. Version 2 requires about 5.1 seconds to perform the integration over the $32,182$ iterations, while the optimal time is equal to 1.3 seconds. The performance metrics collection resulting from version 2 is illustrated in Figures 2-5. It is clear from Figure 2 that the resulting throughput efficiency of this version, which is about 3.3 GB/s, is about half of what one Thread-block can achieve (8.1 GB/s). The percentage of the stalling reasons is better than version 1, but the amount of data to write remains the same (see Figure 5). This is not surprising since version 2 follows the same algorithmic order as version 1; the only difference is that it uses the reduction tree summation for large vectors, which works on shared memory only, and thus, does not decrease any data write 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 does not resolve the issue of branching as well. We did several attempts to parallelize the sequential summation over different warps but the resulting implementation was slower.

The technique to minimize bank conflicts 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 bounds, if possible, to allow the compiler to unroll some portions of the code (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 expect to never reach the optimal timing, which assumes that data is carried from main memory only once, and sits in the cache all the way till the end of the computation. Indeed, techniques like locality and reusing data can be very helpful.

Thus, in our third design, we propose to reorder the data storage as well as the computation, "when possible", in order 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 ($m$, $nb$), as illustrated in Figure 7, where the size $nb$ is 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 (where a warp will be working of 32 consecutive data elements). Moreover, the loop is ordered by the index

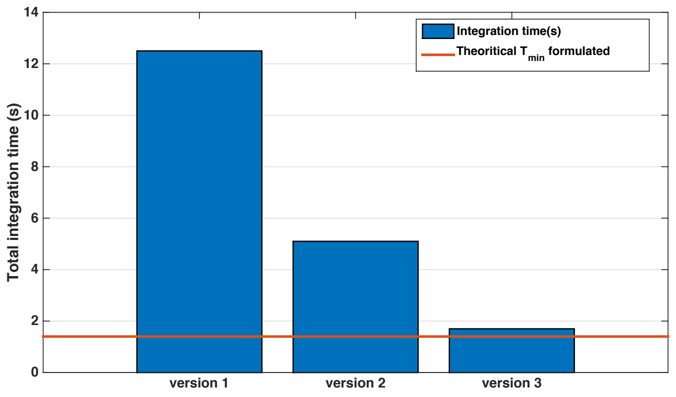

Fig. 6. The elapsed time to perform the computation of the whole kinetic integration process.

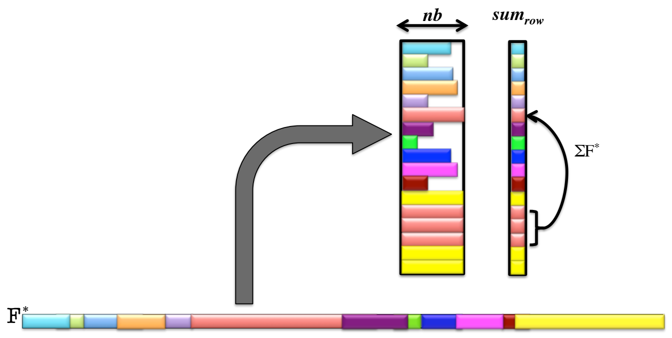

Fig. 7. Reshaping data structures and reordering the computations for continuous access and improved vectorization, as well as for higher parallelism.

of the columns "$j$" and since $nb$ is predefined, the compiler is able to unroll the loop over $j$, allowing predicted data access patterns which increases the load throughput. Within this proposition, the summation process must be split over two phases. First, phase (a), is the row-wise summation "$sum_{row}$", i.e., summation over the columns of $F^+$ and $F^-$. Second, phase (b), is for the species with reaction components larger than $nb$; they 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, the chunk of elements colored in "red" in Figure 7 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 at a warp level. During the computation of every element of $F^*$ (step 2), its value can be directly accumulated into a register or a shared memory variable "$v$". Once a row of $F^*$ is computed, its summation $sum_{row}$ is also computed in $v$. As a consequence, we expect that this version will exhibit better locality, less instructions, and higher throughput. Figures 2-5 show that this version exhibit very high read throughput, close to the peak bandwidth, and requires $7\times$ less instructions than the previous two versions. Moreover, the percentage of time the threads are idles is about half of that for version 1, and the amount of data to be written to the main memory is about 2.3× less than the other versions (see Figure 5). We also propose to merge the $F^+$ and $F^-$ data structure into one matrix in order to expose more parallelism, which increases the occupancy of the kernel and minimizes the amount of time threads "idle". This version runs close to the optimal bound, as shown in Figure 6. This GPU implementation is about 6× faster than its CPU counter part. Note that the CPU implementation of the explicit integration approach presented in [3], [4], [5], [6] was also about 6 × faster than the state-of-the art implicit method for our 150-isotope example.

## IV. Performance Analysis for Multiple Network

A factor of 36 speedup (6× from the explicit approach and 6× from the GPU acceleration) over current state of the art for thermonuclear networks coupled to hydrodynamics is impressive, but the GPU is computing a single kinetic network, and thus is highly under-utilized. There is only one Thread-block performing the computation of one network. In typical applications the CPUs will host multiple independent fluid dynamics zones, which results in independent networks. This motivates us to investigate launching many networks in parallel on the GPU, as illustrated schematically in Figure 8.

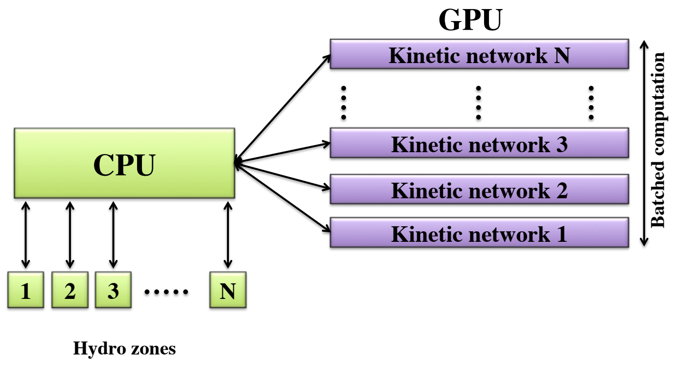

Fig. 8. Integrating multiple kinetic networks in parallel on a GPU by a batched computation.

There are two possible design strategies to execute concurent networks on a GPU: either using CUDA streams or using batched computations [19]. 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 [20], which are similar to our test cases. We investigated and developed prototypes using both design strategies.

Timing results for the launch of many representative networks of 150-isotopes running in parallel using either the streamed or the batched implementation are displayed in Figure 9. We see that the time to run $n$ networks scales almost perfectly. For the cuda-stream design, we observe that the time to solve up to 15 networks is roughly the same as solving one network. The period of 15 concurrent networks in the steps reflects the availability of 15 SMX of the K40c GPU.

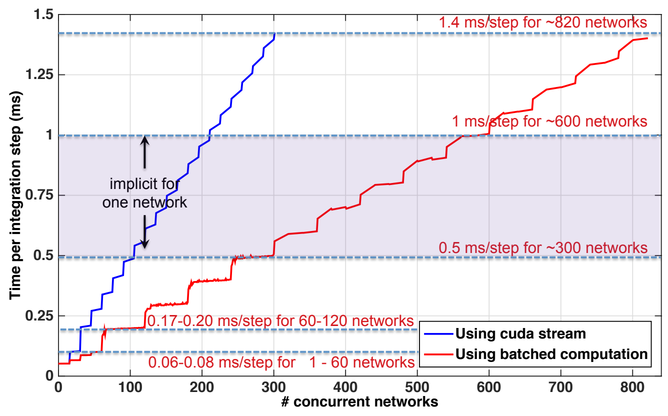

Fig. 9. Massively parallel integration of many 150-isotope networks on a Tesla K40c GPU. The approximate range of integration times for integrating a single network on a CPU with current implicit codes is indicated by the horizontal band. In this example the GPU has 15 available streaming multiprocessors 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.

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 is also able to schedule about 3-4 Thread-block per SMX. As a result, our batched design was about 3-4× faster than the cuda-stream design, as shown in Figure 9. The slight rise in execution time on any given period reflects a small increase in the CPU overhead associated with the preparation of increasing number of networks concurrently (since the timing includes the CPU processing and copying overhead as well as kernel execution time).

The results implied by Figure 9 have large implications for simulations in a variety of scientific fields. They demonstrate that not only a single realistic network can run fast enough to couple the fluid dynamics, but in fact many such networks can be executed in a time short enough to make the simulation feasible. Here we see that the new algorithms are capable of running $\sim 300-600$ realistic networks (150-isotopes) in a time that a standard implicit code can run one such network on a CPU.

## V. Conclusion

In summary, our new algebraically-stabilized explicit algorithms are intrinsically faster than standard implicit algorithms by factors of $\sim 5-10$ for network with several hundred species. For a single network, GPU acceleration increases this to a factor of $\sim 20-40$ for networks with several hundred species. The GPU is capable of running $\sim 300-600$ 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. This work has been supported by the US Department of Energy, Office of Nuclear Energy, and by the ORNL Postgraduate Research Participation Program, which is sponsored by ORNL and administered jointly by ORNL and the Oak Ridge Institute for Science and Education (ORISE). ORNL is managed by UT-Battelle, LLC, for the US Department of Energy under contract no. DE-AC05-00OR22725. ORISE is managed by Oak Ridge Associated Universities for the US Department of Energy under contract no. DE-AC05-00OR22750.

## 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] 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.

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

[19] 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.

[20] J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Hogg, P. Valero-Lara, S. D. Relton, S. Tomov, and M. Zounon, "A proposed API for Batched Basic Linear Algebra Subprograms," Manchester Institute for Mathematical Sciences, The University of Manchester, UK, MIMS EPrint 2016.25, Apr. 2016. [Online]. Available: http://eprints.ma.man.ac.uk/2464/