20th Australasian Fluid Mechanics Conference
Perth, Australia
5-8 December 2016

On the Effect of Workload Ordering for Reacting Flow Simulations using GPUs

K. A. Damm[1], **R. J. Gollan**[1] **and A. Veeraragavan**[1]

[1]Centre for Hypersonics, School of Mechanical & Mining Engineering,
The University of Queensland, Brisbane, Queensland 4072, Australia

Abstract

In reacting flow simulations, considerable computational effort is spent on updating the change of composition due to chemical reactions. As a means of accelerating the simulation of reacting flows, we have been investigating the use of graphics processing units (GPUs) to compute the chemistry update in a massively parallel manner. We have some evidence from previous work that our use of the GPU is less than optimal because of imbalances in the workload that is sent to the GPU. In the present work, we implement several new strategies to better order the workload of chemistry updates that are processed by the GPU and test the performance of these strategies. The results show that we can achieve a speed-up of 1.4x for a particular flow case with hydrogen/ oygen combustion when using a GPU accelerator. The results also show that the performance of the GPU accelerator is insensitive to the workload odering.

Introduction

When performing numerical modelling of reacting flows, a source term appears in each of the species continuity equations. This source term represents the production or loss of species due to chemical reactions in the flow. In operator-splitting methods, the integration of these chemical source terms are solved in a separate update step from the fluid dynamic update. In this case, the update of the chemistry problem becomes one of solving a set of coupled ordinary differential equations (ODEs) in each finite-volume cell of a computational domain. The computational effort required to solve these ODE problems scales with the number of species in the flow and the number of chemical reactions between those species. For example, when using our in-house flow solver `Eilmer` [4], we have determined that approximately 40% of the compute time is spent on the chemistry update in a combusting flow simulation with a 19-species, 53-reactions mechanism for methane-air combustion. This fraction of computational effort spent on the chemistry update increases with increaing numbers of species and reactions. Therefore, we believe there is benefit to focussing efforts on reducing the computational expense of the chemistry update, particularly as we seek to improve the fidelity of our modelling with larger reaction mechanisms.

One avenue to explore for reducing the computational expense is to exploit the inherent parallelism in performing the chemistry update of a reacting flow simulation. These chemistry ODE problems are local in space, that is, they are not dependent on the properties in neighbouring finite-volume cells. This then presents a great opportunity to exploit parallel processing at a fine scale because the independent ODE problems in each finite-volume cell can be computed in parallel without any data dependency across cells. To exploit this fine-grained parallelism, one desires a compute architecture that offers many (on the order of hundreds) of concurrent execution threads. Many researchers in computational science are now turning their attention to graphics processing units (GPUs) as one such architecture that offers many concurrent threads in a single computational unit.

In this paper, we are particularly focussed on hybrid CPU/GPU implementations for the simulation of reacting flows. We are interested in the model of using a CPU/GPU algorithm where the CPU is used to: 1) provide overall of control of the simulation; 2) perform the fluid dynamic updates; and 3) control the offload of chemistry update to a GPU and collation of results from the GPU. In this implementation model, the GPU is used exclusively to perform the chemistry update and return results to the CPU. The idea of using a GPU in this manner, as an accelerator for reacting flow simulations, has been studied by several investigators. Niemeyer and Sung [8] compared the time of a single-core CPU implementation to that of a CPU/GPU implementation for reacting flow. They reported performance improvements of 126x, 59x and 4.5x faster for the CPU/GPU implemenation as compared to the CPU-only for various reaction mechanisms. Shi et al [11] implemented the CHEMEQ2 solver [7] on a GPU. Their results showed that the GPU solver using the CHEMEQ2 algorithm had comparable performance to 13 CPU processors working in parallel on the same task. Encouraged by these results, we began work on a GPU-chemistry module to augment our in-house compressible flow solver, `Eilmer`.

In our earlier work [1, 2], we developed a GPU chemistry module in the OpenCL programming language and coupled this to our flow solver written in C++. For an idealised test case, but with only a small number of species and reactions, we showed a speed-up of 4x when using the CPU/GPU hybrid method as compared to the CPU on its own to solve the same problem However, the same reaction mechanism used on a more realistic reacting flow problem only showed a performance improvement of approximately 2x when using the GPU module for chemistry updates. We suspected that we had encountered the problem known as 'branch divergence' in the GPU calculations. This concept is discussed in next section. In this paper, we present a new implementation of the GPU module that couples with the latest version of `Eilmer`, version 4, which is now implemented in the D programming language [5]. The goal of this work is to investigate changes to the CPU/GPU algorithm that may mitigate the branch divergence problem.

Background on branch divergence in GPU calculations

A standard GPU architecture consists of one or more compute units. Housed on each compute unit are a number of processing threads. These threads are the elements on which floating point operations are executed. GPUs are built on a Single Instruction Multiple Data (SIMD) architecture. This format allows the concurrent execution of a single operation on multipled sets of data.

The processing threads on a GPU are grouped into smaller subsets called warps. Warps vary in size for different vendors. For example, in NVIDIA devices (which are used in the present work), a warp consists of 32 processing elements. Only one warp on a GPU will be executing at one point in time. Hence, the 32 processing elements will be truly executing concurrently. For peak run-time performance, all 32 processing threads in a warp execute the same instruction path. If any conditional statements (such as *if* or *else*) cause two threads in a warp to travel down different instructional paths then branch divergence is said to occur. This is a direct consequence of the SIMD structure. When divergence occurs in a warp, the thread which is taking a different instructional path must wait until the other 31 threads have finished executing before its own execution can begin again. In the worst case, where the 32 threads are all on differing instructional paths, this leads to the code running in serial. Branch divergence is a major concern since the single-threaded performance of a GPU thread is far inferior to that of a CPU thread when performing tasks serially. The focus of this paper is to report on efforts to enhance our CPU/GPU hybrid reacting flow simulator to reduce or avoid branch divergence as much as possible. In particular, we look at strategies for packaging up the workload of cells that are sent to the GPU for processing. These strategies are all implemented on the CPU-side of the algorithm. No changes were made to the GPU kernel code.

CPU/GPU reacting flow algorithm and extensions

As mentioned earlier, `Eilmer` employs operator-splitting to integrate the reacting compressible flow equations in time. In the operator-splitting method, the fluid dynamic update (motion due to convection and diffusion) is performed first, and then followed by the update due to chemistry. The details of the governing equations and the implementation of the update is provided in the second author's thesis [3]. One advantage of separating the update is that the best numerical methods for each of these updates can be used. For example, one often encounters stiff ODE systems for the chemistry update and these are best tackled with specialised ODE solvers that exploit the nature of these chemistry systems. A detailed discussion of operator-splitting in the context of reacting flow simulation is provided in the text by Oran and Boris [9].

On each step of the update, the flow field is advanced in time by an amount dt_{flow}. Often this value of dt_{flow} is too large to use in the update of the chemistry problem. To remedy this, the chemistry problem in each finite-volume cell is solved with smaller timesteps, dt_{chem}, and these smaller steps are repeated until the accumulated time equals the dt_{flow} value. We call this method subcycling because we perform several cycles of solving the chemistry ODE problem within the larger flow update. In this approach, there is one single dt_{flow} value for the entire computational domain (that may change as the simulation proceeds). However, there is a unique dt_{chem} value for every finite-volume cell because we use the most appropriate value for dt_{chem} in each cell based on the local flow conditions. This idea of variation of dt_{chem} across the flow field is shown in Figure 1 in which the contour colouring follows the size of the local dt_{chem} value. In this simulation, $dt_{flow} \approx 3.0 \times 10^{-6}$s whereas the dt_{chem} values are several orders of magnitude smaller. In Figure 1, the deep blue regions can be interpreted as regions where the most computational effort is spent on the chemistry update since these areas have the smallest dt_{chem} value and require the most subcycles. The regions towards the red end of the spectrum represent the computationally cheap chemistry updates.

This dt_{chem} value is stored as one of the properties in the data structure for the cell for subsequent reuse on the next chemistry update. The assumption here is that the dt_{chem} value from a previous flow update will be a good value for the next update. This works well when the flow conditions have only varied slightly between updates. In cases where flow conditions have changed dramatically between flow timesteps, such as a passing shock wave, there are correction mechanisms in the chemistry ODE solver to adjust the dt_{chem} value to something more appropriate. We provide this description of the operator-splitting and subcycling to give context to how the CPU/GPU reacting flow algorithm is implemented.

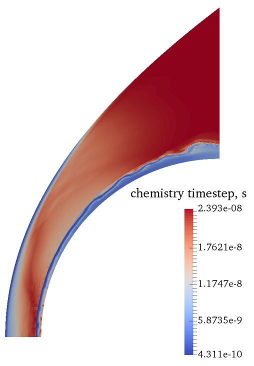

Figure 1: Variation of dt_{chem} values for a representative reacting flow field. For reference, the dt_{flow} value is approximately 3.0×10^{-6}s.

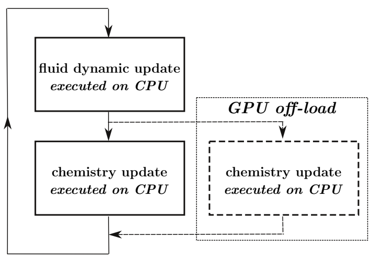

Figure 2: Flow diagram for update algorithm with both CPU-only and CPU/GPU updates shown.

CPU-only algorithm

When solving a reacting flow problem in `Eilmer` using a CPU only, each timestep consists of a fluid dynamic update followed by a chemistry update. This approach is depicted in the control flow diagram in Figure 2 by following the arrows in bold. When using the CPU-only approach, each chemistry problem is solved sequentially: the algorithm loops over all cells in the domain in turn solving for the new chemical state after timestep dt_{flow}. Note that the chemistry update in each cell might require a different number of subcycles depending on the local value of dt_{chem} in that cell. In the performance testing, presented later, we designate this algorithm as *'CPU-only'*.

CPU/GPU algorithm: original implementation

In earlier work [1, 2], we reported on a CPU/GPU hybrid algorithm in which the CPU was used to perform the fluid dynamic update, and then the chemistry update work was off-loaded to the GPU. This approach is depicted in Figure 2 by following the dashed arrows. Note that the solution of the individual ODE problems can be performed in parallel on the GPU. The exact number of parallel ODE problems that can be handled simultaneously is dependent upon the memory and thread count available on the GPU. In typical reacting flow simulations, the number of cells passed to the GPU far exceeds the number of parallel tasks the GPU can handle. As such, the GPU stages its workload: it compute the chemistry update for as many cells as it can handle at one time in parallel, then loads up a new set of cells. This continues until all cells have had their chemistry update computed. We designate this algorithm as *'GPU-orig'* in the Results section.

In this original implementation, we took a very simple approach as to how we passed the workload of cells to the GPU. That

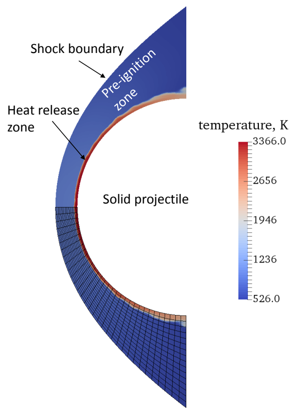

Figure 3: Temperature contours for supersonic flow of hydrogen/oxygen mixture over a projectile. On the bottom half, the computational domain of finite-volume cells is overlayed. For clarity, the number of cells has been reduced by a factor of 8 in each dimension.

simple approach was to load the cells in the logical order they are stored on the CPU. We made no attempt to group together cells with similar workload. As discussed earlier, we believed this might lead to branch divergence in the GPU worker threads, and so diminish the benefit of the parallel execution.

In what follows, we describe changes to how we distribute the workload on to the GPU. In effect, this amounts to changes in the ordering of how cells are passed to the GPU. Our idea was to group cells of similar flow conditions or computational expense together so that when processed by the GPU, the effect of branch divergence was minimised or even completely avoided.

GPU workload ordering based on flow structure

Figure 3 shows the temperature contours of a reacting flow field of a hydrogen/oxygen mixture. The hemispherical projectile is driven into the combustible mixture at Mach 3.55 and causes a detonation in the shock layer gas in front of the projectile. This is used as our test case in the Results section and the details are presented there. For the moment, note that the flow field has some inherent structure: it naturally divides into a pre-ignition zone and a heat release zone. The bottom-half of Figure 3 shows the finite-volume cells overlayed on the flow field. One can see that if the workload for the GPU could be grouped based on those zones, then we are likely to send cells ordered in such a way that they have similar computational workload in terms of chemistry.

Indeed, there is an easy change to our algorithm for packaging cell data such that we can group cells based on the flow structure. That change is to reorder our loop indices `i` and `j` when packaging the cells since the `j` logical direction runs in the body tangential direction. We call this algorithm *'GPU-reorder'*.

We acknowledge that this particular reordering is very specific to this flow field, and not a general solution. Our goal is simply to test if the reordering gives a noticeable improvement in computational performance. If it does, then it would be worth investigating how to apply this idea in a more general and dynamic way. By dynamic, we mean a packaging of cells that changes throughout the simulation in response to changes in the flow field structure.

GPU workload ordering based on chemistry timestep

Referring again to Figure 1, we note there is a range of dt_{chem} values across the flow field. When we simply package the cells for the GPU off-load in logical ordering — for example, looping over i and j indices in a structured grid — it seems likely that cells with very different flow conditions will be processed concurrently by the GPU. When the flow conditions vary a lot, then so too do the instructional paths for the execution of the chemistry ODE update. This large variation is exactly what one would like to avoid to mitigate branch divergence. The *'GPU-reorder'* algorithm, discussed above, is one attempt to reduce this variation in workload. However, that approach still has deficiencies in terms of minimising the variation in workload that is off-loaded to the GPU.

A second strategy of workload ordering that we attempt in this work is to order the cells for processing based on their dt_{chem} value. The assumption here is that cells with a similar dt_{chem} value will present similar computational workloads for the GPU.

In the implementation, we use the `sort` algorithm that is part of the D standard library to order the cells from smallest dt_{chem} to largest. The sorting occurs on the CPU and then the cells are off-loaded to the GPU in sorted order. During a simulation, the dt_{chem} value in a given cell varies as the flow field evolves and the local conditions change in the cell. As such, performing a sort on dt_{chem} just once may not be an optimal strategy since the flow field changes with time. Any initial benefit in workload ordering might rapidly degrade as the flow conditions change in individual cells. We wondered if there would be a trade-off between the time spent sorting cells against any performance gains by having the cells sorted by workload. To quantify this trade-off, we tried three variants of our sorting approach:

1. *'GPU-sort-every'*, in which the sorting of cells occurs on every timestep before off-load to the GPU;
2. *'GPU-sort-once'*, in which the sorting of cells occurs once at the beginning of the simulation; and
3. *'GPU-sort-10'*, in which the sorting of cells occurs on every 10th timestep.

Results

A hydrogen/oxygen reacting flow case was chosen to test the performance of the various workload ordering algorithms. The particular flow case corresponds to an experiment performed by Lehr [6] in which a hemispherical projectile was fired into a stoichiometric mixture of hydrogen and oxygen at supersonic speeds. We use his experimental results at Mach 3.55 as our test case. The quasi-steady flow field for this set of flow conditions is depicted in Figure 3. A previous validation study of the CPU-only algorithm showed a very good comparison between simulation and experimental results. As such, this seemed a good starting point to test the new CPU/GPU algorithms.

In order to test the various algorithms, each test was performed starting from the same initial condition where the flow had reached a quasi-steady state. That quasi-steady state was computed with the CPU-only algorithm using the shock-fitting mode in `Eilmer4`. The use of shock-fitting mode is why the shock lies perfectly on the inflow boundary in Figure 3. For each algorithm test, the flow field was advanced forward in time a further 1000 iterations. The test was repeated 5 times for each algorithm so that an average time for simulation could be reported. There is some variation in run time on the CPU/GPU systems because we do not have precise control over the linux scheduler during tests. However, the averaging of results over five runs accounts for this variability. In these tests, the Rogers and Schexnayder [10] reaction mechanism was used, which contains 9 species and 28 reactions for a hydrogen/oxygen mixture. Our in-house implementation of Mott's α-QSS integra-

Table 1: Timing results in seconds for various workload ordering algorithms performing 1000 iterations of the Lehr M=3.55 test case.

Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Avg.	Std. dev.	Speed-up (GPU_{avg}/CPU_{avg})
CPU-only	6777	6757	6796	6780	6800	6782	17.1	-
GPU-orig	4850	4866	4816	4806	4803	4828	28.2	1.40
GPU-reorder	4838	4831	4825	4855	4832	4836	11.5	1.40
GPU-sort-every	4872	4890	4932	4858	4831	4877	37.7	1.39
GPU-sort-once	4878	4877	4836	4845	4834	4854	21.9	1.40
GPU-sort-10	4871	4873	4876	4899	4860	4876	14.3	1.39

tor [7] was used on the chemistry ODE problems for the both *CPU-only* and the CPU/GPU hybrid algorithms.

The results of the timing tests are displayed in Table 1. The encouraging result is that the use of the GPU to compute the chemistry update gives a speed-up of 1.4x when compared to the CPU-only simulation. This gives confidence that our new D/OpenCL implementation is working well since these results are consistent with tests performed with our older C++/OpenCl implementation. It also confirms that there is some benefit to augmenting a reacting flow simulator with a GPU chemistry module. The disappointing result is that the various algorithms for ordering the cells off-loaded to the GPU have made no difference to the performance.

The result that the performance of the GPU calculations is insensitive to the workload ordering has several interpretations. It could be argued that branch divergence is of little importance in the GPU performance for this application and so the performance is insensitive to the order in which the cells are processed. We believe this interpretation of the result is incorrect because we have previously collected evidence that branch divergence occurs when moving from idealised tests to more realistic cases [1].

A second interpretation of the result is that branch divergence is occurring, but that it occurs at a finer scale than what we can influence with a reordering of cells. This can be explained as follows. In the algorithms for workload ordering presented here, we have tried to group cells with a similar dt_{chem} value on the assumption that these will require a similar number of steps to accumulate the dt_{chem} values up to the dt_{flow}. This is a very coarse means for ensuring that the instructional paths of grouped cells are similar. These workload ordering methods ignore the fact that there are deeply-nested *if*-statements in the ODE update method. We believe that branch divergence is still occuring at this deeply-nested fine scale. This is supported by the result that the run-time performance of the GPU calculation is consistent across all the various algorithms because these algorithms for workload ordering do not affect the branch divergence at a fine scale.

Conclusion

In this paper, we tested the performance of several algorithms for ordering the off-loaded work sent to a GPU chemistry update in a hybrid CPU/GPU simulator for reacting flows. The timing results showed that the CPU/GPU algorithms had a speed-up of 1.4x compared to the CPU-only for a hydrogen/oxygen combusting flow case. The results also showed that the performance of the GPU accelerator was insensitive to the workload ordering sent to the GPU. This result suggests to us that branch divergence is occurring on a fine scale in the GPU calculations — at a fine scale that is not affected by high-level changes in the grouping of cells as they are off-loaded to the GPU.

In this work, all the attempts at algorithm changes to mitigate branch divergence were implemented on the CPU-side of the code. In future work, we intend to focus our efforts on the GPU-side of the code. Specifically, we will aim to reduce the number of branching statements in the GPU code in an attempt to mitigate the fine-scale branch divergence.

References

[1] Damm, K. A., Using GPUs to reduce wall-clock times of reacting flow simulations, Bachelor of Engineering Thesis, School of Mechanical & Mining Engineering, The University of Queensland, 2015.

[2] Damm, K. A., Gollan, R. J. and Veeraragavan, A., Acceleration of combustion simulations using gpus, in *The Australian Combustion Symposium 2015 (ACS2015)*, 2015, 148–151, 148–151.

[3] Gollan, R. J., *The Computational Modelling of High-Temperature Gas Effects with Application to Hypersonic Flows*, Ph.D. thesis, The University of Queensland, 2009.

[4] Gollan, R. J. and Jacobs, P. A., About the formulation, verification and validation of the hypersonic flow solver Eilmer, *International Journal for Numerical Methods in Fluids*, **73**, 2013, 19–57.

[5] Jacobs, P. A. and Gollan, R. J., Implementation of a compressible-flow simulation code in the D programming language, *Applied Mechanics and Materials*, **846**, 2015, 54–60.

[6] Lehr, H. F., Experiments on shock-induced combustion, *Astronautica Acta*, **17**, 1972, 589–597.

[7] Mott, D. R., Oran, E. S. and van Leer, B., A quasi-steady-state solver for the stiff ordinary differential equations of reaction kinetics, *Journal of Computational Physics*, **164**, 2000, 407–428.

[8] Niemeyer, K. E. and Sung, C.-J., Accelerating moderately stiff chemical kinetics in reactive-flow simulations using GPUs, *Journal of Computational Physics*, **256**, 2014, 854–871.

[9] Oran, E. S. and Boris, J. P., *Numerical Simulation of Reactive Flow*, Cambridge University Press, 2005.

[10] Rogers, R. C. and Schexnayder Jr., C. J., Chemical kinetic analysis of hydrogen-air ignition and reaction times, Technical Paper 1856, NASA, 1981.

[11] Shi, Y., Green, W. H., Wong, H.-W. and Oluwole, O. O., Accelerating multi-dimensional combustion simulations using GPU and hybrid explicit/implicit ODE integration, *Combustion and Flame*, **159**, 2012, 2388–2397.