Accelerating unstructured-grid CFD algorithms on NVIDIA and AMD GPUs

Christopher P. Stone
National Institute of Aerospace
Hampton, VA, USA
christopher.stone@nianet.org

Aaron Walden
Langley Research Center
NASA
Hampton, VA, USA
aaron.c.walden@nasa.gov

Mohammad Zubair
Dept. of Computer Science
Old Dominion U.
Norfolk, VA, USA
zubair@cs.odu.edu

Eric J. Nielsen
Langley Research Center
NASA
Hampton, VA, USA
eric.j.nielsen@nasa.gov

***Abstract*—Computational performance of the FUN3D unstructured-grid computational fluid dynamics (CFD) application on GPUs is highly dependent on the efficiency of floating-point atomic updates needed to support the irregular cell-, edge-, and node-based data access patterns in massively parallel GPU environments. We examine several optimization methods to improve GPU efficiency of performance-critical kernels that are dominated by atomic update costs on NVIDIA V100/A100 and AMD CDNA MI100 GPUs. Optimization on the AMD MI100 GPU was of primary interest since similar hardware will be used in the upcoming Frontier supercomputer. Techniques combining register shuffling and on-chip shared memory were used to transpose and/or aggregate results amongst collaborating GPU threads before atomically updating global memory. These techniques, along with algorithmic optimizations to reduce the update frequency, reduced the run-time of select kernels on the MI100 GPU by a factor of between 2.5 and 6.0 over atomically updating global memory directly. Performance impact on the NVIDIA GPUs was mixed with the performance of the V100 often degraded when using register-based aggregation/transposition techniques while the A100 generally benefited from these methods, though to a lesser extent than measured on the MI100 GPU. Overall, both V100 and A100 GPUs outperformed the MI100 GPU on kernels dominated by double-precision atomic updates; however, the techniques demonstrated here reduced the performance gap and improved the MI100 performance.**

***Index Terms*—Unstructured-grid CFD, GPU Performance, Performance Portability, AMD ROCm, Atomic Update**

I. Introduction

FUN3D [1] is a node-based, unstructured, finite-volume CFD application that supports several linear volumetric cell types such as tetrahedra, pyramids, prisms, and hexahedra along with triangular and quadrilateral boundary surfaces. FUN3D uses an implicit time integration method, which requires the assembly of the Jacobian matrix of the nonlinear right-hand-side residual vector in order to converge the solution at each time-step. This results in a large, sparse linear system of dense block matrices that are solved each nonlinear iteration using an iterative point-implicit solver. Aside from the linear solver, the computation of the approximate Jacobian matrix (i.e., the left-hand side) and the nonlinear residual (i.e., the right-hand-side) are the most expensive parts of a FUN3D simulation.

While the FUN3D solution variables are nodal, different physical processes are implemented using either cell-based (e.g., diffusion) or edge-based (e.g., convection) algorithms. This leads to highly irregular data access patterns as the different connectivity structures are traversed and the nodal data are updated with the cell- or edge-based computations. In the massively parallel GPU environments, atomic updates are used to avoid race conditions while maintaining acceptable levels or parallelism. As such, FUN3D's GPU performance is highly dependent on efficient atomic updates to global memory.

FUN3D has been studied extensively on several many-core acceleration devices [2] and has been scaled to over 16,000 NVIDIA V100 GPUs [3] on the Summit supercomputer at Oak Ridge National Laboratory (ORNL). The next-generation system at ORNL, *Frontier* [4], will be based on AMD CDNA GPUs. AMD and NVIDIA GPUs share a common parallel taxonomy (i.e., single-instruction, multiple thread – SIMT), have many common features (e.g., shared memory for collaborative groups of SIMT threads, register shuffling), and development environments (e.g., CUDA and ROCm/HIP). These commonalities simplify application portability but performance optimizations are still necessary due to key low-level differences such as the SIMT *vector* length: 32 threads per CUDA *warp* vs. 64 SIMT threads per CDNA *wave-front*).

As noted, the multicolor Gauss–Seidel point-implicit iterative solver is the most computationally expensive and, in the context of accelerator computing, is the most extensively studied [2], [5]–[7] aspect of FUN3D. It is possible to improve the performance of kernels that process the irregular cell, edge, and nodal FUN3D data structures and depend upon atomic updates by exploiting collaborative thread groups (CGs) in the SIMT programming models supported by CUDA and ROCm. CGs at the *thread-block* or the finer warp[1] level can efficiently exchange on-chip *shared* or *register* memories and support suitable synchronization mechanisms. When combined, data

This research was sponsored by the NASA Langley Research Center CIF/IRAD program, the NASA Transformational Tools and Technologies (TTT) Project of the Transformative Aeronautics Concepts Program under the Aeronautics Research Mission Directorate, and the National Institute of Aerospace Cooperative Agreement award NNL09AA00A.

[1]We shall use the term *warp* to denote the vector width on both NVIDIA and AMD platforms for simplicity.

can be efficiently reduced, broadcast, transposed and aggregated to improve data access layouts and reduce the frequency and irregularity of atomic updates. Because of the different SIMT widths on the NVIDIA and AMD GPU architectures, different CG approaches may be necessary to achieve high performance.

In this study, we examine different optimization methods using CGs to improve the computational throughput of performance-critical FUN3D kernels that have irregular access patterns and are dependent on atomic updates on NVIDIA V100 and A100 GPUs and AMD MI100 GPUs. Performance studies are provided using several different strategies to efficiently transpose and accumulate results using collaborating threads in order to reduce the dependency on atomic updates. Results of this study will be of use by developers and computational scientists investigating performance portability between the CUDA and ROCm GPU programming environments.

II. Methodology

Most simulation meshes of interest to the FUN3D community are dominated by tetrahedral cells (TET) with a smaller but appreciable number of prismatic cells (PRZ) since prisms allow for greater orthogonality at viscous boundary surfaces. As such, we shall focus on only TET and PRZ cell types for this discussion.

FUN3D supports both a calorically-perfect gas model, suitable for modeling nonreactive, compressible aerodynamics, and a generic gas model that supports reacting gases. For this study, we shall focus on only the perfect gas model in which five conservation equations are solved at each mesh vertex (or node) for mass, momentum (3), and energy.

FUN3D's unstructured Jacobian and residual kernels can be split into edge- and cell-based formulations. The edge-based kernels are used for inviscid fluxes and other node-centric operations (e.g., gradients computed with a least-squares formulation). Viscous (and other diffusive) flux terms and their associated Jacobian terms are computed using cell-based kernels which combine cell-centered gradients and variable values (averages of the nodes) with terms computed at the edge midpoints. All cells of the same type are computed in parallel to allow for function specialization.

Because of the unstructured nature of the cell geometries, values at the nodes in both edge- and cell-based kernels must be updated atomically since vertices have at least three edges and can be members of many neighboring cells or edges that are being evaluated in parallel. In all common scenarios, the nodal value updates use the atomic summation, *atomic-add*[2], operation on either a single- (fp32) or double-precision (fp64) value but do not require the newly updated results (i.e., the *reduction* version of the atomic-add operator).

Note that it is possible to formulate many of the edge- and cell-based unstructured grid kernels without needing atomic updates. For example, edge or cell coloring, like that used in the multicolor Guass-Seidel point-solver, could identify independent work items that can be updated concurrently without race conditions; but, this would reduce the potential parallelism. Another race-free approach is to use a node-centric formulation in which all edges or cells influencing a given node are evaluated sequentially or reduced in parallel. However, this approach would significantly increase the memory bandwidth and the level of redundant work as well as suffer from load imbalances as the number of edges or cells acting on a node can vary widely. For these reasons, the race-free approaches were not pursued.

Atomic updates ensure that a specified memory address can be updated by any number of concurrent threads but do not guarantee the order in which operations are completed. When multiple atomic updates are issued to the same memory location concurrently, each individual update is applied serially but in an arbitrary order. This scenario, called a *collision*, reduces the throughput of the atomic update.

On both NVIDIA and AMD GPUs, atomic updates can be issued to both global and shared memory addresses. The V100, A100, and MI100 GPUs support both fp32 and fp64 global and shared memory atomic-add operations; however, shared-memory floating-point atomics use a *compare-and-swap* (CAS) algorithm, which is generally slower, when contended, than if implemented in hardware. All GPUs have hardware support for fp32 atomic-add to global memory. NVIDIA V100 and A100 GPUs have hardware support for fp64 atomic-add operations to global memory but MI100 lacks hardware support for this operation, which is used heavily in residual computations.

Atomic updates to global memory are applied to the L2 cache on the GPUs. Updating cachelines already in the L2 can improve performance as this avoids a load from global memory. In the massively parallel GPU environments, efficient cacheline access often requires using a structure-of-arrays (SoA) data layout such that contiguous threads access contiguous elements of the array.

For all kernels to be discussed here, atomic-add operations are used to update multiple values per vertex. FUN3D uses an array-of-structure (AoS) data storage format in which 5 (or more) solution variables are stored contiguously at each node. For the perfect gas model studied here, five variables are updated per node for residual computations and up to 25 values for Jacobian computations (i.e., a 5x5 block matrix representing the partial derivatives of the 5 flux terms with respect to the 5 solution variables).

For edge-based residual kernels, it is common for 1 SIMT thread to compute the edge-centered value and then update both the left and right nodes atomically. Edge-based kernels use a lookup structure[3] to obtain the left and right node indices for a given edge. The edge map structure is sorted in ascending order using the left node value. FUN3D uses a node reordering scheme (i.e., reverse Cuthill-McKee [8]) to cluster neighboring nodes close together in memory to improve data locality. As

[2]Atomic summation is implemented in the `atomicAdd` device function in the CUDA and ROCm/HIP languages.

[3]In C/C++, `struct EdgeMap {int left, right;}`.

TABLE I: Example edge-list showing sorted left and unsorted right node index pairs.

Edge index	0	1	2	3	4	5	6	7	8	9	...
Left node index	0	0	0	1	1	1	1	1	2	2	...
Right node index	3	1	2	3	4	2	5	6	3	7	...

a result, the right nodes in the edge list are not sorted but the indices tend to be relatively close in value to the left node indices. Also, the same nodal index may occur on both left and right entries of the edge-list. See Table I for an example edge-list taken from the benchmark mesh used in this study.

The sorted nature of the edge-list is beneficial for cache reuse, especially when loading nodal data needed to compute edge terms. However, there is a high collision rate when consecutive threads atomically update the sorted left nodes of consecutive edges.

For cell-based viscous flux kernels, collaborative groups (CGs) of threads are used per cell. CGs use warp-level functions to efficiently synchronize and exchange register data (i.e., shuffles). The size of a *tile* of threads in a CG are restricted to the SIMT width of the hardware (i.e., 32 on NVIDIA and 64 on AMD) and must be a power-of-2. Note, at the time of this study, ROCm does not directly support a CUDA *cooperative-groups* framework so all functionality needed for CG programming was written into FUN3D.

Within a given cell, the viscous flux of edge-connected nodes are updated using the edge-centered flux. In the baseline implementation, edges in a cell are computed in parallel by the CG threads and the edge-centered flux values are *scattered* to the left and right nodes atomically (e.g., 9 edges in PRZ cells scatter values to 6 nodes). This leads to 12 (i.e., 3×4) atomic-add operations per-node, per-cell in the viscous residual flux kernels. As noted above, FUN3D uses a node reordering scheme to cluster neighboring cells to boost data spatial locality. A consequence is that cells containing the same node are processed concurrently and that the same node may be updated by threads in the same thread-block or warp. This increases the collision rates of the atomic updates.

In the viscous flux Jacobian (*visjac*) kernel, all nodes influence all other nodes within the same cell. Given N_n nodes *per cell*, this leads to N_n^2 node-node interactions of which there are N_n diagonal and $N_n(N_n - 1)$ off-diagonal interactions. For the perfect-gas model, there are only 17 nonzero values in the 5×5 block matrices that must be updated.[4] The block matrices are stored in column-major format and the 17 nonzero terms are not contiguous. Off-diagonal block matrix entries are stored in single-precision since FUN3D uses a mixed-precision iterative solver to approximately solve the linear system each nonlinear iteration. (Diagonal block matrices use full 64b precision.) This reduces the bandwidth and storage requirements significantly.

We have investigated several strategies for improving the performance of kernels dependent upon global atomic-add operations in FUN3D. Broadly, these methods focus on the following goals:

1) Reduce the overall number of atomic-add operations needed to conserve bandwidth to the L2 cache.
2) Reduce the number of atomic-add collisions to avoid serialization.
3) Improve L2 hit rate to conserve global memory bandwidth for kernel load operations.

To address these, we have implemented several methods which are summarized as:

- *Warp-level pre-atomic aggregation*: combine (add) values that will update the same address using warp-level shuffle operations and atomically update once. (Goals: 1 and 2)
- *Warp-level shuffle transposition*: reorder per-thread AoS data to SoA format using warp shuffle operations to increase cacheline reuse in the atomic-add operations. (Goals: 3)
- *Warp or block-level shared memory transposition*: write per-thread AoS data to SoA format using *shared* memory to increase cacheline reuse in the atomic-add operations. (Goals: 3)

For edge-based kernels, we have investigated using the sorted edge-list to locally reduce (accumulate) the values at the warp level and then issuing a single atomic-add per value. This concept, *pre-atomic warp aggregation*, is designed to reduce the frequency of atomic-add's and the collision rates since the left nodes of the edge-list point to the same node. Warp-level atomic aggregation has been used successfully to improve the throughput of data filtering [9] and graph traversal [10] operations on NVIDIA CUDA GPUs. We have implemented two variants of this approachthat target sorted and unsorted node lists. The warp-aggregation algorithm is split into two parts: *partitioning* and *reduction*. The partitioning phase finds lanes in a warp that have the same node indices. Partitioning equates directly to the `match_any_sync` function in CUDA where the node index is the matching value. All lanes that are members of the same partition (i.e., have the same node index) have the same resulting bit-mask after partitioning. The `match_any` function does not exist in ROCm and was created using warp-level ballot and shuffle functions.

The reduction phase sums (accumulates) values for each partition. We created specialized reduction operations when the node indices are known to be sorted and contiguous lanes are all members of the same partition. For unsorted node lists, the partition's lanes are spread arbitrarily across the bit-mask and the lane indices must be found via a search which increases the reduction cost compared to the sorted variant.

Table II shows the median number of unique nodes when processing the edge list in batches of 32 and 64 edges concurrently (i.e., the SIMT widths) on the benchmark mesh. The median values for the left side of the edge list equate to contiguous partitions of 5.3 and 5.8 lanes. For the right nodes, the set lengths are only 1.3 and 1.4 and are not contiguous. The relatively low number of unique nodes in the 9^{th} decile of edges (i.e., 9 and 18 in 90% of edges) indicates the high

[4]For PRZ cells, the visjac updates require $17 \times N_n^2 = 612$ values per cell.

TABLE II: Median number and 9^{th} decile of unique nodes per set of 32 or 64 edges in the edge-to-node mapping from the 1M benchmark.

Width	Unique Nodes			
	Median		9^{th} decile	
	Left	Right	Left	Right
32	6	25	9	28
64	11	45	18	50

probability of having useful aggregates. That is, statistically, the number of colliding atomic-add operations for the sorted node list is reduced by a factor of 5-6x in 50% of edges (or 3-4$\times$ in 90% of edges). This also shows that there is limited benefit from the unsorted right node list.

The warp-aggregation approach can reduce the frequency of atomic-add collisions; however, it does not address the cacheline efficiency of the atomic-add operation when multiple values per node must be updated. As noted, it is common for each node to require 4-5 values for residual kernels and up to 25 values for Jacobian kernels. We have observed that it is beneficial to have contiguous threads issue atomic-add operations to contiguous array locations. It is likely that this is due to cacheline reuse since multiple threads in the same warp would issue the same instruction concurrently thereby queuing the atomic-add event concurrently. This is analogous with the common GPU memory optimization in which it is more efficient to read/write contiguous data using contiguous threads (i.e., *coalesced* memory access).

Due to FUN3D's AoS nodal data storage format, it is common for 1 thread to compute all flux or other update terms for a given node. To facilitate contiguous atomic-add operations, we must transpose the data such that contiguous threads will issue atomic updates to contiguous array locations for the same nodal index. A common approach for this is to use *shared* memory as a staging area to transpose the AoS to SoA format. Shared memory on both CUDA and AMD GPUs can be written efficiently using non-unit-stride access so long as bank conflicts are avoided (i.e., concurrent writes to the same bank). When the number of values per node is prime (e.g., 5 or 17), this is generally not an issue. Once the data are written to shared-memory and the threads synchronize appropriately, the CG threads traverse the data in transposed order and atomically update the global memory. This approach is straightforward but adds overhead for thread synchronization, shared-memory read and write latency, and shared-memory storage.

Another approach is to transpose the data within each CG tile using register shuffle functions. We have investigated two algorithms depending upon the number of values per node (k) that must be transposed. When k is a power-of-2 (e.g., 4 in *visrhs*) and not greater than the CG size (which must be a power-of-2 by definition), each set of k adjacent lanes transpose their values using $k-1$ register shuffles. Once transposed, the k adjacent lanes issue atomic updates to contiguous memory locations.

When k is a prime value (e.g., 5 in *roe-flux*), it is possible to transpose the data using k shuffle exchanges. Unlike the

Lane	Initial Values				Transposed			
0	0	1	2	3	0	4	8	12
1	4	5	6	7	1	5	9	13
2	8	9	10	11	2	6	10	14
3	12	13	14	15	3	7	11	15
4	16	17	18	19	16	20	24	28
5	20	21	22	23	17	21	25	29
6	24	25	26	27	18	22	26	30
7	28	29	30	31	19	23	27	31

(a) $k = 4$ (power-of-2)

Lane	Initial Values			Transposed		
0	0	1	2	0	8	16
1	3	4	5	1	9	17
2	6	7	8	2	10	18
3	9	10	11	3	11	19
4	12	13	14	4	12	20
5	15	16	17	5	13	21
6	18	19	20	6	14	22
7	21	22	23	7	15	23

(b) $k = 3$ (prime)

Fig. 1: Example data layout before and after shuffle-based transposition of CG tile with 8 threads and (a) $k = 4$ (i.e., power-of-2) and (b) $k = 3$ (i.e., prime) values per thread.

method for k is a power-of-2, which transposes only k adjacent lanes *in place*, the prime-k method transposes all L lanes (where $L \geq k$) from a virtual `v[L][k]` rectangular array to `v[k][L]` and requires k temporary storage per thread. The additional storage is needed for the more complex exchange network. Specifically, lane p seeks to send its i^{th} value and receive the j^{th} value from lane q where j and q are found algebraically for $i \in [0, k)$ using the following equations:

$$\begin{aligned} q &= (pS_{L,k} + iS_{L,k}(L-1)) \pmod L \\ j &= (kq + i) \backslash L. \end{aligned} \tag{1}$$

Here, $S_{L,k}$ is a prime-valued stride that is dependent upon both L and k. For $L \leq 64$ and a power-of-2, $S_{L,3} = 43$ and $S_{L,5} = 13$. After the CG tile transposes its data, the k values per node can be atomically updated with unit-stride access. See Figures 1(a) and 1(b) for example data layouts before and after the power-of-2 and prime-k transpositions.

III. Results

We have implemented the three strategies introduced in the previous section designed to improve the performance of device kernels using atomic-add operations. The first two kernels to be discussed are edge-based and update the fp64 right-hand-side residual vector with an AoS format (e.g., `double res[N_nodes][5]`). The *roe-flux* kernel computes the inviscid flux vector and updates all 5 terms at each vertex based on data computed at the edge midpoints. One thread is mapped to each edge in a thread-block and both left and right nodes are updated per-edge. The *lstgs* kernel computes the contribution of an edge to the nodal gradient vector resulting in 15 values per-node that are atomically updated per-edge. Two threads are mapped per edge, one for

TABLE III: Median device run-time (ms) and speed-up relative to the *Global atomic* method for the edge-based inviscid flux kernel (*roe-flux*).

Method	V100		A100		MI100	
Global Atomic	2.49	–	1.75	–	6.30	–
Smem Trans.†	1.52	(1.64)	1.01	(1.73)	3.49	(1.81)
Hybrid Agg.	2.30	(1.08)	1.03	(1.70)	2.30	(2.74)
Warp Agg.	N/A		1.02	(1.72)	2.62	(2.40)
Warp Trans.	2.41	(1.03)	1.23	(1.42)	3.09	(2.04)

TABLE IV: Median device run-time (ms) and speed-up (parenthesis) relative to the *Global atomic* method for the edge-based, least-squares gradient kernel (*lstgs*).

Method	V100		A100		MI100	
Global Atomic†	5.95	–	4.09	–	19.18	–
Smem Trans.	2.70	(2.20)	1.93	(2.12)	6.90	(2.78)
Warp Agg.	4.69	(1.27)	2.07	(1.98)	4.36	(4.40)
Hybrid AggT.	4.73	(1.26)	1.39	(2.94)	3.19	(6.01)

each node, since the computations are based largely at the nodes, not the edge midpoints.

For all kernels reported here, the thread-block layout (i.e., the number of threads in the x/y/z directions) was tuned to find the fastest combination with the constraint that whole warps or wave-fronts are allocated (e.g., multiples of 32 or 64) for the NVIDIA and AMD GPUs. The kernels were run for at least 20 samples and the median device times were recorded to remove spurious fast or slow run-times. Data were collected using the vendor-provided profiling tools (e.g., *rocprof*). All results presented are based on ROCm release 4.2.0 for the MI100 and CUDA release 11.2 for the V100 and A100 GPUs.

All results reported here are based on a FUN3D mesh with 1.124 million nodes, 5.968 million edges, 3.040 TET cells, and 1.172 million PRZ cells. This same benchmark, the *1M* benchmark, has been used extensively in previous single-device FUN3D accelerator performance studies [2], [5].

Table III gives the device run-time and speed-up relative to the *Global atomic* method (i.e., updating the global memory locations directly). The original method[5] for *roe-flux* used shared-memory to store the transposed edge-centered flux (*Smem Trans.*) and the 5 left and right nodal values per-edge are atomically updated by 10 contiguous threads. This approach is approximately 64-82% faster than if each thread atomically updated the global memory directly (i.e., *Global atomic*) on all three GPUs with the largest impact on the AMD MI100. We observe that while the performance of the MI100 was improved significantly using the *Smem Trans.* method, the relative performance difference between the NVIDIA and AMD GPUs still shows a high penalty for the lack of fp64 atomic-add hardware support. The *Smem Trans.* method improves the cacheline reuse but not the penalty of atomic update collisions, especially when using software-based fp64 atomic-add operations on the MI100.

Two variants of warp-aggregation were attempted. In *Hybrid Agg.*, the sorted left node was aggregated and atomically updated sequentially by the first lane of the aggregate (i.e., the *leader*) and the unsorted right node used the *Smem Trans.* method. In *Warp Agg.*, both the sorted left and unsorted right nodes of the edge list were aggregated and the aggregate *leaders* updated the 5 values sequentially. On MI100, *Hybrid Agg.* was the fastest approach giving a 52% improvement over *Smem Trans.*. Both warp-aggregation methods are ineffective on the V100; the performance is degraded significantly and is only slightly faster than *Global Atomic*. However, the A100 is statistically unchanged with either warp-aggregation method. With the *Hybrid Agg.* method, the performance gap between the NVIDIA and MI100 GPUs narrows but the MI100 is still significantly slower (e.g., 51% slower than V100).

The prime-value warp-transposition (*Warp Trans.*) method was also tested for *roe-flux*. Both transposition methods (e.g., shared-memory or shuffle-based) lead to the same atomic-add access pattern and the performance on MI100 is comparable. The shuffle-based approach is 13% faster, which may be due to reduced shared-memory and synchronization latency. On A100, the warp-level transposition is slower than the shared-memory variant though the deficit is less than on the V100.

For *lstgs*, the *Global Atomic* method updates all 15 values per node and per edge directly using global atomic-add operations. As seen in Table IV, the *Smem Trans.* method resulted in greater than a factor of 2 improvement on all platforms. The *lstgs* kernel updates 3 times as much data as *roe-flux* and the higher cost can be seen in both *Global Atomic* and *Smem Trans.* methods. *Warp Agg.*, applied to *both* left and right nodes, improves the MI100 performance by nearly a factor of 4 over the baseline but has a lesser impact on the V100 and A100 platforms. Note that two approaches are possible with *Warp Agg.* in *lstgs*: aggregate all 5 nodal values per Cartesian direction (i.e., 3 aggregations) or aggregate the 3 gradient components per variable (i.e., 5 aggregations). For MI100 and V100, performing 3 aggregations on the 5 values was more efficient but the converse occurred on A100 for unknown reasons. The *Warp Agg.* results in Table IV show the faster of these two approaches for each platform.

A hybrid aggregation and transposition strategy using shared-memory similar to that used in *roe-flux* was also investigated. In the *lstgs Hybrid AggT* approach, both left and right nodes were aggregated and then the aggregate leaders wrote to shared-memory in a compressed format such that all aggregated data were contiguous. The memory offsets into the shared-memory array were found by performing a warp-level prefix-sum (i.e., scan), a common approach for compressing data based on a conditional. Contiguous data were needed in order to efficiently traverse the transposed data after remapping from 1 thread per node and edge to 5. This approach further accelerated the MI100 and A100 performances over either *Smem Trans* or *Warp Agg.* alone. The

[5]The original atomic update method for each kernel is identified with † to give an indication of the realized performance improvement.

net effect was approximately a factor of 3 and 6 performance acceleration on the A100 and MI100 GPUs, respectively. However, the V100 did not benefit, likely due to the relatively poor performance with *Warp Agg.*.

We have implemented several variants of the residual viscous flux (*visrhs*) and viscous Jacobian (*visjac*) kernels. In both, the different cell types are processed separately to allow for specialization of the kernels based on N_n, N_e edges, and N_f faces per cell. Except for the TET version of *visrhs*, CG tiles are assigned to each cell and the size of the CG tiles is a tuning parameter along with the thread-block dimensions. For simplicity, `blockDim.x` is customarily defined as the CG tile size and `blockDim.y` represents the number of cells per thread-block to be processed in cell-based kernels.

As discussed earlier, the baseline versions of the generic cell-based *visrhs* and *visjac* kernels loop over the edges per cell and scatter the results of the two nodes of each edge (i.e., *edge-parallel* or *scatter* algorithm). We have investigated several methods to avoid the edge-to-node race condition. One strategy is to use atomic-add to global memory directly. However, as was seen in the edge-based kernels, this approach is not efficient due to the high atomic-add collision rates. Recall that adjacent cells in the reordered cell indexing are likely to share an edge or face, which results in higher collision rates since they will then share 2 or more nodes.

Table V shows the baseline results for *visrhs* using the *Global Atomic* update method using the *scatter* algorithm. Note, only PRZ data are available since the TET version of *visrhs* does not use the generic cell-based implementation.[6] The relative difference in performance between the MI100 and NVIDIA GPUs is comparable to that of *roe-flux* kernel using global atomic updates. This is expected since both kernels update fp64 values and number of values per node is similar (i.e., 4 vs. 5).

In *visrhs*, we have evaluated warp-aggregation on the left and right nodes of each edge. This is done for all N_e edges per cell. Unlike the edge-based kernels, the warp-aggregate nodes are not sorted and the more expensive unsorted reduction kernel must be used. *Warp Agg.* improves the performance by 42% for MI100 and 46% for A100; however, the V100 performance is significantly degraded as before.

An alternative method to avoid the edge-to-node scatter has been implemented. This method uses a *node-parallel* (or *gather*) approach in which the edges for a given node are summed without any possible race condition within the same cell. This reduces the parallelism per-cell since $N_n < N_e$ for all cell types; however, this has no local race conditions and reduces the number of global atomic-add operations by a factor of 3. The *gather* results in Table V are given for both PRZ and TET cells. The specialized *visrhs* kernel for TET cells explicitly uses this approach and was written specifically for 1 thread per cell and all internal loops over N_n, N_e, and N_f have been effectively unrolled, fused and collapsed. Even with *Global Atomic* updates, *visrhs* is improved by 61% and 63% on the NVIDIA GPUs and more than 120% on MI100.

The benefit of *Warp Agg.* in the gather approach is highly dependent upon cell type and platform. Because 1 thread is assigned to each cell in the specialized TET version of *visrhs*, there is a high probability of common nodes in the same warp. The large aggregate population counts with TET cells leads to a 75% and 82% benefit on the A100 and MI100. A similar reason leads to *lower* performance on PRZ: CG tiles with more than 1 thread reduce the number of cells per warp and reduce the probability of matching nodes. Similarly, a CG tile with less than N_n threads reduces the number of nodes that will participate in the aggregation at the same time. It is important to note that this latter effect occurs in the TET algorithm as well. Only 1 of 4 nodes is aggregated since the 4 nodes are processed sequentially in each thread. That is, the probability of the i^{th} *logical* node in the warp's cells matching is lower than the probability of *any* nodes matching. As before, V100 performs poorly with warp aggregation regardless of cell type.

The power-of-2 shuffle-based warp transposition algorithm was also used to accelerate the node-parallel, gather version of *visrhs* kernels. The *Warp Trans.* results in Table V show substantial improvements over *Warp Agg.* on the V100 and MI100 for TET cells but a decrease on A100. Interestingly, the acceleration from *Warp Trans.* on MI100 gives better performance on TET cells than V100 despite the lack of fp64 atomic-add hardware support.

A final test was conducted blending both *Warp Agg.* and *Warp Trans.* on MI100 in an attempt to both reduce the collision rate and increase L2 cache efficiency. While this approach does give a small improvement over *Warp Agg.* alone, at least for TET cells, this is less performant than using *Warp Trans.* alone.

Table VI shows the run-times for the *visjac* kernel. As can be seen, this kernel is substantially more expensive than the previously studied kernels and accounts for a large portion[7] of the overall FUN3D run-time. As with *visrhs*, the *visjac* kernel is specialized for different cell types. The two kernels are very similar as they represent the same underlying formulation for viscous fluxes.

We have investigated three strategies for controlling the thread-safety in the local shared-memory node updates in the edge-parallel (scatter) algorithm. The first uses atomic-add operations on the shared-memory data and then updating the global memory from the transposed shared-memory data atomically. We have also investigated two scheduling methods such that the node data in shared-memory are updated without the need for an atomic update. The first, *serialization*, explicitly updates one edge at a time though multiple Jacobian entries can still be updated in parallel. The second, *independent sets*, identifies sets of edges with independent nodes that can be processed concurrently. Two or three edges can be updated concurrently for TET or PRZ cells, respectively. This reduces

[6]The cell-based viscous flux formulation greatly simplifies when applied to TET cells warranting a special, high-performance implementation.

[7]The Jacobian matrix can be reused for several nonlinear iterations if the convergence rate is above a threshold. This reduces the overall cost of the Jacobian assembly though it still remains a performance-critical kernel.

TABLE V: Median run-time (ms) and speed-up (parenthesis) relative to the *Global Atomic* methods for TET and PRZ kernels using several atomic update acceleration methods in the cell-based, viscous flux (*visrhs*) kernel using edge- and node-parallel per-cell cooperative-group parallel algorithms. Note: *Warp AggT* uses both warp-aggregation and warp-transposition.

Method	V100				A100				MI100			
	TET		PRZ		TET		PRZ		TET		PRZ	
Edge-parallel												
Global Atomic†			2.92	–			2.25	–			6.92	–
Warp Agg.			5.92	(0.49)			1.54	(1.46)			4.87	(1.42)
Node-parallel												
Global Atomic‡	1.72	–	1.81	(1.61)	1.33	–	1.38	(1.63)	3.90	–	3.09	(2.24)
Warp Agg.	3.49	(0.49)	2.40	(1.22)	0.76	(1.75)	1.42	(1.58)	2.14	(1.82)	3.29	(2.10)
Warp Trans.	1.99	(0.86)	2.32	(1.26)	1.44	(0.92)	1.89	(1.19)	1.31	(2.98)	2.77	(2.50)
Warp AggT	2.94	(0.56)	3.19	(0.58)	1.19	(1.12)	1.86	(0.74)	1.90	(2.05)	3.31	(2.09)

TABLE VI: Median run-time (ms) and speed-up (parenthesis) relative to *Global atomic* method using different atomic update acceleration methods and per-cell cooperative group parallel algorithms (i.e., edge-parallel (scatter) or node-parallel (gather)) in the cell-based, viscous flux Jacobian (*visjac*) kernel for TET and PRZ cell types. Note: all implementations, except *Global atomic*, use shared-memory transposition of the data.

Method	V100				A100				MI100			
	TET		PRZ		TET		PRZ		TET		PRZ	
Edge-parallel												
Global Atomic	55.57	–	67.29	–	43.39	–	37.32	–	154.47	–	133.46	–
Smem Atomic	25.33	(2.19)	18.06	(3.73)	16.67	(2.60)	18.25	(2.40)	44.73	(3.45)	43.86	(3.04)
Smem serial†	20.65	(2.69)	20.89	(3.22)	15.08	(2.88)	13.13	(2.84)	40.74	(3.79)	50.30	(2.65)
Smem indep.	17.49	(3.18)	17.65	(3.81)	11.62	(3.73)	12.30	(3.03)	46.96	(3.29)	43.39	(3.08)
Node-parallel												
Global Atomic	23.37	(2.38)	24.90	(2.70)	16.38	(2.65)	15.62	(2.39)	50.72	(3.05)	47.07	(2.84)
Smem Trans.	14.38	(3.86)	13.47	(5.00)	8.92	(4.86)	9.29	(4.02)	30.19	(5.12)	29.53	(4.52)

the potential edge-based parallelism by a factor of 3 for TET and PRZ cells but avoids the need for local atomic updates or the need to serialize the updates to local shared-memory. It is important to note that while the off-diagonal Jacobian matrix terms are stored in single-precision, all *internal* computations, including shared-memory atomic-add operations, are done in full fp64 precision.

As shown in Table VI, *Global Atomic* is approximately 2-4 times slower than methods using shared-memory for all platforms due to the large amount of data that must be atomically updated. That is, all shared-memory methods reduce the number of global atomic-add operations by a factor of 3 times. While no platform has hardware support for atomic-add operations in shared-memory, the *Smem Atomic* performance is comparable, and at times better, than the original *Smem serial* method. On the NVIDIA GPUs, the *Smem indep.* method (i.e., independent edge sets) is consistently more efficient than the other edge-parallel methods.

The node-parallel algorithm for *visjac* follows the same strategy as in *visrhs*. We see that with *Global atomic* updates, the performance is approximately 3 times faster than in the edge-parallel method since the number of total updates per node is reduced by a factor of 3. When combined with shared-memory transposition (*Smem Trans.*), the performance improves for all devices and all cell types. Overall, this approach is between 35 and 70% faster than the original *Smem serial* method.

Warp-aggregation has not been studied for the Jacobian assembly. There is possible aggregation across the N_n atomic updates to the block diagonal matrix but our studies indicate that the dominant cost is the $N_n(N_n-1)$ off-diagonal matrix updates per-cell, even with the lower precision. The matching diagonal matrix updates coincide with common nodes across neighboring cells and, as noted earlier, occurs frequently. However, the probability of updating the same off-diagonal entry is much lower. For example, when processing 4 to 8 cells concurrently in PRZ cells per warp (practical values based on the fastest CG tile sizes), only 20 to 30% of off-diagonal entries would match at least half of the time. As a result, the off-diagonal atomic updates appear to be more limited by bandwidth than collisions.

IV. Conclusions

In this study, we investigated methods to improve the performance of select performance-critical FUN3D GPU kernels that are dependent upon floating-point atomic updates. Atomic updates are used to maintain a high degree of parallelism while modifying the irregular, unstructured mesh data structures in FUN3D. Three GPU platforms were tested, the NVIDIA V100 and A100 and the AMD MI100 GPUs. The optimization methods were (i) transposition of the array-of-structures to structure-of-arrays vertex data layouts before atomic updates to improve spatial locality (cache efficiency) of the parallel atomic updates and (ii) pre-atomic aggregation (reductions)

of the nodal data to reduce the frequency of atomic updates. Transposition kernels using shared-memory and register shuffles were demonstrated within cooperating groups (CG) of threads. Pre-atomic reductions within CGs were demonstrated using register shuffles for sorted edge and unsorted cell data.

The following items summarize the findings from this study:

- Algorithm modifications that avoid atomic updates provided the most consistent and effective performance increases across the three GPU devices. Refactoring the viscous flux and viscous Jacobian kernels to use a node-based *gather* approach compared to the original edge-based *scatter* approach, resulted in 40-70% performance improvement over the baseline implementation even though the available parallelism was reduced.
- Shared-memory floating-point atomic updates, even without dedicated hardware support, provided significant improvement over equivalent atomic updates to global memory. We observed performance improvement between a factor of 2 and 3.7 on all platforms on the costly viscous Jacobian kernel.
- Atomic updates to contiguous memory had a large impact on all platforms. This is likely due to cacheline reuse in the last-level (L2) cache when updating global memory.
- Transposing the array-of-structure data before atomic updates improved cache efficiency.
  - Shared-memory transposing is viable on all tested platforms provided resources are available. Synchronization overhead is reduced by limiting data sharing to threads within a CG tile (or warp).
  - Register-based transposing when the per-thread array length is a small power-of-2 (e.g., 4) was found to be faster than shared-memory transposing on MI100 and did not increase resource requirements. This size can be challenging for shared-memory transposing due to increased likelihood of bank conflicts.
- Warp aggregation was beneficial when applied to the sorted components of the edge list compared to unsorted lists. This is attributed to the high rate of node index repetitions, which would be atomic update collisions, in the sorted edge list and the more efficient reduction algorithm that is possible with the sorted data.
- Aggregation was most performant when combined with shared memory transposition as this reduced the frequency of updates and improved the cache efficiency. The aggregated data could be easily compacted when writing to shared-memory, which improved the performance compared to register-based transposition.
- The register-based methods were not effective on the NVIDIA V100 GPU and, in general, reduced the performance while the performance was generally improved on the A100 GPUs. The MI100 performance was improved the most with the registered-based methods.

Overall, the performance on the AMD MI100 GPU was significantly improved using transposition, aggregation, or their combination reducing the overhead due to lacking hardware support for double-precision floating point atomic updates. Despite these improvements, in all but the viscous flux kernel (with TET cells), the MI100 remained between 18-53% slower than the V100 GPU and the tested kernels were consistently faster on the A100 GPU with differences of between 72% and 238%. However, these performance differences should narrow on future generations of the AMD GPU if floating-point double-precision atomic updates were supported in hardware.

ACKNOWLEDGMENT

This research was sponsored by the NASA Langley Research Center CIF/IRAD program, the NASA Transformational Tools and Technologies (TTT) Project of the Transformative Aeronautics Concepts Program under the Aeronautics Research Mission Directorate, and the National Institute of Aerospace Cooperative Agreement award NNL09AA00A. The authors would like to thank Advanced Micro Devices, Inc., NVIDIA Corporation, and Dr. Sameer Shende of the University of Oregon. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. The authors are also grateful for the support of the Frontier Center of Excellence.

REFERENCES

[1] R. Biedron, J.-R. Carlson, J. Derlaga, P. Gnoffo, D. Hammond, K. Jacobson, W. Jones, W. Kleb, E. Lee-Rausch, E. Nielsen, M. Park, C. Rumsey, J. Thomas, K. Thompson, A. Walden, L. Wang, and W. Wood, *FUN3D Manual 13.7*, NASA/TM-2020-5010139, 2020.

[2] A. Walden, M. Zubair, and E. Nielsen, "Performance and Portability of a Linear Solver Across Emerging Architectures," in *Seventh Workshop on Accelerator Programming Using Directives*, ser. WACCPD 2020, 2020.

[3] G. Nastac, A. Walden, E. Nielsen, and A. Frendi, "Implicit Thermochemical Nonequilibrium Flow Simulations on Unstructured Grids Using GPUs," *AIAA SciTech Forum*, 2021.

[4] ORNL. (2021) Frontier. [Online]. Available: "https://www.olcf.ornl.gov/frontier"

[5] M. Zubair, E. Nielsen, J. Luitjens, and D. Hammond, "An Optimized Multicolor Point-Implicit Solver for Unstructured Grid Applications on Graphics Processing Units," in *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 2016. Piscataway, NJ, USA: IEEE Press, 2016, pp. 18–25.

[6] A. Walden, E. Nielsen, B. Diskin, and M. Zubair, "A Mixed Precision Multicolor Point-Implicit Solver for Unstructured Grids on GPUs," in *Ninth Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 2019, 2019.

[7] A. Walden, M. Zubair, and E. Nielsen, "Performance Portability Issues for a Large-Scale Computational Fluid Dynamics Application on Emerging High-Performance Architectures," in *Performance, Portability, and Productivity in HPC Workshop*, September 2020.

[8] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th National Conference*, ser. ACM '69. New York, NY, USA: Association for Computing Machinery, 1969, p. 157–172. [Online]. Available: https://doi.org/10.1145/800195.805928

[9] A. Adinets. (2017) Cuda pro tip: Optimized filtering with warp-aggregated atomics. [Online]. Available: "https://developer.nvidia.com/blog/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics"

[10] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on gpus," *SIGPLAN Not.*, vol. 51, no. 10, p. 1–19, Oct. 2016. [Online]. Available: https://doi.org/10.1145/3022671.2984015

Appendix A
Edge Kernel Examples

```
template <class BlockDims, int WarpSize, ...>
__global__ void
EdgeKernel_Direct ( double *odata, ...)
{
  static_assert( BlockDims::Y == 2 );
  /* Map threads to edge and node. */
  auto side = threadIdx.y;
  int edge_id = /* block offset */ + threadIdx.x;
  int node_id = EdgeMap[edge_id][side];
  /* Complex computation on this node/edge. */
  double node_vals[5] = { ... };
  for (int i = 0; i < 5; ++i)
    atomicAdd( &odata[i+5*node_id], node_vals[i] );
}
```

Listing 1: Baseline example edge kernel directly updating global memory.

```
template <class BlockDims, int WarpSize, ...>
__global__ void
EdgeKernel_SmemTranpose ( double *odata, ...)
{
  /* BlockDims compile-time blockIdx. */
  static_assert( BlockDims::X % WarpSize == 0
            and BlockDims::Y == 2 );
  __shared__ double smem[2][BlockDims::X][5];
  auto side = threadIdx.y;
  {
    /* Map threads to edge / node. */
    int edge_id = /* block offset */ + threadIdx.x;
    int node_id = EdgeMap[edge_id][side];
    /* Complex computational for this edge/node. */
    double node_vals[5] = { ... };
    for (int i = 0; i < 5; ++i)
      smem[side][threadIdx.x][i] = node_vals[i];
  }
  __syncwarp();
  int warp_id = threadIdx.x / WarpSize;
  int lane_id = threadIdx.x % WarpSize;
  int offset  = warp_id * WarpSize;
  for (int i = 0; i < 5; ++i) {
    /* Remap threads to edge in warp tile. */
    int linear_id = i * WarpSize + lane_id;
    int edge_id = linear_id / 5 + offset;
    int node_id = EdgeMap[edge_id][side];
    int var_id = linear_id % 5;
    auto val = smem[side][edge_id][var_id];
    atomicAdd( &odata[i+5*node_id], val );
  }
}
```

Listing 2: Edge kernel with data transposed in shared-memory.

Appendix B
Shuffle-based transposition kernels

```
template <class BlockDims, int N, ...>
__device__ void
transpose_vector_2k ( double v[N], Mask mask )
{
  constexpr auto L = BlockDims::X;
  static_assert( isPowerOfTwo(N) and
                 L >= N and
                 L % N == 0 and
                 L <= WarpSize );
  for (int s = 1; s < N; ++s) {
    /* Index of lane to take from. */
    int q = threadIdx.x ^ s;
    /* Index of variable to give/take. */
    int j  = q % N;
    v[j] = SHFL( mask, v[j], q, L );
  }
}
```

Listing 3: Shuffle-based transposition algorithm for per-thread data that are power-of-2 in length.

```
template <class BlockDims, int N, ...>
__device__ void
transpose_vector_prime ( double v[N], Mask mask )
{
  /* Fetch fixed stride for this L and N. */
  constexpr auto S = PrimeShift<L,N>::S;
  static_assert( isPowerOfTwo(L) and
                 N < L and
                 L <= WarpSize );
  /* Make a copy of input v[]. */
  double w[N] = {v[0], ...};
  for (int i = 0; i < N; ++i) {
    auto offset = threadIdx.x * S +
                  i * (L-1) * S;
    /* Source lane. */
    auto q      = offset % L;
    /* Output index in v[]. */
    auto j      = (N * q + j) / L;
    v[j] = SHFL( mask, w[i], q, L );
  }
}
```

Listing 4: Shuffle-based transposition algorithm for per-thread data lengths of 3, 5, or 17 (i.e., prime lengths).