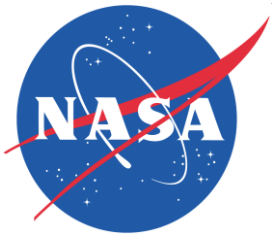


Accelerating unstructured-grid CFD algorithms on NVIDIA and AMD GPUs

Christopher P. Stone
National Institute of Aerospace

Aaron Walden and Eric Nielsen
NASA Langley Research Center

Mohammad Zubair
Old Dominion Univ.

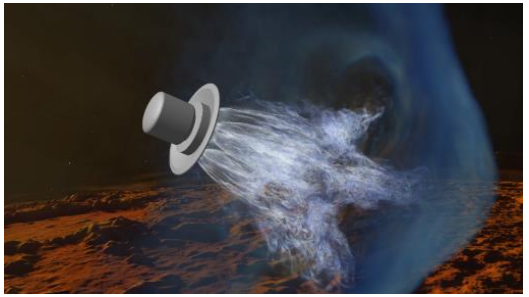
11th Workshop on Irregular Applications: Architectures and Algorithms
SC'21
Nov. 15, 2021

Acknowledgements:

- 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.

Motivation

- FUN3D (unstructured CFD solver) is being used to simulate supersonic, retro-propulsion systems for future, human-scale Mars landers.
  - Reacting (generic) gas formulation with unsteady RANS methods.
- Currently running at scale on *Summit* at Oak Ridge National Laboratory (ORNL).
  - Multi-GPU nodes with NVIDIA V100 GPUs.
  - Extensive performance optimization effort for perfect and generic gas formulations.
- Frontier (ORNL), potentially the first ExaFLOP supercomputer, expected in 2022-2023.
  - Multi-GPU nodes with AMD CDNA GPUs (MI200).
  - Current FUN3D application readiness efforts focusing on MI100 as precursor.
  - Current study focuses on addressing performance on kernels dependent upon atomic memory updates (atomicAdd) relative to V100.



Nastac, G. et al. “Implicit Thermochemical Nonequilibrium Flow Simulations on Unstructured Grids using GPUs,” AIAA SciTech 2021.

GPU Computing

- NVIDIA and AMD HPC GPUs both use
  - *Single Instruction, Multiple Thread* (SIMT) parallel paradigm
  - Multi-level parallel hierarchy: Thread → Warp → ThreadBlock → Kernel
    - O(1) Kernel is launched with many blocks of threads (ThreadBlock).
    - O(1,00-1,000) ThreadBlocks mapped to O(100) GPU multiprocessors.
    - O(1-10) Warps mapped to each ThreadBlock.
    - O(10) Threads per Warp (32 or 64).
- Threads in a Warp follow SIMT paradigm; divergence is serialized.
  - Threads can branch but performance is impacted.
  - No impact of branching for Threads in different Warps.
- Threads in a ThreadBlock can synchronize; share scratchpad memory.
- No global synchronization mechanism during kernel execution.
  - Atomic memory updates used *in lieu* of synchronization.
- Latency hidden by oversubscription of many Threads (Warps) onto multiprocessors.
  - Launch many ThreadBlocks per multiprocessor and switch to any Warp with data ready to be processed.

AMD vs. NVDIA GPUs

NATIONAL INSTITUTE OF AEROSPACE

Similarities

- Single instruction, multiple thread (SIMT) paradigm.
- On-chip L1 cache / thread scratchpad (shared-memory).
- Shared L2 cache for global memory.
- Register shuffling among threads in the same Warp.
- Programming environments support common device functions.
  - ROCm HIPCC and CUDA NVCC are not compatible but are mutually comprehensible.
    - HIP code can compile to both CDNA and CUDA (i.e., one-way portability).
- Atomic update support for both global and shared memory locations.
  - Support for “reduction” atomicAdd (i.e., no returned result).

Differences

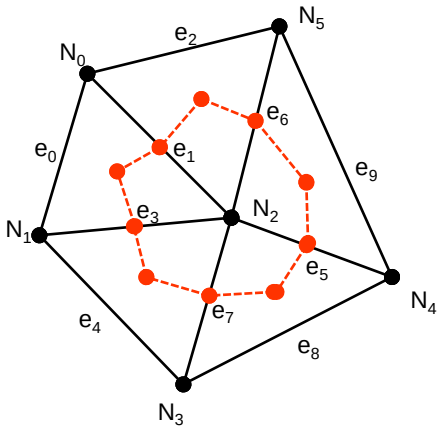
- SIMT widths:
  - CUDA: 32 Threads / Warp
  - CDNA: 64 Threads / Wave-front
    - Wider SIMT increases thread divergence impact.
    - Wider SIMT increases cooperative thread group (CG) programming potential.
- V100+ Tesla GPUs support independent scheduling of threads per Warp.
  - Breaks historic lock-step SIMT model.
    - CUDA CG class facilitates programmability.
  - CDNA fully SIMT.
- Atomic update support in hardware:
  - V100+ Tesla GPUs have hardware support for fp32 and fp64 atomicAdd to global memory.
  - MI100 CDNA only has hardware support for fp32 atomicAdd to global memory.
    - fp64 uses software-based atomicAdd.

FLUDA

- FUN3D Library for Universal Device Acceleration (FLUDA)
  - FUN3D's GPU library for multiple device types.
  - Perfect gas path:
    - Non-reacting, "air" with constant-γ, unsteady RANS
    - Heavily optimized for NVIDIA V100 GPUs
      - See Zubair et al. MCHPC'21
    - *Current focus of study*.
  - Generic gas path:
    - Reacting, multi-species, thermochemistry.
- Bandwidth-bound:
  - Relies heavily upon fp32 and fp64 "reduction" atomicAdd operations for concurrent, irregular updates to unstructured node, edge, cell data structures.
    - Tetrahedra, Pyramids, Prisms, and Hexahedra cells supported.
  - fp32 used for mixed-precision approximate Jacobian block matrix elements.
    - Left-hand side of block sparse linear system (BSR).
    - N_{eq} x N_{eq} block terms. 5x5 for perfect gas.
  - fp64 used for all residual vector ("physics") terms.
    - Right-hand side of linear system.
    - 5 flux terms for perfect gas.

Edge-based Kernels

- Inviscid fluxes are computed via edge-based, finite-volume kernels
  - Flux through dual-mesh face along edges.
  - Costly vector-valued inviscid fluxes (e.g., 5 for perfect-gas).
  - Edge-centered flux updates both left and right nodal residuals.
- Edge fluxes can be computed in parallel.
  - Every node has at least 3 edges in 3D; O(10) is typical with wide variation about the mean.
  - Avoid race conditions with atomicAdd updates to node values.
- Common parallel design:
  - 1 or 2 threads per edge.



Edge	Nodes	
0	0	1
1	0	2
2	0	5
3	1	2
4	1	3
5	2	4
6	2	5
7	2	3
8	3	4
9	4	5

Atomic Updates

- Atomic updates (e.g., atomicAdd) ensure that parallel updates of the same memory location are serialized.
  - Avoids race conditions *without* global synchronization.
  - Ordering of concurrent updates is ambiguous.
    - Addition commutes: the order does not matter (within fp round-off).
  - Performance issues:
    - *Collisions*: High contention increases cost.
    - *Bandwidth*: L2 cache must be updated. Global bandwidth stressor.
- Alternative methods:
  - *Edge-coloring*: color edges to find independent sets.
    - Reduces parallelism; cache inefficient.
  - *Node-parallel*: compute all fluxes for a given node (serially or in parallel).
    - 2x work due to redundant calculation at edge.
    - ragged number of edges per node.
  - Both are viable but have not been investigated to date on the GPU.

Edge-based Kernels

- High probability of threads in the same GPU warp updating the same node location.
  - Edge list is sorted (ascending) by left node index. (primary)
  - Node reordering (Reverse Cuthill-McKee) used to improve cache efficiency by clustering connected nodes. (secondary)
    - Neighboring nodes are “close in index-space”
  - Both increase the atomicAdd collision rate.

Edge Index	0	1	2	3	4	5	6	7	8	9	10	…
Left node index	0	0	0	1	1	1	1	1	2	2	2	…
Right node index	3	1	2	3	4	2	5	6	3	9	20	…

SIMT Width	Percent Duplicate Nodes per SIMT Warp			
Edges / Warp	50%		90%	
	Left	Right	Left	Right
32	81.3%	21.9%	71.9%	12.5%
64	82.8%	29.7%	71.9%	21.9%

Optimization Methods

- Reduce atomic update contention with *pre-atomic warp aggregation*:
  - Aggregate common node updates at the warp level:
    - Exploit sorted structure of edge-list to explicitly aggregate (sum) nodal updates within the same warp.
      - Register shuffle-based reduction algorithm for matching node indices.
    - Issue 1 atomicAdd per unique node index *per warp* instead of *per thread*.
    - Extendable to unsorted node index lists (e.g., cells).
- Improve atomicAdd data access pattern by array transposition:
  - Transpose per-thread update values to enable contiguous updates of global residual vector in AoS format.
  - AtomicAdd are not vector load / store operations like other global memory operations. But concurrently updating contiguous array elements should improve L2 cache hit rates by increasing likelihood of cacheline reuse.
  - Two approaches: (i) shared-memory buffering; (ii) warp-level register shuffling.

Warp Node Aggregation:

- Find matching left (and/or right) node indices in edge-list across threads in the same warp.
  - CUDA “match” warp-level SIMT function: `__match_sync(edge.left_node)`
  - AMD: created shuffle-based function to mimic CUDA’s match function.
    - Iterative sequence of register broadcasts (`__shfl`) and votes (`__ballot`) .
  - All lanes with the same destination node index produce the same SIMT bitmask.
- First non-zero bit in mask is defined as subset leader: `__ffs(mask)`
- All lanes in subset bitmask accumulate values onto leader lane via reduction.
  - Sorted lists: all member lanes are contiguous starting at leader.
    - Parallel (binary) reduction.
  - Unsorted lists: search algorithm used to find member lanes in arbitrary locations.
    - Sequential reduction used due to low population.
- Multiple values per thread reduced concurrently to reduce overhead.
- Leader issues atomicAdd updates on accumulated update values.

Edge Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	…
Left node index	0	0	0	1	1	1	1	1	2	2	2	…
Right node index	3	1	2	3	4	2	5	6	3	9	20	…

Transposition:

- 1 thread typically evaluates all nodal updates per edge.
  - Efficient computations but inefficient data layout for global memory stores (writes) or atomic updates.
  - Strided access to nodal residual array in AoS format reduces L2 cache reuse during atomic updates.
- 3 Methods investigated:
  - Shared-memory transpose.
  - Register-based (shuffle) transpose.
    - Prime number of values.
    - Power-of-2 (2^n) number of values.

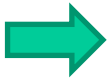
Thread	Values
0	0 1 2 3 4
1	5 6 7 8 9
2	10 11 12 13 14
3	15 16 17 18 19
4	20 21 22 23 24
…	…

Node Residual Array (AoS in global memory)																					
Node Index	10					11					12					13					14 …
Subindex	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	…

Transposition:

- Register: K is power of 2
  - K-1 *in-place* shuffle operations using XOR butterfly exchange network pattern.
  - Diagonal untouched.
  - Results in K contiguous values available for update.

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

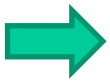


Thread	Values			
0	**0**	4	8	12
1	1	**5**	9	13
2	2	6	**10**	14
3	3	7	11	**15**
4	**16**	20	24	28
5	17	**21**	25	29
6	18	22	**26**	30
7	19	23	27	**31**

Transposition:

- Register: K is prime (3, 5, 17)
  - Data is transposed from row-major to column-major.
  - More expensive transposition algorithm, *not* in-place.
  - Algebraic equations for send/recv shuffle indices.
- Shared-memory: K is arbitrary
  - Threads write per-thread data into 2D smem buffer in transposed order.
    - Strided access to smem is efficient if bank conflicts are avoided.
      - Only if K is 2^n.
  - After warp-level synchronize, threads atomically update values from smem in unit-stride order to gmem.
  - Possible impact on thread occupancy due to smem usage.

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



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

Benchmark Method

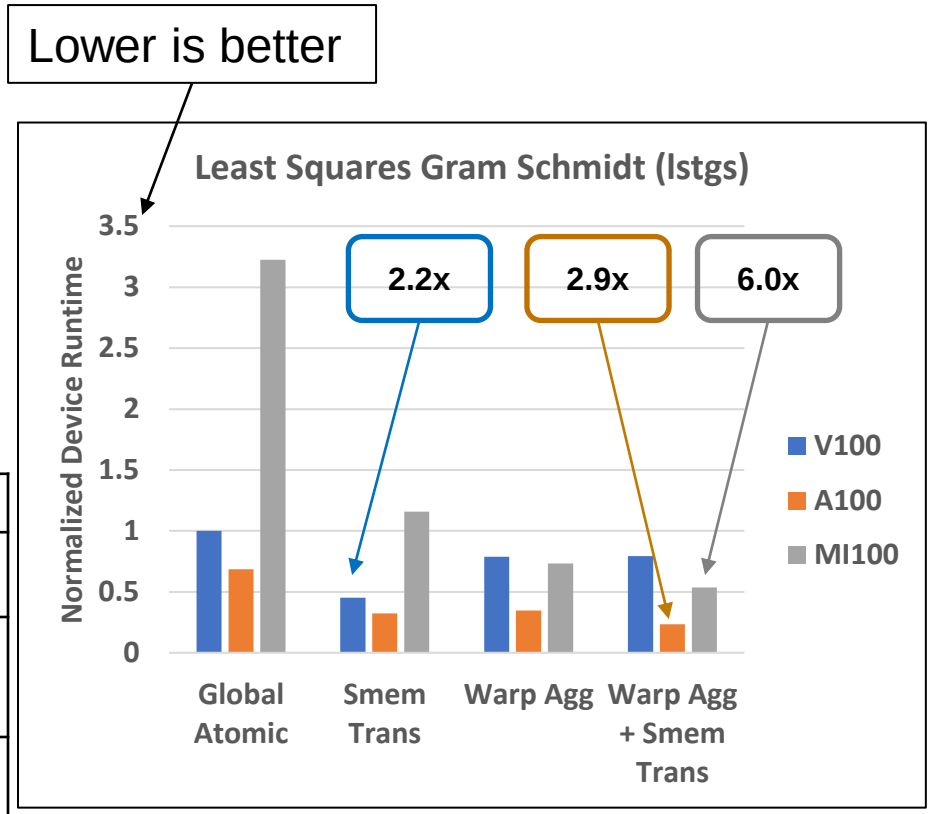
- NVIDIA V100 and A100 GPUs.
  - CUDA 11.2
- AMD MI100 GPU.
  - ROCm 4.2.0
- Best performing ThreadBlock size (blockDim) searched for each kernel via autotuning framework.
  - Constraints:
    - Product(blockDim.xyz) must be multiple of SIMT width.
    - blockDim.x ≤ WarpSize and power-of-2 to support cooperative thread groups (CG) framework.
- Optimal blockDim run 20 times and profiled with nvprof, nsys (nsight), or rocprof.
- Median device time reported to remove any spurious device times.
- 1M Benchmark used for all reported timings.

1M Benchmark	**Nodes**	**Edges**	**Cells**	
			PRZ	TET
	1.124M	5.968M	1.172M	3.040M

Edge-based kernel: LSTGS

- LeaST squares Gram Schmidt formulation to compute gradient vector of all solution variables at the nodes.
- Sum[(edge difference vector) x (nodal gradient coefficient matrix)]
  - 15 fp64 values per node per edge.
  - 2 threads per edge.

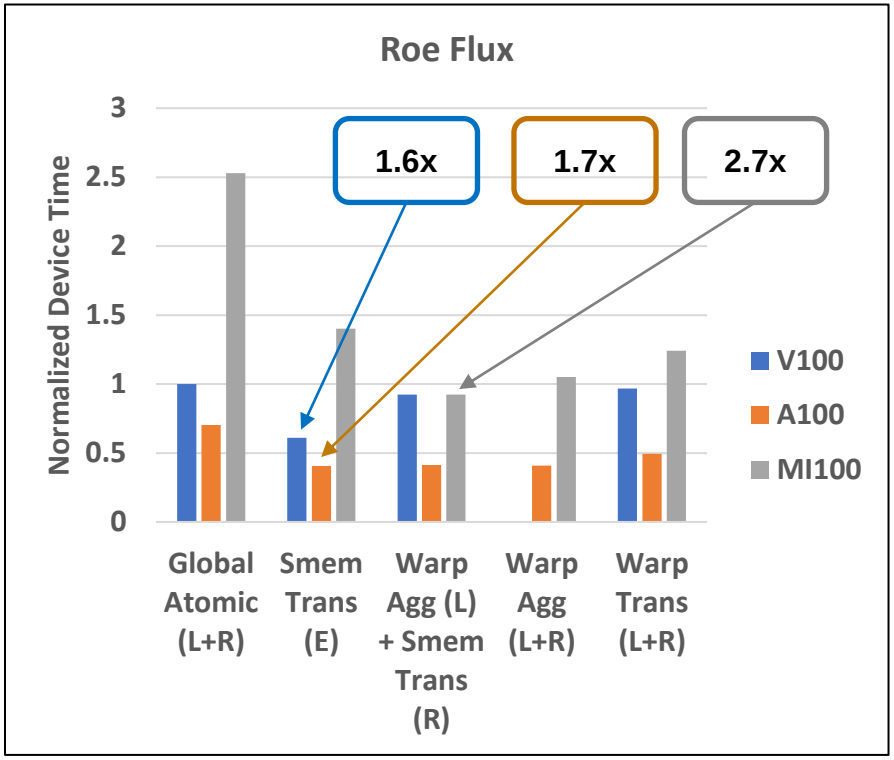
Method	Description
Direct Gmem	Threads directly issue atomicAdd to gmem
Smem Transpose	Threads write to smem in transpose order; All threads then issue atomicAdd to gmem in transposed order.
Warp Agg	Left & Right nodal data is aggregated; leader atomically updates gmem.
Warp Agg + Smem Trans	*Hybrid approach*: Left and Right nodal aggregation; leader writes to compressed smem buffer; All threads then atomically update gmem in transposed order.



Edge-based kernel: Inviscid flux

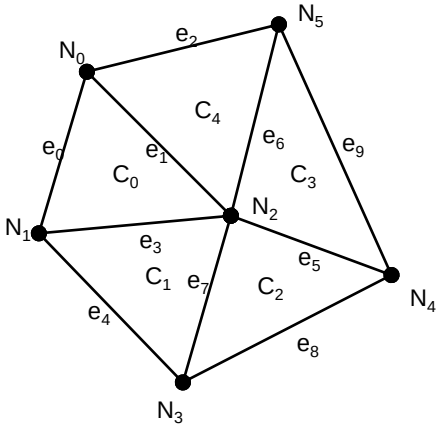
- Upwind Roe Flux-difference Splitting with reconstruction at the dual-mesh face along the edge.
  - Edge-centered flux vector computed then ± to left/right nodes.
  - 5 fp64 values on left & right nodes.
  - 1 thread per edge.

Method	Description
Direct Gmem	Thread atomically updates left & right nodes in gmem.
Smem Transpose	Threads write private edge flux vector to smem; All threads in Warp atomically update left and right nodes in transposed order.
Warp Agg	Left & Right nodal data is aggregated; leader atomically updates gmem.
Warp Agg (L) + Smem Trans (R)	Hybrid: Left node is aggregated, and leader atomically updates gmem. Right node (unsorted) transposed in smem buffer, and all threads atomically update gmem in transposed order.
Warp Transpose	Flux vector transposed (Prime-K); Warp threads atomically update left and right nodes in transposed order from flux vector.



Cell-based kernel: Viscous flux

- Cell-based formulation equivalent to a Galerkin finite-element approach on TET element augmented with edge-gradient info for other cell types.
  - Combines cell-centered gradients and primitive values with edge-gradients.
  - 4 fp64 viscous flux values per edge per node.
    - No mass diffusion in “air” perfect gas model.
  - Node indices are not sequential in cells but tend to be “close” in index-space due to RCM reordering.
  - TET cells: 1 thread per cell
  - Non-TET cells: 1+ cells per Warp, Q = 2^n threads per cell.

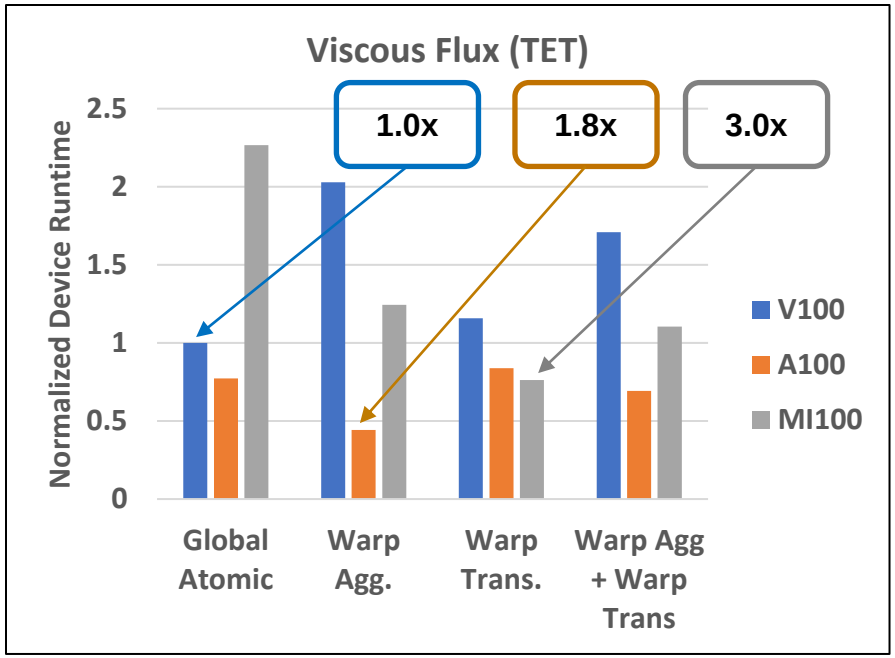


Cell	Nodes		
0	0	1	2
1	1	3	2
2	3	4	2
3	2	4	5
4	0	2	5
…	…		

Cell-based kernel: Viscous flux (TET)

- Specialized implementation for TET cells.
  - 4 fp64 values per node.
  - 4 nodes per cell.
  - 1 thread per cell.

Method	Description
Direct Gmem	1 Thread atomically updates all 4 fluxes on all 4 nodes per cell in gmem.
Warp Agg	Aggregate *unsorted* node index fluxes; leader atomically updates gmem.
Warp Transpose	Flux vectors transposed (2^k method); Subgroup threads atomically update each node in sequence using transposed values and broadcasted node index.
Warp Agg + Warp Transpose	Combines both unsorted warp aggregation and warp transposition.



Summary

- MI100 2.3-3.2x slower than V100 on kernels with fp64 atomicAdd dependencies.
  - Expected due to lack of hardware-based fp64 atomicAdd.
- A100 1.3-1.45x faster than V100 on all tested kernels.
  - Improvement less than theoretical bandwidth improvement (73%).
- Transposition in shared-memory of per-thread, flux vectors before atomicAdd provides consistent performance improvement on all three GPU devices.
  - 1.6-2.2x (V100), 1.7-2.2x (A100), 1.8-2.8x (MI100)
  - Constraining to Warp-level threads reduces synchronization overhead; i.e., cooperative thread group (CG) programming.
- Register-based transposition generally ineffective on all devices.
  - Power-of-2 method (viscous flux) on MI100 only instance this approach provided performance increase.
- Warp-aggregation improved performance on edge- and cell-based kernels on MI100 and A100 GPUs.
  - Direct to global memory improvement: 1.4-1.8x (A100), 1.4-4.4x (MI100)
  - Consistently poor performance on V100: 0.5-1.3x
  - Most effective when combined with shared-memory transposition, especially using compressed format from prefix scan (e.g., lstgs): 35% faster than aggregation alone.

Future Work

- Continue on-going generic gas GPU optimization effort.

NATIONAL INSTITUTE OF AEROSPACE

THANK YOU.