Performance and Portability of a Linear Solver Across Emerging Architectures

Eric Nielsen and Aaron Walden
NASA Langley Research Center (LaRC)

Mohammad Zubair
Old Dominion University (ODU)

https://fun3d.larc.nasa.gov

This research was supported in part by the NASA Aeronautics Research Mission Directorate Transformational Tools and Technologies Project and the NASA Langley Research Center High Performance Computing Incubator (LaRC HPCI).

Overview

- FUN3D is a CFD software suite from NASA Langley that solves the Navier-Stokes equations on fully unstructured mixed element meshes
- This work is an exploration of the performance and portability of FUN3D's principal linear solver across a diverse set of established and emerging HPC architectures
- We first attempt to establish a "speed of light" benchmark implementation of the solver on each architecture
- We then attempt to achieve that speed with a variety of higher-level programming models, including those which emphasize performance portability
- We have not yet studied the portability of the same code across architectures, which is the goal of future work

- FUN3D solves the Navier-Stokes equations of fluid dynamics using implicit time integration on general unstructured grids
- 2nd order finite volume discretization
- An approximate nearest-neighbor linearization of the residual equations for each control volume gives rise to a large tightly-coupled system of block-sparse linear equations
- Written predominantly in Fortran 90; MPI parallelism with limited OpenMP support on CPUs and CUDA support on NVIDIA GPUs

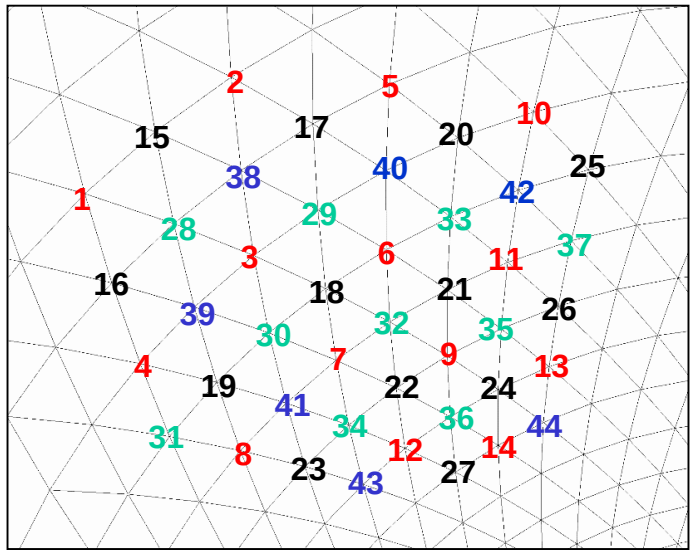
Algorithm 1 SOLVER

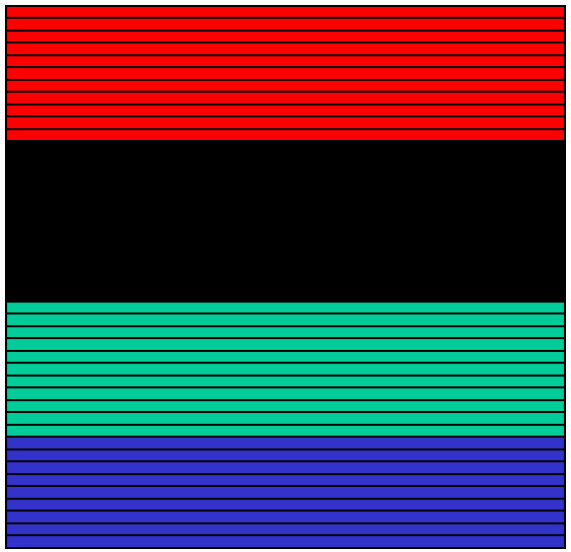
1: $q \leftarrow 0$
2: **for** $i = 1$ to $maxiter$ **do**
3: Construct Jacobian matrix A at q
4: Construct vector b at q
5: Solve for Δq in linear system $A\Delta q = b$
6: $q \leftarrow q + \Delta q$
7: **end for**

Multicolor Linear Solver

FUN3D uses a series of multicolor point-implicit sweeps to form an apx. solution to **Ax = b**

- Color by rows which share no adjacent unknowns; re-order rows by color contiguously
- Unknowns of the same color carry no data dependency and may be updated in parallel
- Updates of unknowns for each color use the latest updated values for other colors
- The overall process may be repeated using several outer sweeps over the entire system





Algorithm 2 LINEAR SOLVER

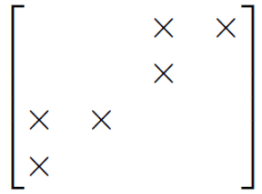
1: **for** $i = 1$ to $niter$ **do**
2: **for** $c = 1$ to nc **do**
3: $\Delta \mathbf{r} = \mathbf{b_c} - \mathbf{O_c}\Delta \mathbf{q}$
4: $\Delta \mathbf{q} = \mathbf{D_c^{-1}}\Delta r$
5: **end for**
6: **end for**

Multicolor Linear Solver: Basics

- Implicit scheme results in linear systems of equations:
  - $A\,\Delta q = b$, A is a sparse $n \times n$ block matrix
  - Typically 14-19 blocks per row
  - block is of size $nb \times nb$ (typically, nb = 5)
- Matrix A is segregated into two separate matrices:
  - $A \equiv O + D$, where O and D represent the off-diagonal and diagonal blocks of A
  - D is always stored in double precision (FP64)
  - O is typically stored in single precision (FP32)
- Prior to performing each linear solve, each diagonal block D is decomposed in-place into lower and upper triangular matrices

Multicolor Linear Solver: Memory Layout

Sparse Structure of Matrix *O*



[x indicates a non-zero block]

CSR Storage for the Matrix to the left

$$
\begin{aligned}
ia &= [1, 3, 4, 6, 7] \\
ja &= [3, 4, 3, 1, 2, 1] \\
data &= [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, \\
&\quad 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
\end{aligned}
$$

Matrix *O* with a 2×2 Block Size

$$
\begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 3 & 5 & 7 \\
0 & 0 & 0 & 0 & 2 & 4 & 6 & 8 \\
0 & 0 & 0 & 0 & 9 & 11 & 0 & 0 \\
0 & 0 & 0 & 0 & 10 & 12 & 0 & 0 \\
13 & 15 & 17 & 19 & 0 & 0 & 0 & 0 \\
14 & 16 & 18 & 20 & 0 & 0 & 0 & 0 \\
21 & 23 & 0 & 0 & 0 & 0 & 0 & 0 \\
22 & 24 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

Multicolor Linear Solver: Challenges

- Indirect memory addressing (for vector Δq)
- Low arithmetic intensity (≈0.5 flops/byte) – memory bound on CPU and GPU
- Number of blocks per row varies, meaning most accesses are not aligned
- The number of rows associated with a color, and thus the coarse-grained parallelism available, can vary significantly
- To support strong scalability, the single node performance for light workloads should be good

HPC Architectures

	Abbreviation	Cores / SMs / Compute Units	Vector / Warp Length, SP	Peak Bandwidth, GB/s
Dual-socket Intel Xeon Skylake 6148	SKL	40	16	256
Intel Knights Landing 7230	KNL	64	16	485
Dual-socket Marvell Thunder X2	TX2	56	4	318
NEC SX-Aurora Tsubasa (NUMA-2)	VE	8	512	1220
NVIDIA Tesla V100	V100	80	32	900
NVIDIA Tesla A100	A100	128	32	1600
AMD Radeon Instinct MI50	MI50	60	64	1024

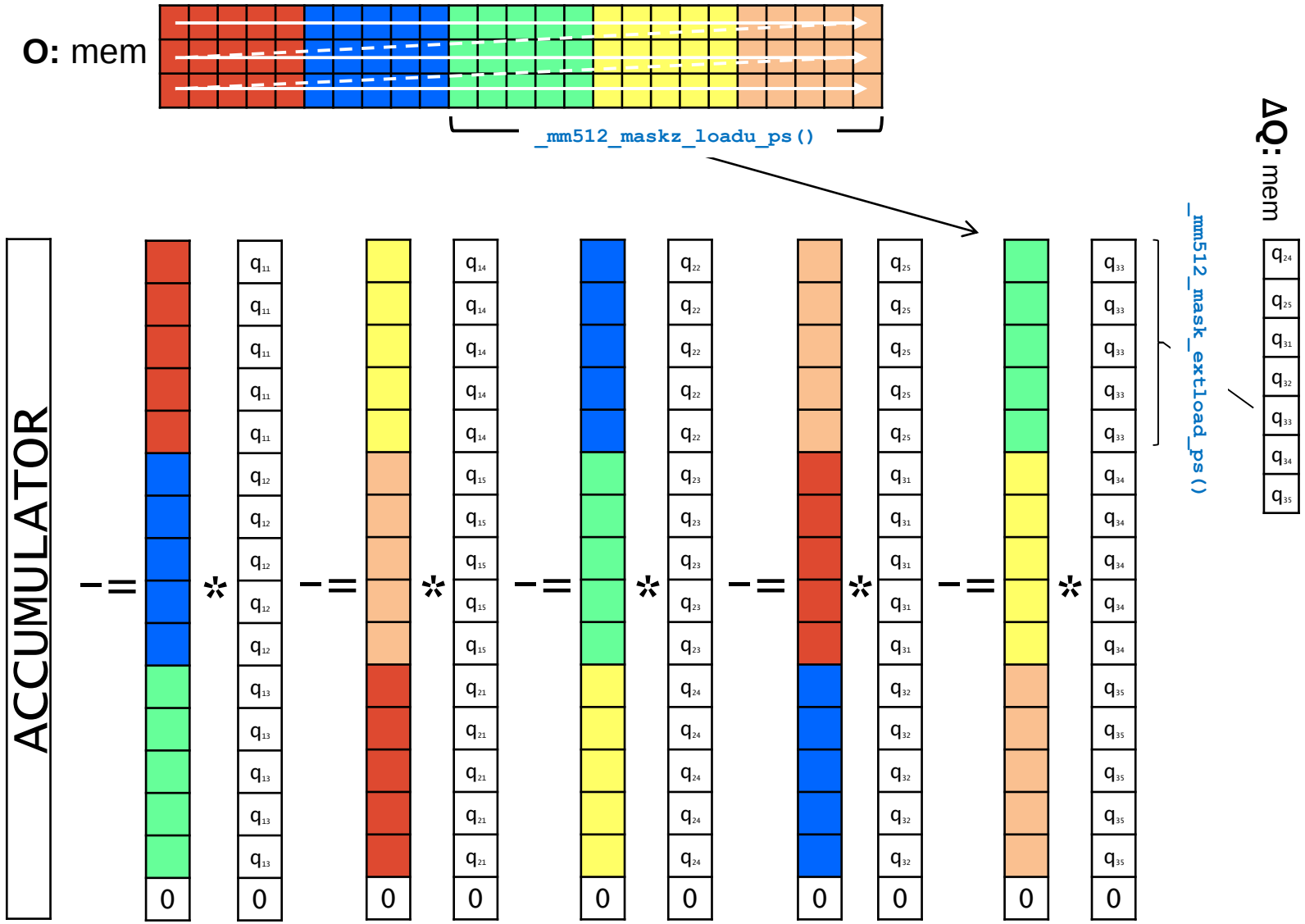
Experimental Setup

- Transonic turbulent flow over a semispan wing-body configuration
- 1,123,718 grid vertices, 1,172,171 prisms, 3,039,656 tetrahedra, and 7,337 pyramids
- 18,998,518 nonzero off-diagonal blocks; average 17 off-diagonal blocks per row; 4.5 GB memory footprint
- The domain is decomposed over a number of MPI ranks; typically, this number is 1 per NUMA domain (or device)
- For CPUs, OpenMP threads are added if the number of ranks is less than the number of processing elements on the node
- Timings are recorded for 15 sweeps over the linear system on a single device (GPU) or node (CPU)

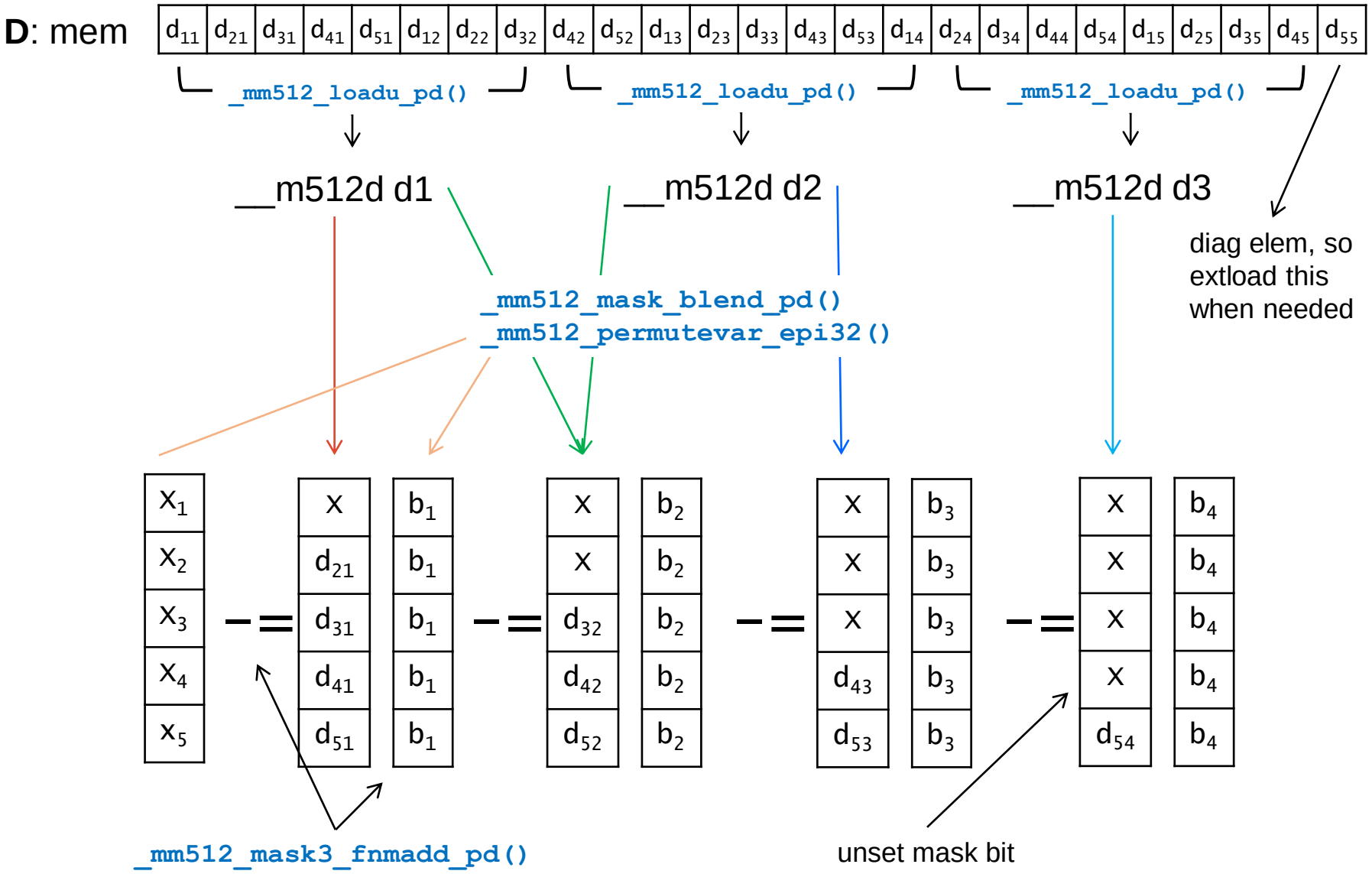
SKL/KNL Solver Benchmark

- Written in AVX512 vector intrinsics
- Map 3 5x5 blocks → 5 vector registers, accumulate partial sums
- Use register permutations to add partial sums
- Triangular solves are vectorized, but limited due to data dependencies
- Prefetch current row's data into L1 and next row's data into L2
- Speedup over legacy Fortran apx. 1.7x for KNL, 1.13x for SKL, though for SKL, this rises to 1.5x if run on a single core
- On KNL, half the speedup is due to prefetching, on SKL it does not help

SKL/KNL Solver Benchmark: Matvec

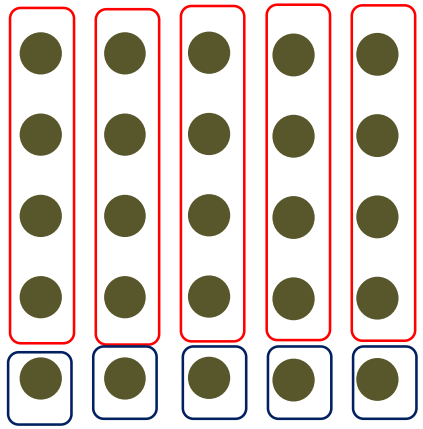


SKL/KNL Solver Benchmark: Triangular Solves



TX2 Solver Benchmark

- Written in Neon vector intrinsics; 128-bit vector holds 4xFP32
- Map 4 rows per column to 1 register, 5th row done with scalars



- Some performance improvement by prefetching the vector for the subsequent block
- Overall speedup over legacy Fortran solver of apx. 1.13x, similar to AVX512 on CPU

VE Solver Benchmark

- The long vector (512 for FP32) is difficult to use efficiently with CSR layout
- Legacy Fortran (CSR) is 10.0x slower on VE than SKL
- By interchanging the loop over blocks and the loop over rows, performance improves from 10.0x relative slowdown to 2.2x
- ELLPACK layout improves performance a further 3.0x (to 1.35x speedup over SKL), but limited by padding as the max number of nonzero blocks is 1.7x the average
- SELL-C-σ layout improves on ELLPACK by sorting groups of σ rows and zero-padding groups of C rows to the maximum row length in the group
- SELL-C-σ layout improves performance by a further 1.25x
- We sort all rows in each color and pad rows in groups of the vector length
- Use NEC compiler `vreg` directives to treat local arrays as vector registers

V100/A100 Solver Benchmark

- Written in CUDA C++
- Map 25 threads to a 5x5 block, 1 thread per entry
- Loop over blocks in a row, multiplying by the vector and accumulating
- Use shuffle instructions to reduce 25 sums to 5
- Use 25 threads to load D into shared memory
- Tune the number of warps per thread block to maximize performance; each warp processes 1 row of the matrix

MI50 Solver Benchmark

- Written in HIP
- Code is very similar to the CUDA solver, but each wavefront processes 2 rows: the first 32 threads (25 active) process 1 row, the remaining process a second row
- Aggregation is done in shared memory instead of using shuffles

Higher-level Framework Optimization

- We attempted to match the performance of our low-level optimized benchmarks using higher-level frameworks
- Thus far, this has only been done for NVIDIA GPUs
- The question to answer is whether or not the framework can deliver the same performance as a lower-level optimized code
- The structure of the CUDA C++ solver is used as a template; having this available makes the optimization process much easier
- Thus far, the frameworks studied are: OpenACC, SYCL, HIP, and OCCA

OpenACC/SYCL

OpenACC

- Implementation closely follows the CUDA C++ solver
- CUDA block and thread launch parameters are replaced by loops
- Shared memory is used for aggregation instead of shuffles

SYCL

- Uses Codeplay DPC++ compiler for NVIDIA GPUs (CUDA)
- Implementation closely follows the CUDA C++ solver
- Shared memory is used for aggregation instead of shuffles

HIP/OCCA

HIP

- *HIPify* tool converts CUDA C++ to HIP, which leaves the CUDA C++ solver code unchanged in this case

OCCA

- Implementation closely follows the CUDA C++ solver
- CUDA block and thread launch parameters are replaced by for loops
- Shared memory is used for aggregation instead of shuffles

Results

- The results table on the following slide shows relative performance normalized to the performance of the legacy Fortran code on SKL
- It also shows the percent of peak bandwidth obtained (value is computed based on the minimum number of bytes that must pass through main memory)
- The higher-level frameworks can match the performance of the CUDA C++ solver to within 3.0%

Caveats

- TX2 performance should not be taken as representative as we used a prototype system with significant anomalies in memory performance
- The VE optimized benchmark should not be considered complete
- A100 results use code tuned for V100

	SKL	KNL	TX2	VE	V100	A100	MI50
Fortran (CSR)	1.0 69.1%	0.79 31.3%	0.97 53.9	0.53 7.7%			
Fortran (SELL-C-σ)				1.84 26.7%			
OpenACC					3.77 74.1%	5.22 57.8%	
CUDA C++					3.86 75.8%	6.08 67.3%	
HIP					3.85 75.8%		2.90 51.3%
SYCL for CUDA					3.79 74.5%		
Vector Intrinsics	1.13 78.3%	1.34 52.8%	1.13 62.6%				
OCCA					3.76 74.0%		2.89 51.2%

Summary

- Optimized linear solver benchmarks were implemented for a variety of emerging and established HPC architectures
- For NVIDIA GPUs, the solver was implemented using a number of higher-level frameworks
- Results show the higher-level frameworks were able to match the performance of optimized CUDA C++ benchmark to within 3.0%

Future Work

- Additional frameworks and architectures will be studied
- Performance portability of the higher-level frameworks will be studied; i.e., when optimized for NVIDIA GPUs, how do the framework solvers perform on other architectures and how do optimizations for one architecture affect performance on the others?