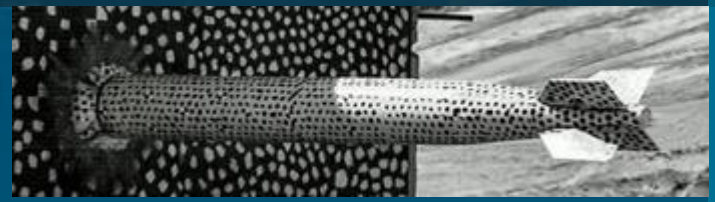
SAND2022-9567C

Sandia National Laboratories

A performance portable implementation of SIMD vector intrinsics on high-order, entropy-stable spectral collocation schemes for compressible turbulent flows



July 19, 2022

PRESENTED BY

Jerry Watkins, Victor Brunini, and Travis Fisher

Contributors: Jungyeoul (Brad) Maeng

North American High Order Methods Conference
San Diego, California

SAND

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

U.S. DEPARTMENT OF ENERGY NNSA

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Outline

- Introduction
  - Motivation – High-fidelity simulations
  - Motivation – Exascale computing
- High-order methods and performance portability
  - High-order, entropy-stable methods
  - High-order communication and operators
  - High-order methods on modern hardware
  - Performance portable C++ frameworks
- Performance portable SIMD vector intrinsics
  - SIMD vector intrinsics
  - SIMD performance portability
  - SIMD example for high-order
- Numerical Results
  - Case/Study setup
  - SIMD Performance

Motivation

Why are we interested in performance and portability?

High-fidelity simulations

High-fidelity simulations on exascale systems for analysis/design in hypersonics

Multi-fidelity design tools

Direct Numerical Simulation (DNS)
- Purpose: Model Development and Uncertainty Quantification
- Methods: High-order structured or unstructured methods

Wall-modeled LES and hybrid RANS/LES
- Purpose: Higher-fidelity engineering analysis
- Methods: High-order or low-dissipation finite volume

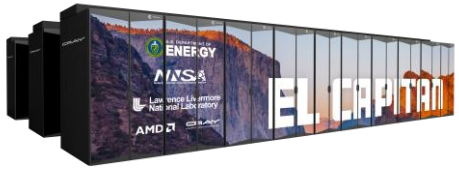
RANS
- Purpose: Engineering calculations
- Methods: Second-order finite volume

Reduced-order and semi-empirical models
- Purpose: Engineering
- Methods: Various

Target systems

LANL Trinity
Intel (KNL)

Sierra
NVIDIA (V100)

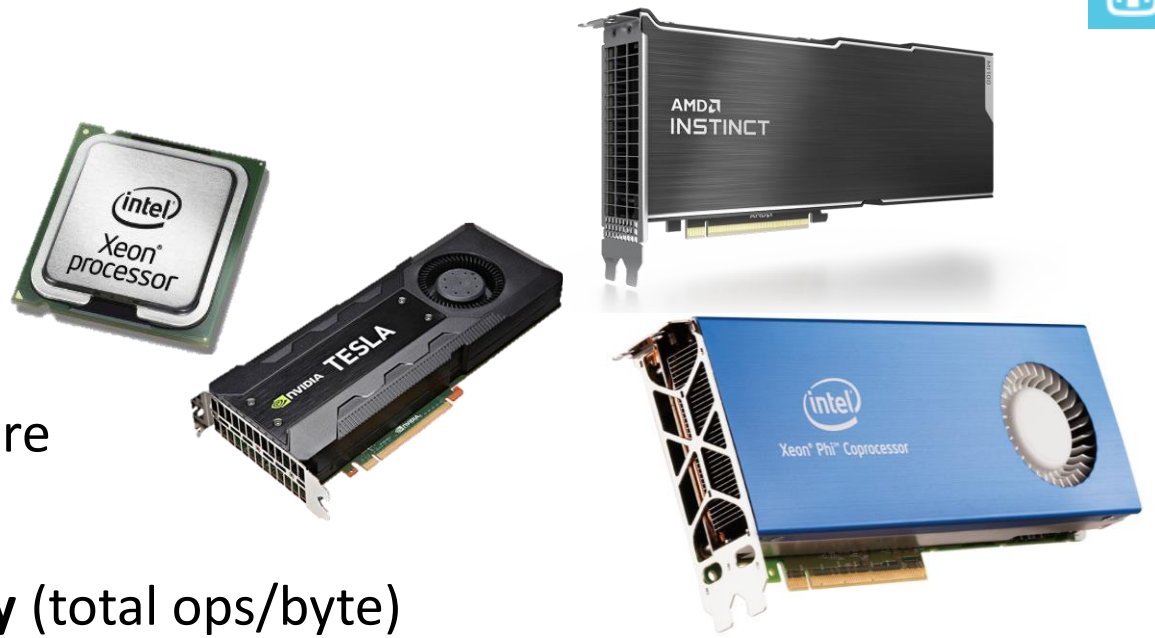


El Capitan
AMD

CROSSROADS

Crossroads
Intel

Exascale computing



Challenges:

- Diverse set of HPC vendors and architectures
  - Intel, AMD, NVIDIA, IBM, ARM-based
  - CPUs with vector processing; GPUs
- Software life cycle is much longer than hardware

Different architectures, trend remains the same

- Need algorithms with **high arithmetic intensity** (total ops/byte)
- Need fundamental **abstractions** during code development

Performance portability: A reasonable level of performance is achieved across a wide variety of computing architectures with the same source code.

Approaches:

- **Libraries** – High-level abstractions with specified input/output (e.g. BLAS)
- **Task-based** – Data-centric abstractions for mapping tasks to resources (e.g. Legion)
- **MPI+X** – Algorithmic-level abstractions for distributed (MPI) and shared (X) memory parallelism (e.g. **Directives**: OpenMP, OpenACC; **Frameworks**: Kokkos, RAJA, OCCA)

High-order methods and performance portability

What strategies are we using for high-order methods and performance portability?

High-order, entropy-stable methods

Entropy Stable Methods → Shock Capturing → Evaluate Turbulent Dissipation

Entropy Stable Summation-by-Parts Methods:

- Unstructured spectral collocation elements (**SCE**)

Shock capturing:

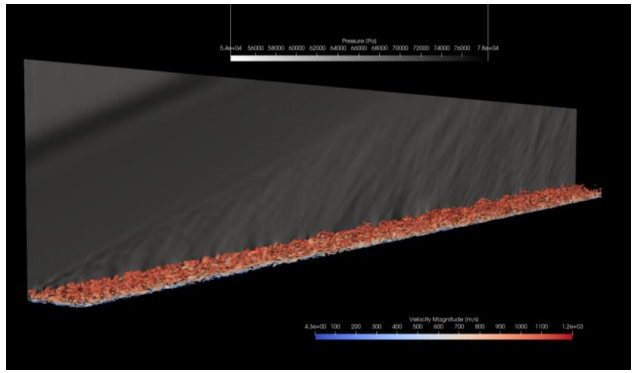
- Artificial viscosity
- Hybridized with Larsson shock sensor

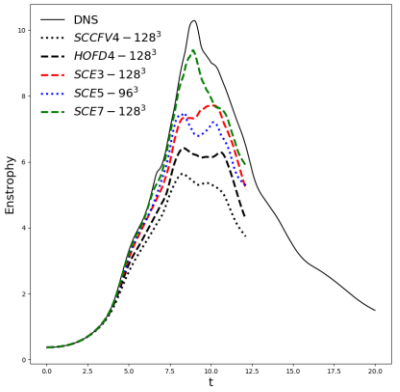
Evaluate Turbulent Dissipation:

- Need accurate and robust methods

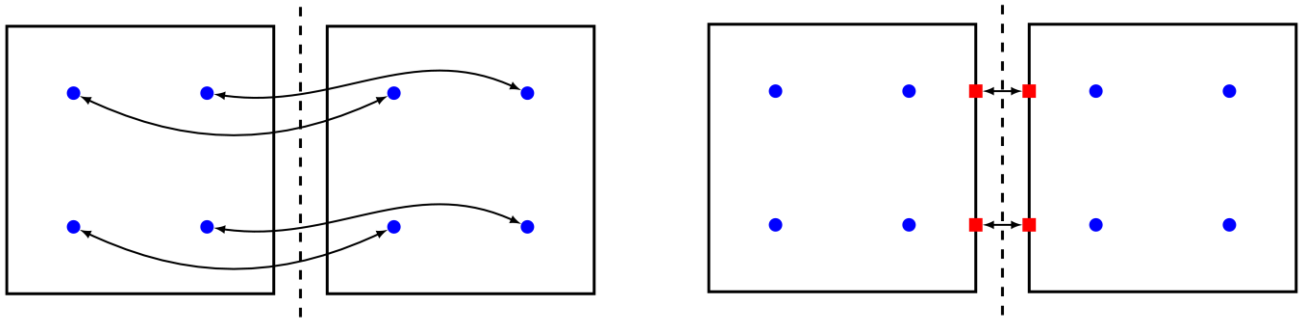
Where do discontinous Galerkin (DG) methods fit?

- **SCE** schemes are **nodal DG schemes** where nonlinear operators are used in place of linear operators to achieve entropy stability
- **Entropy stability** is used to help ensure **robustness** in the presence of **shocks**





High-order communication and operators



Communication:

- Ghost cell communication (LG)
- Ghost face communication (LGL)

Three major operators: Volume, Interface and Boundary

Linear Operators

$$\mathcal{D}w$$

- Gradient operators
- Matrix-vector operations in each cell (matrix-matrix including cells)
- w vector is reused for local assembly

Nonlinear Operator (Entropy Stability)

$$\left[\mathcal{D} \circ \mathcal{F}\right] 1$$

- Flux divergence operators
- Batch vector inner product in each cell (Batch vector-matrix including cells)
- $\mathcal{F}$ matrix is not reused

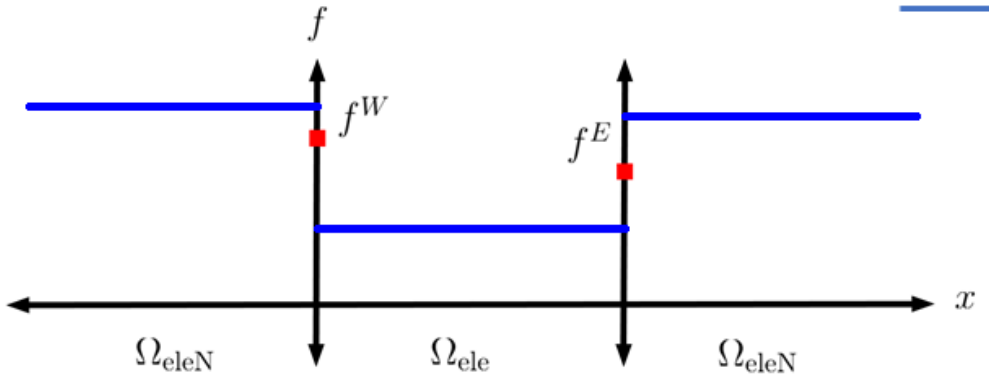
Two strategies used to avoid race conditions: **graph coloring** and **atomics**

High-order methods on modern hardware

Higher arithmetic intensity to efficiently utilize modern hardware

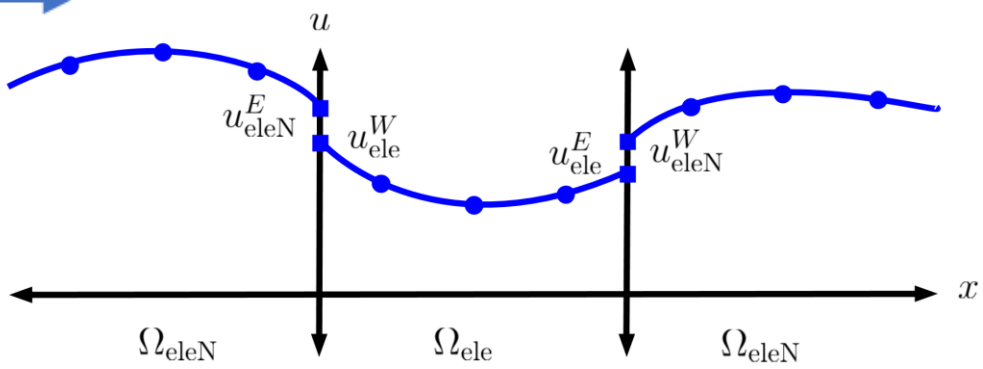
Example: No extrapolation operator in finite volume

Finite Volume



Higher FLOPS per byte

Nodal DG (LG)



Increasing polynomial order

- More operations per degree of freedom
  - Increases computational throughput
- Majority of operations are element-local
  - Allows for efficient use of shared memory
- Improves strong scaling; reduces error
- **Challenges:**
  - Diminishing returns
  - Better performance given a fixed error metric

Performance portable C++ frameworks

MPI+X: C++ frameworks within **Trilinos** for performance portability

- Distributed memory linear algebra (***Tpetra***)
- Shared memory parallelism (***Kokkos***)

Abstract **data layouts** and **hardware features** on current and future architectures

- Allocation: `U = Kokkos::DualView<double***[5]>(Ncells,Nspts,Nv)`
- Memory transfer: `U.modify_host(); U.sync_device();`
- Memory layout: `Kokkos::LayoutLeft` (col-major)
- Data parallelism: `Kokkos::parallel_for(policy, functor)`
  - `policy` defines iteration range: `Kokkos::RangePolicy(N)`
  - `functor` defines function to be parallelized

Allows researchers to focus more on **algorithm development** instead of **architecture specific programming**

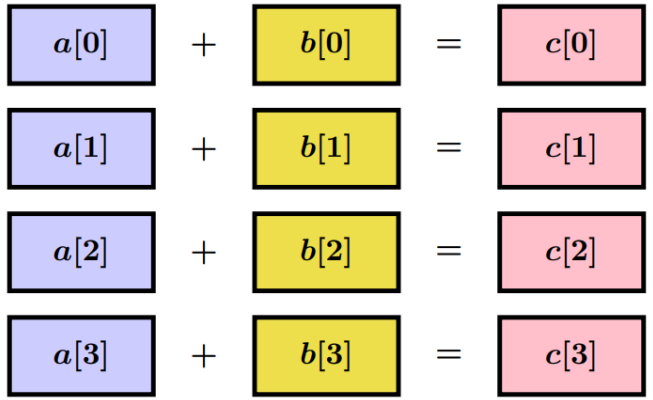
https://github.com/trilinos/Trilinos/
https://github.com/kokkos/kokkos/

Performance portable SIMD vector intrinsics

What does a performance portable implementation of SIMD vector intrinsics look like?

SIMD vector intrinsics

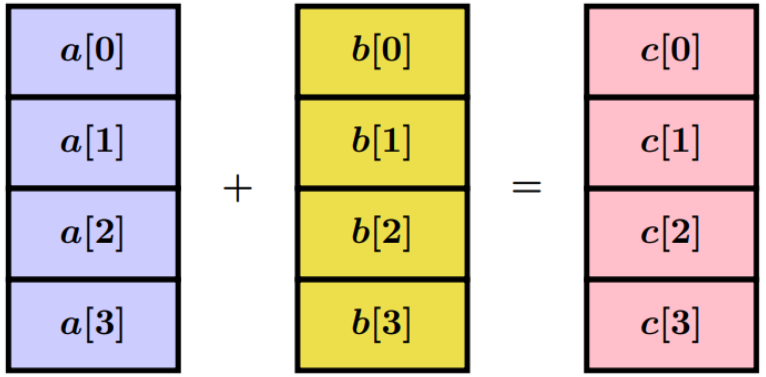
SISD: Single Instruction Single Data




```
for (int i = 0; i < N; ++i) {
  c[i] = a[i] + b[i];
}
```

- Compilers may auto-vectorize simple loops but not always
- Explicit vectorization with intrinsics improves performance

SIMD: Single Instruction Multiple Data




```
for (int i = 0; i < N/4; ++i) {
  __m256d a256 = _m256_loadu_pd(a+4*i);
  __m256d b256 = _m256_loadu_pd(b+4*i);
  __m256d c256 = _m256_add_pd(a256, b256);
  _m256_storeu_pd(c+4*i, c256);
}
for (int i = N-N%4; i < N; ++i) {
  c[i] = a[i] + b[i];
}
```

Data Type → `__m256d`

Func → `_m256_storeu_pd`

Remainder

SIMD performance portability

Architectures:

- Intel CPUs: AVX2, AVX-512
- ARM64: ARM Neon
- CUDA: SIMT model
  - maintain performance with same source code

Strategy:

- **SIMD_Double:** Data type for explicit vector
- Operators for **SIMD_Double**
- Functions for **SIMD_Double**
  - (e.g. loads/stores, math functions, if_then_else())

Libraries:

- Trilinos/STK – utilizes Kokkos simd-math library for SIMD data types support
  - Portable across AVX2/AVX-512/Neon
- Plan to transition to Kokkos SIMD
  - Portable across current and future SIMD architectures
  - Kokkos core will provide SIMD data types (Work-in-progress)
  - Dan Ibanez providing initial implementation: https://github.com/kokkos/kokkos/pull/5016

SIMD example for high-order

Example: Inviscid volume term

- Template parameter for SIMD index types (masked/unmasked)
- Array class operators for SIMD types
- Arithmetic operators for SIMD types
- Use of `auto` type for portability
- Function overloads in some cases

```
template <typename SIMDIndexT>
KOKKOS_FORCEINLINE_FUNCTION void
compute(const SIMDIndexT &cell, const int &spti) const
{
  for (int dir = 0; dir < num_dims; ++dir)
  {
    for (int j = jstart; j < jend; ++j)
    {
      ComputeFluxT(cell_V(cell,spti), cell_V(cell, sptj),
          tmpflux);

      for (int var = 0; var < numVars; ++var)
        hoflux[var] += coeffj * tmpflux[var];
    }
  }

  const auto lid = lid_map(cell, spt, 0);
  AddToResidual(lid, hoflux);
}
```

Numerical Results

How well does performance portable SIMD perform?

Case setup

Gradient Test (GradTest):

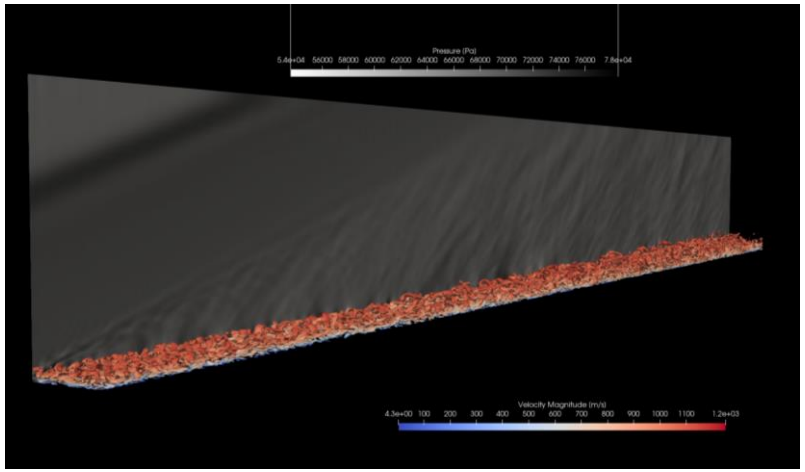
- Initialize solution with exact polynomial on unit cube
- Compute gradient, check exactness and performance

Taylor-Green Vortex (TGV):

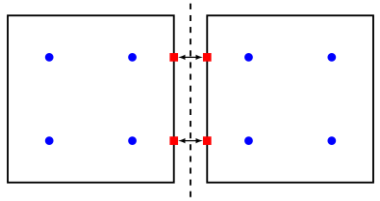
- Initialize solution on 3D Cartesian mesh
- Wall-clock time over 100 RK44 iterations

Mach 3.5 Flat Plate Boundary Layer ILES (FP):

- Synthetic turbulent inflow
- Shock capturing: Shock sensor limiting artificial viscosity
- BDF2 implicit time integration
- Low-order preconditioned Jacobian-free Newton-Krylov



Study setup



Methods:

- Structured cell-centered finite volume (**SCCFV**)
- Unstructured spectral collocation element (**SCE**)
  - P = 1-7; LGL

MPI+X Notation

$$r(\mathrm{MPI} + j\mathrm{X}),\ \ \mathrm{X} \in \{\mathrm{OMP}, \mathrm{OMPV}, \mathrm{GPU}\}$$

r = # MPI ranks
j = # OpenMP threads or GPUs/rank
X = architecture for shared memory parallelism

Node Architectures

- Intel Haswell (**HSW**) – 32 cores, 64 threads, AVX2 (4 doubles)
- Intel Knights Landing (**KNL**) – 64 cores, 256 threads, AVX512 (8 doubles)
- Intel Cascade Lake (**CLX**) – 48 cores, 96 threads, AVX512 (8 doubles)
- ARM64 Cavium ThunderX2 (**TX2**) – 56 cores, 112 threads, Neon (2 doubles)
- NVIDIA Volta (**V100)** – 4 GPUs, Cuda (no simd)

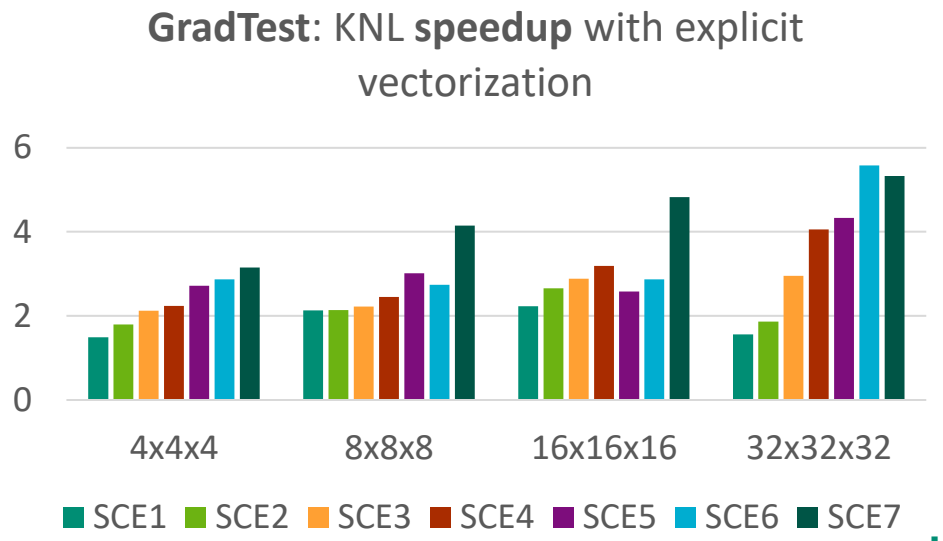
SIMD performance on LayoutLeft

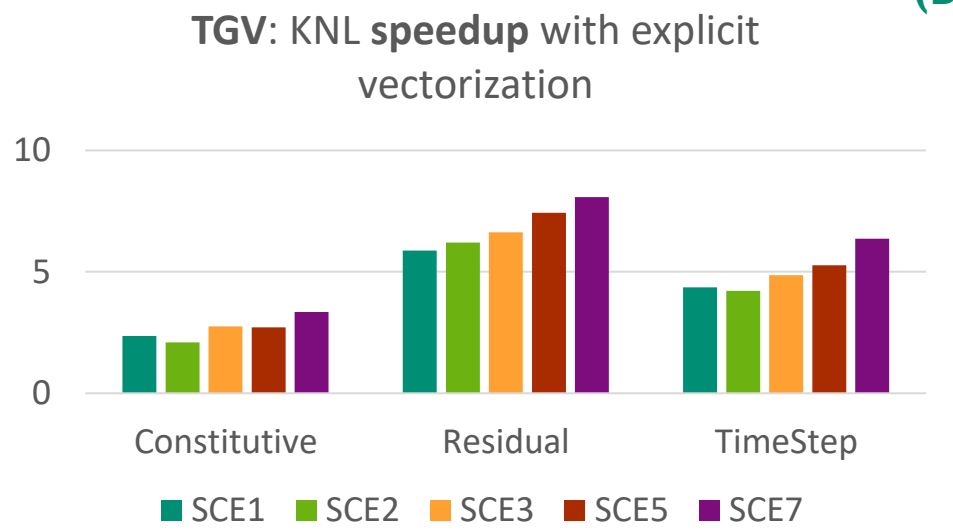
Setup:

- Compare with and without explicit vectorization given cell contiguous data
  - i.e. LayoutLeft on Array(**cell**,spt,var)
  - GradTest and TGV

Results:

- Speedup with explicit vectorization
  - GradTest: up to 3.2x (HSW), 5.6x (KNL)
  - TGV Residual: up to 3.9x (HSW), 8.1x (KNL)
  - TGV Step: up to 3.3x (HSW), 6.4x (KNL)
- Larger speedups at higher orders
  - GradTest: 1.5->3.2x (HSW), 1.6->5.6x (KNL)
  - TGV Residual: 2.5->3.9x (HSW), 5.9->8.1x (KNL)
  - TGV Step: 2.3->3.3x (HSW), 4.4->6.4x (KNL)





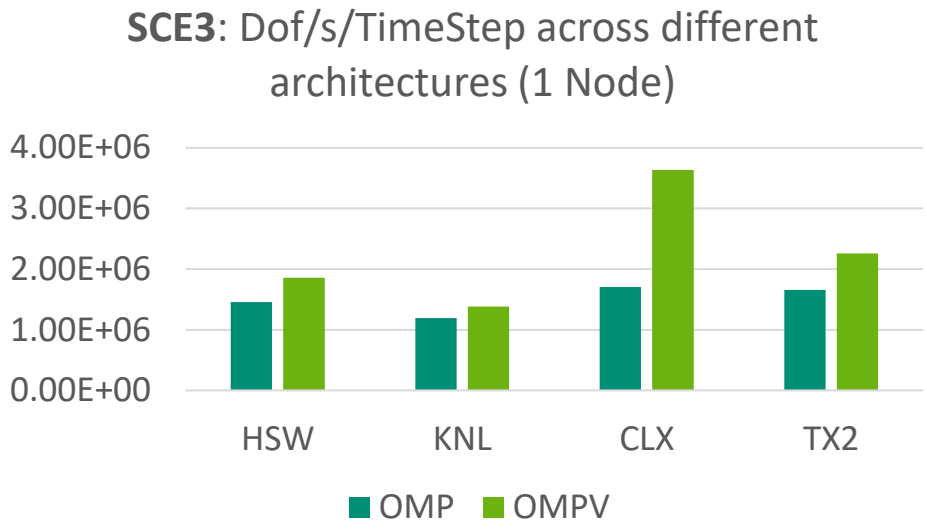
SIMD performance on TGV

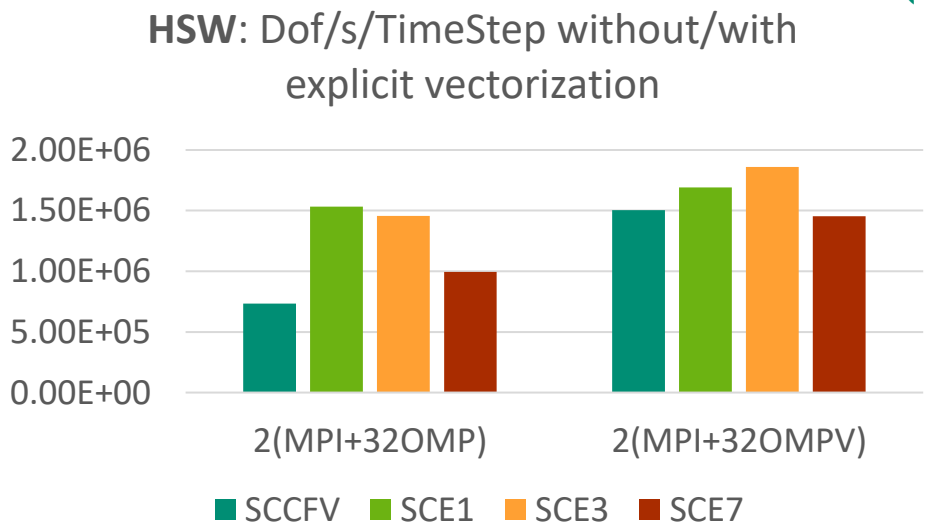
Setup: 128^3 Degrees of Freedom (Dof)

- Compare best MPI+OpenMP cases
  - OMP: LayoutRight on Array(cell,spt,**var**)
    - No explicit vectorization
  - OMPV: LayoutLeft on Array(**cell**,spt,var)
    - Explicit vectorization

Results:

- Higher throughput with explicit vectorization
  - LayoutRight performs better in rare cases
    - Better caching?
- Larger throughput at higher orders
  - SCE7 has smaller throughput
- Large benefit in SCCFV
  - High-order improvements are relatively modest





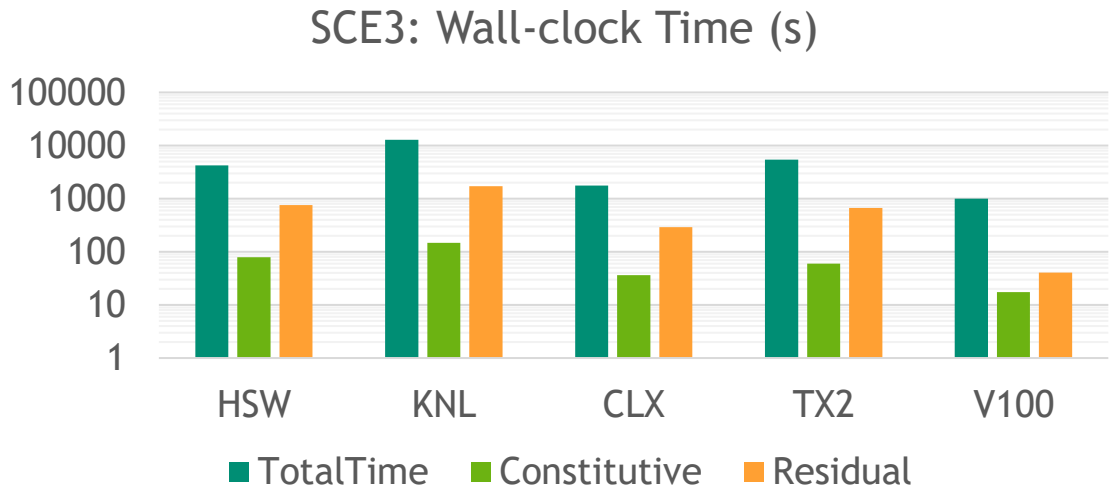
SIMD performance on flat plate

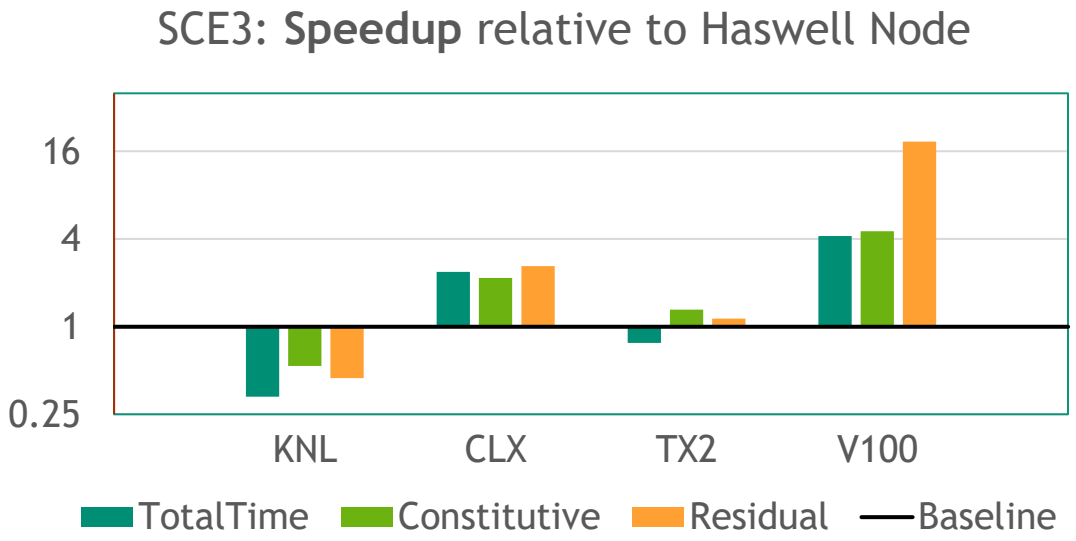
Setup:

- Jacobian-free Newton-Krylov
  - Frechet approximation
  - Low-order, Block Jacobi preconditioner
- All results are using **8 nodes**
- Speedup relative to Haswell (**HSW**) node

Results:

- Largest speedup from GPU node (18.7x)
- Largest CPU speedup from CLX (2.6x)





Discussion

Discussion

- HPC architectures are changing rapidly which poses a significant challenge
- **Trilinos/Kokkos** offers an efficient way to meet this challenge for large scale, high-fidelity simulations
- SIMD performance portable libraries offer a means to perform explicit vectorization on a variety of different architectures
- Performance results are promising but more R&D is needed to improve robustness and performance for high-order methods and hypersonics

Future Work

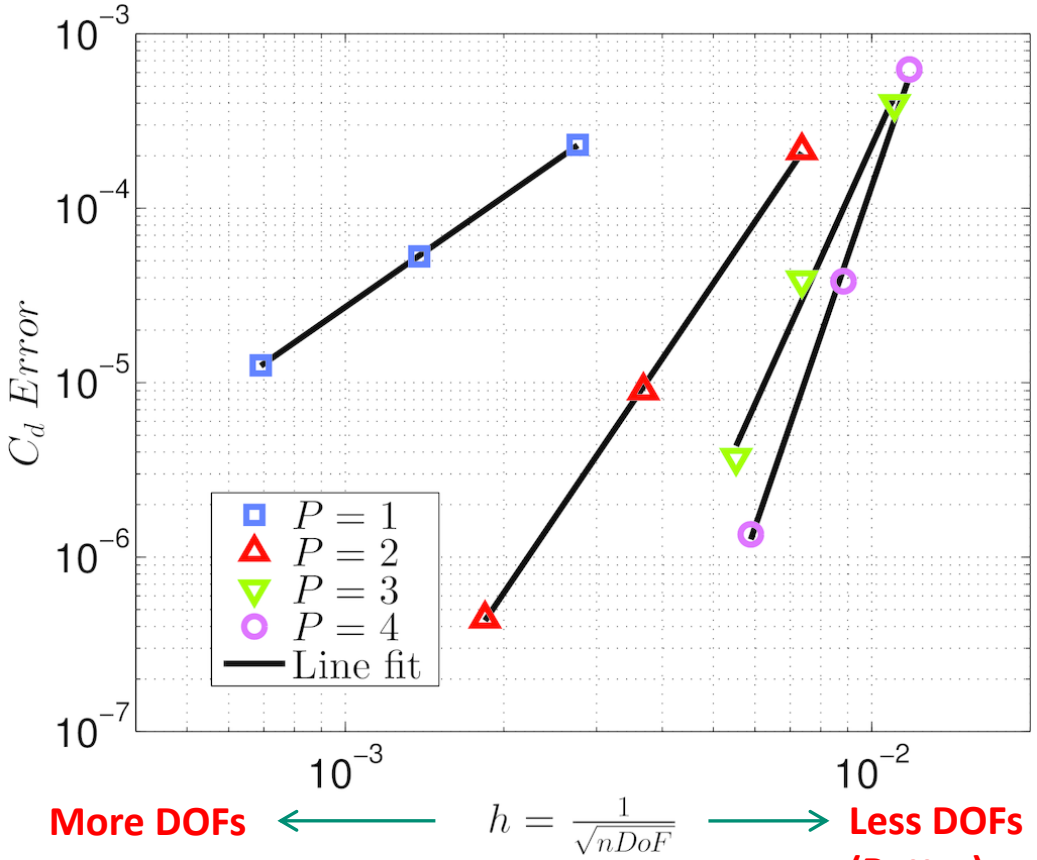
- Additional data layout testing (strided or AoSoA)
- Full integration of Kokkos SIMD
- Improve end-to-end performance of implicit solver

Backup slides

High-order methods – Introduction

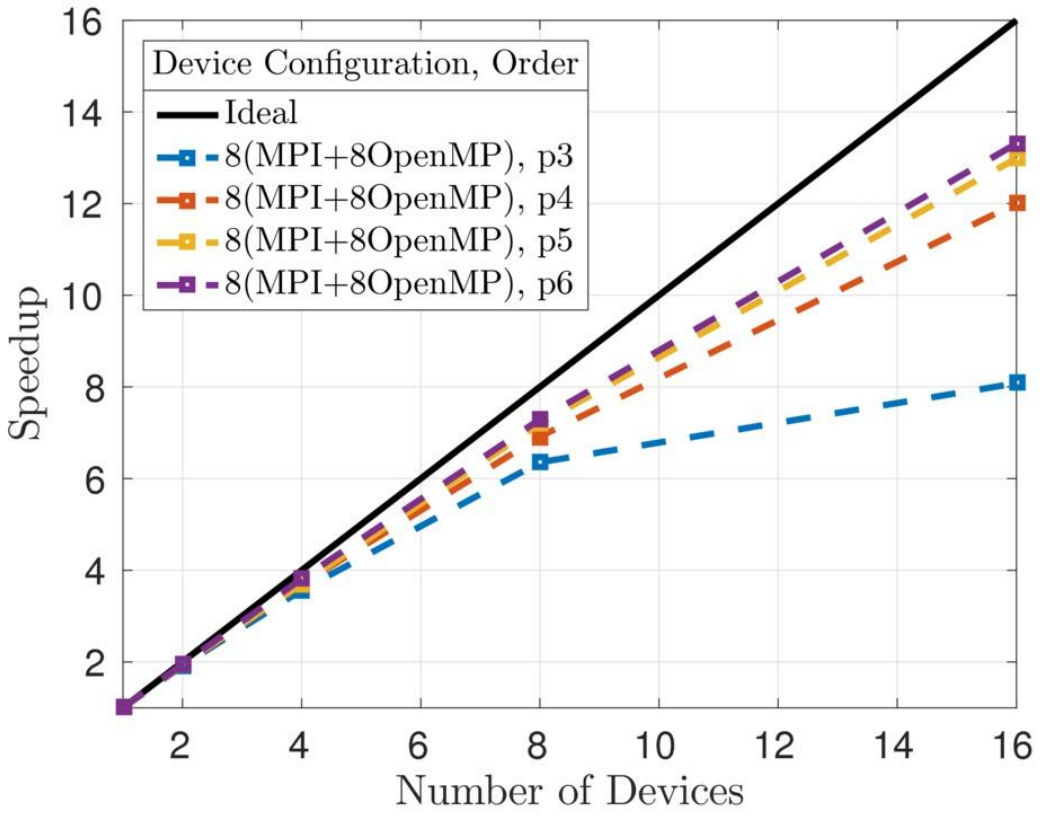
Increasing polynomial order allows for more efficient simulations

Drag coefficient error for Joukowski airfoil (Ma=0.5, Re=1000) using ZEFR (2017)



More DOFs ← → Less DOFs (Better)

Strong Scalability for cubed-sphere, shallow water equations using Aeras (2018)



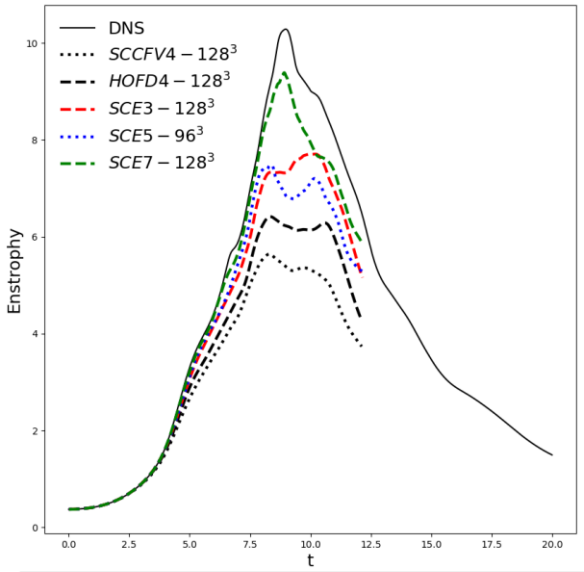
Higher P (Better)

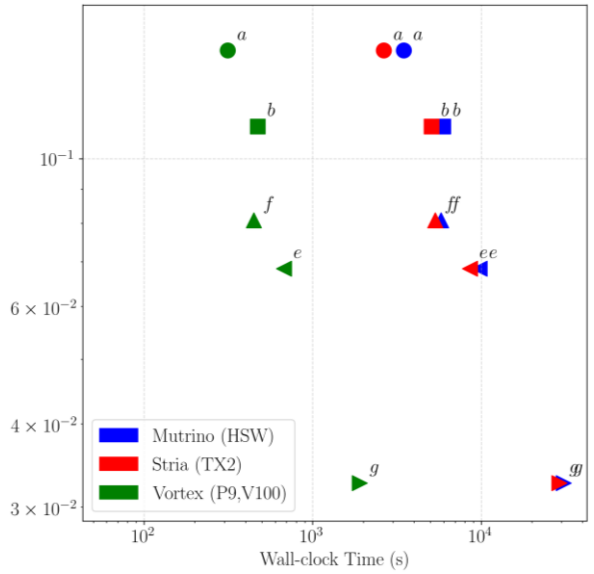
TGV Results

First attempt to quantify performance of high-order

$$Figure\ of\ Merit = \frac{T_{SCCFV} \cdot err^{\varepsilon}_{SCCFV}}{T \cdot err^{\varepsilon}}$$

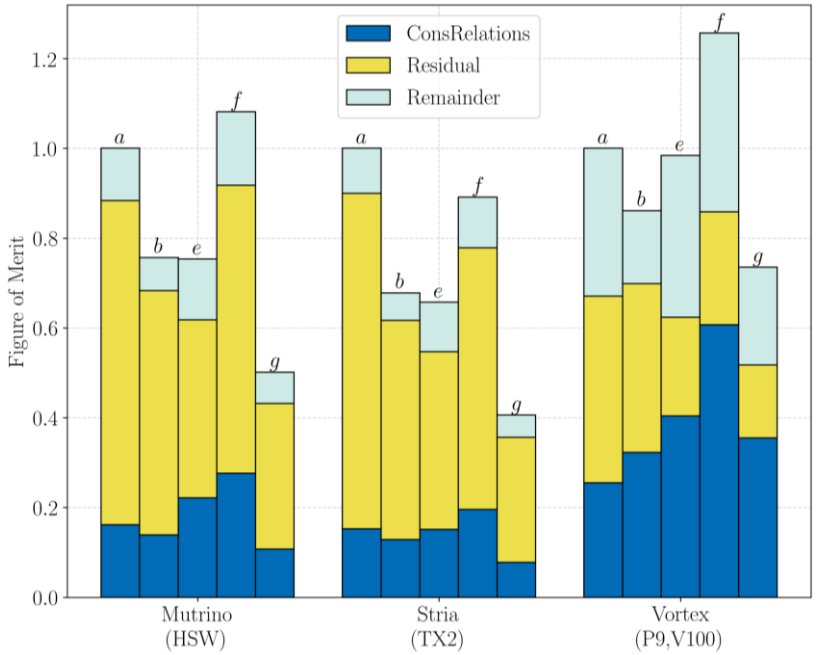
- T - Wall-clock time for 30s simulation time (s)
- $err^{\varepsilon} = \int |\varepsilon(t) - \varepsilon_{ref}(t)|^2 dt$ - Enstrophy error
- Reference: Spectral element solution, 512^3
- Note: results are architecture independent





Analysis:

- Current optimal is near **SCE5**
- High-order performing better on GPUs
- Bottlenecks
  - CPU – Residual (computation)
  - GPU – ConsRelations (gradient, communication)
  - Remainder also needs more profiling/improvement



Discretization - DoF

a: SCCFV4 - 128^3
b: HOFD4 - 128^3
e: SCE3 - 128^3
f: SCE5 - 96^3
g: SCE7 - 128^3

Matrix-free methods – Introduction

Time Stepping Schemes:

Semi-discrete equation

Solution

Divergence of the flux

$$\frac{\partial \boldsymbol{U}}{\partial t} = -\boldsymbol{V}^{-1}\,\hat{\nabla}^{\delta}\cdot\hat{\boldsymbol{F}} \qquad R(\boldsymbol{U}) = -\boldsymbol{V}^{-1}\,\hat{\nabla}^{\delta}\cdot\hat{\boldsymbol{F}}$$

Determinant geometric Jacobian matrix

Residual

Runge-Kutta methods

$$\boldsymbol{U}^{n+1} = \boldsymbol{U}^{n} + \Delta t^{n}\sum_{s=1}^{N_{\text{stages}}} b_s \boldsymbol{R_s}$$

Stages can be solved sequentially (No linear system)

BDF1 for steady-state

$$\Delta \boldsymbol{U}^{m} = \Delta \mathbf{T}^{m}\, R\left(\boldsymbol{U}^{m+1}\right)$$

$$\left(\left(\Delta \mathbf{T}^{m}\right)^{-1} - \frac{\partial \boldsymbol{R}^{m}}{\partial \boldsymbol{U}^{m}}\right)\Delta \boldsymbol{U}^{m} = \boldsymbol{R}^{m}$$

Residual Jacobian matrix (Large sparse matrix)

Explicit methods (Ex: RK44)

- Pros:
  - No matrix and linear system solve required
  - Performance limited by residual
- Cons:
  - Time step often limited by numerical stability
  - Difficult to determine reliable time step
  - High-order time step restriction h/P^2

Implicit methods (Ex: BDF1)

- Pros:
  - Time step tuned to accuracy
  - More computation per sequential time step
- Cons:
  - Matrix and linear system solve required
  - Nonlinear stability is not guaranteed
  - High-order matrix size P^3

Matrix-free methods – Introduction

Jacobian-free Newton-Krylov:

- Using GMRES to solve the linear system only requires matrix-vector products
  - Less memory (no need to store a large matrix) which allows an increase in utilization
  - Less data movement (no need to assemble a large matrix) which allows efficient bandwidth use
  - More computation (evaluate matrix-vector product at each linear iteration)
- May need matrix if preconditioning is required

Matrix-free approximate

Use Frechet derivative

- Pros:
  - Performance limited by residual
- Cons:
  - Residual evaluation at each linear iteration
  - Approximation may limit stability

Matrix-free exact

Use automatic differentiation (AD)

- Pros:
  - Quadratic convergence at best
- Cons:
  - AD evaluation at each linear iteration
  - Nonlinear stability is not guaranteed
  - More difficult to solve for stiff equations

Matrix-free methods – Results

Example: Turbulent flat plate boundary layer (M=0.2)

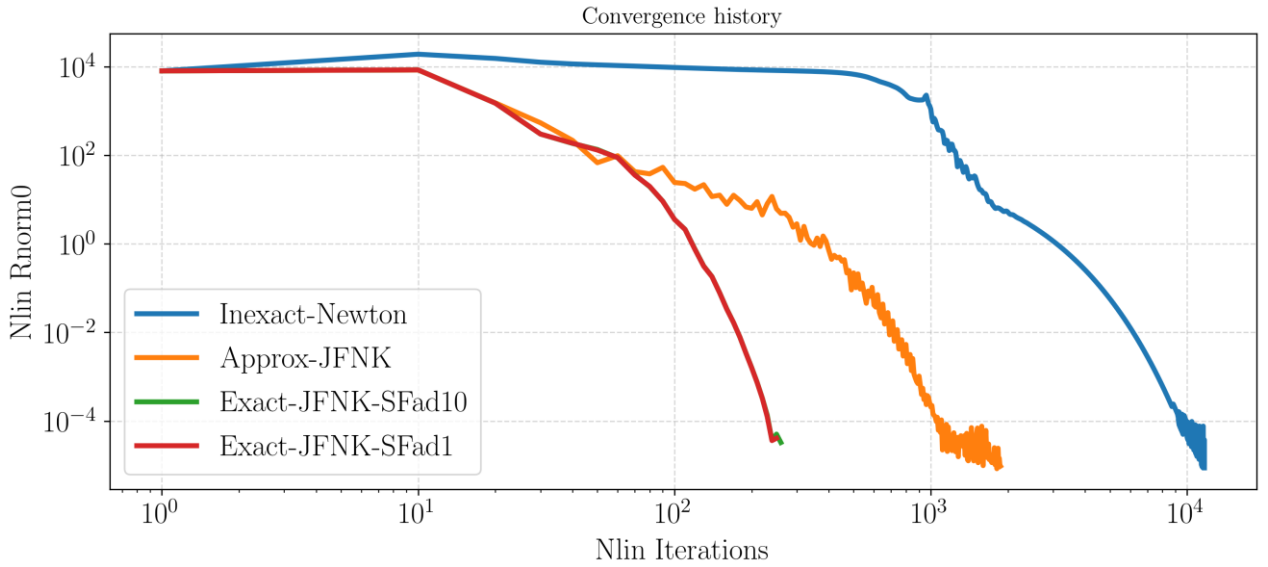
Steady-state: pseudo-transient continuation

- Inexact Jacobian
  - Second-order finite volume discretization
  - First-order inviscid Jacobian, neglect viscous cross-terms
  - Used for inexact Newton and preconditioned JFNK
- Linear solve
  - Block tridiagonal solver or GMRES/ILU

Results:

- Exact matrix-free led to 7x speedup over inexact Newton for SA turbulence model
- Robustness issues when applying matrix-free methods to SST turbulence model

Convergence history for flat plate, Spalart-Allmaras (SA) turbulence model



	Nlin Iterations	Problem Solve Time (s)	Belos Solve Time (s)
Inexact-Newton	11684	109.216	58.7482
Approx-JFNK	1873	51.9301	43.1881
Exact-JFNK-SFad10	262	46.3098	44.2337
Exact-JFNK-SFad1	256	15.2841	13.4616

Matrix-free methods – Results

Focusing on performance with Jacobi:

All results are using **8 nodes** on each platform

- Intel Haswell (**HSW**) CPU (32 cores/node)
- ARM64 Cavium ThunderX2 (**TX2**) CPU (56 cores/node)
- NVIDIA Volta (**V100**) GPU (4 GPUs/node)

Inexact Newton performed better on **CPU** platforms

Matrix-free approximate performed better on **GPU** platforms

Matrix-free exact performed better on platforms with **more threads**

Fastest wall-clock time was on **GPU**

- **5x** over fastest **HSW** time