SAND2019-6444C

Embedded UQ on Emerging Computing Architectures

Eric Phipps (etphipp@sandia.gov)
Scalable Algorithms Department
Sandia National Laboratories

2019 NNSA/ASC-CEA/DAM

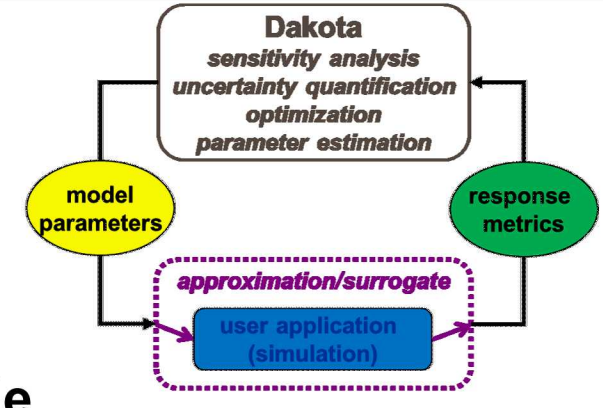
June 25-27, 2019

SAND2019-xxxx

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.

Emerging Architectures Motivate New UQ Approaches

- Sampling-based UQ approaches traditionally implemented as an outer loop:
  - Repeatedly call forward simulation for each sample realization
  - Coarse-grained parallelism over samples
- Many important scientific simulations struggle with emerging architectures
  - Irregular memory access patterns
  - Difficulty in exploiting fine-grained parallelism
- Can we improve aggregate UQ performance by breaking the outer-loop pattern
  - Reduce number of samples through gradients computed by automatic differentiation
  - Improve sampling performance through embedded ensemble propagation



http://dakota.sandia.gov

Fast Gradient Computations Reduce Aggregate UQ Cost

- FENL example in Trilinos-couplings package in Trilinos
  - Simple nonlinear diffusion equation
  - 3-D, linear FEM discretization on 1x1x1 cube, unstructured mesh
  - CG iterative solver (Belos) with smoothed aggregation AMG preconditioning (MueLu)
  - PCE methods implemented by Dakota

$$-\nabla \cdot (\kappa(x,y)\nabla u) + u^2 = 0,$$

$$\kappa(x,y) = \kappa_0 + \sigma \sum_{i=1}^{M} \sqrt{\lambda_i}\kappa_i(x) y_i$$

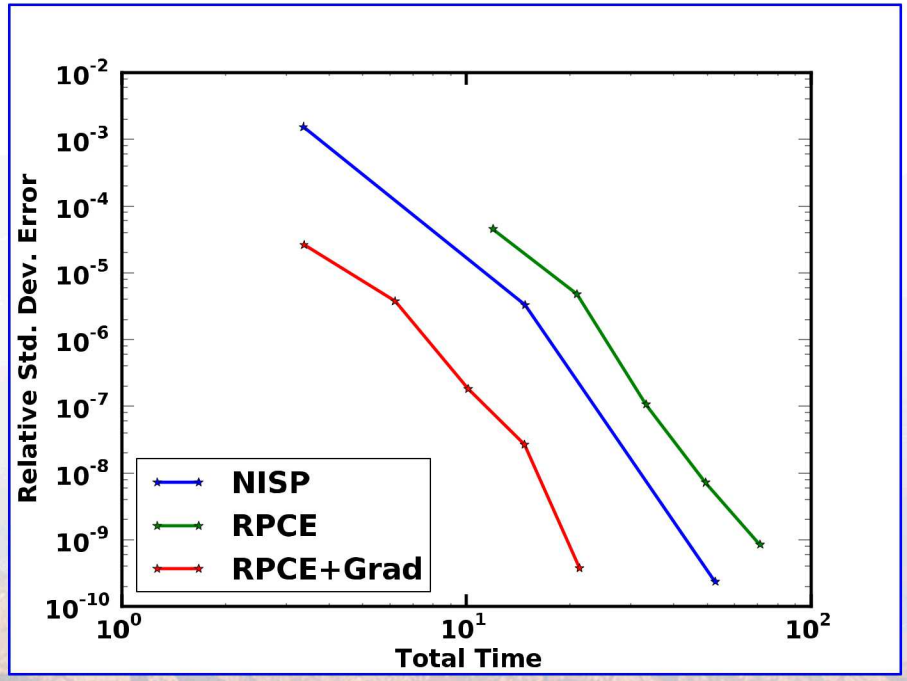
$$h(u) = \|u\|^2$$

DAKOTA

http://dakota.sandia.gov

Trilinos

http://trilinos.org



Computing derivatives efficiently in large-scale codes

- These techniques require accurate and fast evaluations of partial derivatives
- These can always be derived and coded by-hand
  - Time consuming
  - Error prone
- One alternative is numerical differentiation
  - Difficult to make accurate, robust
  - Can be very expensive
- A better alternative is automatic differentiation
  - Evaluate *analytic* derivatives automatically, efficiently
  - Works by transforming *code* to compute analytic derivatives

What is Automatic Differentiation (AD)?

- Technique to compute analytic derivatives without hand-coding the derivative computation
- How does it work -- freshman calculus
  - Computations are composition of simple operations (+, *, sin(), etc…) with known derivatives
  - Derivatives computed line-by-line, combined via chain rule
- Derivatives accurate as original computation
  - No finite-difference truncation errors
- Provides analytic derivatives without the time and effort of hand-coding them

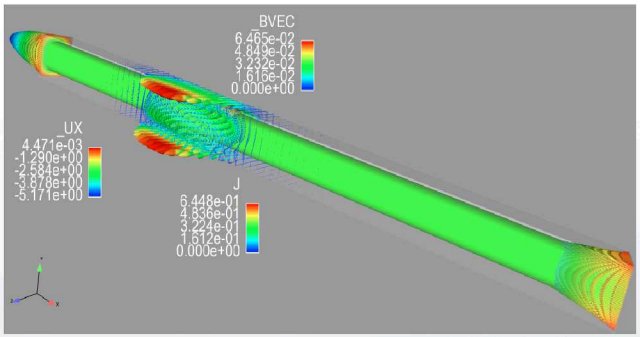
$$y = \sin(e^x + x \log x), \quad x = 2$$

	x	$\frac{d}{dx}$
$x \leftarrow 2$	2.000	1.000
$t \leftarrow e^x$	7.389	7.389
$u \leftarrow \log x$	0.693	0.500
$v \leftarrow xu$	1.386	1.693
$w \leftarrow t + v$	8.775	9.082
$y \leftarrow \sin w$	0.605	-7.233

Sacado: AD Tools for C++ Applications

- Package in Trilinos
  - http://trilinos.org or http://github.com/trilinos
  - Open source license
- Operator overloading-based approach
  - Sacado provides C++ data types implementing AD
  - Type of variables in code replaced by AD data type
  - AD object for each variable stores value of that variable and its derivatives
  - Mathematical operations replaced by overloaded versions implementing chain-rule
  - Expression templates reduce overhead
- Careful software engineering required to use effectively
  - Manually exploit simulation structure/sparsity
  - AD only applied at "element" level

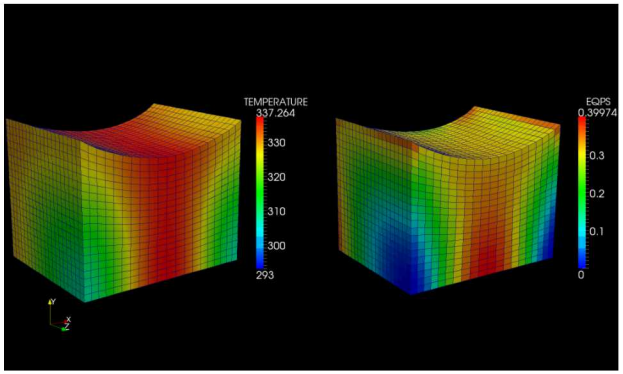
http://trilinos.org



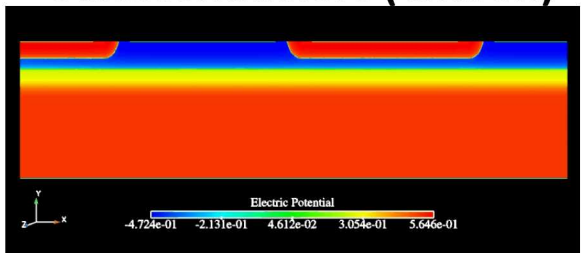
Iso-velocity adjoint surface for fluid flow in a 3D steady MHD generator in Drekar computed via Sacado (Courtesy of T. Wildey)

Tools and Approach Have Impacted Many Sandia Applications

Thermal Mechanics

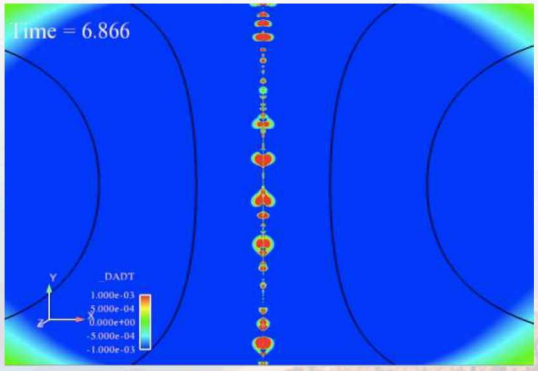


Semiconductors (Charon)

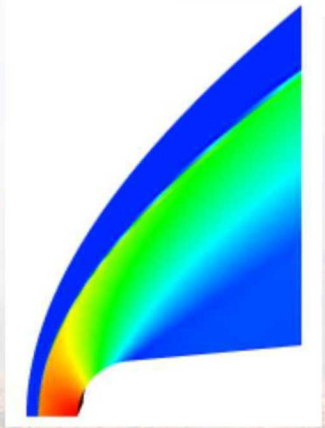


- Quantum Devices (QCAD)
- Circuits (Xyce)
- EM-Plasma (EMPIRE)
- Fluids (Aria)

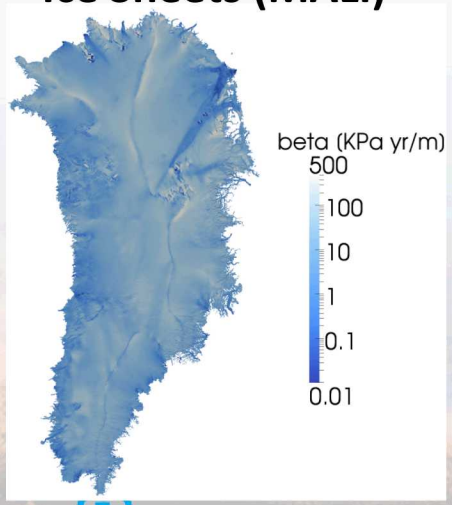
MHD (Drekar)



CFD (SPARC)

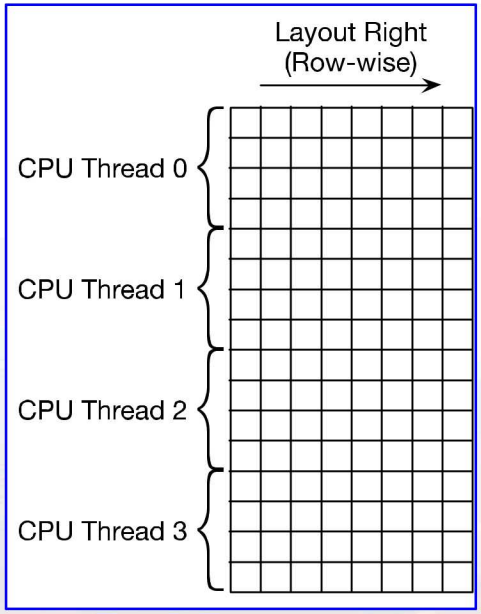


Ice Sheets (MALI)

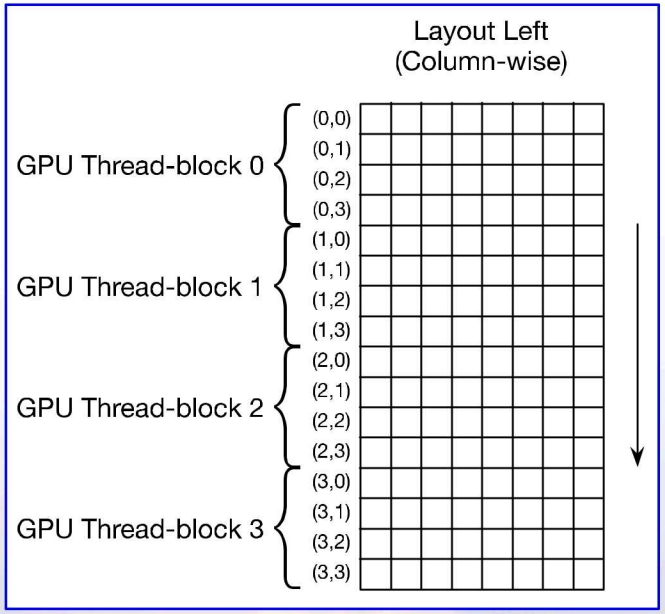


Kokkos Layout Polymorphism for Performant Memory Accesses

- **CPU/MIC**
  - Each thread accesses contiguous range of entries
  - Ensures neighboring values are in cache

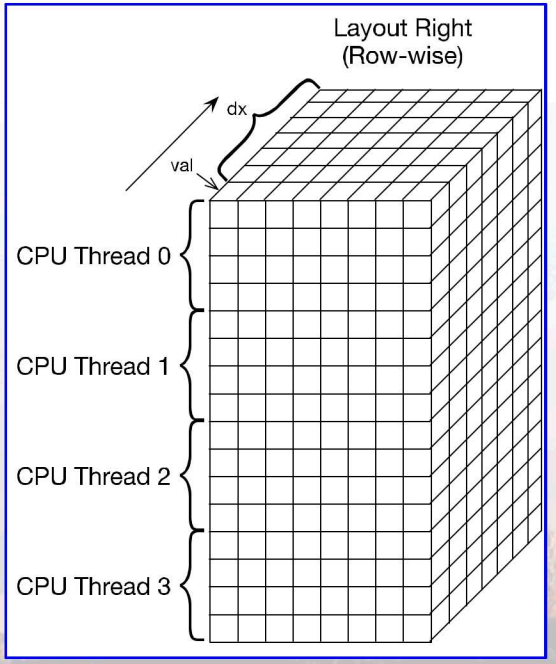


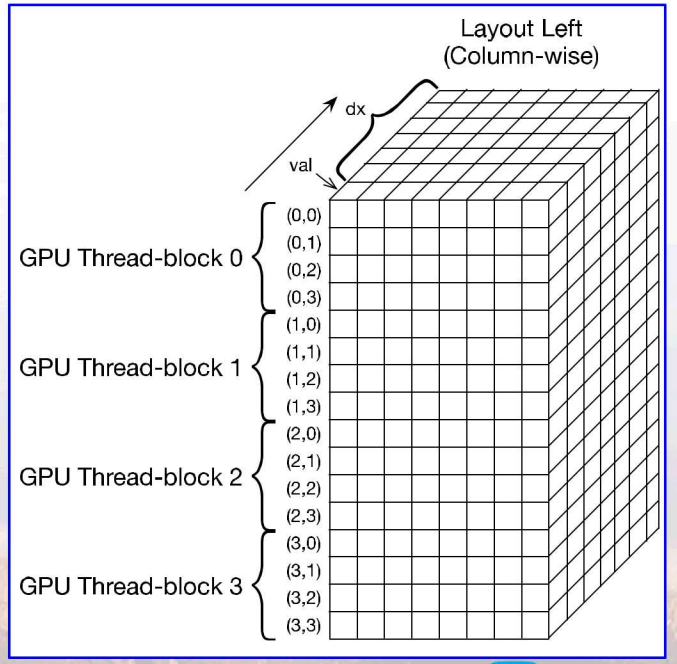
- **GPU**
  - Each thread accesses strided range of entries
  - Ensures coalesced accesses (consecutive threads access consecutive entries)



What happens when we use Sacado in Kokkos parallel kernels?

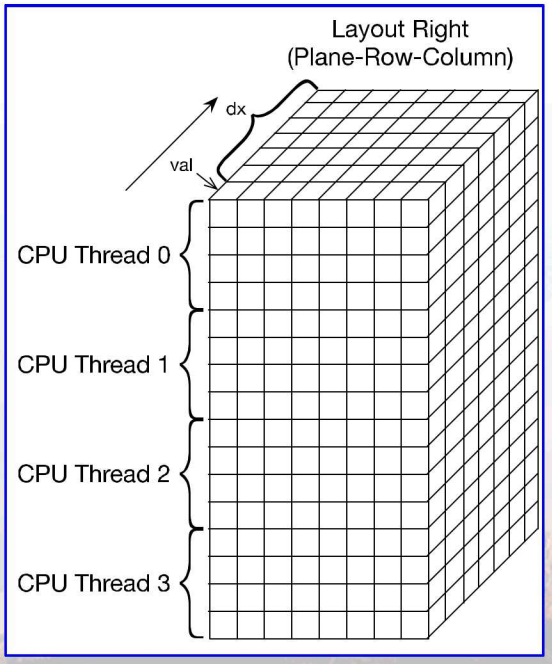
- Derivative components always stored consecutively
  - CPU: Good cache, vector performance
  - GPU: Large stride causes bad coalescing

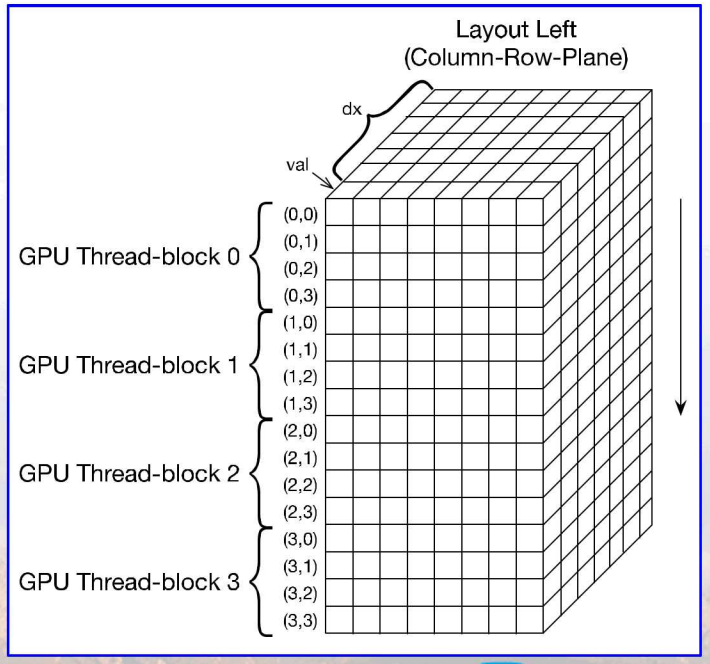




Want good AD performance with no modifications to Kokkos kernels

- Achieved by specializing data structures for Sacado scalar types
  - Rank-r Kokkos::View internally stored as a rank-(r+1) array of doubles
  - Kokkos layout applied to internal rank-(r+1) array



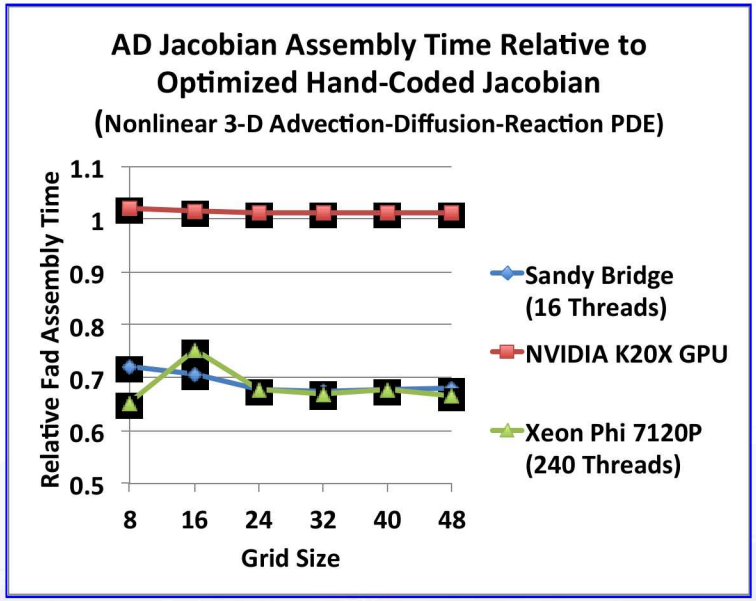


Sacado + Kokkos Performance

- **Performance test for measuring Jacobian/Residual assembly using Sacado**

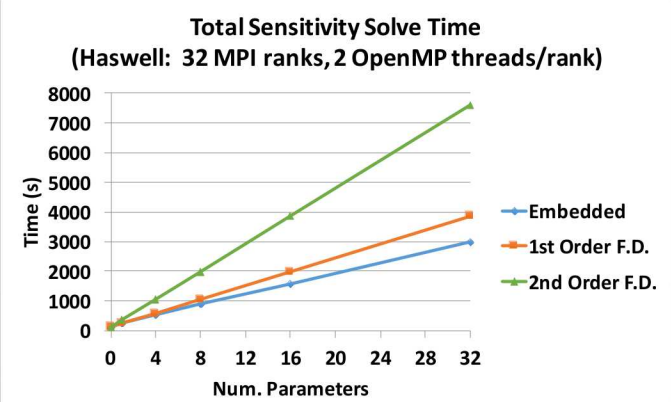
$$-\nabla \cdot (\kappa \nabla u) + \alpha v \cdot \nabla u + \beta u^2 = 0$$

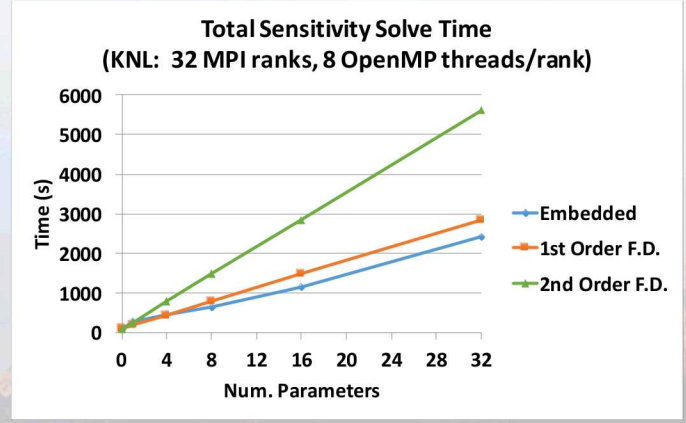
  - **3-D, linear FEM discretization**
  - **1x1x1 cube, unstructured mesh**
  - **Derived from FENL Kokkos example (H.C. Edwards)**
  - **Thread-parallel matrix/residual assembly**

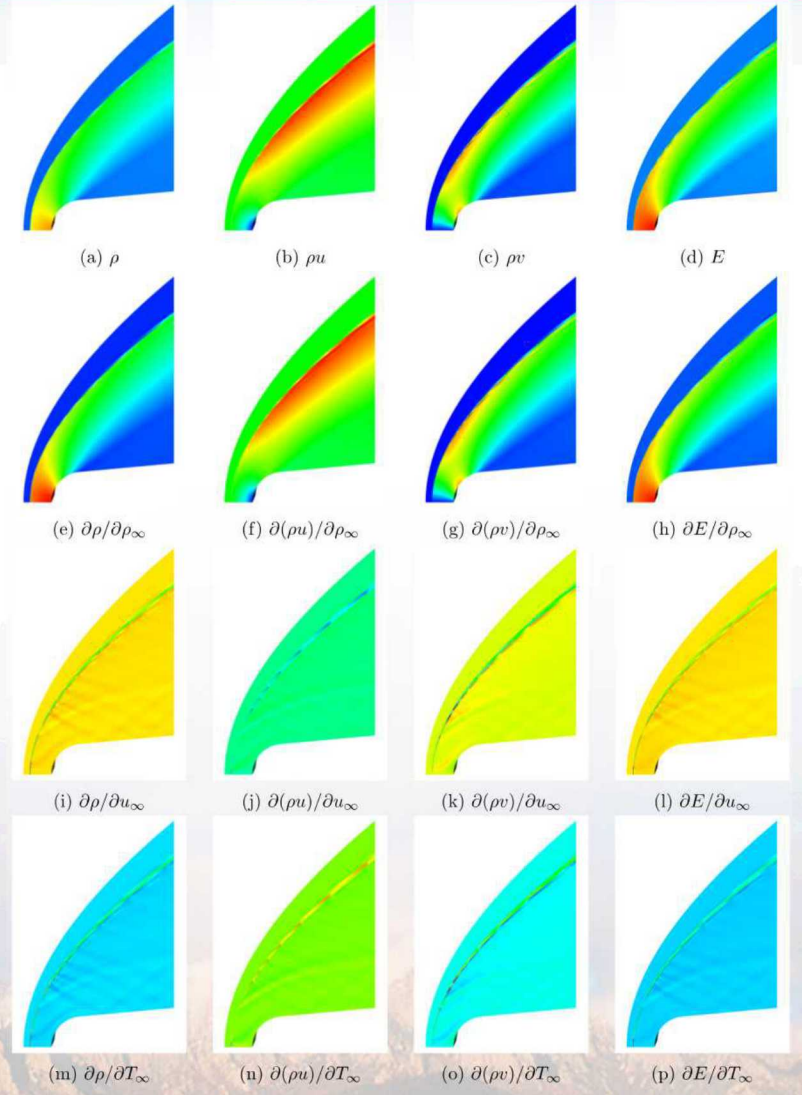


Supersonic flow past a blunt wedge

- SPARC ATDM application code
  - M. Howard et al
  - Structured grids
  - Cell-centered finite volume scheme

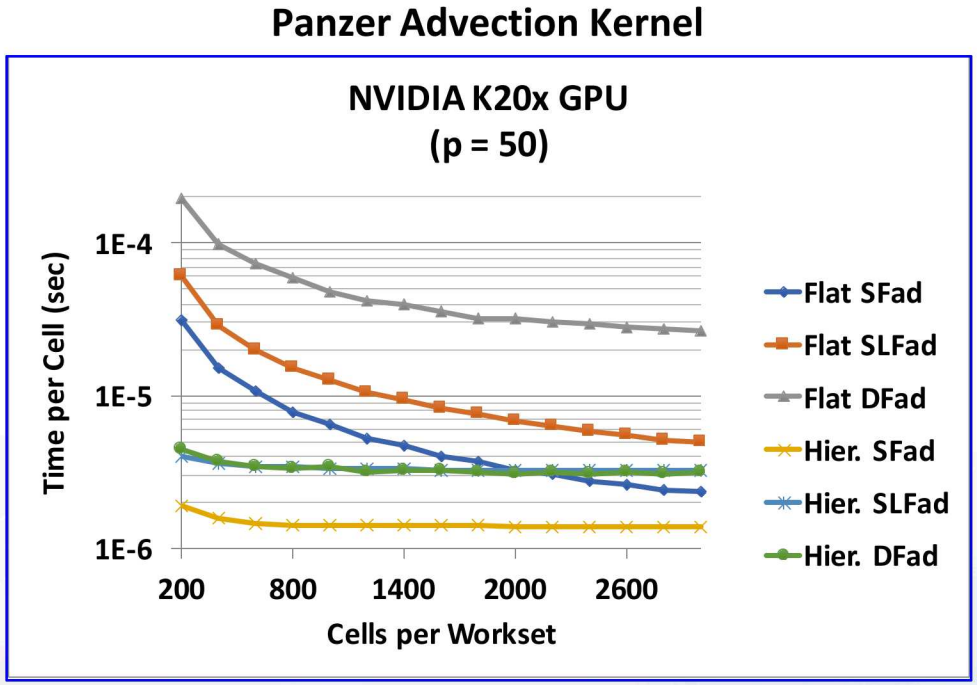






Hierarchical Parallelism

- Layout approach was explored to minimize code user-code changes for Sacado
- Derivative propagation provides good opportunities for exposing more parallelism
  - Parallelism across derivative array
  - Code may not expose enough parallelism natively (e.g., small workset)
- Exploring modifications to Sacado, Kokkos to map GPU thread parallelism across derivative calculations
  - Outcome of ATDM FY16 L2 Milestone
  - With Christian Trott, Eric Cyr, Matt Bettencourt, Roger Pawlowski



Embedded Ensemble Propagation

- Divide samples into small groups called ensembles
  - ~32 samples/ensemble
- Replace sample-dependent data with ensemble arrays
  - Same templating approach as with Sacado
  - Provide overloaded operators for math on ensemble arrays
- Better exploit fine-grained hardware parallelism
  - Map fine-grained threads, vector instructions across ensemble arrays
- Improved memory access patterns
  - Scatter/gathers for sample-dependent data replaced with ensemble packed/coalesced loads/stores
- Increased re-use and reduced memory bandwidth needs
  - Non sample-dependent data stored and accessed once per ensemble
- Reduced communication costs
  - Latency of message passing amortized across ensemble
  - Fewer but larger messages

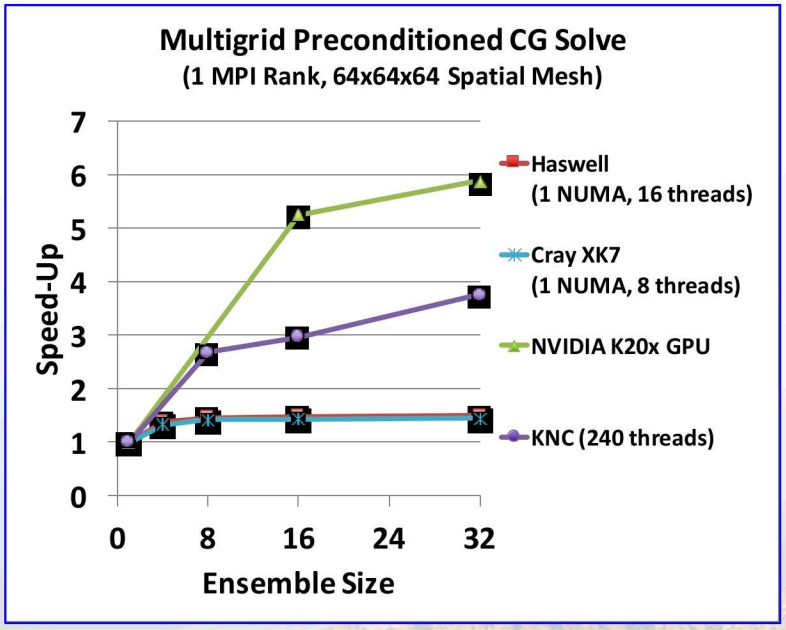
```
// Ensemble scalar type
template <int S>
struct Ensemble {
    double val[S];
    Ensemble(const double& v) { for (int e=0; e<S; ++e) val[e] = v; }
    Ensemble& operator=(const Ensemble& a) {
        for (int e=0; e<S; ++e) val[e] = a.val[e];
        return *this;
    }
    Ensemble& operator+=(const Ensemble& a) {
        for (int e=0; e<S; ++e) val[e] += a.val[e];
        return *this;
    }
    // ...
};

template <int S>
Ensemble<S> operator*(const Ensemble<S>& a, const Ensemble<S>& b) {
    Ensemble<S> c;
    for (int e=0; e<S; ++e) c.val[e] = a.val[e]*b.val[e];
    return c;
}

// ...
```

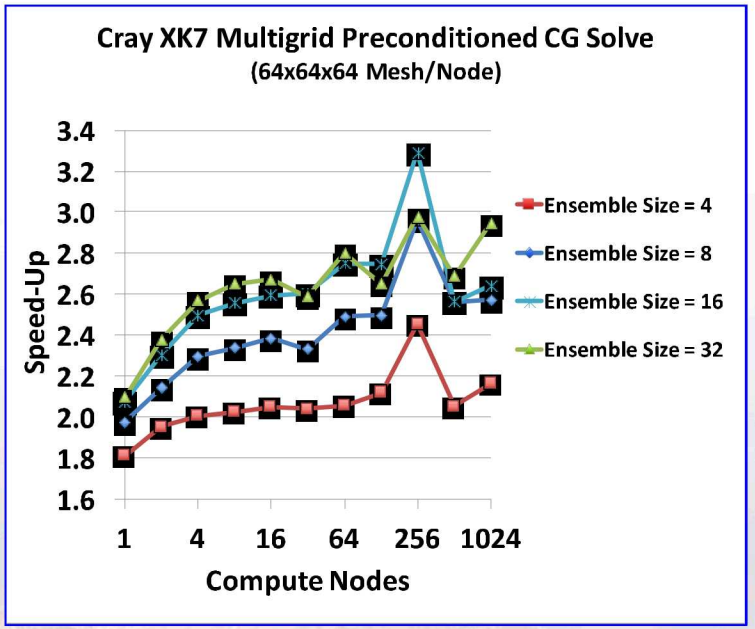
Techniques Prototyped in FENL Mini-App*

- **Simple nonlinear diffusion equation**
  - 3-D, linear FEM discretization
  - 1x1x1 cube, unstructured mesh
  - KL truncation of exponential random field model for diffusion coefficient
  - Trilinos-couplings package

$$-\nabla \cdot (\kappa(x,y)\nabla u) = 0,$$

$$\kappa(x,y) = \kappa_0 + \sigma \sum_{i=1}^{M} \sqrt{\lambda_i}\kappa_i(x)y_i$$





*Phipps, et al, *Embedded Ensemble Propagation for Improving Performance, Portability and Scalability of Uncertainty Quantification on Emerging Computational Architectures*, SISC, 2017

Conclusions

- Templates, operator overloading are a powerful enabling capability for embedded analysis
  - Automated code transformation for moving beyond single-point, forward simulation
- Enables many types of advanced analysis
  - Optimization
  - Sensitivity Analysis
  - Error Estimation
  - UQ
- Kokkos integration provides effective embedded analysis path for next-generation architectures
  - Increase fine-grained parallelism
  - Improved memory access patterns
  - Reduced communication costs

Auxiliary Slides

Polynomial Chaos Expansions (PCE)

- **Steady-state finite dimensional model problem:**

$$\text{Find } u(\xi) \text{ such that } f(u,\xi) = 0,\ \xi : \Omega \to \Gamma \subset R^M,\ \text{density } \rho$$

- **(Global) Polynomial Chaos approximation:**

$$u(\xi) \approx \hat{u}(\xi) = \sum_{i=0}^{P} u_i \psi_i(\xi), \quad \langle \psi_i \psi_j \rangle \equiv \int_\Gamma \psi_i(y)\psi_j(y)\rho(y)dy = \delta_{ij}\langle \psi_i^2 \rangle$$

- **Non-intrusive polynomial chaos (NIPC, NISP):**

$$u_i = \frac{1}{\langle \psi_i^2 \rangle} \int_\Gamma \hat{u}(y)\psi_i(y)\rho(y)dy \approx \frac{1}{\langle \psi_i^2 \rangle} \sum_{k=0}^{Q} w_k u^k \psi_i(y^k), \quad f(u^k, y^k) = 0$$

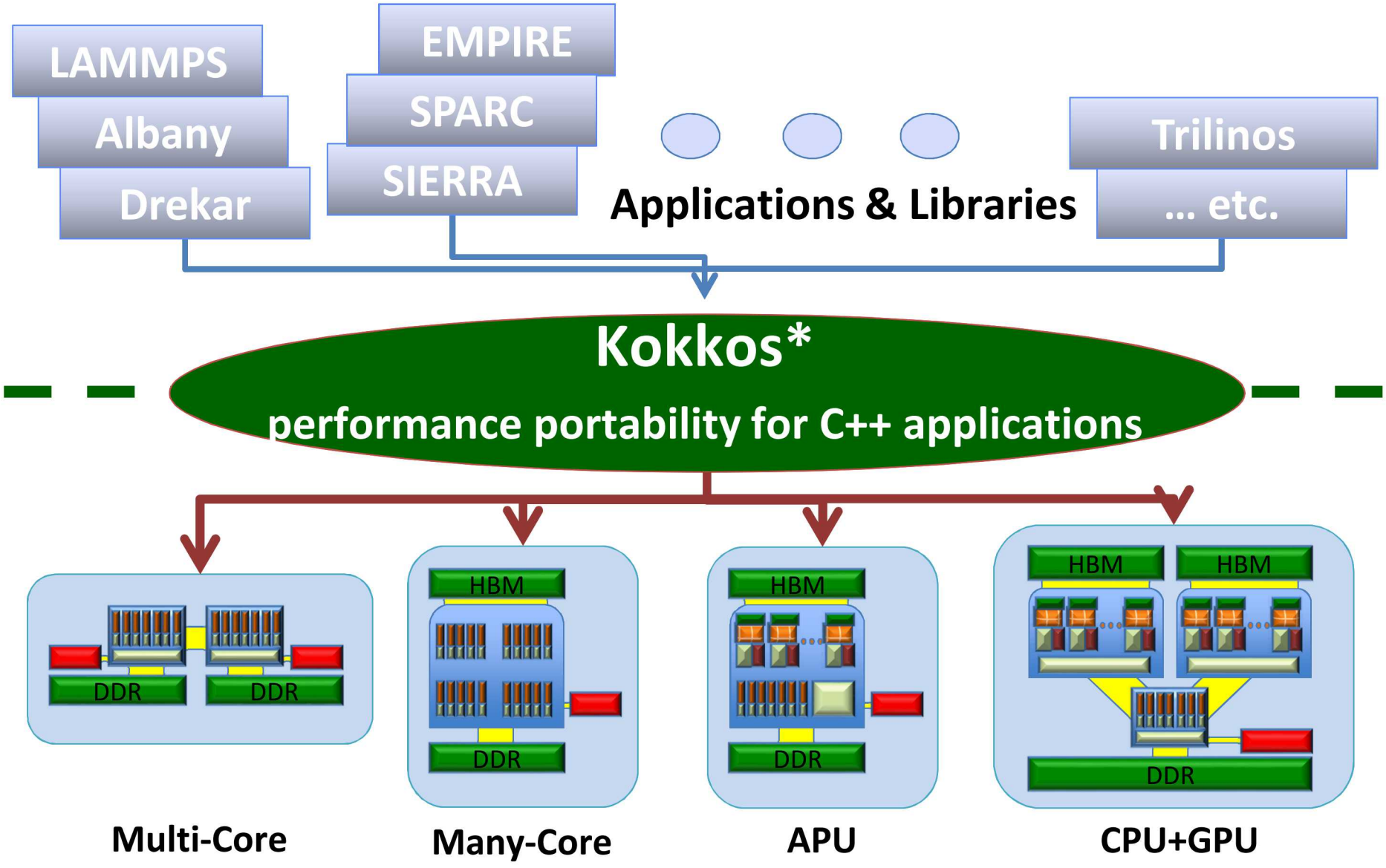
- **Linear regression approach for approximating PCE coefficients**

$$\hat{u}(y_k) = \bar{u}_k \implies \sum_{i=0}^{P} u_i \psi_i(y_k) = \bar{u}_k, \quad f(\bar{u}_k, y_k) = 0, \quad k = 0, \ldots, Q$$

- **Reduce number of samples by adding derivative equations**

$$\sum_{i=0}^{P} u_i \psi_i(y_k) = \bar{u}_k, \quad k = 0, \ldots, \frac{Q}{M+1}$$

$$\sum_{i=0}^{P} u_i \frac{\partial \psi_i}{\partial y}(y_k) = \frac{\partial \bar{u}_k}{\partial y_k}, \quad k = 0, \ldots, \frac{Q}{M+1}$$



κόκκος : "granule" or "grain" ; *like grains of sand on a beach*

*H.C. Edwards et al, https://github.com/kokkos

Kokkos Dense Matrix-Vector Product Example

```
template <typename ViewTypeA, typename ViewTypeB, typename ViewTypeC>
void run_mat_vec(const ViewTypeA& A, const ViewTypeB& b, const ViewTypeC& c) {
  typedef typename ViewTypeC::value_type scalar_type;         // The scalar type
  typedef typename ViewTypeC::execution_space execution_space; // Where we are running

  const int m = A.dimension_0();
  const int n = A.dimension_1();
  Kokkos::parallel_for(
    Kokkos::RangePolicy<execution_space>( 0,m ), // Iterate over [0,m)
    KOKKOS_LAMBDA (const int i) {              // "[=]" (capture by value)
      scalar_type t = 0.0;
      for (int j=0; j<n; ++j)
        t += A(i,j)*b(j);
      c(i) = t;
    }
  );
}

// Use default execution space (OpenMP, Cuda, ...) and memory layout for that space
Kokkos::View<double**> A("A",m,n); // Create rank-2 array with m rows and n columns
Kokkos::View<double* > b("b",n);   // Create rank-1 array with n rows
Kokkos::View<double* > c("c",m);   // Create rank-1 array with m rows

// ...

run_mat_vec(A,b,c);
```

Performance Portability

Architecture	Description	Execution Space	Measured Bandwidth (GB/s)*	Expected Throughput (GFLOP/s)	Measured Throughput (GFLOP/s)	Wrong Layout (GFLOP/s)
Haswell (1 socket)	Intel Xeon E5-2698 v3, 32 threads	OpenMP	47.4	11.9	13.0	6.8
MIC	Intel Xeon Phi 7120P, 240 threads	OpenMP	147	36.8	35.9	3.4
GPU	NVIDIA K80	Cuda	150	37.5	42.0	7.4

- **m = 1e6, n=100**
- **Expected Throughput = Measured Bandwidth x 2 FLOPS / 8 Bytes**
- **Manually specify incorrect layout for "Wrong Layout", e.g.,**

```
Kokkos::View<double**, Kokkos::LayoutRight, Kokkos::Cuda> A("A",m,n);
Kokkos::View<double* , Kokkos::LayoutRight, Kokkos::Cuda> b("b",n);
Kokkos::View<double* , Kokkos::LayoutRight, Kokkos::Cuda> c("c",m);
```

* Bandwidth measured through STREAM (Triad) benchmark

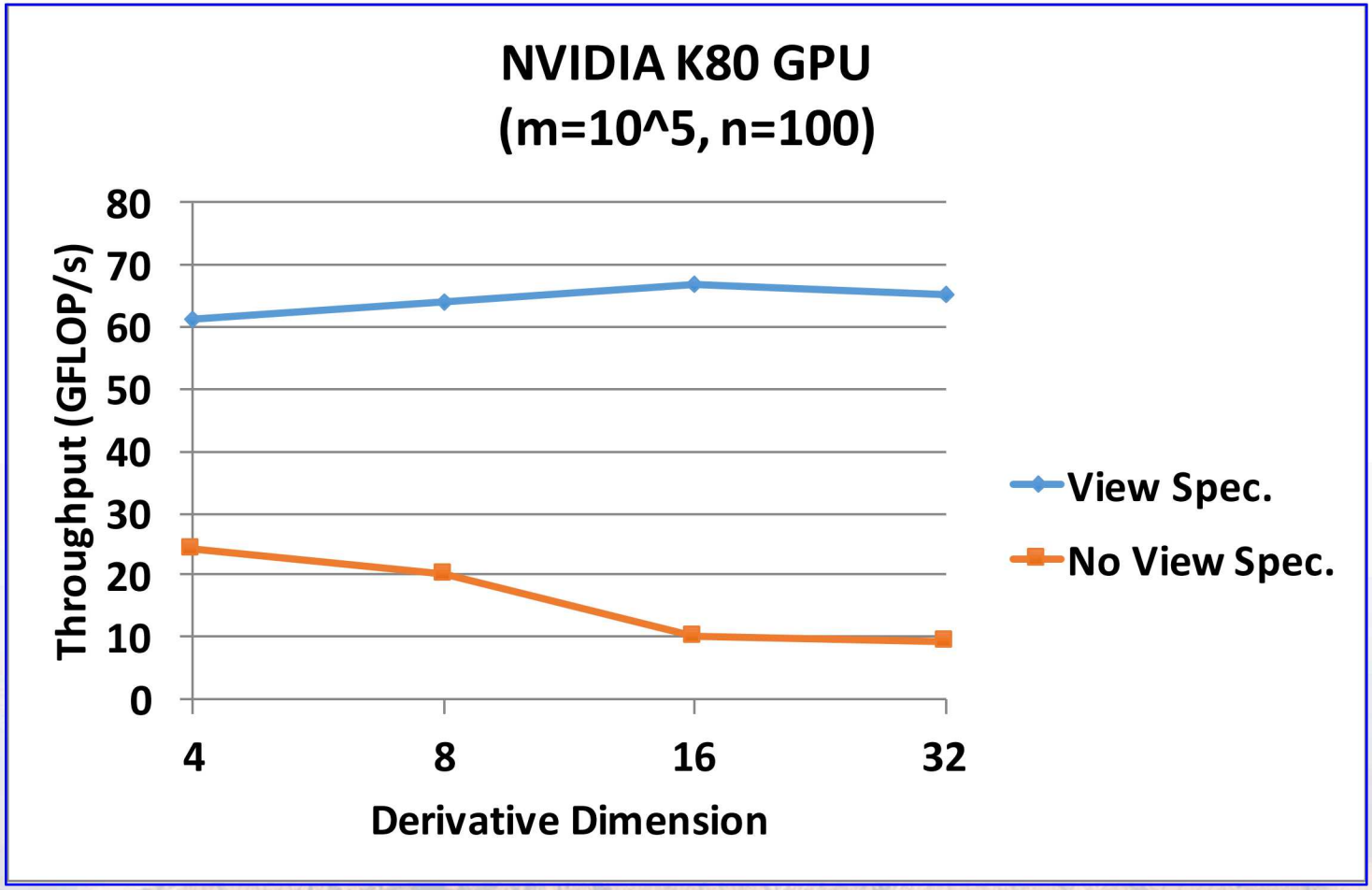
AD Performance Portability

```
Kokkos::View<Sacado::Fad::SFad<double,p>**> A("A",m,n,p+1);  // Create rank-2 array with m rows and n columns
Kokkos::View<Sacado::Fad::SFad<double,p>* > b("b",n,p+1);    // Create rank-1 array with n rows
Kokkos::View<Sacado::Fad::SFad<double,p>* > c("c",m,p+1);    // Create rank-1 array with m rows

// ...

run_mat_vec(A,b,c);
```

Architecture	Measured Bandwidth (GB/s)	Expected Throughput (GFLOP/s)	Measured Throughput (GFLOP/s)	No View Specialization (GFLOP/s)
Haswell	47.4	22.4	24.3	23.1
MIC	147	69.4	69.4	43.2
GPU	150	70.8	81.2	35.1

- m = 1e6, n=100, p = 8 (derivative dimension)
- Expected Throughput ~ Measured Bandwidth x (4p+2) FLOPS / 8(p+1) Bytes
- SFad<double,p> AD data type

Throughput Varying Derivative Dimension



Embedded Transient Sensitivity Analysis

$$f(\dot{u}, u, p, t) = 0 \xrightarrow{t.d.} F(u_{n+1}, u_n, p, t_n, \Delta t_n) = 0$$

$$\downarrow \frac{\partial}{\partial p}$$

$$Z \equiv \frac{\partial u}{\partial p}, \qquad \frac{\partial F}{\partial u_{n+1}} Z_{n+1} + \frac{\partial F}{\partial u_n} Z_n + \frac{\partial F}{\partial p} = 0$$

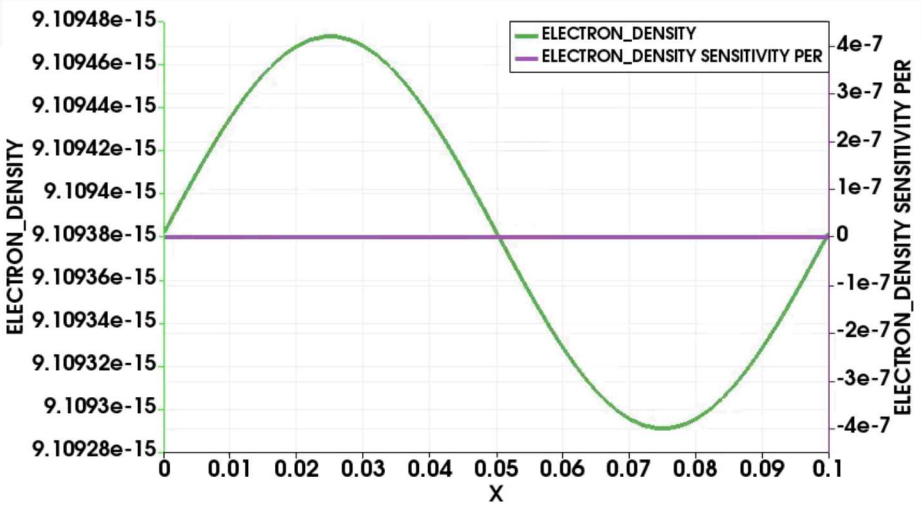
- Sensitivity equations define additional (linear) equations that must be solved after each time step
  - Treat as general nonlinear problem to allow for inexact state Jacobian and inexact linear solves
- Compute sensitivity (tangent) residual by
  - Templating physics residual code on scalar type
  - Instantiating templated residual on Sacado forward mode AD data types to compute tangent

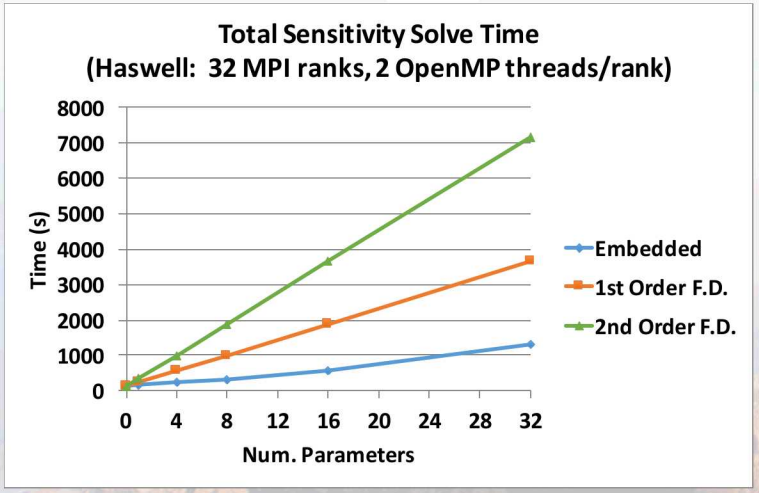
$$\begin{bmatrix} \frac{\partial F}{\partial u_{n+1}} & \frac{\partial F}{\partial u_n} & \frac{\partial F}{\partial p} \end{bmatrix} \begin{bmatrix} Z_{n+1} \\ Z_n \\ I \end{bmatrix} = \frac{\partial F}{\partial u_{n+1}} Z_{n+1} + \frac{\partial F}{\partial u_n} Z_n + \frac{\partial F}{\partial p}$$

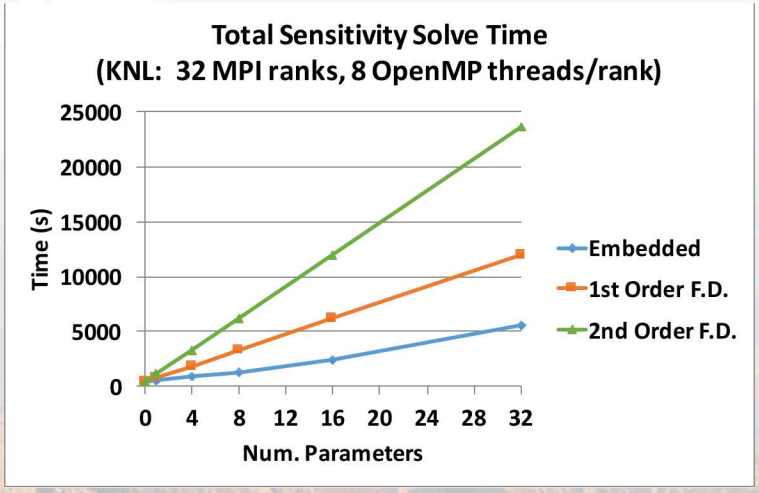
- Augment transient/pseudo-transient time stepping to solve sensitivity equations using tangent residual computed by Sacado

Collisionless, Unmagnetized Electron Plasma

- EMPIRE ATDM application code
  - M. Bettencourt et al
  - Unstructured, finite element
- 1-D verification problem
  - Sensitivity w.r.t. permittivity

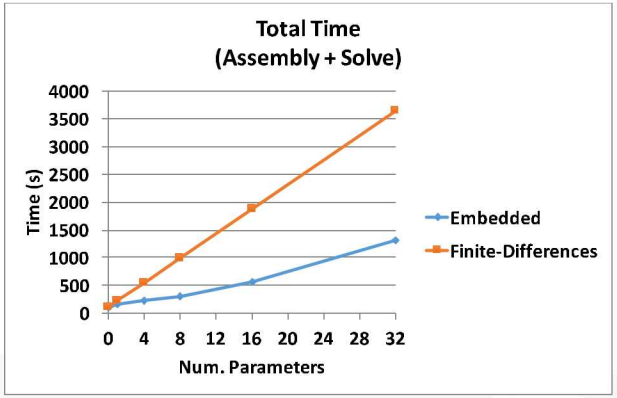


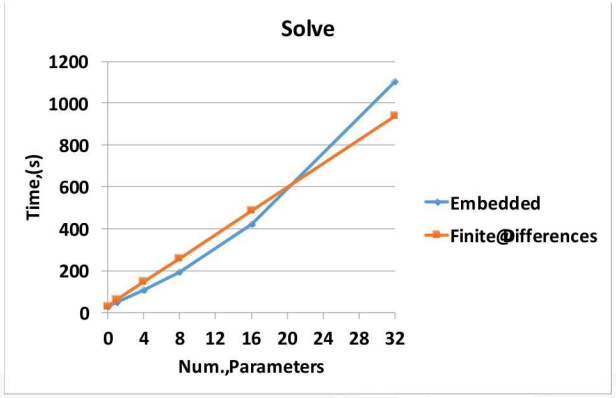




EMPIRE Sensitivity Performance

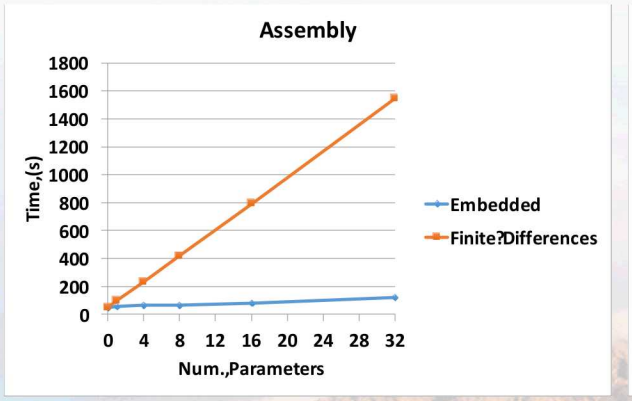
- Collisionless, unmagnetized electron plasma verification problem (3D)
- 1st order finite differences used as benchmark
- Haswell architecture (32 MPI ranks, 2 OpenMP threads/rank)

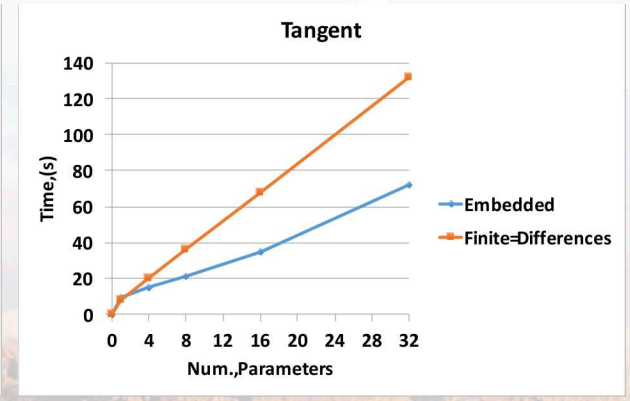


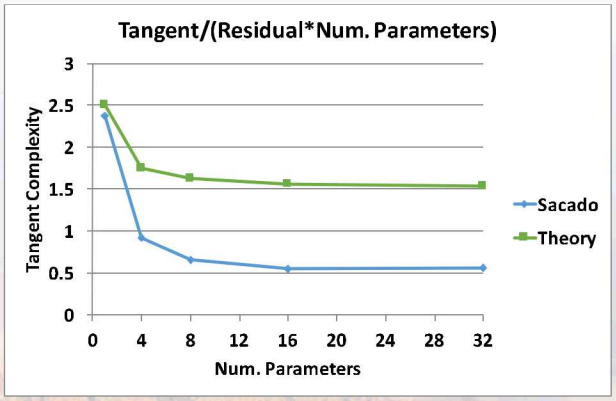


Theory:

$$time(Tangent) \sim \left(1 + \frac{3}{2}m\right) time(Residual)$$

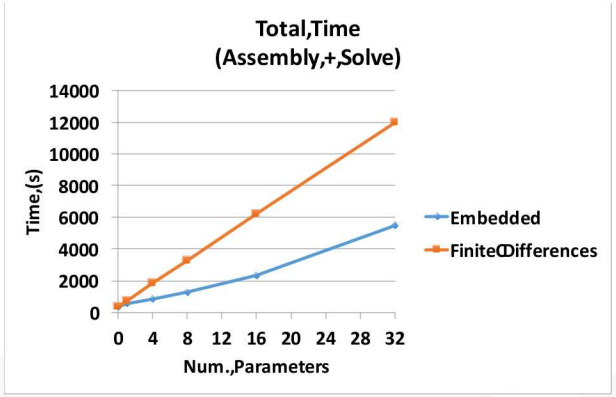


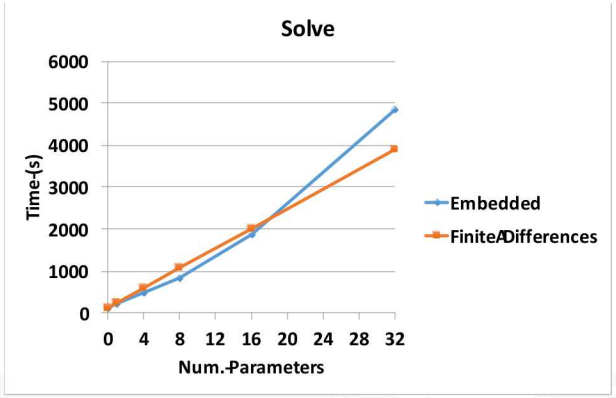




EMPIRE Sensitivity Performance

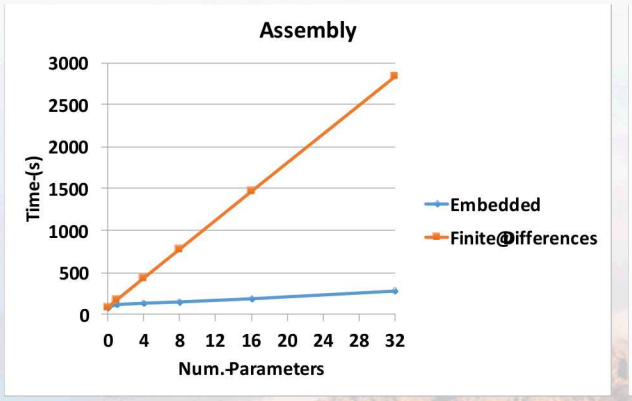
- Collisionless, unmagnetized electron plasma verification problem (3D)
- 1st order finite differences used as benchmark
- KNL architecture (32 MPI ranks, 8 OpenMP threads/rank)

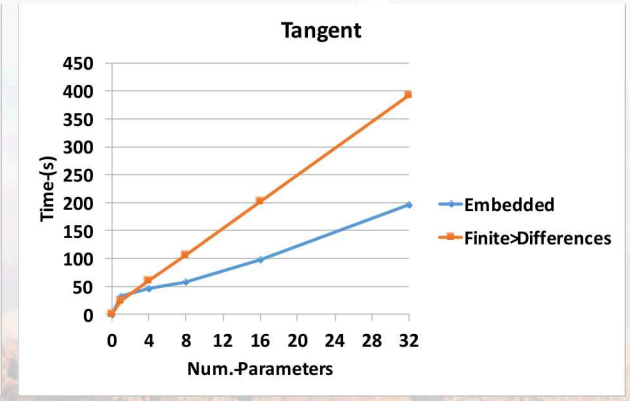


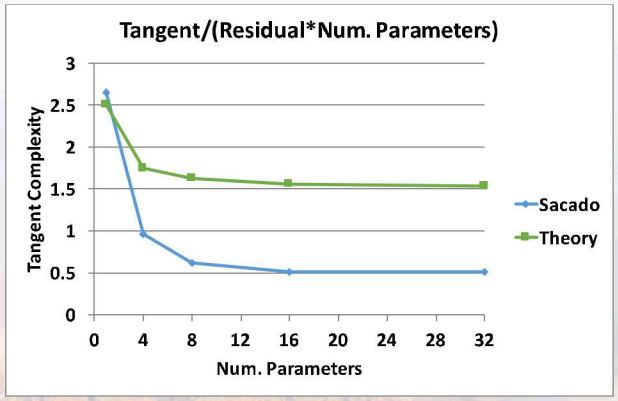


Theory:

$$time(Tangent) \sim \left(1 + \frac{3}{2}m\right) time(Residual)$$

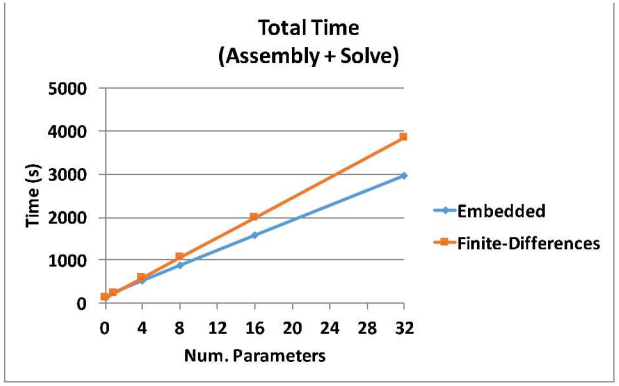


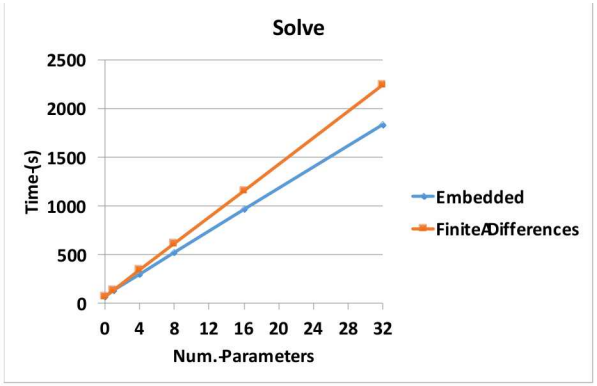




SPARC Sensitivity Performance

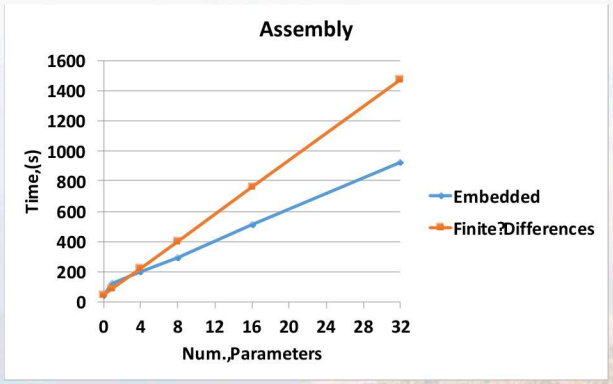
- Generic RV problem
- Haswell architecture (32 MPI ranks, 2 OpenMP threads/rank)

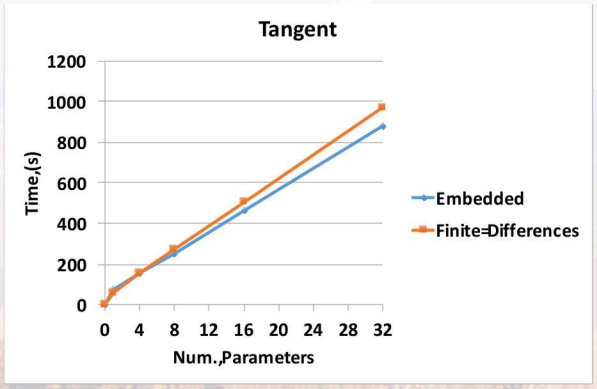


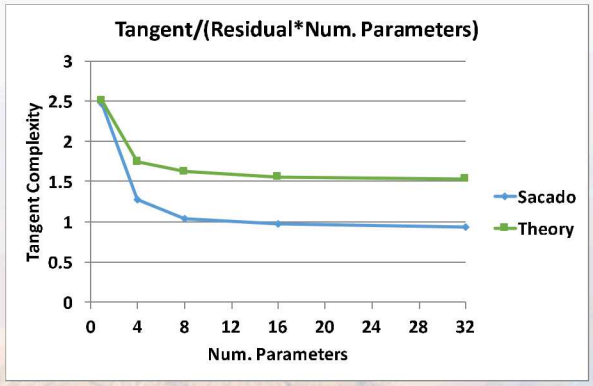


Theory:

$$time(Tangent) \sim \left(1 + \frac{3}{2}m\right) time(Residual)$$

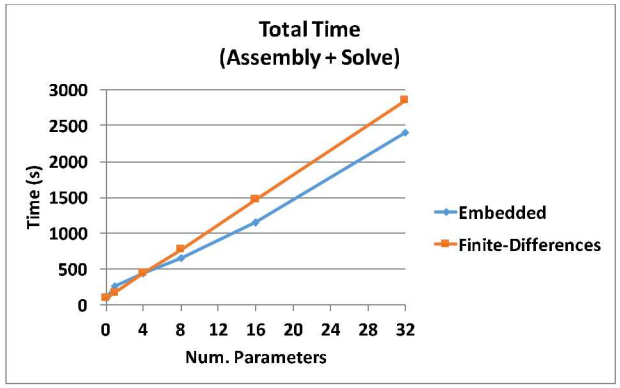


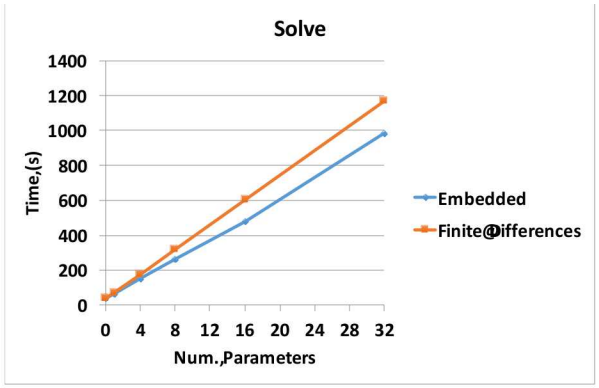




SPARC Sensitivity Performance

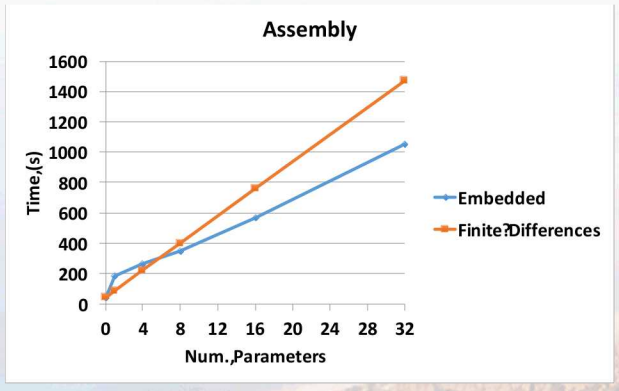
- Generic RV problem
- KNL architecture (32 MPI ranks, 8 OpenMP threads/rank)

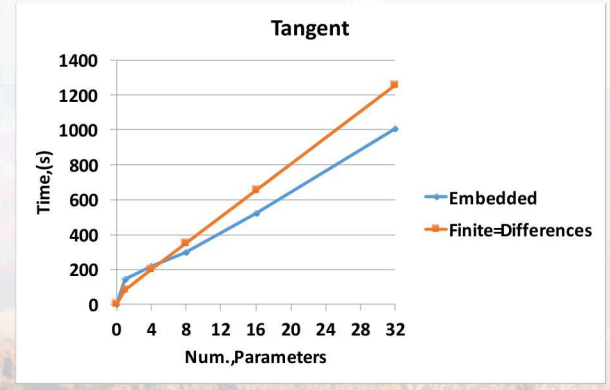


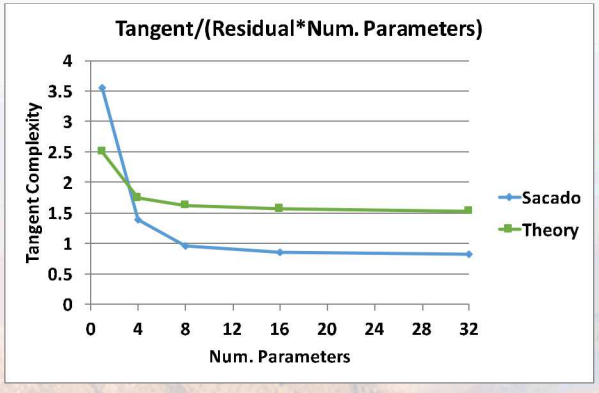


Theory:

$$time(Tangent) \sim \left(1 + \frac{3}{2}m\right) time(Residual)$$







Speedup in EMPIRE Assembly Kernels