Center for Computing Research

Automatic Differentiation of C++ Codes on Emerging Manycore Architectures with Sacado

Eric Phipps (etphipp@sandia.gov), Roger Pawlowski, and Christian Trott,
Sandia National Laboratories
Albuquerque New Mexico USA

EuroTUG 2022

September 12-14, 2022

Sandia National Laboratories

Exceptional service in the national interest

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

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.
SAND2020-8010 C

Why Derivatives?

- Derivatives are a very useful tool for computational simulation
  - Jacobians for implicit time-stepping, steady-state solves
  - Adjoints for error estimation, optimization
  - Parameter sensitivities for sensitivity analysis, stability analysis, optimization, UQ
- Derivatives can always be derived and coded by-hand
  - Time consuming and 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

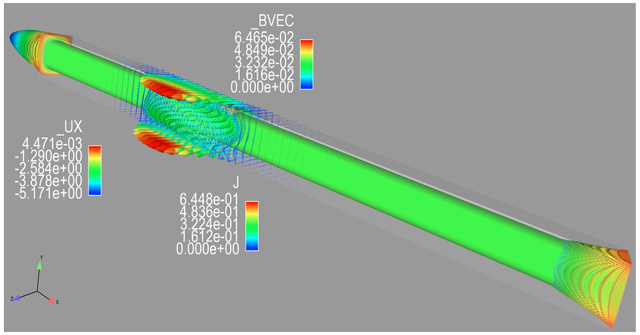
$$y = \sin(e^x + x \log x), \quad x = 2$$

	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

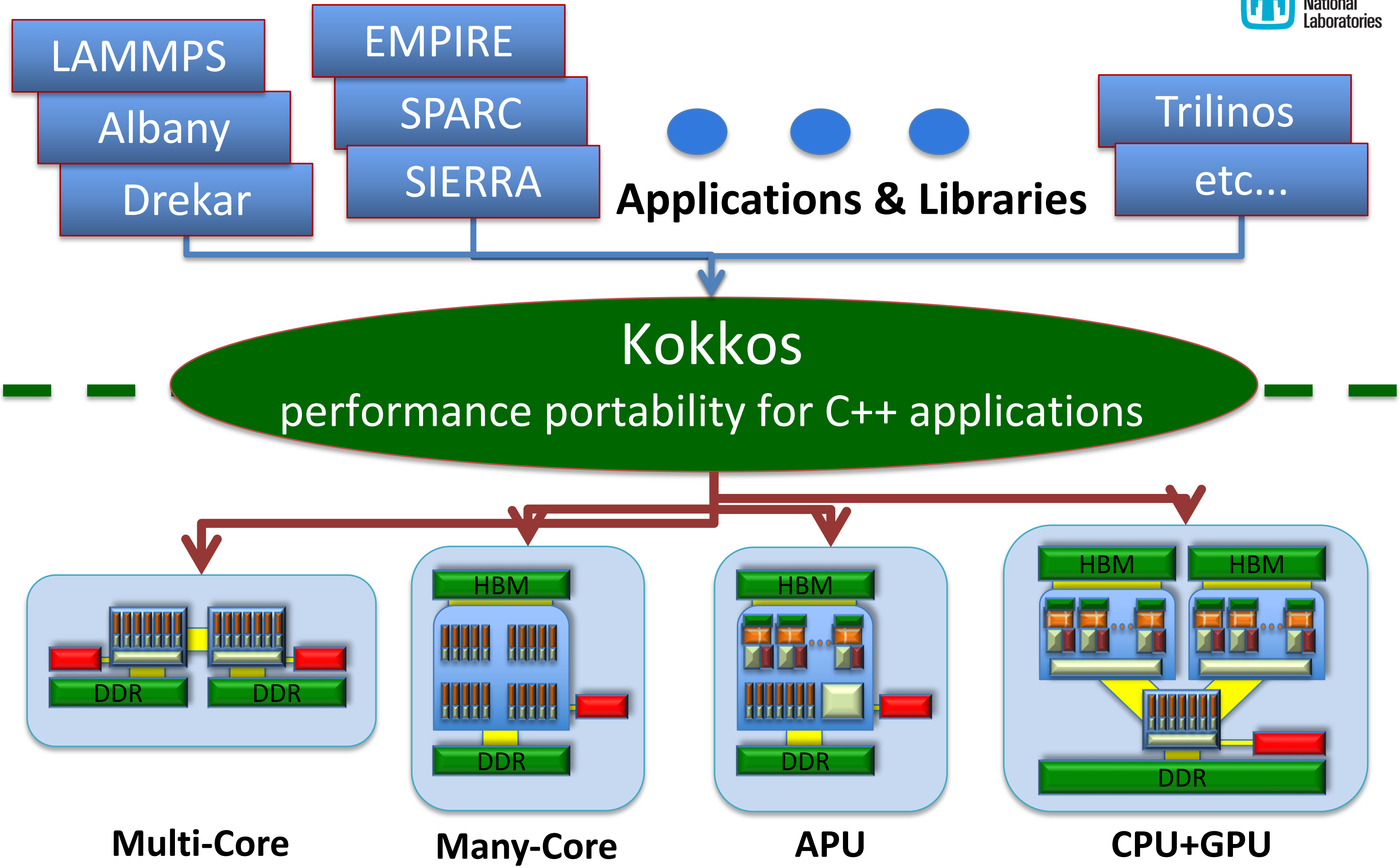
Sacado: AD Tools for C++ Applications

- Package in Trilinos
  - http://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
- Integrates with Kokkos for efficient differentiation of thread-parallel programs

http://github.com/trilinos



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



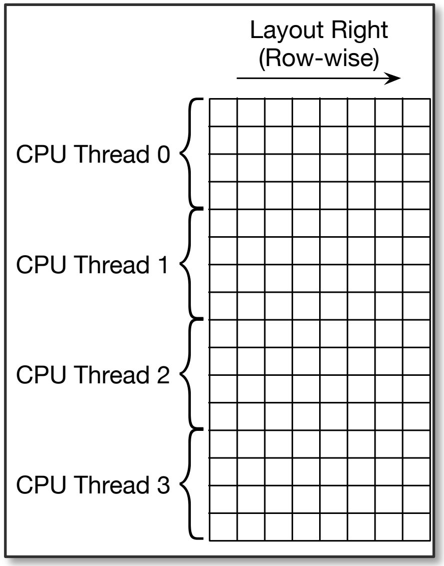
C. Trott, *et al.*, https://github.com/kokkos, https://kokkosteam.slack.com

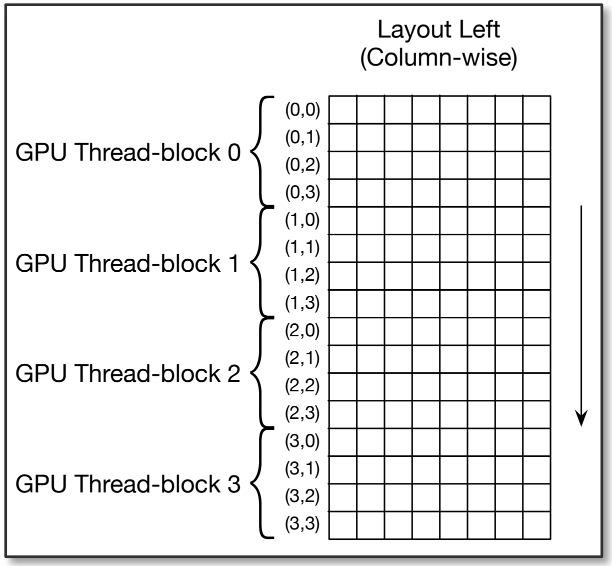
Kokkos Example

```
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.extent(0);
  const int n = A.extent(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);
```

Layout Polymorphism for Performant Memory Accesses

- CPU
  - 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)





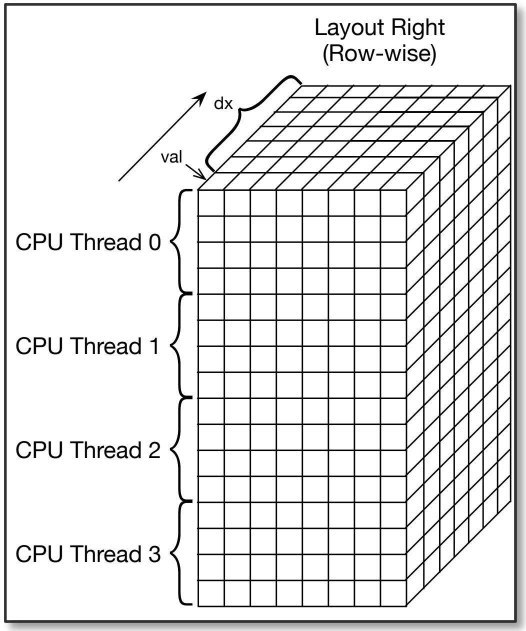
$M = 10^6$, $n = 100$

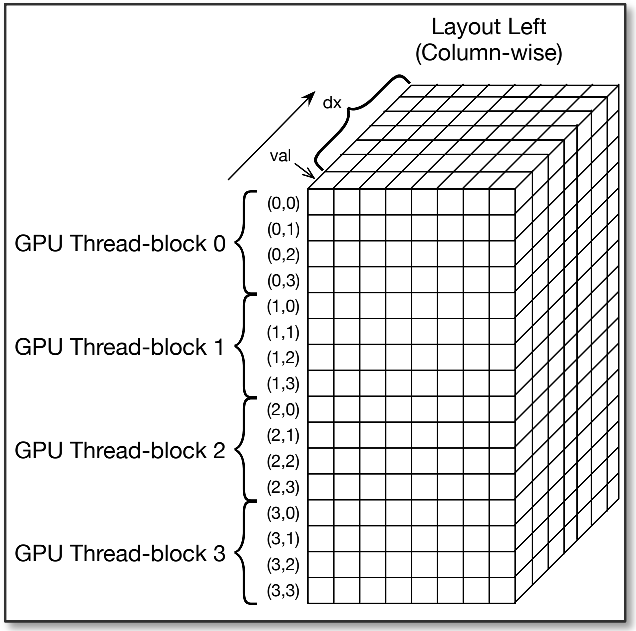
Architecture	Description	Execution Space	Measured Bandwidth (GB/s)	Expected Throughput (GFLOP/s)	Measured Throughput (GFLOP/s)	Wrong Layout (GFLOP/s)
Skylake (1 socket)	Intel Xeon Gold 6154, 36 threads	OpenMP	64.4	16.1	18.0	15.3
GPU	NVIDIA V100	Cuda	833	208	213	26.3

Sacado and Kokkos?

- What happens when we use Sacado AD on manycore architectures with Kokkos?

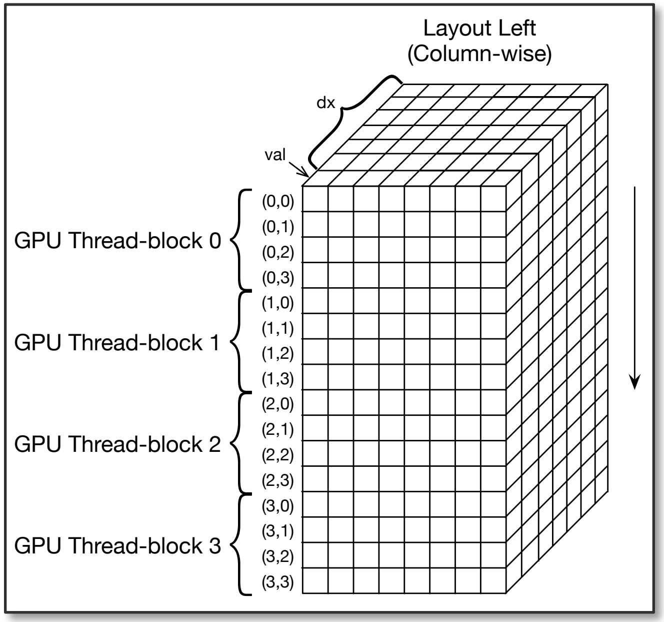
- Kokkos::View< Sacado::Fad::SFad<double,p>**>:
  - Derivative components always stored consecutively
  - CPU: Good cache, vector performance
  - GPU: Large stride causes bad coalescing





Sacado/Kokkos Integration

- Want good AD performance with no modifications to Kokkos kernels
- Achieved by specializing Kokkos::View data structure for Sacado scalar types
  - Rank-r Kokkos::View internally stored as a rank-(r+1) array of double
  - Kokkos layout applied to internal rank-(r+1) array

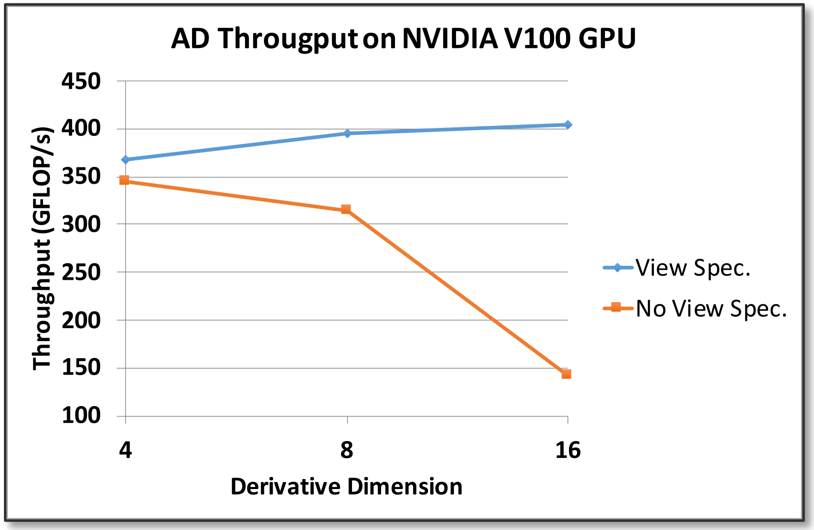


AD Performance Portability

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

// ...

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

SFad, Derivative dimension p=8

Architecture	Expected Throughput (GFLOP/s)	Measured Throughput (GFLOP/s)	No View Specialization (GFLOP/s)
Skylake	30.4	34.1	34.0
GPU	393	395	317



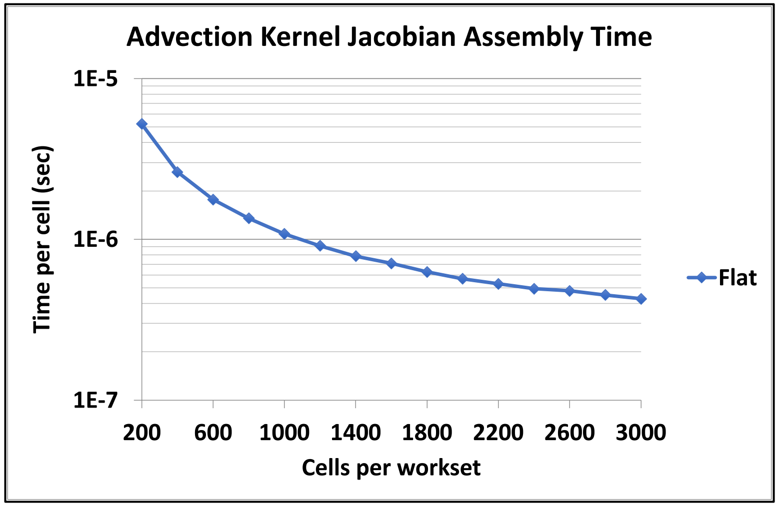
Hierarchical Parallelism

- Layout approach was explored to minimize code user-code changes for Sacado
  - Differentiate code without changing parallel scheduling
- Derivative propagation provides good opportunities for exposing more parallelism
  - Parallelism across derivative array
  - Code may not expose enough parallelism natively (e.g., small workset)
- Motivation is PDE assembly using worksets
  - Many codes group mesh cells into batches called worksets
  - Threaded parallelism over cells in each workset: want large worksets for GPUs with very high concurrency
  - Memory required proportional to size of workset: want small worksets because of limited high-bandwidth memory on GPUs
- Solution: apply fine-grained (warp-level) parallelism across derivative dimension on GPUs
  - Implementation uses Cuda code hidden behind Sacado's overloaded operators

Advection Kernel Example

$$r = \int_e \left(\vec{f}(x) \cdot \nabla \varphi(x) + s(x)\varphi(x) \right) dx$$

```
Kokkos::View<ScalarT****, Layout, ExecSpace> wgb;
Kokkos::View<ScalarT***,  Layout, ExecSpace> flux;
Kokkos::View<ScalarT***,  Layout, ExecSpace> wbs;
Kokkos::View<ScalarT**,   Layout, ExecSpace> src;
Kokkos::View<ScalarT**,   Layout, ExecSpace> residual;
```

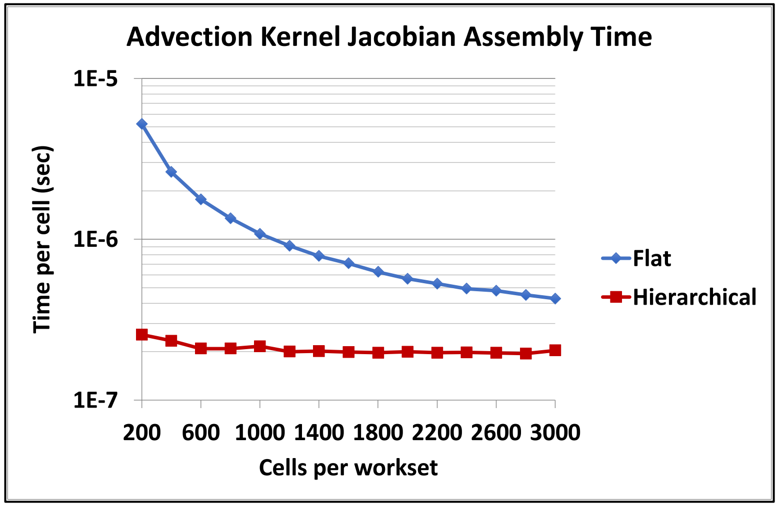



```
typedef Kokkos::RangePolicy<ExecSpace> Policy;
Kokkos::parallel_for(
  Policy( 0,num_cell ),
  KOKKOS_LAMBDA( const int cell )
  {


    for (int basis=0; basis<num_basis; basis+=1) {
      ScalarT value(0),value2(0);
      for (int qp=0; qp<num_points; ++qp) {
          for (int dim=0; dim<num_dim; ++dim)
            value += flux(cell,qp,dim)*wgb(cell,basis,qp,dim);
          value2 += src(cell,qp)*wbs(cell,basis,qp);
      }
      residual(cell,basis) = value+value2;
    }
  });
```

Advection Kernel Example

$$r = \int_e \left(\vec{f}(x) \cdot \nabla\varphi(x) + s(x)\varphi(x) \right) dx$$

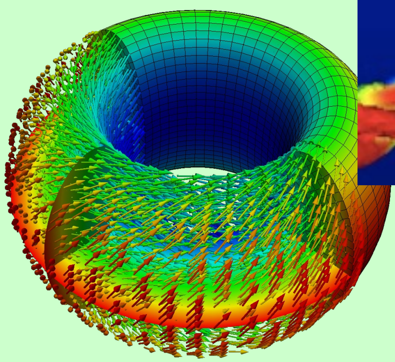
```
const int VectorSize = 32;
typedef Kokkos::LayoutContiguous<Layout,VectorSize> ContLayout;
Kokkos::View<ScalarT****, ContLayout, ExecSpace> wgb;
Kokkos::View<ScalarT***,  ContLayout, ExecSpace> flux;
Kokkos::View<ScalarT***,  ContLayout, ExecSpace> wbs;
Kokkos::View<ScalarT**,   ContLayout, ExecSpace> src;
Kokkos::View<ScalarT**,   ContLayout, ExecSpace> residual;

typedef typename ThreadLocalScalarType<decltype(src)>::type
  local_scalar_type;
typedef Kokkos::TeamPolicy<ExecSpace> Policy;
Kokkos::parallel_for(
  Policy( num_cell, Kokkos::AUTO, VectorSize ),
  KOKKOS_LAMBDA( const typename Policy::member_type& team )
  {
    const int cell = team.league_index();
    const int ti   = team.team_index();
    const int ts   = team.team_size();
    for (int basis=ti; basis<num_basis; basis+=ts) {
      local_scalar_type value(0),value2(0);
      for (int qp=0; qp<num_points; ++qp) {
          for (int dim=0; dim<num_dim; ++dim)
            value += flux(cell,qp,dim)*wgb(cell,basis,qp,dim);
          value2 += src(cell,qp)*wbs(cell,basis,qp);
      }
      residual(cell,basis) = value+value2;
    }
  });
```

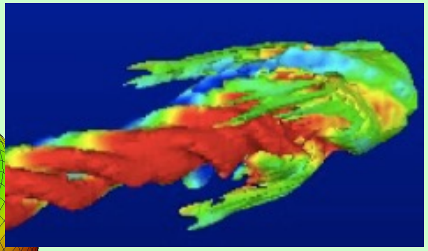


Drekar/Panzer PDE Tools

(Pawlowski, Cyr, Shadid, Smith)

Applications

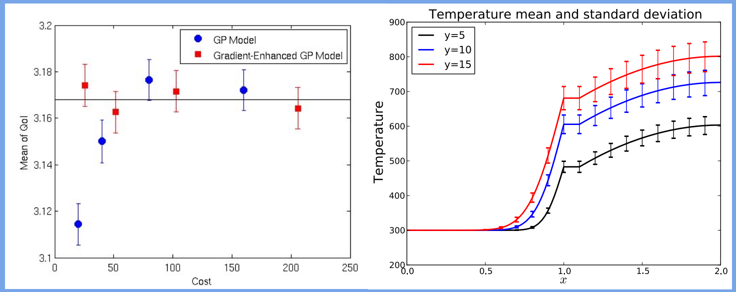




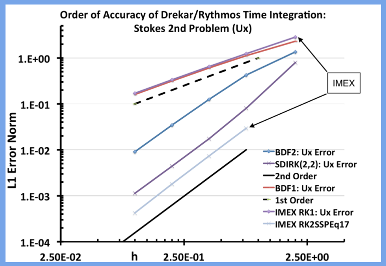
Turbulent CFD

Magnetohydrodynamics

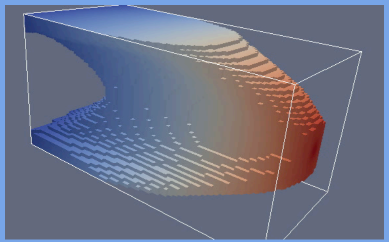
Discretizations & Algorithms



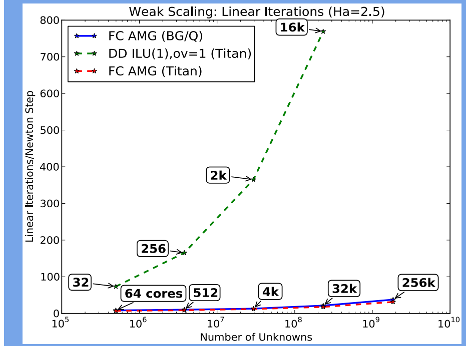
Uncertainty Quantification



IMEX



PDE Constrained Optimization

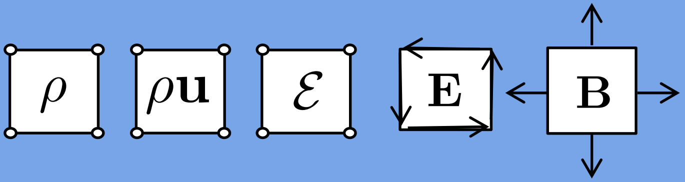


Algebraic Multigrid (>100k cores)

$$\mathcal{A} = \begin{bmatrix} I & \\ BF^{-1} & I \end{bmatrix} \begin{bmatrix} F & B^T \\ & S \end{bmatrix}$$

$$S = C - BF^{-1}B^T$$

Block Preconditioning



Compatible Discretizations

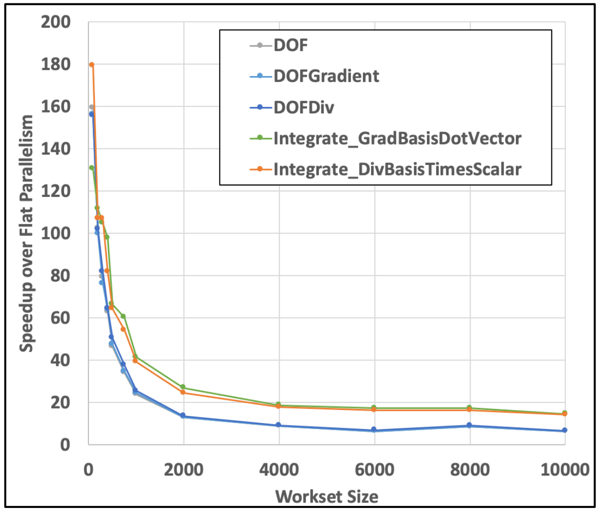
Hierarchical Parallelism in Panzer

Sandia National Laboratories

- Diffusion problem with mixed finite element discretization:

$$\left.\begin{aligned}\nabla^2\phi = f \quad &\text{on } \Omega\\ \phi = \phi_\Gamma \quad &\text{on } \Gamma = \partial\Omega\end{aligned}\right\} \implies \left\{\begin{aligned}&\int_\Omega (\nabla\cdot \mathrm{g} - f)(\nabla\cdot \mathrm{w})d\Omega = 0 \;\; \forall \mathrm{w} \in \mathcal{H}_{(\nabla\cdot)}\\ &\int_\Omega (\nabla\phi - \mathrm{g})\cdot(\nabla q)d\Omega = 0 \;\; \forall q \in \mathcal{H}_{(\nabla)}\end{aligned}\right.$$

Description	Operator	Panzer C++ Class Name
1. Evaluate $\mathbf{g}$ at Quadrature Points	$\mathbf{g} = \sum_i g_i \mathbf{w}_i$	DOF
2. Evaluate $\nabla\phi$ at Quadrature Points	$\nabla\phi = \sum_i \phi_i \nabla q_i$	DOFGradient
3. Evaluate $\nabla\cdot\mathbf{g}$ at Quadrature Points	$\nabla\cdot\mathbf{g} = \sum_i g_i \nabla\cdot\mathbf{w}_i$	DOFDiv
4. Integrate Eq. 6 with $\mathbf{h} = \nabla\phi - \mathbf{g}$	$\int_\Omega(\mathbf{h})\cdot(\nabla q)d\Omega$	Integrate_GradBasisDotVector
5. Integrate Eq. 5 with $s = \nabla\cdot\mathbf{g} - f$	$\int_\Omega(s)(\nabla\cdot\mathbf{w})d\Omega$	Integrate_DivBasisTimesScalar



Concluding Remarks

- Derivatives are an important ingredient in scientific computing
  - AD is a powerful technique for computing derivatives accurately & efficiently
- Sacado provides efficient AD capabilities for C++ codes
  - https://github.com/trilinos/Trilinos
- Sacado integrates with Kokkos for portable and thread-scalable differentiation of shared-memory parallel computations
  - Leverage layout polymorphism to enable AD of Kokkos kernels without modification
  - Incorporate GPU vector/warp-level parallelism for improved performance
- Sacado+Kokkos impacting numerous projects at Sandia
  - Albany (https://github.com/SNLComputation/Albany)
  - Panzer/Drekar
  - ECP/ATDM
- Future work:
  - Higher derivatives (Kokkos specializations for nested Sacado AD types)
  - Reverse mode with Kokkos for scalable adjoint computations

Auxiliary Slides

Some Negative Implications

- View access operator returns AD *handle* object (pointing into rank-(r+1)array)
  - `View<SFad<double,p>**>::operator(i,j)` returns `ViewFad<double>(ptr+offset(i,j), stride, p)` temporary
  - Not the same as `SFad<double,p>&`
  - Can't take address of return value
- View constructor needs AD derivative dimension (
  - Needed to properly allocate internal array
  - `View<SFad<double,p>**>(m,n,p)`
- Introduces challenges for
  - Templating on scalar type: `View<T**>` operates differently depending on type of T
  - Porting codes to use Kokkos: Can't get pointer of type `T*` to pass to legacy code
- Unclear how to efficiently extend this to nested Fad objects for higher derivatives

AD Takes Three Basic Forms

$$x \in \mathbb{R}^n,\ f : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

- Forward Mode:
  $$(x,\ V) \longrightarrow \left(f,\ \frac{\partial f}{\partial x}V\right)$$
  - Propagate derivatives of intermediate variables w.r.t. independent variables forward
  - Directional derivatives, tangent vectors, square Jacobians, $\partial f/\partial x$ when $m \geq n$
- Reverse Mode:
  $$(x,\ W) \longrightarrow \left(f,\ W^T\frac{\partial f}{\partial x}\right)$$
  - Propagate derivatives of dependent variables w.r.t. intermediate variables backwards
  - Gradient of a scalar value function with complexity $\approx 4\ \mathrm{ops}(f)$
  - Gradients, Jacobian-transpose products (adjoints), $\partial f/\partial x$ when $n > m$
- Taylor polynomial mode:
  $$x(t) = \sum_{k=0}^{d} x_k t^k \longrightarrow \sum_{k=0}^{d} f_k t^k = f(x(t)) + O(t^{d+1}),\quad f_k = \frac{1}{k!}\frac{d^k}{dt^k}f(x(t))$$
- Basic modes combined for higher derivatives: $\frac{\partial}{\partial x}\left(\frac{\partial f}{\partial x}V_1\right)V_2,\quad W^T\frac{\partial^2 f}{\partial x^2}V,\quad \frac{\partial f_k}{\partial x_0}$

Differentiating Element-Based Codes

- Global residual computation (ignoring boundary computations):

$$f(x) = \sum_{i=1}^{N} Q_i^T e_{k_i}(P_i x)$$

- Jacobian computation:

$$\frac{\partial f}{\partial x} = \sum_{i=1}^{N} Q_i^T J_{k_i} P_i, \quad J_{k_i} = \frac{\partial e_{k_i}}{\partial x_i}, \quad x_i = P_i x$$

- Jacobian-transpose product computation:

$$w^T \frac{\partial f}{\partial x} = \sum_{i=1}^{N} (Q_i w)^T J_{k_i} P_i$$

- Hybrid symbolic/AD procedure
  - Element-level derivatives computed via AD
  - Exactly the same as how you would do this "manually"
  - Avoids parallelization issues