SAND2020-13005PE

Center for Computing Research

Automatic Differentiation of C++ Codes with Sacado

Eric Phipps (etphipp@sandia.gov)
Sandia National Laboratories
Albuquerque New Mexico USA

BYU Spline-based Finite Element Analysis Class

Dec. 9, 2020

Sandia National Laboratories

Exceptional service in the national interest

U.S. DEPARTMENT OF ENERGY

National Nuclear Security Administration

Outline

- Overview of AD techniques/theory
- AD software
- Sacado: AD tools for C++ codes
- Brief overview of using Sacado
- Selected performance results
- AD on multicore/manycore architectures using Kokkos

Analytic Derivatives Enable Robust Simulation and Design Capabilities

Sandia National Laboratories

- Analytic first & higher derivatives are useful for predictive simulations
  - Computational design, optimization and parameter estimation
  - Stability analysis
  - Uncertainty quantification
  - Verification and validation

- Analytic derivatives improve robustness and efficiency

- Infeasible to expect application developers to code analytic derivatives
  - Time consuming, error prone, and difficult to verify
  - Thousands of possible parameters in a large code
  - Developers must understand what derivatives are needed

- Automatic differentiation solves these problems

What Is Automatic Differentiation (AD) ?

- Analytic derivatives without hand-coding
- All differentiable computations are composition of simple operations
  - sin(), log(), +, *, /, etc…
- We know the derivatives of these simple operations
- We have the chain rule from calculus
- Systematic application of the chain rule through your computation differentiating each statement line-by-line.

A Simple Example

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

		x	$\frac{d}{dx}$
$x \leftarrow 2$		2.000	1.000
$t_1 \leftarrow e^x$		7.389	7.389
$t_2 \leftarrow \log x$		0.693	0.500
$t_3 \leftarrow x t_2$		1.386	1.693
$t_4 \leftarrow t_1 + t_3$		8.775	9.082
$y \leftarrow \sin t_4$		0.605	-7.233

Analytic derivative evaluated to machine precision

Related Methods

Sandia National Laboratories

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

Automatic Differentiation

$$
\begin{aligned}
x &\leftarrow 2 & \frac{dx}{dx} &\leftarrow 1 \\
t_1 &\leftarrow e^x & \frac{dt_1}{dx} &\leftarrow t_1 \frac{dx}{dx} \\
t_2 &\leftarrow \log x & \frac{dt_2}{dx} &\leftarrow \frac{1}{x}\frac{dx}{dx} \\
t_3 &\leftarrow x t_2 & \frac{dt_3}{dx} &\leftarrow t_2 \frac{dx}{dx} + x\frac{dt_2}{dx} \\
t_4 &\leftarrow t_1 + t_3 & \frac{dt_4}{dx} &\leftarrow \frac{dt_1}{dx} + \frac{dt_3}{dx} \\
y &\leftarrow \sin t_4 & \frac{dy}{dx} &\leftarrow \cos(t_4)\frac{dt_4}{dx}
\end{aligned}
$$

$$\frac{dy}{dx} = -7.233\ 340\ 400\ 802\ 3158$$

Symbolic Differentiation

$$\frac{dy}{dx} = \cos(e^x + x \log x) \cdot (e^x + \log x + 1)$$

$$
\begin{aligned}
x &\leftarrow 2 \\
t_1 &\leftarrow e^x \\
t_2 &\leftarrow \log x \\
t_3 &\leftarrow x t_2 \\
t_4 &\leftarrow t_1 + t_3 \\
y &\leftarrow \sin t_4
\end{aligned}
$$

$$
\begin{aligned}
s_1 &\leftarrow \cos t_4 \\
s_2 &\leftarrow t_1 + t_2 \\
s_3 &\leftarrow s_2 + 1 \\
\frac{dy}{dx} &\leftarrow s_1 s_3
\end{aligned}
$$

$$\frac{dy}{dx} = -7.233\ 340\ 400\ 802\ 3167$$

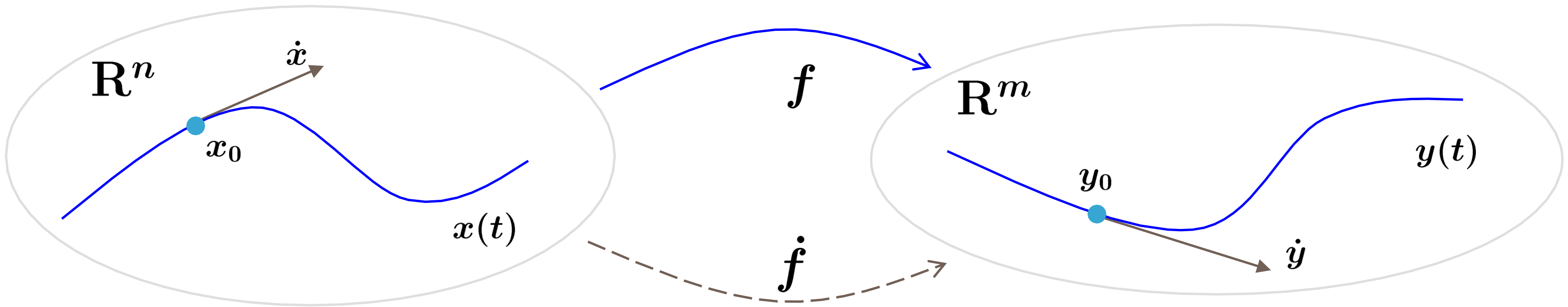
Finite Differencing

$$\frac{dy}{dx} \approx \frac{y(2+\varepsilon) - y(2)}{\varepsilon}$$

$$\approx -7.233\ 343\ 187$$

Tangent Propagation

$$y = f(x),\ f : \mathrm{R}^n \rightarrow \mathrm{R}^m$$



- Tangents

$$y(t) = f(x(t)) \implies \dot{y} \equiv \left.\frac{dy}{dt}\right|_{t=t_0} = \frac{\partial f}{\partial x}\dot{x}$$

- For each intermediate operation

$$c = \varphi(a, b) \implies \dot{c} = \frac{\partial \varphi}{\partial a}\dot{a} + \frac{\partial \varphi}{\partial b}\dot{b}$$

- Tangents map forward through evaluation

Operation	Tangent Rule
$c = a + b$	$\dot{c} = \dot{a} + \dot{b}$
$c = a - b$	$\dot{c} = \dot{a} - \dot{b}$
$c = ab$	$\dot{c} = a\dot{b} + \dot{a}b$
$c = a/b$	$\dot{c} = (\dot{a} - c\dot{b})/b$
$c = a^b$	$\dot{c} = c(\dot{b}\log(a) + \dot{a}b/a)$
$c = \sin(a)$	$\dot{c} = \cos(a)\dot{a}$
$c = \log(a)$	$\dot{c} = \dot{a}/a$

A Simple Tangent Example

$$y_1 = \sin(e^{x_1} + x_1 x_2)$$
$$y_2 = \frac{y_1}{y_1 + x_1^2}$$

$$\begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \end{bmatrix} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix}$$

Given x_1, x_2, $\dot{x}_1$, $\dot{x}_2$:

$$\begin{aligned}
s_1 &\leftarrow e^{x_1} & \dot{s}_1 &\leftarrow s_1 \dot{x}_1 \\
s_2 &\leftarrow x_1 x_2 & \dot{s}_2 &\leftarrow x_1 \dot{x}_2 + \dot{x}_1 x_2 \\
s_3 &\leftarrow s_1 + s_2 & \dot{s}_3 &\leftarrow \dot{s}_1 + \dot{s}_2 \\
y_1 &\leftarrow \sin(s_3) & \dot{y}_1 &\leftarrow \cos(s_3)\dot{s}_3 \\
s_4 &\leftarrow x_1^2 & \dot{s}_4 &\leftarrow 2x_1 \dot{x}_1 \\
s_5 &\leftarrow y_1 + s_4 & \dot{s}_5 &\leftarrow \dot{y}_1 + \dot{s}_4 \\
y_2 &\leftarrow y_1 / s_5 & \dot{y}_2 &\leftarrow (\dot{y}_1 - y_2 \dot{s}_5)/s_5
\end{aligned}$$

Return y_1, y_2, $\dot{y}_1$, $\dot{y}_2$

Forward Mode AD via Tangent Propagation

- Choice of space curve $x(t)$ is arbitrary
- Tangent $\dot{y}$ depends only on x_0, $\dot{x}$
- Given x_0 and v:

$$y(t) = f(x_0 + vt) \implies \dot{y} = \frac{\partial f}{\partial x_0} v$$ Jacobian vector product

- Propagate p vectors $v_1, \ldots, v_p$ simultaneously

$$[\dot{y}_1 \ldots \dot{y}_p] = \frac{\partial f}{\partial x_0} [v_1 \ldots v_p] = \frac{\partial f}{\partial x_0} V$$ Jacobian matrix product

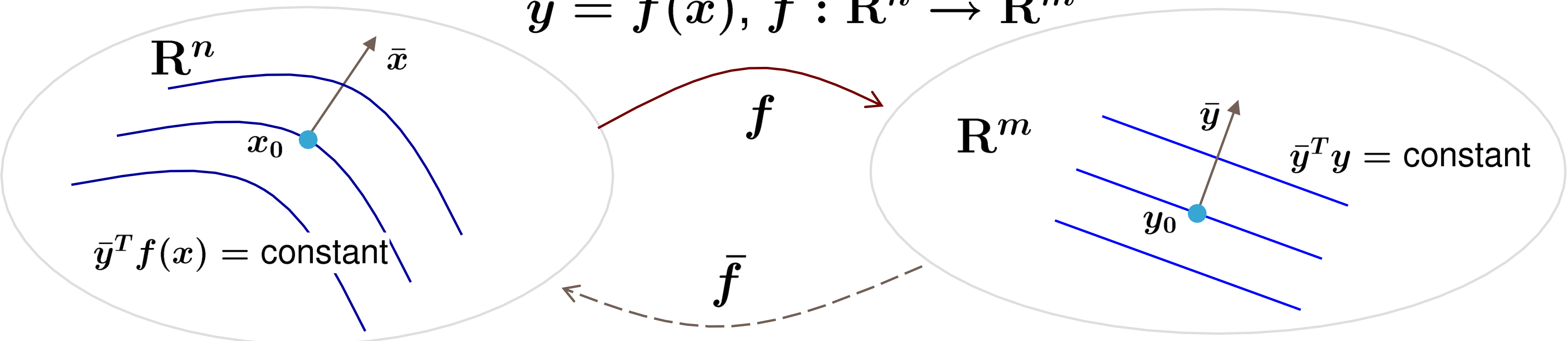
- Forward mode AD:

$$(x, V) \rightarrow \left(f(x), \frac{\partial f}{\partial x} V \right)$$

- V is called the seed matrix. Setting equal to identity matrix yields full Jacobian
- Computational cost $\approx (1 + 1.5p)\text{time}(f)$
- Jacobian-vector products, directional derivatives, Jacobians for $m \geq n$

Gradient Propagation

$$y = f(x),\ f : \mathrm{R}^n \rightarrow \mathrm{R}^m$$



- Gradients

$$z = \bar{y}^T y = \bar{y}^T f(x) \implies \bar{x} \equiv \left(\frac{\partial z}{\partial x}\right)^T = \left(\frac{\partial f}{\partial x}\right)^T \bar{y}$$

- For each intermediate operation

$$c = \varphi(a, b) \implies \begin{aligned} \bar{a} &= \frac{\partial z}{\partial a} = \frac{\partial z}{\partial c}\frac{\partial c}{\partial a} = \bar{c}\frac{\partial \varphi}{\partial a}, \\ \bar{b} &= \frac{\partial z}{\partial b} = \frac{\partial z}{\partial c}\frac{\partial c}{\partial b} = \bar{c}\frac{\partial \varphi}{\partial b} \end{aligned}$$

- Gradients map backward through evaluation

Operation	Gradient Rule
$c = a + b$	$\bar{a} = \bar{c}, \quad \bar{b} = \bar{c}$
$c = a - b$	$\bar{a} = \bar{c}, \quad \bar{b} = -\bar{c}$
$c = ab$	$\bar{a} = \bar{c}b, \quad \bar{b} = \bar{c}a$
$c = a/b$	$\bar{a} = \bar{c}/b, \quad \bar{b} = -\bar{c}c/b$
$c = a^b$	$\bar{a} = \bar{c}c\log(a),\ \bar{b} = \bar{c}cb/a$
$c = \sin(a)$	$\bar{a} = \bar{c}\cos(a)$
$c = \log(a)$	$\bar{a} = \bar{c}/a$

A Simple Gradient Example

$$y_1 = \sin(e^{x_1} + x_1 x_2)$$

$$y_2 = \frac{y_1}{y_1 + x_1^2}$$

$$\begin{bmatrix} \bar{x}_1 \\ \bar{x}_2 \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \end{bmatrix}^T \begin{bmatrix} \bar{y}_1 \\ \bar{y}_2 \end{bmatrix}$$

$$c = \varphi(a, b) \implies \begin{array}{l} \bar{a} = \bar{c}\frac{\partial \varphi}{\partial a}, \\ \bar{b} = \bar{c}\frac{\partial \varphi}{\partial b} \end{array}$$

Given x_1, x_2, $\bar{y}_1$, $\bar{y}_2$:

$s_1 \leftarrow e^{x_1}$

$s_2 \leftarrow x_1 x_2$

$s_3 \leftarrow s_1 + s_2$

$y_1 \leftarrow \sin(s_3)$

$s_4 \leftarrow x_1^2$

$s_5 \leftarrow y_1 + s_4$

$y_2 \leftarrow y_1 / s_5$

$\bar{y}_1 \leftarrow \bar{y}_1 + \bar{y}_2 / s_5, \quad \bar{s}_5 \leftarrow -y_2 \bar{y}_2 / s_5$

$\bar{y}_1 \leftarrow \bar{y}_1 + \bar{s}_5, \quad \bar{s}_4 \leftarrow \bar{s}_5$

$\bar{x}_1 \leftarrow 2\bar{s}_4 x_1$

$\bar{s}_3 \leftarrow \bar{y}_1 \cos(s_3)$

$\bar{s}_1 \leftarrow \bar{s}_3, \quad \bar{s}_2 \leftarrow \bar{s}_3$

$\bar{x}_1 \leftarrow \bar{x}_1 + \bar{s}_2 x_2, \quad \bar{x}_2 \leftarrow \bar{s}_2 x_1$

$\bar{x}_1 \leftarrow \bar{x}_1 + \bar{s}_1 s_1$

Return y_1, y_2, $\bar{x}_1$, $\bar{x}_2$

Reverse Mode AD via Gradient Propagation

- Choice of normal $\bar{y}$ is arbitrary
- Gradient $\bar{x}$ depends only on x_0, $\bar{y}$
- Given x_0 and w:

$$\bar{y} = w, y = f(x) \implies \bar{x} = \left(\frac{\partial f}{\partial x}\right)^T w$$ Jacobian-transpose vector product

- Propagate q vectors $w_1, \dots, w_q$ simultaneously

$$[\bar{x}_1 \dots \bar{x}_q] = \left(\frac{\partial f}{\partial x}\right)^T [w_1 \dots w_q] = \left(\frac{\partial f}{\partial x}\right)^T W$$ Jacobian-transpose matrix product

- Reverse mode AD: $$(x, W) \rightarrow \left(f(x), \left(\frac{\partial f}{\partial x}\right)^T W\right)$$
- W is called the seed matrix. Setting equal to identity matrix yields full Jacobian
- Computational cost $\approx (1.5 + 2.5q)$time(f) $m = q = 1 \implies$ cost ≈ 4 time(f)
- Jacobian-transpose products, gradients, Jacobians for $n > m$

Software Implementations

- Tools implementing AD have been created for many popular programming languages
  - C/C++: ADOL-C, ADIC, Sacado, …
  - Fortran: ADIFOR, OpenAD, Tapenade, …
  - Matlab: ADiMAT, MAD, …
  - Python: pyADOL-C, AD, …
- See http://www.autodiff.org/ for a comprehensive listing
- Tools fall into two general categories
  - Source transformation
  - Operator overloading

Source Transformation

- AD implemented by preprocessor
  - Preprocessor reads code to be differentiated
  - Uses AD to generate derivative code
  - Writes-out differentiated code in original source language
  - Differentiated code is then compiled using a standard compiler
- Resulting derivative computation is usually very efficient
- Works well for simple languages (FORTRAN, some C)
- ADIFOR/ADIC/OpenAD out of Argonne
- Extremely difficult for C++

ADIFOR* Example

```
      subroutine func(x, y)
C
      double precision x(2), y(2)
      double precision u, v, w
C
      u = exp(x(1))
      v = x(1)*x(2)
      w = u+v
      y(1) = sin(w)
C
      u = x(1)**2
      v = y(1) + u
      y(2) = y(1)/v
C
      return
      end
```

```
      subroutine g_func(g_p_, x, g_x, ldg_x, y, g_y, ldg_y)

C     Initializations removed for clarity…

      d2_v = exp(x(1))
      d1_p =  d2_v
      do g_i_ = 1, g_p_
        g_u(g_i_) = d1_p * g_x(g_i_, 1)
      enddo
      u = d2_v
C--------
      do g_i_ = 1, g_p_
        g_v(g_i_) = x(1) * g_x(g_i_, 2) + x(2) * g_x(g_i_, 1)
      enddo
      v = x(1) * x(2)
C--------
      do g_i_ = 1, g_p_
        g_w(g_i_) = g_v(g_i_) + g_u(g_i_)
      enddo
      w = u + v
C--------
      d2_v = sin(w)
      d1_p = cos(w)
      do g_i_ = 1, g_p_
        g_y(g_i_, 1) = d1_p * g_w(g_i_)
      enddo
      y(1) = d2_v

C     continues…
```

*ADIFOR 2.0D
www.mcs.anl.gov/research/projects/adifor/

Operator Overloading

- AD implemented within source language constructs
  - New data types are created for forward, reverse, Taylor modes
  - Intrinsic operations/elementary operations are overloaded to compute derivatives as a side-effect
  - Data type (e.g., double) in original code is replaced with AD type
- Generally easy to incorporate into C++ codes
- Generally slower than source transformation due to function call overhead
  - This can generally be eliminated
- Requires changing data types from floats/doubles to AD types
  - C++ templates greatly help
- ADOL-C, FAD/TFAD, Sacado

(Naive) Operator Overloading Example

```
void func(const double x[], double y[]) {
  double u, v, w;
  u = exp(x[0]);
  v = x[0]*x[1];
  w = u+v;
  y[0] = sin(w);

  u = x[0]*x[0];
  v = y[0] + u;
  y[1] = y[0]/v;
}

void func(const Tangent x[], Tangent y[]) {
  Tangent u, v, w;
  u = exp(x[0]);
  v = x[0]*x[1];
  w = u+v;
  y[0] = sin(w);

  u = x[0]*x[0];
  v = y[0] + u;
  y[1] = y[0]/v;
}
```

```
class Tangent {
public:
  static const int N = 2;
  double val;
  double dot[N];
};

Tangent operator+(const Tangent& a, const Tangent& b) {
  Tangent c;
  c.val = a.val + b.val;
  for (int i=0; i<Tangent::N; i++)
    c.dot[i] = a.dot[i] + b.dot[i];
    return c;
}
Tangent operator*(const Tangent& a, const Tangent& b) {
  Tangent c;
  c.val = a.val * b.val;
  for (int i=0; i<Tangent::N; i++)
    c.dot[i] = a.val * b.dot[i] + a.dot[i]*b.val;
    return c;
}
Tangent exp(const Tangent& a) {
  Tangent c;
  c.val = exp(a.val);
  for (int i=0; i<Tangent::N; i++)
    c.dot[i] = c.val * a.dot[i];
    return c;
}
```

Expression Template Operator Overloading

```
void func(const Tangent x[], Tangent y[]) {
  y[0] = sin(exp(x[0]) + x[0]*x[1]);
  //…
}
```

↓

```
SinExpr< PlusExp< ExpExpr<Tangent>,
                  MultExpr<Tangent,Tangent>
                 >
        >
```

↓

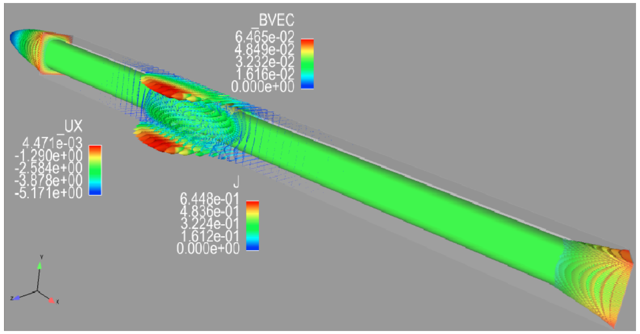
```
y[0].val = sin(exp(x[0]) + x[0]*x[1]);
for (int i=0; i<N; i++) {
  y[0].dot[i] = cos(exp(x[0]) + x[0]*x[1])*
     (exp(x[0])*x[0].dot[i] +
      x[0]*x[1].dot[i] + x[1]*x[0].dot[i]);
```

Public domain Fad/TFad package

```
template <class E1, E2> class PlusExpr {
  double val() const { return e1.val() + e2.val(); }
  double dx(int i) const { return e1.dx(i) + e2.dx(i); }
  const E1& e1;
  const E2& e2;
};
template<class E1, class E2> PlusExpr<E1,E2>
operator+(const E1& a, const E2& b) {
  return PlusExpr<E1,E2>(a,b);
}
template <class E1> class SinExpr {
  double val() const { return sin(e1.val())] }
  double dx(int i) const { return cos(e1.val())*e1.dx(i); }
  const E1& e1;
};
template<class E1> SinExpr<E1> sin(const E1& a) {
  return SinExpr<E1>(a);
}
class Tangent {
public:
  double val() const { return val; }
  double dx(int i) const { return dot[i]; }
  template <class E> Tangent& operator=(const E& e) {
    val = e.val();
    for (int i=0; i<N; i++)
      dot[i] = e.dx(i);
   }
};
```

Sacado: AD Tools for C++ Apps

- Package in Trilinos
  - https://github.com/trilinos
  - Open source license
- Forward mode AD
  - Based on Fad<> library of Di Césaré, Aubert and Pironneau
  - Tries to eliminates OO overhead via expression templates
  - DFad<double>: Derivative array determined at run-time
  - SFad<double,N>: Derivative array length = N
  - SLFad<double,N>: Derivative array length at most N
- Reverse mode AD
  - David Gay's Rad library
- AD applied through template-based generic programming
  - Template on scalar type
  - Instantiate on AD data types
- Manually exploit simulation structure/sparsity
  - AD applied at "element" level
  - Template "physics"
  - Manually incorporate derivatives into global linear algebra objects



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

How to use Sacado

- Template code to be differentiated: double -> ScalarT
- Replace independent/dependent variables with AD variables
- Initialize seed matrix
  - Forward: Derivative array of i'th independent variable is i'th row of seed matrix
  - Reverse: Derivative array of i'th dependent variable is i'th row of seed matrix
- Evaluate function on AD variables
  - Instantiates template classes/functions
- Extract derivatives
  - Forward: Derivative components of dependent variables
  - Reverse: Derivative components of independent variables

Primary Sacado AD Classes

- #include "Sacado.hpp"

- All classes are templated on the Scalar type

- Forward AD classes:
  - Sacado::Fad::DFad<ScalarT>: Derivative array is allocated dynamically
  - Sacado::Fad::SFad<ScalarT>: Derivative array is allocated statically and dimension must be known at compile time
  - Sacado::Fad::SLFad<ScalarT>: Like SFad except allocated length may be greater than "used" length
  - Sacado::Fad::SimpleFad<ScalarT>: Dynamically allocated array that doesn't use expression templates

- Similar forward AD classes in other namespaces that use different forward AD approaches (research ideas)
  - Sacado::ELRFad, Sacado::CacheFad, Sacado::ELRCacheFad

- Reverse mode AD classes:
  - ADvar<ScalarT>

Basic Fad Example

Forward or Reverse?

- Forward: Computes derivatives column-wise
  - Number of independent variables <= number of dependent variables
  - Square Jacobians for Newton's method
  - Sensitivities with small numbers of parameters
  - Algorithm naturally calls for Jacobian-vector/matrix products
    - (Block) Matrix-free Newton-Krylov
- Reverse: Computes derivatives row-wise
  - Number of independent variables >> number of dependent variables
  - Gradients of scalar valued functions
  - Sensitivities with respect to large numbers of parameters
  - Algorithm naturally calls for Jacobian-transpose-vector/matrix products
    - (Block) Matrix-free solves of transpose matrix
    - Optimization

Choosing AD Types

- DFad
  - Derivative array allocated dynamically
  - Most flexible
  - Slowest
  - Very slow in threaded environments
- SFad
  - Derivative array size fixed at compile time
  - Must know exact number of derivative components
  - Fastest
  - Best choice in threaded environments
- SLFad
  - Fixed-length derivative array, can use only a portion of it at run-time
  - Compromise between the two
  - Usually just a little slower than SFad
  - Good choice for threaded environments
- ADvar (reverse mode)
  - Due to overhead, need substantially more independent variables than dependent variables (at least 40 more)
  - Currently not appropriate for threaded environments

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

Performance (Charon semiconductor physics code)

Scalability of the element-level derivative computation

Set of N hypothetical chemical species:

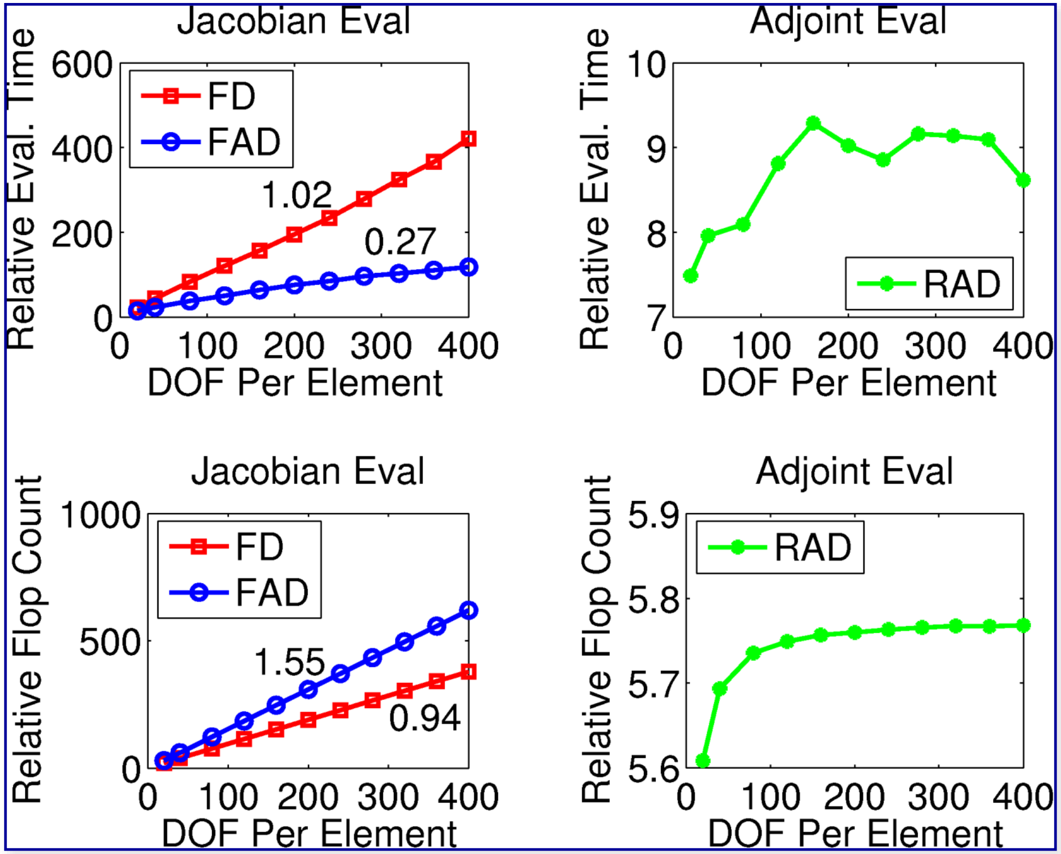
$$2X_j \rightleftharpoons X_{j-1} + X_{j+1}, \quad j = 2, \ldots, N-1$$

Steady-state mass transfer equations:

$$\mathbf{u} \cdot \nabla Y_j + \nabla^2 Y_j = \dot{\omega}_j, \quad j = 1, \ldots, N-1$$

$$\sum_{j=1}^{N} Y_j = 1$$

- Forward mode AD
  - Faster than FD
  - Better scalability in number of PDEs
  - Analytic derivative
  - Provides Jacobian for all Charon physics
- Reverse mode AD
  - Scalable adjoint/gradient



DOF per element = 4*N

Matrix/Residual Assembly Performance Test

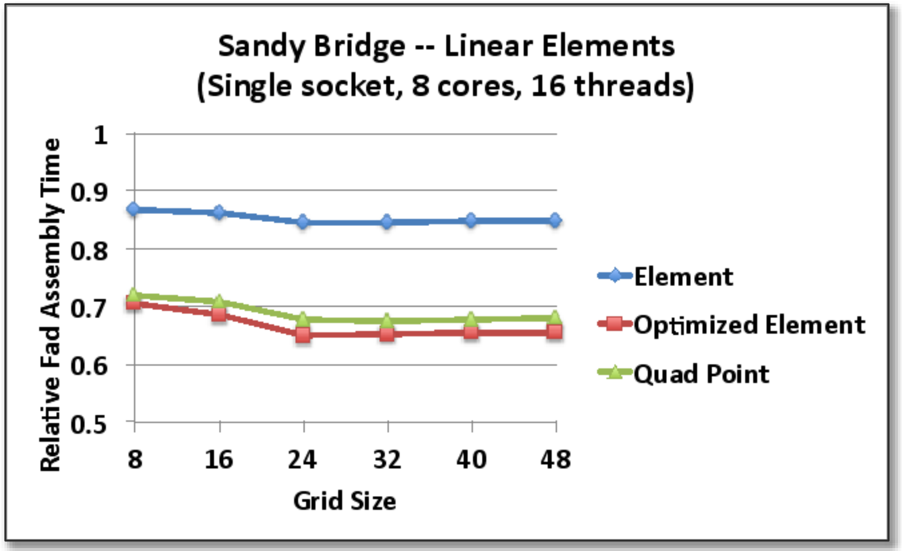
- Performance test for measuring Jacobian/Residual assembly using Sacado

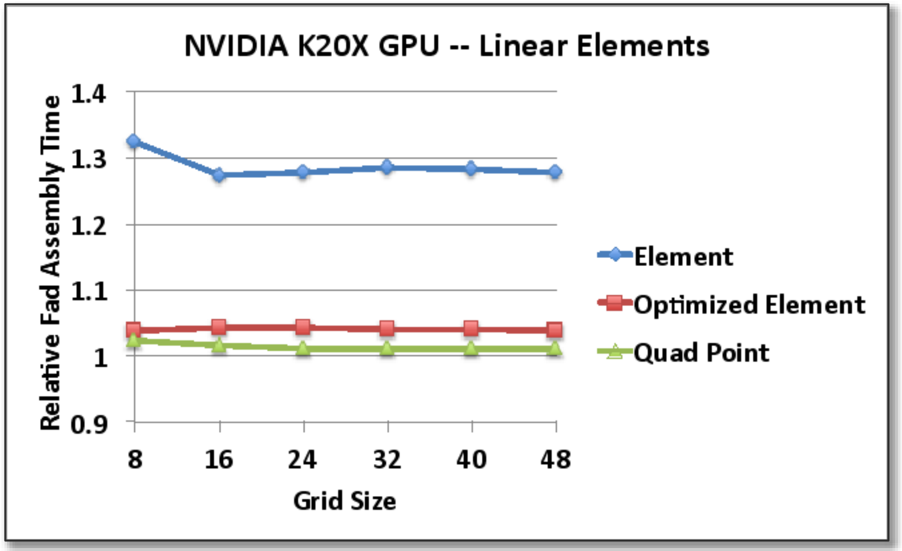
$$-\nabla \cdot (\kappa \nabla u) + \alpha v \cdot \nabla u + \beta u^2 = 0$$

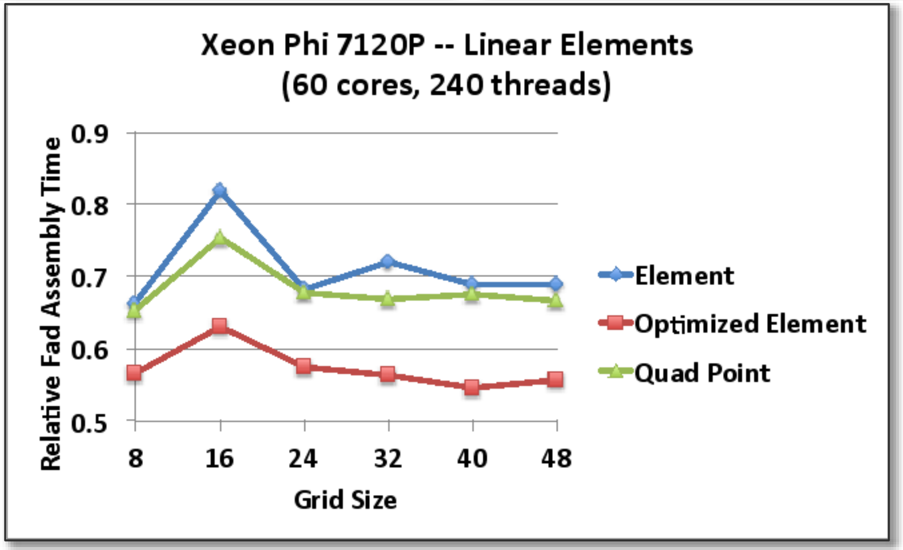
  - 3-D, linear FEM discretization
  - 1x1x1 cube, unstructured mesh
  - Derived from FENL Kokkos example (H. Carter Edwards)
  - Thread-parallel matrix/residual assembly
    - Mesh cell loop parallelized with OpenMP/CUDA
    - Atomic instructions for assembling into matrix/residual

- 3 algorithms studied
  - Traditional element derivative w.r.t. nodal solution (AD size = # nodes/element x # equations)
  - Element derivative with optimized derivative of interpolation of nodal solution, gradient at quadrature points
  - Derivative at each quadrature point w.r.t. nodal solution and gradient interpolated at quadrature point (AD size = 4 x # equations)

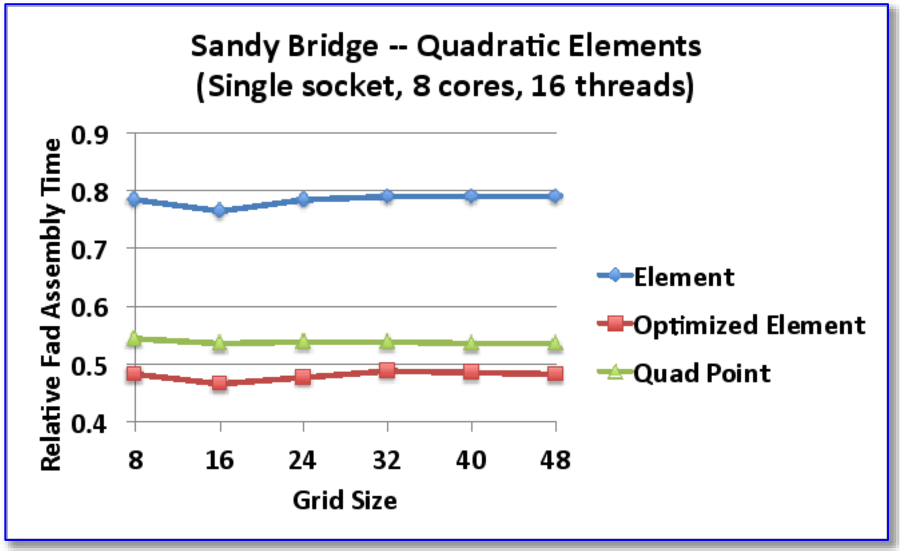
Sacado Assembly Performance

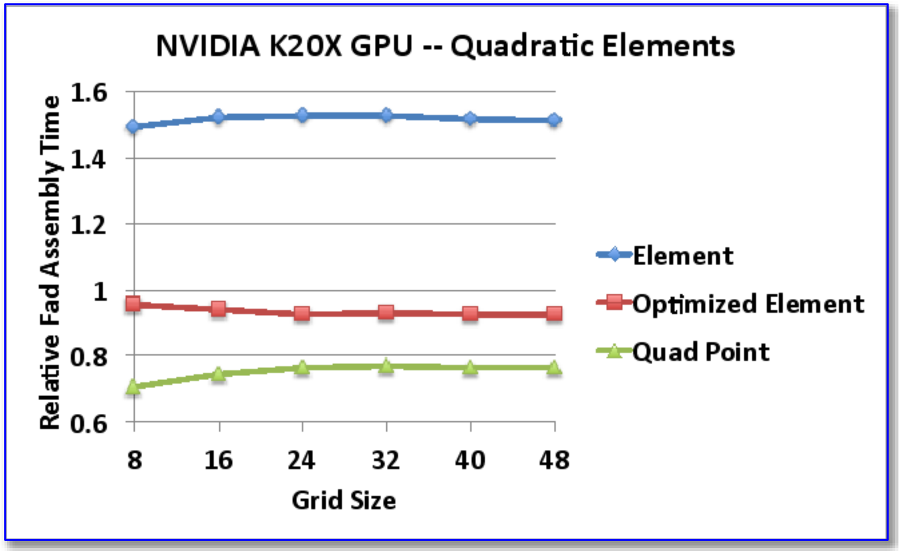


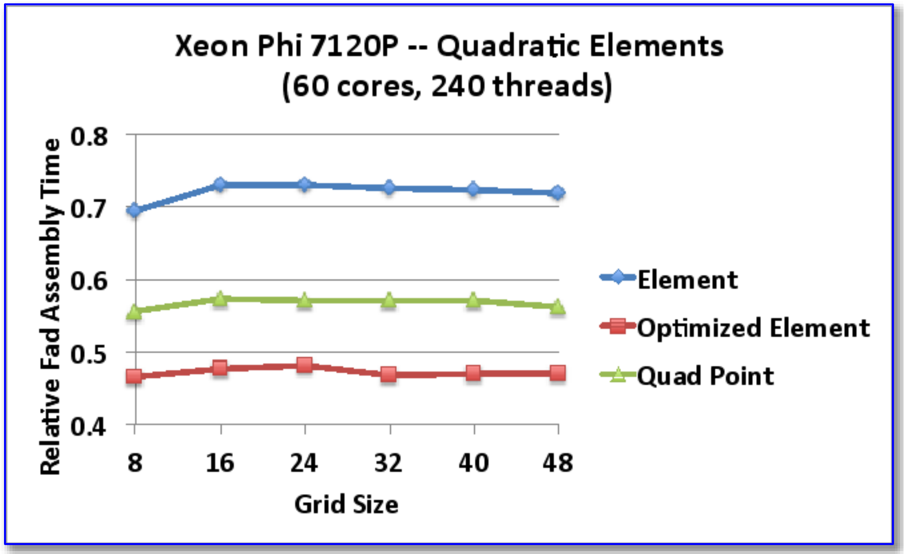




Sacado Assembly Performance

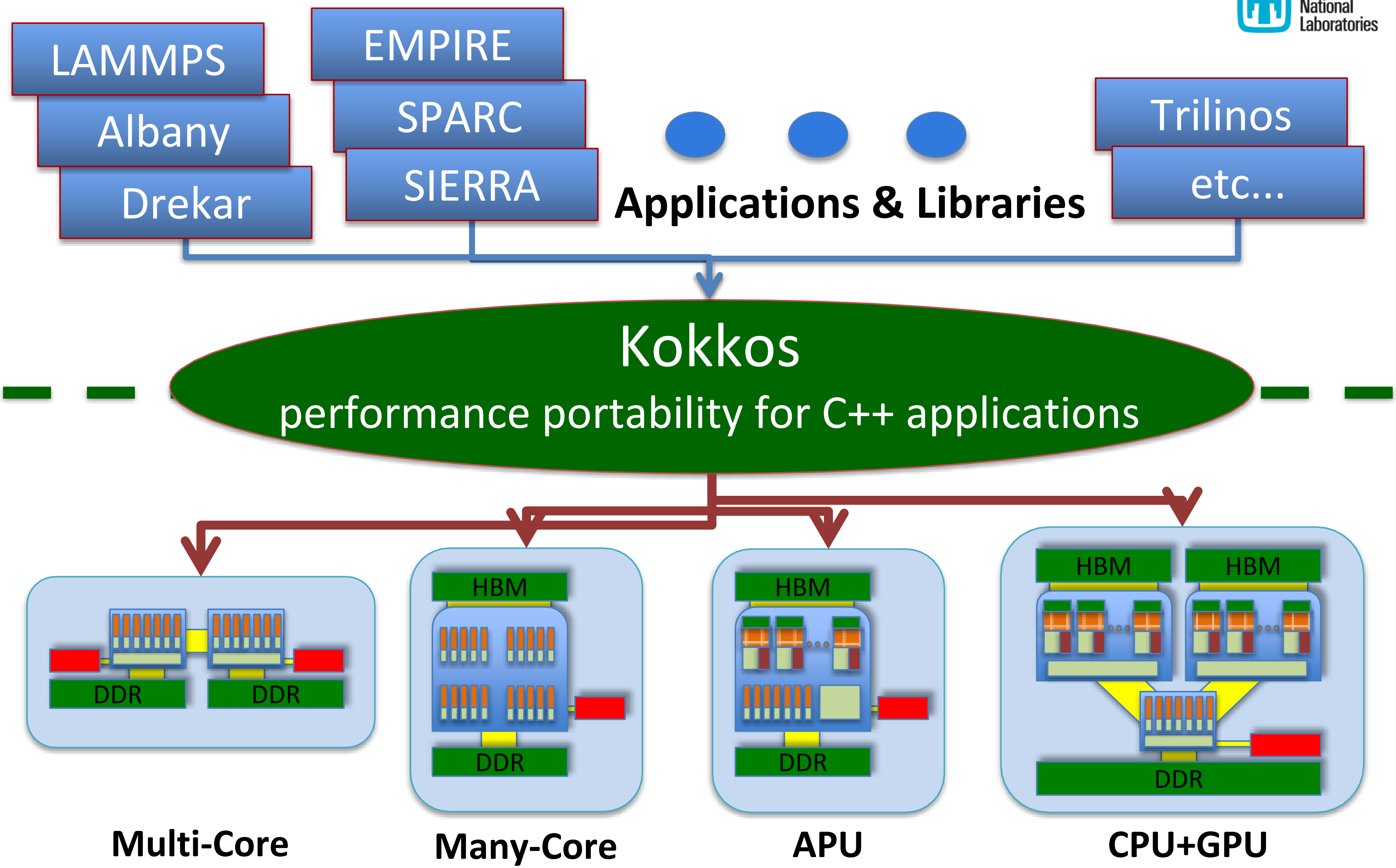






AD Research

- Efficiently deploying AD in modern programming environments
  - Expression templates for C++
  - AD in interpreted languages (Matlab, Python, …)
- Reducing overhead of reverse-mode AD
- AD in threaded-environments
  - Automatically differentiating thread-parallel programs
  - Exploiting thread parallelism within AD tools
- Finding most efficient way to differentiate a given program
  - Column/row compression
  - Cross-country elimination
- Efficiently evaluating higher derivatives
- Automatically detecting and exploiting sparsity in derivatives



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

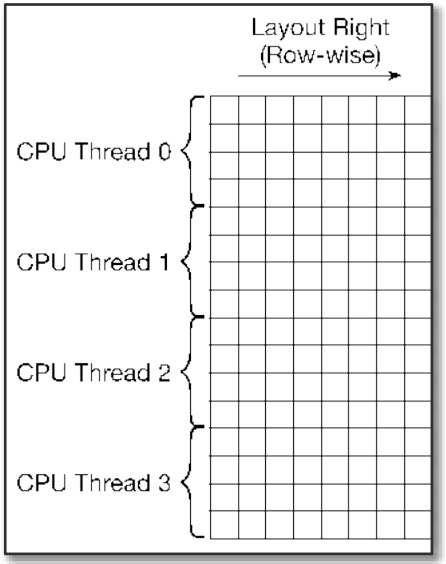
Kokkos Example

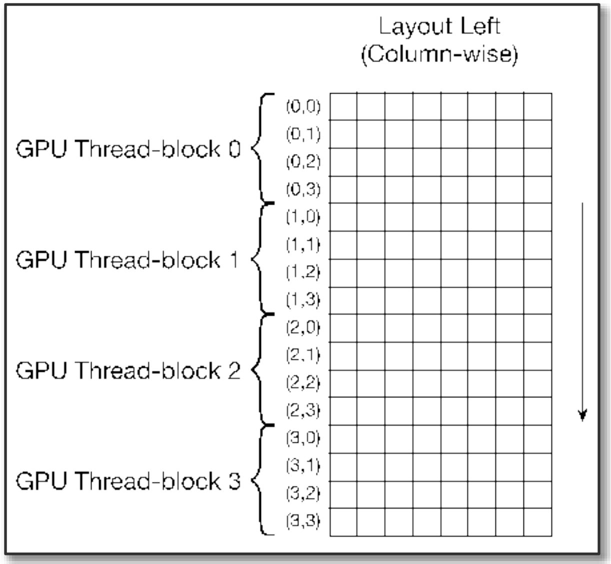
Sandia National Laboratories

```
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)





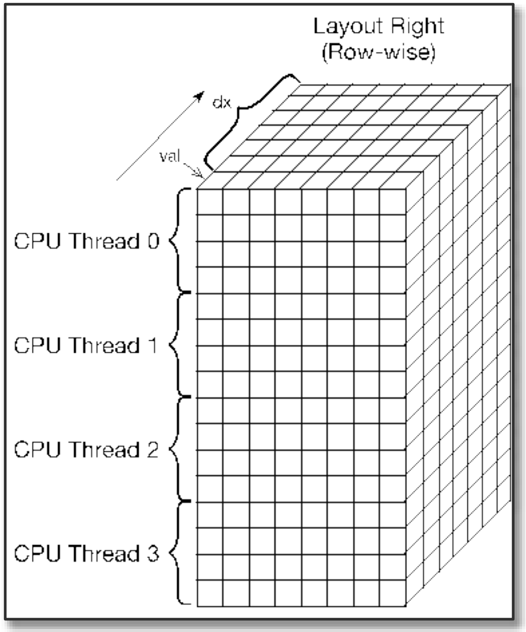
$M = 10^6$, $n = 100$

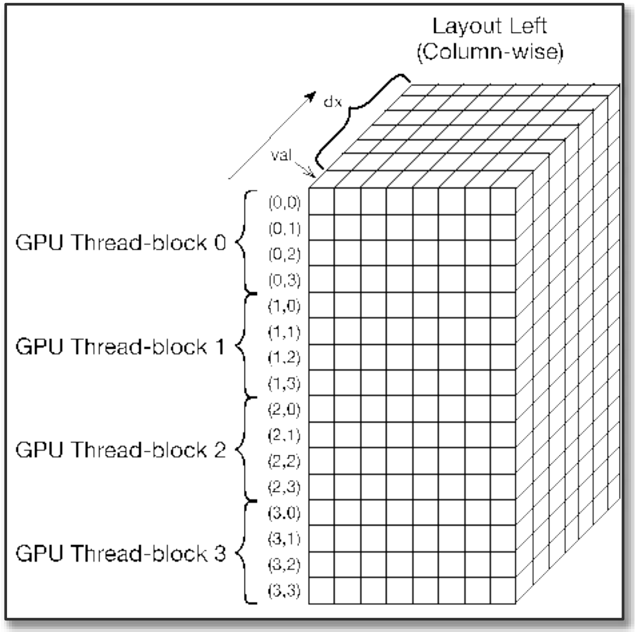
Architecture	Description	Execution Space	Measured Bandwidth (GB/s)	Expected Throughput (GFLOP/s)	Measured Throughput (GFLOP/s)	Wrong Layout (GFLOP/s)
Skylake (1 socket)	Intel Xeon Gold 6154, 36 threads	OpenMP	64.4	16.1	18.0	15.3
GPU	NVIDIA V100	Cuda	833	208	213	26.3

Sacado and Kokkos?

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

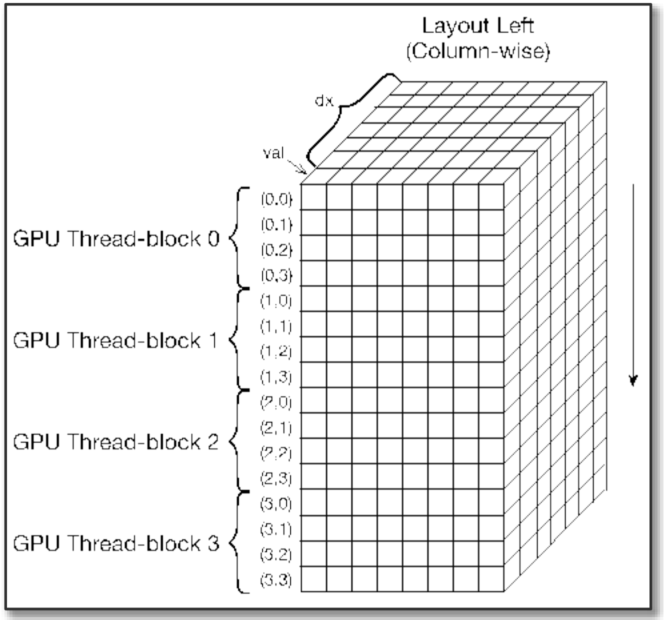
- Kokkos::View< Sacado::Fad::SFad<double,p>**>:
  - Derivative components always stored consecutively
  - CPU: Good cache, vector performance
  - GPU: Large stride causes bad coalescing





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

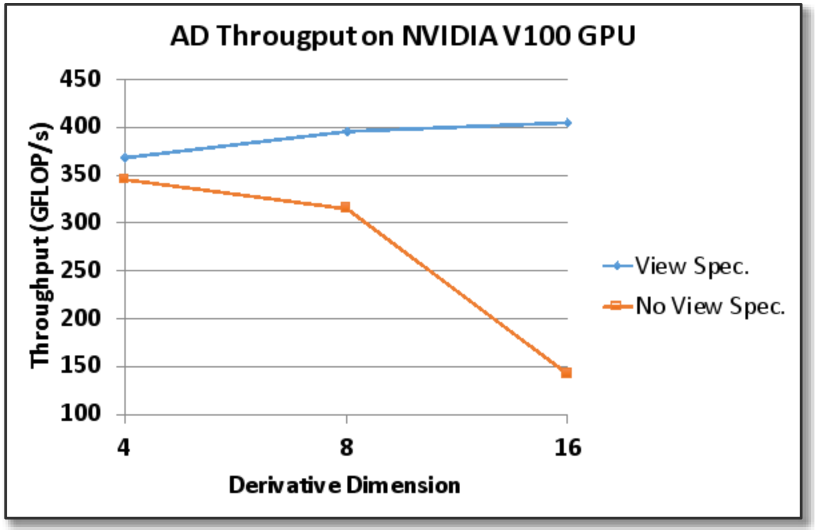


AD Performance Portability

```
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

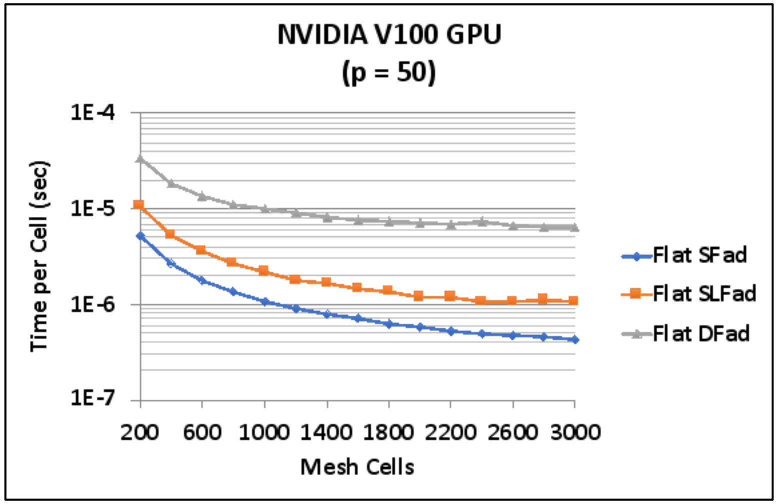


Hierarchical Parallelism

- 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

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



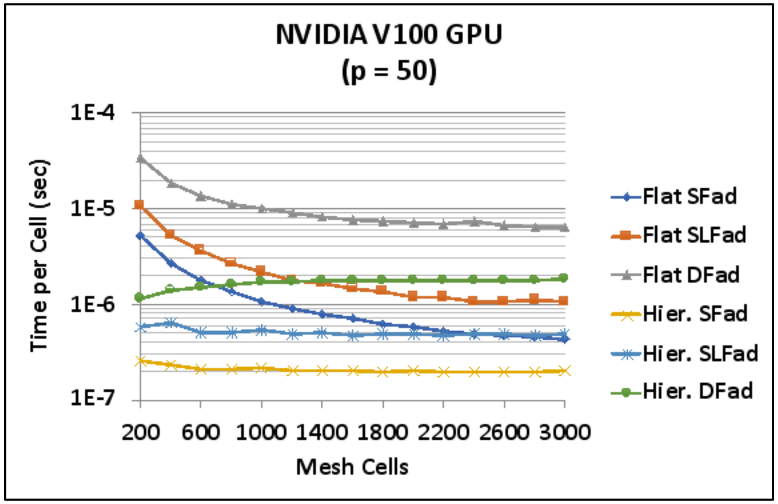

```
Kokkos::View<ScalarT****, Layout, ExecSpace> wgb;
Kokkos::View<ScalarT***,  Layout, ExecSpace> flux;
Kokkos::View<ScalarT***,  Layout, ExecSpace> wbs;
Kokkos::View<ScalarT**,   Layout, ExecSpace> src;
Kokkos::View<ScalarT**,   Layout, ExecSpace> residual;
```

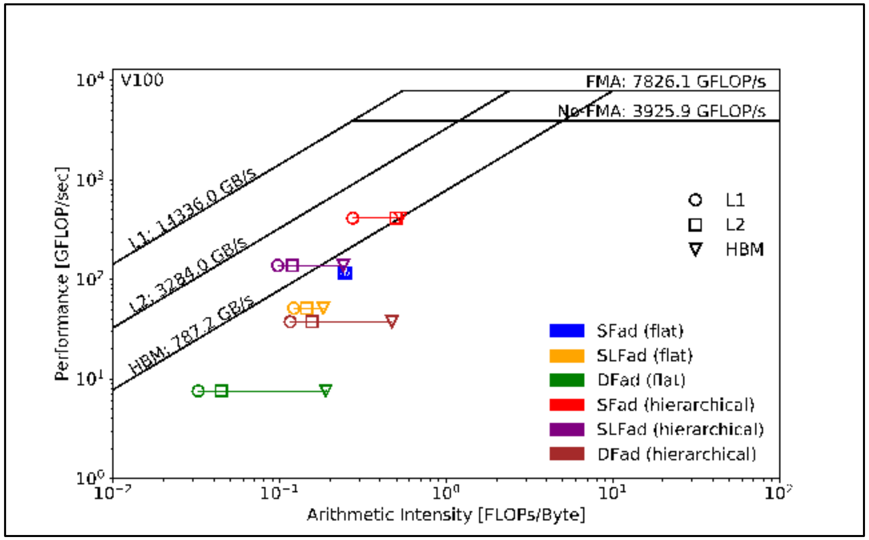
```
typedef Kokkos::RangePolicy<ExecSpace> Policy;
Kokkos::parallel_for(
  Policy( 0,num_cell ),
  KOKKOS_LAMBDA( const int cell )
  {



    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;
    }
  });
```

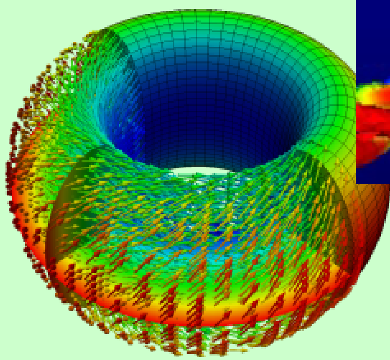


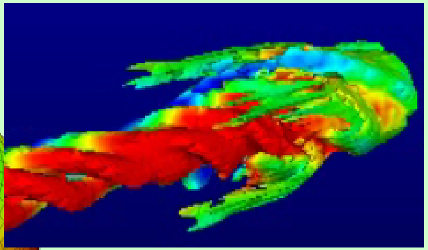


Drekar/Panzer PDE Tools

(Pawlowski, Cyr, Shadid, Smith)

Applications

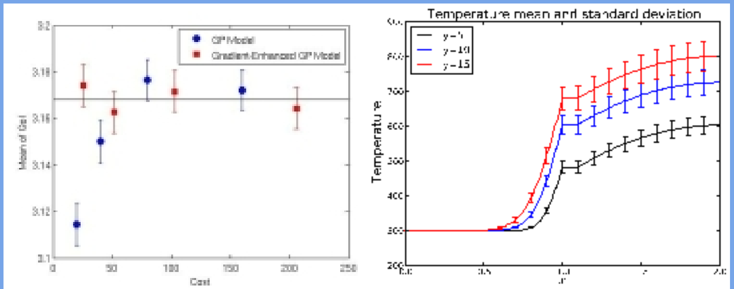




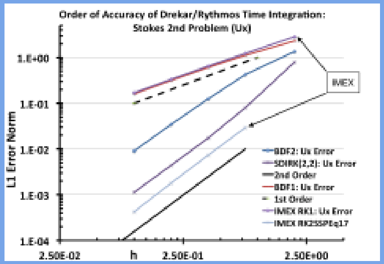
Turbulent CFD

Magnetohydrodynamics

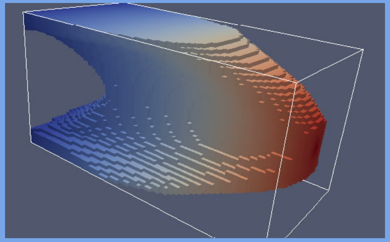
Discretizations & Algorithms



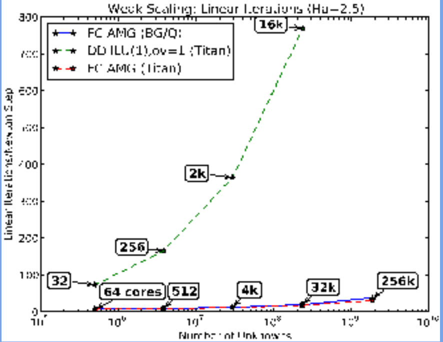
Uncertainty Quantification



IMEX



PDE Constrained Optimization

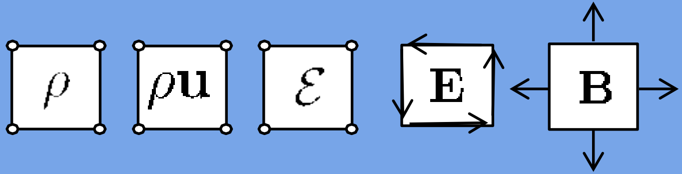


Algebraic Multigrid (>100k cores)

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

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

Block Preconditioning



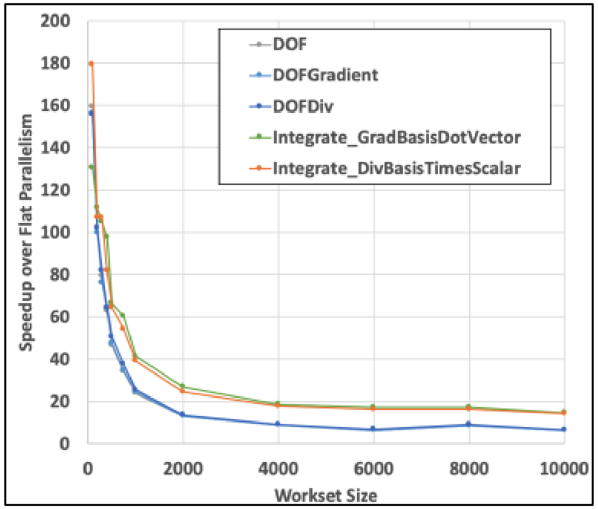
Compatible Discretizations

Hierarchical Parallelism in Panzer

- Diffusion problem with mixed finite element discretization:

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

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

- Analytic derivatives are an important enabling technology for simulation and analysis
  - Automatic differentiation is a powerful means for obtaining these derivatives
- Sacado provides efficient AD capabilities to C++ codes
  - Widespread use within Sandia simulation codes
- Highly parallel architectures like GPUs are here
  - AD tools and techniques need to work in these environments
- Sacado solves this problem through integration with Kokkos
  - Leverage layout polymorphism to enable AD of Kokkos kernels without modification
  - Incorporate GPU vector/warp-level parallelism for improved performance
- Code and performance tests are available within Sacado (and Panzer)
  - https://github.com/trilinos