Operator Overloading-based Automatic Differentiation of C++ Codes on Emerging Manycore Architectures

Eric Phipps (etphipp@sandia.gov) and H. Carter Edwards
Sandia National Laboratories

AD2016 - 7th International Conference on Algorithmic Differentiation


SAND2016-8683 C
Sandia Physics Simulation Codes

• "Element-based"
  – Finite element, finite volume, finite difference, network, etc…

• Large-scale
  – Billions of unknowns

• Parallel
  – Distributed memory
  – MPI

• C++
  – Object oriented
  – Run-time & compile-time polymorphism
Sacado: AD Tools for C++ Applications

• Package in Trilinos
  – [https://trilinos.org](https://trilinos.org) or [https://github.com/trilinos](https://github.com/trilinos)
  – Open source license

• Forward mode
  – Based on Fad<> library of Di Césaré, Aubert and Pironneau
  – Tries to eliminate 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
  – 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)
Computer Architectures Are Changing Dramatically

• Historically (super)computers have gotten faster by
  – Increasing clock frequency
  – Adding more compute nodes that communicate through an interconnect

• Power requirements make this approach untenable for future performance increases

• Instead performance increases are now achieved through increases in node-level fine-grained parallelism
  – Many, many threads executing simultaneously
  – Memory access, arithmetic on wide vectors
  – Complex memory hierarchies that require processing units to share data

Herb Sutter, “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software”, Dr. Dobb’s Journal
Applications & Libraries

Kokkos*
performance portability for C++ applications

LAMMPS
Albany
Drekar

EMPIRE
SPARC
SIERRA

Trilinos
... etc.

Applications & Libraries

Kokkos*: “granule” or “grain”; like grains of sand on a beach

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) {
            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/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)
## Performance Portability

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

- \( m = 1e6, n=100 \)

- Expected Throughput = Measured Bandwidth x 2 FLOPS / 8 Bytes

- Manually specify incorrect layout for “Wrong Layout”, e.g.,

  ```cpp
  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
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
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 Performance Portability

- **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**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Measured Bandwidth (GB/s)</th>
<th>Expected Throughput (GFLOP/s)</th>
<th>Measured Throughput (GFLOP/s)</th>
<th>No View Specialization (GFLOP/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haswell</td>
<td>47.4</td>
<td>22.4</td>
<td>24.3</td>
<td>23.1</td>
</tr>
<tr>
<td>MIC</td>
<td>147</td>
<td>69.4</td>
<td>69.4</td>
<td>43.2</td>
</tr>
<tr>
<td>GPU</td>
<td>150</td>
<td>70.8</td>
<td>81.2</td>
<td>35.1</td>
</tr>
</tbody>
</table>

```cpp
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);
```
Throughput Varying Derivative Dimension

NVIDIA K80 GPU
(m=10^5, n=100)

Throughput (GFLOP/s)

Derivative Dimension

View Spec.
No View Spec.
PDE Jacobian Assembly

• Sacado performance test based on Kokkos FENL example
  – 3-D advection-diffusion-reaction equation
  – Unstructured FEM assembly
  – Hex mesh, linear and quadratic basis functions
  – Compare AD assembly with SFad to well-optimized, hand-coded

AD Jacobian Assembly Time Relative to Optimized Hand-Coded Jacobian
(Nonlinear 3-D Advection-Diffusion-Reaction PDE, Linear Elements)

Relative Fad Assembly Time

8 16 24 32 40 48
Grid Size

Haswell (16 Threads)
NVIDIA K80 GPU
Xeon Phi 7120P (240 Threads)

AD Jacobian Assembly Time Relative to Optimized Hand-Coded Jacobian
(Nonlinear 3-D Advection-Diffusion-Reaction PDE, Quadratic Elements)

Relative Fad Assembly Time

8 16 24 32 40 48
Grid Size

Haswell (16 Threads)
NVIDIA K80 GPU
Xeon Phi 7120P (240 Threads)
Concluding Remarks

• Highly parallel architectures are the future
  – High thread concurrency
  – Vector architectures

• AD tools and techniques need to work in these environments
  – Tools must be aware of data layout for performance

• Sacado solves this problem through integration with Kokkos
  – These ideas are applicable to source transformation too!

• Another worthwhile approach: hierarchical (nested) parallelism:
  – Map portion of GPU threads across derivative dimension
  – Likely requires modification to kernel source and/or execution policy
  – Exploring how to do this effectively with Sacado and Kokkos

• Code and performance tests are available within Sacado
  – https://github.com/trilinos

• Sacado+Kokkos part of several large code projects at Sandia
  – Albany (https://github.com/gahansen/albany)
  – Drekar
  – ATDM
Differentiating Element-Based Simulation Codes

- Global residual computation (ignoring boundary computations):

\[ f(\dot{u}, u, p, t) = \sum_{i=1}^{N} Q_i^T e_{ki}(P_i \dot{u}, P_i u, p, t) \]

- Time-step Jacobian computation:

\[ \alpha \frac{\partial f}{\partial \dot{u}} + \beta \frac{\partial f}{\partial u} = \sum_{i=1}^{N} Q_i^T \left( \alpha \frac{\partial e_{ki}}{\partial \dot{u}_i} + \beta \frac{\partial e_{ki}}{\partial u_i} \right) P_i \]

- Parameter derivative computation:

\[ \frac{\partial f}{\partial p} = \sum_{i=1}^{N} Q_i^T \frac{\partial e_{ki}}{\partial p} \]

- Hybrid symbolic/AD procedure
  - Only use AD for element derivatives
  - Not exploiting intra-element-derivative structure
Sandia Strategy for Emerging Architectures

• Build on existing expertise in MPI

• Incorporate hybrid parallelism for new architectures (MPI+X)
  – OpenMP for Intel (HSW/KNL) & IBM (POWER)
  – Cuda for NVIDIA
  – ???

• Library approach for performance portability
  – Kokkos

• Explore alternative parallel programming models
  – Asynchronous many tasking (AMT)
Kokkos: Performance Portability for C++ Applications

• Performance Portable Thread-Parallel Programming Model
  – E.g., “X” in “MPI+X” ; not a distributed-memory programming model
  – Application identifies its parallelizable grains of computations and data
  – Kokkos maps those computations onto cores and that data onto memory

• Fully Performance Portable C++11 Library Implementation
  – Not a language extension (e.g., OpenMP, OpenACC, OpenCL, …)
  – Production – open source at https://github.com/kokkos/kokkos
  – Multicore CPU - including NUMA architectural concerns
  – Intel Xeon Phi (KNC) – toward DOE’s Trinity (ATS-1) supercomputer
  – NVIDIA GPU (Kepler) – toward DOE’s Sierra (ATS-2) supercomputer
  – IBM Power 8 – toward DOE’s Sierra (ATS-2) supercomputer
  – AMD Fusion – back-end in collaboration with AMD via HCC
Kokkos Programming Model

- **Parallel Pattern** of user’s computation
  - parallel_for, parallel_reduce, parallel_scan, task-graph, ... *(extensible)*

- **Execution Policy** tells **how** user computation will be execute
  - Static scheduling, dynamic scheduling, thread-teams, ... *(extensible)*

- **Execution Space** tells **where** user computations will execute
  - Which cores, numa region, GPU, ... *(extensible)*

- **Memory Space** tells **where** user data resides
  - Host memory, GPU memory, high bandwidth memory, ... *(extensible)*

- **Layout (policy)** tells **how** user data is laid out in memory
  - Row-major, column-major, array-of-struct, struct-of-array ... *(extensible)*

- **Differentiating: Layout and Memory Space**
  - Versus other programming models (OpenMP, OpenACC, ...)

  - This is critical for performance portability
Kokkos Functor Example

template<typename ViewTypeA, typename ViewTypeB, typename ViewTypeC>
struct MatVecFunctor {
    typedef typename ViewTypeC::value_type scalar_type; / Scalar type used

    // Data needed by functor
    const ViewTypeA A;
    const ViewTypeB b;
    const ViewTypeC c;
    const int n;

    // Constructor
    MatVecFunctor(const ViewTypeA& A_arg, const ViewTypeB& b_arg, const ViewTypeC& c_arg) :
        A(A_arg), b(b_arg), c(c_arg), n(A.dimension_1()) {}

    // Function to compute matrix–vector product for a given row i
    KOKKOS_INLINE_FUNCTION
    void operator() (const int i) const {
        scalar_type t = 0.0;
        for (int j=0; j<n; ++j)
            t += A(i,j)*b(j);
        c(i) = t;
    }
};
Kokkos Functor Example

template <typename ViewTypeA, typename ViewTypeB, typename ViewTypeC>
void run_mat_vec(const ViewTypeA& A, const ViewTypeB& b, const ViewTypeC& c) {
    typedef typename ViewTypeC::execution_space execution_space; // Space we are running on

    const int m = A.dimension_0();
    Kokkos::parallel_for(
        Kokkos::RangePolicy<execution_space>( 0,m ), // Iterate over [0,m)
        MatVecFunctor<ViewTypeA, ViewTypeB, ViewTypeC>(A,b,c)
    );
}

// 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);
Sacado Assembly Performance

Haswell-- Linear Elements  
(Single socket, 16 cores, 32 threads)

NVIDIA K80 GPU -- Linear Elements

Xeon Phi 7120P -- Linear Elements  
(60 cores, 240 threads)
Sacado Assembly Performance

Haswell--Quadratic Elements
(Single socket, 16 cores, 32 threads)

Haswell--Quadratic Elements
Optimized Element
Quad Point

NVIDIA K80 GPU -- Quadratic Elements

NVIDIA K80 GPU -- Quadratic Elements
Element
Optimized Element
Quad Point

Xeon Phi 7120P -- Quadratic Elements
(60 cores, 240 threads)

Xeon Phi 7120P -- Quadratic Elements
Element
Optimized Element
Quad Point
Hierarchical Parallelism

- Layout approach was explored to minimize code user-code changes for Sacado

- (Forward mode) Derivative propagation provides good opportunities for exposing more parallelism
  - Parallelism across derivative array
  - Code may not expose enough parallelism natively

- Exploring modifications to Sacado, Kokkos to map GPU thread parallelism across derivative calculations

- Two approaches are being explored:
  - Create thread-local, derivative-strided subviews
    - SFad<double,32> becomes SFad<double,1> (per thread)
  - Directly reference thread index in overloaded operators
    - Requires efficient memory pool to manage allocation of temporaries