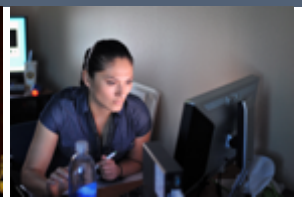
SAND2020-13188C

Sandia National Laboratories

Finite Element Tools for Performance Portability of Implicit and IMEX Simulations on Next Generation Architectures



PRESENTED BY

Roger P. Pawlowski, Eric T. Phipps, C. R. Trott, John N. Shadid and Eric C. Cyr

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.

Motivating Example: Fast/Stiff/Implicit Modes in Plasma Physics

- Application codes are significant investments and need to cover a wide variety of problems
  - Step over fast modes to simulate time scales of interest: Implicit or IMEX schemes
- 5-Moment Plasma Model
  - Speed of light arises from coupling of electromagnetic field: explicit CFL $\sim c\Delta t/\Delta x$
  - Plasma oscillation arises from Ampere's law to momentum conservation: explicit CFL $\sim \Delta t$
  - Collisions explicit CFL$\sim \Delta t$
  - Cyclotron frequency explicit CFL$\sim |\mathrm{B}|\Delta t$

$$\frac{\partial \rho_\alpha}{\partial t} + \nabla \cdot (\rho_\alpha \mathbf{u}_\alpha) = \sum_{\text{srcs}} m_\alpha \Gamma^{\text{src}} - \sum_{\text{sinks}} m_\alpha \Gamma^{\text{sink}}$$

$$\frac{\partial (\rho_\alpha \mathbf{u}_\alpha)}{\partial t} + \nabla \cdot (\rho_\alpha \mathbf{u}_\alpha \otimes \mathbf{u}_\alpha + p_\alpha I + \Pi_\alpha) = \frac{q_\alpha}{m_\alpha} \rho_\alpha (\mathbf{E} + \mathbf{u}_\alpha \times \mathbf{B})$$

$$+ \sum_{\text{srcs}} m_\alpha \mathbf{u}_{\text{src}} \Gamma^{\text{src}} - \sum_{\text{sinks}} m_\alpha \mathbf{u}_\alpha \Gamma^{\text{sink}} + \sum_{\beta \neq \alpha} \mathbf{R}^{\alpha,\beta}$$

$$\frac{\partial \mathbf{E}}{\partial t} - c^2 \nabla \times \mathbf{B} = -\frac{1}{\epsilon_0} \sum_\alpha \frac{q_\alpha}{m_\alpha} \rho_\alpha \mathbf{u}_\alpha$$

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \times \mathbf{E} = 0$$

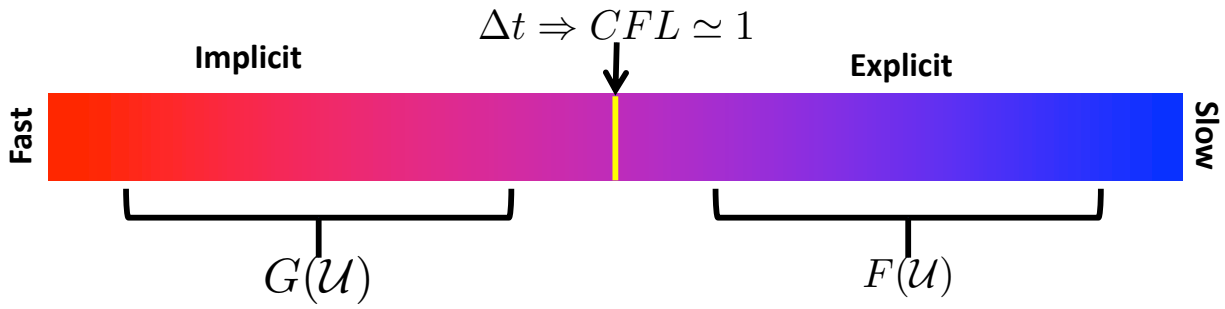
Stiff Modes:

- Speed of light
- Plasma Oscillation
- Collisions
- Cyclotron frequency

Motivation

- Implicit/IMEX Solution Methods
  - Large-scale Newton-based solvers leveraging iterative linear solvers (GMRES, CG)
  - Requires Jacobian-vector product, Jacobian sensitivities for preconditioning
  - Combine with block/physics-based preconditioning for implicit solves (Algebraic Multigrid within blocks)

- IMEX methods split fast and slow modes
  - Implicit terms solve for stiff modes (plasma oscillation, speed of light)
  - Explicit terms are accurately resolved
  - IMEX assumes an additive decomposition: $\dot{\mathcal{U}} + F(\mathcal{U}) + G(\mathcal{U}) = 0$



- Requirements:
  - Performance Portability
  - Keep the code base manageable: write physics once in a clear and concise manner
  - Move terms from explicit to implicit depending on time scale of interest (evaluate sensitivities)

Requirement: Performance Portable Automatic Differentiation

Trilinos Discretization Tools Overview

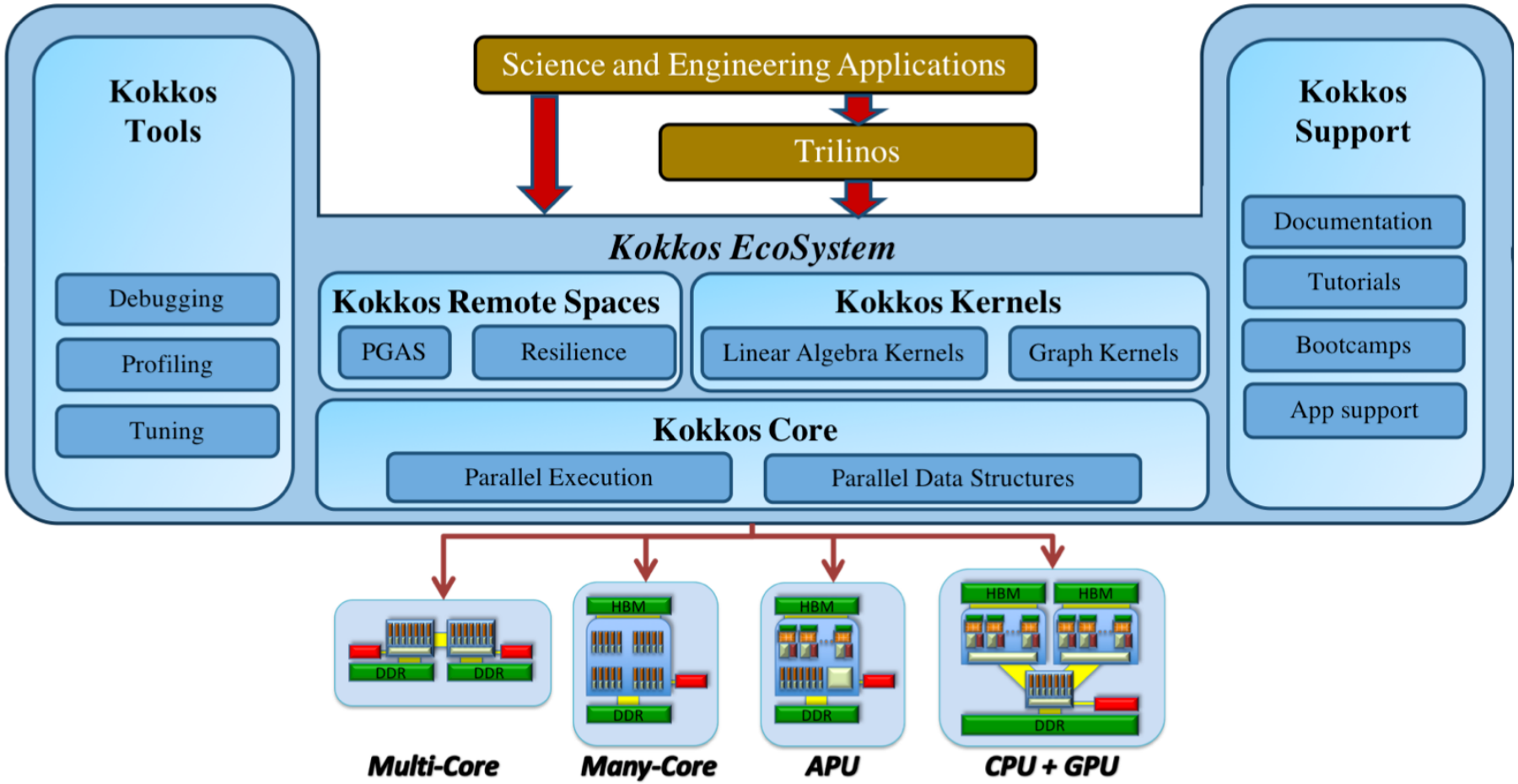
- MPI Related
  - Panzer (Multiphysics Assembly and Utilities)
    - DOF Manager: Global Indexing for mixed bases, mixed equations
    - Connection Manager: Mesh DB abstraction
    - Workset Builder: Mesh over-decomposition for AMT
    - Linear Algebra Builder: Epetra/Tpetra/Thyra
    - Disc-FE: Multiphysics assembly, Mixed Eq Sets, Mixed Bases, BCs, Compatible discretizations, Projections
- Local Node
  - Intrepid2: FE Basis Library
  - Shards: Cell/Element Topology
  - Phalanx: DAG Assembly: flexibility/complexity
  - Sacado: Automatic differentiation scalar types
  - Kokkos: Performance portability

github.com/trilinos/Trilinos

Performance Portable Automatic Differentiation with Kokkos, Sacado and Phalanx

Performance Portability: Kokkos

- Write algs once, run on many architectures: e.g. multi-core CPU, Nvidia GPU, Xeon Phi, AMD GPU, ARM, …
- Performance Portable Shared-memory Programming Model in C++
- Supports clear, concise, thread-scalable parallel patterns
  - for, reduce, scan, task
- Multidimensional array with compile-time polymorphic layouts.
- Kokkos devs are on the c++ standards committee
  - C++23 std::mdspan is Kokkos::View



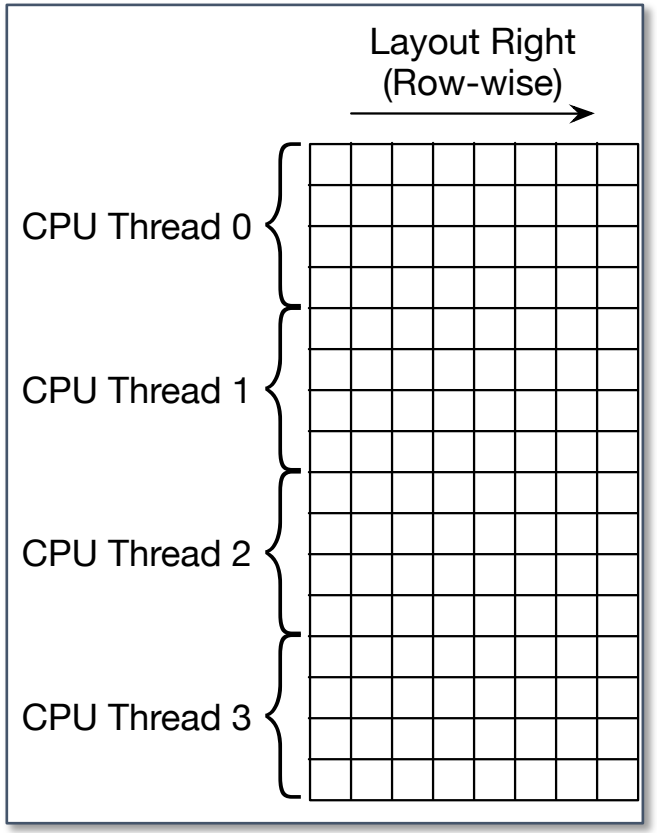
https://github.com/kokkos/kokkos

Layouts for Performant Memory Access

```
View<double**> A("A", m, n);
```

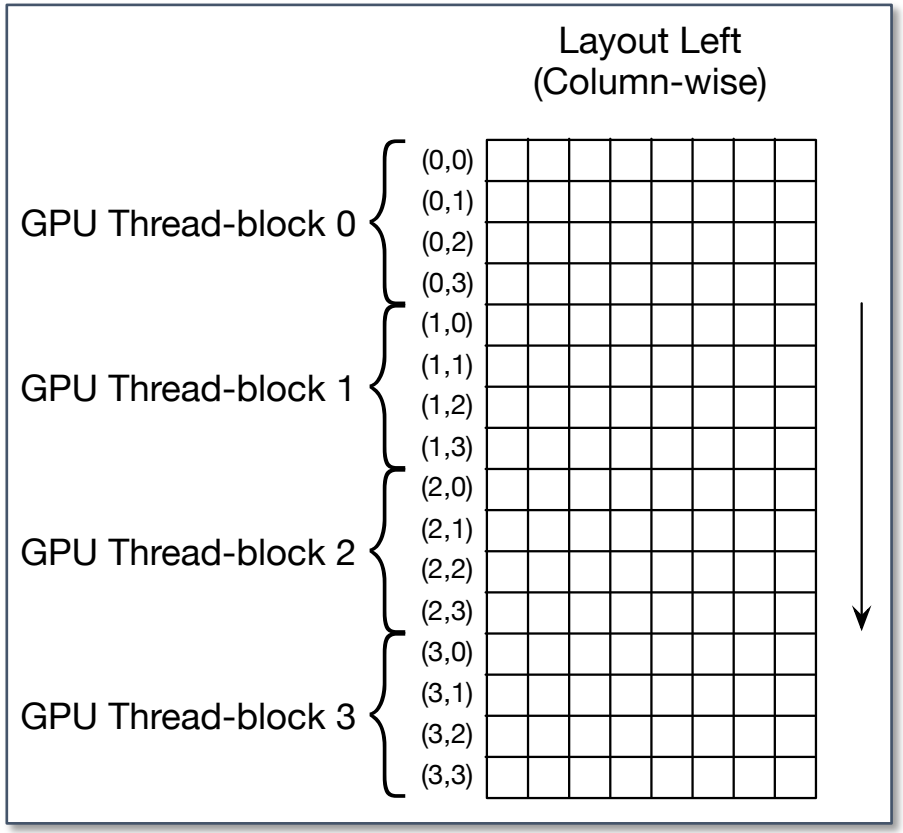
CPU/MIC

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



GPU

- **Each thread accesses strided range of entries**
- **Thread group can read all values in one memory transaction**
- **Ensures coalesced accesses (consecutive threads access consecutive entries)**



Example: Dense Matrix-vector Multiply

- Kokkos::View has shared pointer semantics
- Allocates memory in a MemorySpace
- Executes Kernel using an ExecutionPolicy coupled to an ExecutionSpace.
- Execution policy determines parallel distribution of work to threads
- ExecutionSpace is where to run the kernel

```
Kokkos::View<double**> A("A",m,n); // Create rank-2 array with m rows and n columns
                                   // using default execution space and layout
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
```

```
// Function implementing dense matrix-vector product.
// The function is templated on the types of views storing A, b, and c
template <typename ViewTypeA, typename ViewTypeB, typename ViewTypeC>
void matvec(const ViewTypeA& A, const ViewTypeB& b, const ViewTypeC& c) {

  // The scalar type used in this calculation, e.g., double
  typedef typename ViewTypeC::value_type scalar_type;

  // The best ordinal type for the architecture we are running on, e.g., int or size_t
  typedef typename ViewTypeC::size_type size_type;

  // The execution space where this functor will run
  typedef typename ViewTypeC::execution_space execution_space;

  // Matrix dimensions
  const size_type m = A.extent(0);
  const size_type n = A.extent(1);

  Kokkos::RangePolicy<execution_space> policy(0,m)
  Kokkos::parallel_for("MatVec", policy, KOKKOS_LAMBDA(const size_type i)
  {
    scalar_type t = 0.0;
    for (size_type j=0; j<n; ++j)
      t += A(i,j)*b(j);
    c(i) = t;
  });
}

// Execute matrix-vector product for given views A, b, and c
matvec( A, b, c );
```

Example: Dense Matrix-vector Multiply

Architecture Name	Architecture Description	Execu-tion Space	Compiler	Bandwidth Measured (GB/s)
Skylake	Single-socket 3.0 GHz, Intel Xeon Gold 6154 CPU, 18 cores, 2 threads/core	OpenMP	Intel 19.0	64.4
GPU	NVIDIA V100 GPU	CUDA	NVCC 10.1	833

Architecture Name	Expected Throughput	Measured Throughput	Throughput with Wrong Layout	
Skylake	16.1	18.0	15.3	GFLOP/s
GPU	208	213	26.3	

Memory layouts have a critical impact on performance!

Throughput measured using STREAM Triad Benchmark:
McCalpin, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, 1995

Sacado: Template-based Automatic Differentiation (AD)

- Implement equations templated on the scalar type
- Libraries provide new scalar types that ***overload the math operators*** to propagate embedded quantities
  - Hidden derivative array
  - Uses chain rule for internal derivatives
  - Expression templates for performance
- Analytic Values (NO finite differencing involved)!
- Algorithms: Forward mode (FAD) and reverse mode (RAD)
  - **SFAD**: Static FAD - compile time derivative length
  - **DFAD**: Dynamic FAD – runtime derivative length
  - **SLFAD**: In between SFAD and DFAD – compile time MAX derivative length, runtime sets true length
- For various AD tools, see www.autodiff.org

`double`	`Fad<double>`
Operation	Forward AD rule
$c = a \pm b$	$\dot{c} = \dot{a} \pm \dot{b}$
$c = ab$	$\dot{c} = a\dot{b} + \dot{a}b$
$c = a/b$	$\dot{c} = (\dot{a} - c\dot{b})/b$
$c = a^r$	$\dot{c} = ra^{r-1}\dot{a}$
$c = \sin(a)$	$\dot{c} = \cos(a)\dot{a}$
$c = \cos(a)$	$\dot{c} = -\sin(a)\dot{a}$
$c = \exp(a)$	$\dot{c} = c\dot{a}$
$c = \log(a)$	$\dot{c} = \dot{a}/a$

```
template <typename ScalarT>
void computeF(ScalarT* x, ScalarT* f)
{
  f[0] = 2.0 * x[0] + x[1] * x[1];
  f[1] = x[0] * x[0] * x[0] + sin(x[1]);
}
```

Residual

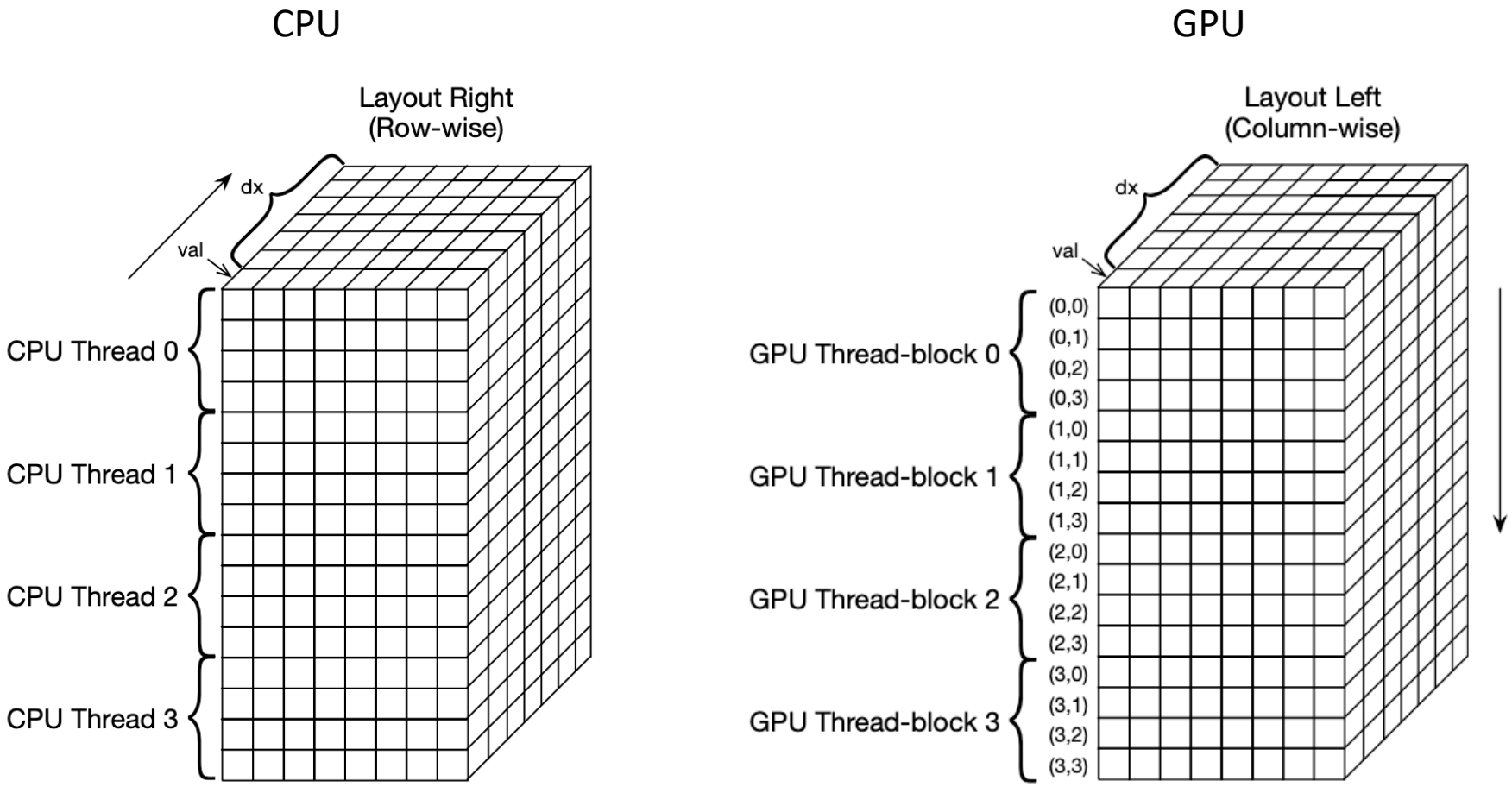
```
double* x;
double* f;
…
computeF(x,f);
```

Jacobian

```
DFad<double>* x;
DFad<double>* dfdx;
…
computeF(x,dfdx);
```

AD performance without modifications to Kokkos kernels

```
View<DFad<double>**> A("A", m, n);
```



- Achieved by specializing Kokkos::View data structure for Sacado scalar types
  - Rank-r Kokkos::View is internally stored as a rank-(r+1) array of doubles
  - Specialized Kokkos layout applied to internal rank-(r+1) array for coalesced access on GPU (derivative values are strided, not consecutive on GPU)

AD Matrix-Vector Multiply

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

// ...

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

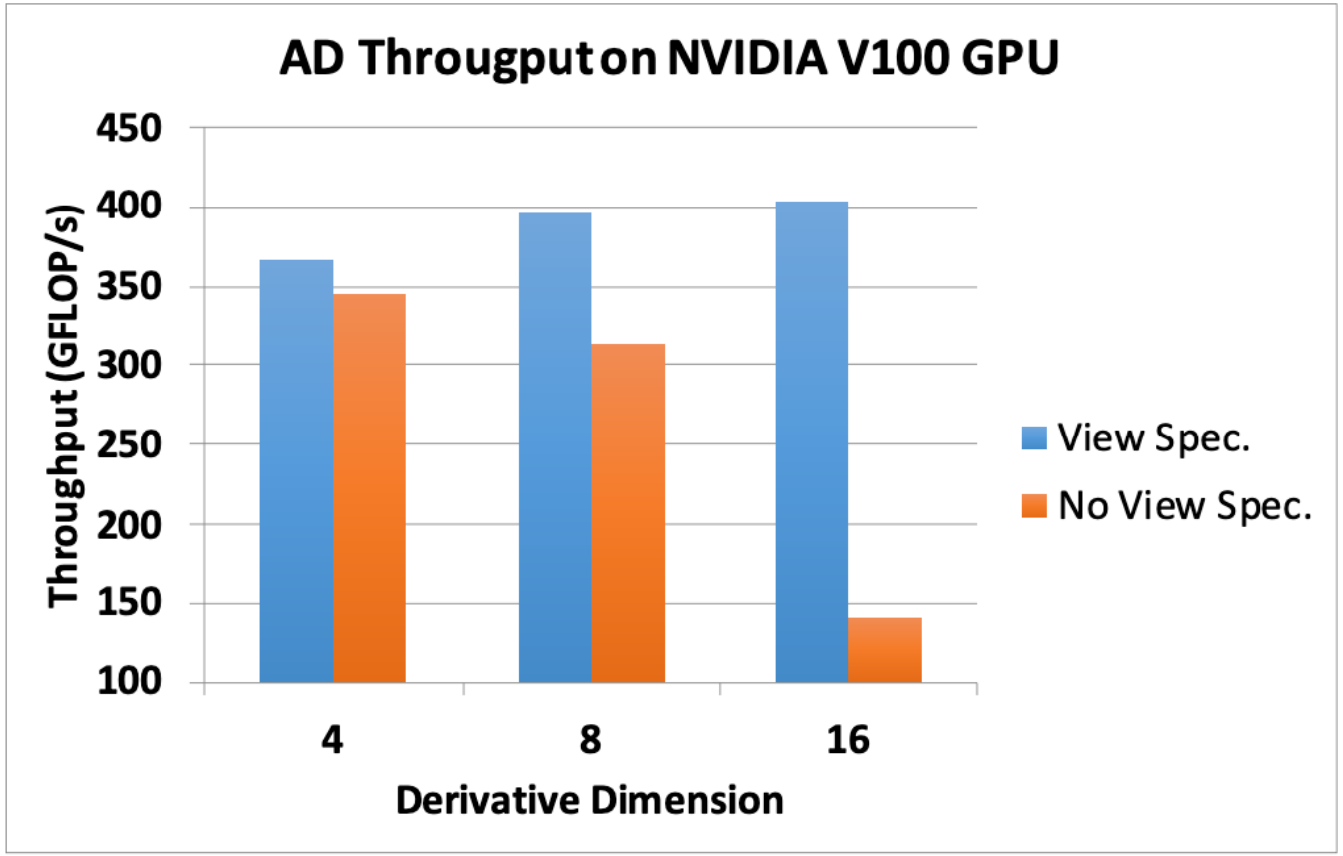
Architecture Name	Expected Throughput	Measured Throughput	No View Specialization
Skylake	30.4	34.1	34.0
GPU	393	395	317

Derivative dimension: p = 8

Throughput measured using STREAM Triad Benchmark:
McCalpin, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, 1995

Layout Impact on Derivative Array Size

- Performance loss from incorrect layout is more apparent for larger derivative sizes
- Small derivative size might get multiple values in same memory transaction (more cache hits)



Hierarchic Parallelism

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

- Flat parallelism is performant IF you can saturate the GPU with work.
- Not always possible especially for implicit methods.
- Parallelism
  - Flat: one thread per cell
  - Hierarchic: Team of threads work on each Cell/FAD entry
- Three level hierarchy:
  - League
  - Team (CUDA thread block)
  - Vector (warp/hyperthreads)

Hierarchic Diffusion/Rxn Kernel for “double” scalar type

```
typedef double ScalarT;
typedef Kokkos::LayoutRight Layout;
typedef Kokkos::TeamPolicy<ExecSpace> Policy;

Kokkos::View<ScalarT****, Layout, ExecSpace> grad_phi;
Kokkos::View<ScalarT***,  Layout, ExecSpace> f;
Kokkos::View<ScalarT***,  Layout, ExecSpace> phi;
Kokkos::View<ScalarT**,   Layout, ExecSpace> s;
Kokkos::View<ScalarT**,   Layout, ExecSpace> r;

const int num_cell   = grad_phi.extent(0);
const int num_basis  = grad_phi.extent(1);
const int num_points = grad_phi.extent(2);
const int num_dim    = grad_phi.extent(3);
Kokkos::parallel_for(
  Policy( num_cell, 8, 1 ), // League size == num_cell, team size == 8, vector size == 1
  KOKKOS_LAMBDA( const typename Policy::member_type& team )
  {
    const int cell = team.league_rank();
    const int team_index = team.team_rank();
    const int team_size = team.team_size();
    ScalarT value, value2;
    for (int basis=team_index; basis<num_basis; basis+=team_size) {
      value = 0.0; value2 = 0.0;
      for (int qp=0; qp<num_points; ++qp) {
        for (int dim=0; dim<num_dim; ++dim)
          value += f(cell,qp,dim)*grad_phi(cell,basis,qp,dim);
        value2 += s(cell,qp)*phi(cell,basis,qp);
      }
      r(cell,basis) = value+value2;
    }
  });
```

Hierarchic AD Kernel

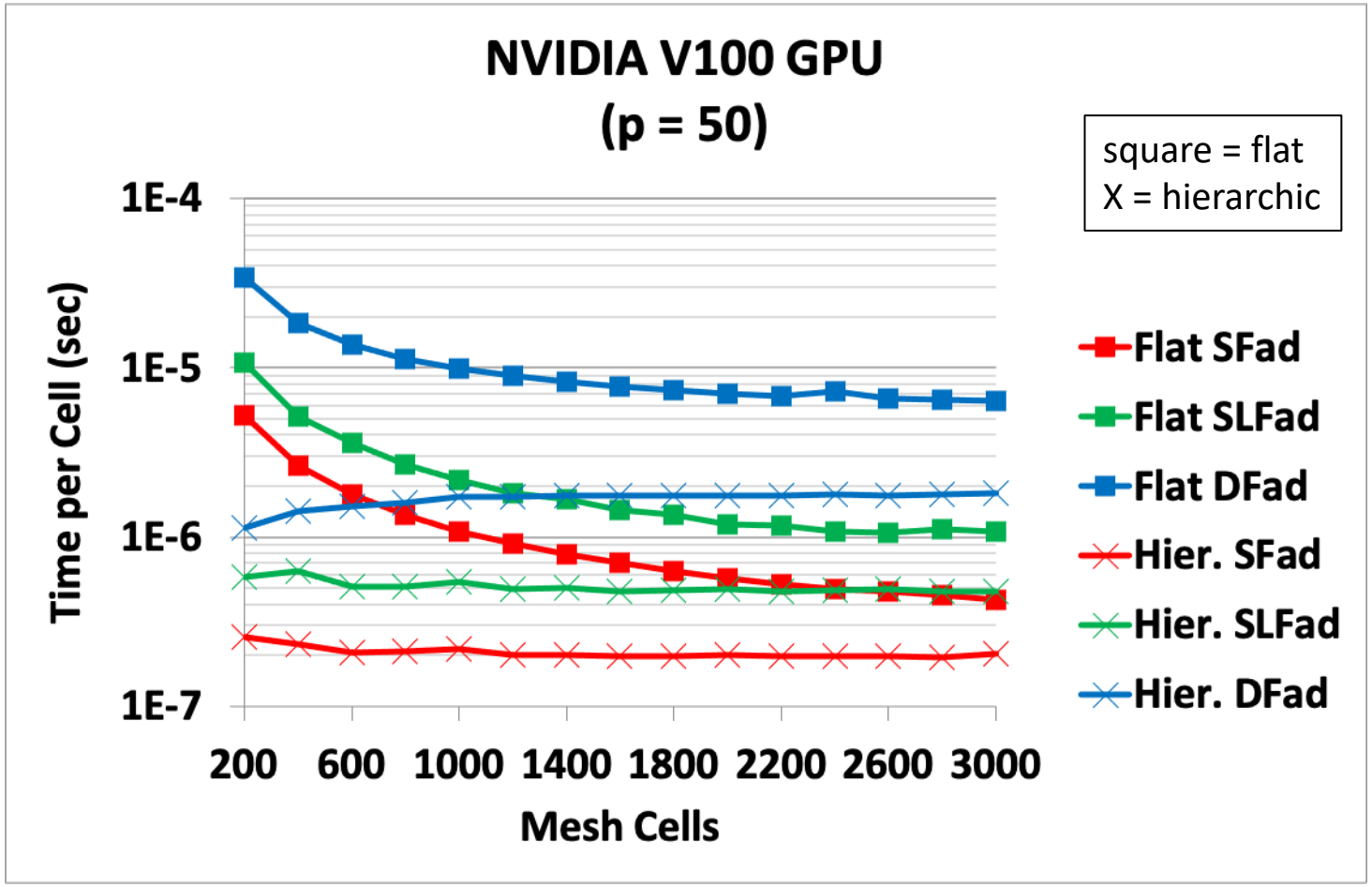
- Sacado FAD → parallelize over derivative dimension
- Need a specialized Layout for contiguous derivative values in the View
- Declare the layout explicitly:
  - Allows using both flat and hierarchic kernels in the same code base!
  - If we don't declare, uses layout for flat parallelism
- Layout construction with vector striding shown for clarity
  - Set based on node architecture at compile time.
- Vector loop is handled internally in Sacado FAD type

Hierarchic Diffusion/Rxn Kernel for "DFad<double>" scalar type

```
typedef Sacado::Fad::SFad<double,p> ScalarT;
typedef Kokkos::LayoutRight Layout;
const int Stride = 32;
typedef Kokkos::LayoutContiguous<Layout,Stride> ContLayout;

typedef Kokkos::TeamPolicy<ExecSpace> Policy;

Kokkos::View<ScalarT****, ContLayout, ExecSpace> grad_phi;
Kokkos::View<ScalarT***,  ContLayout, ExecSpace> f;
Kokkos::View<ScalarT***,  ContLayout, ExecSpace> phi;
Kokkos::View<ScalarT**,   ContLayout, ExecSpace> s;
Kokkos::View<ScalarT**,   ContLayout, ExecSpace> r;

// Typename of local scalars within the kernel
typedef typename Kokkos::ThreadLocalScalarType<decltype(s)>::type local_scalar_type;

const int num_cell   = grad_phi.extent(0);
const int num_basis  = grad_phi.extent(1);
const int num_points = grad_phi.extent(2);
const int num_dim    = grad_phi.extent(3);
Kokkos::parallel_for(
  Policy( num_cell, 256/Stride, Stride ), // League size == num_cell, team size == 8,
                                          // vector size == 32
  KOKKOS_LAMBDA( const typename Policy::member_type& team )
  {
    const int cell = team.league_rank();
    const int team_index = team.team_rank();
    const int team_size = team.team_size();
    local_scalar_type value, value2;
    for (int basis=team_index; basis<num_basis; basis+=team_size) {
      value = 0.0; value2 = 0.0;
      for (int qp=0; qp<num_points; ++qp) {
        for (int dim=0; dim<num_dim; ++dim)
          value += f(cell,qp,dim)*grad_phi(cell,basis,qp,dim);
        value2 += s(cell,qp)*phi(cell,basis,qp);
      }
      r(cell,basis) = c*(value+value2);
    }
  });
```

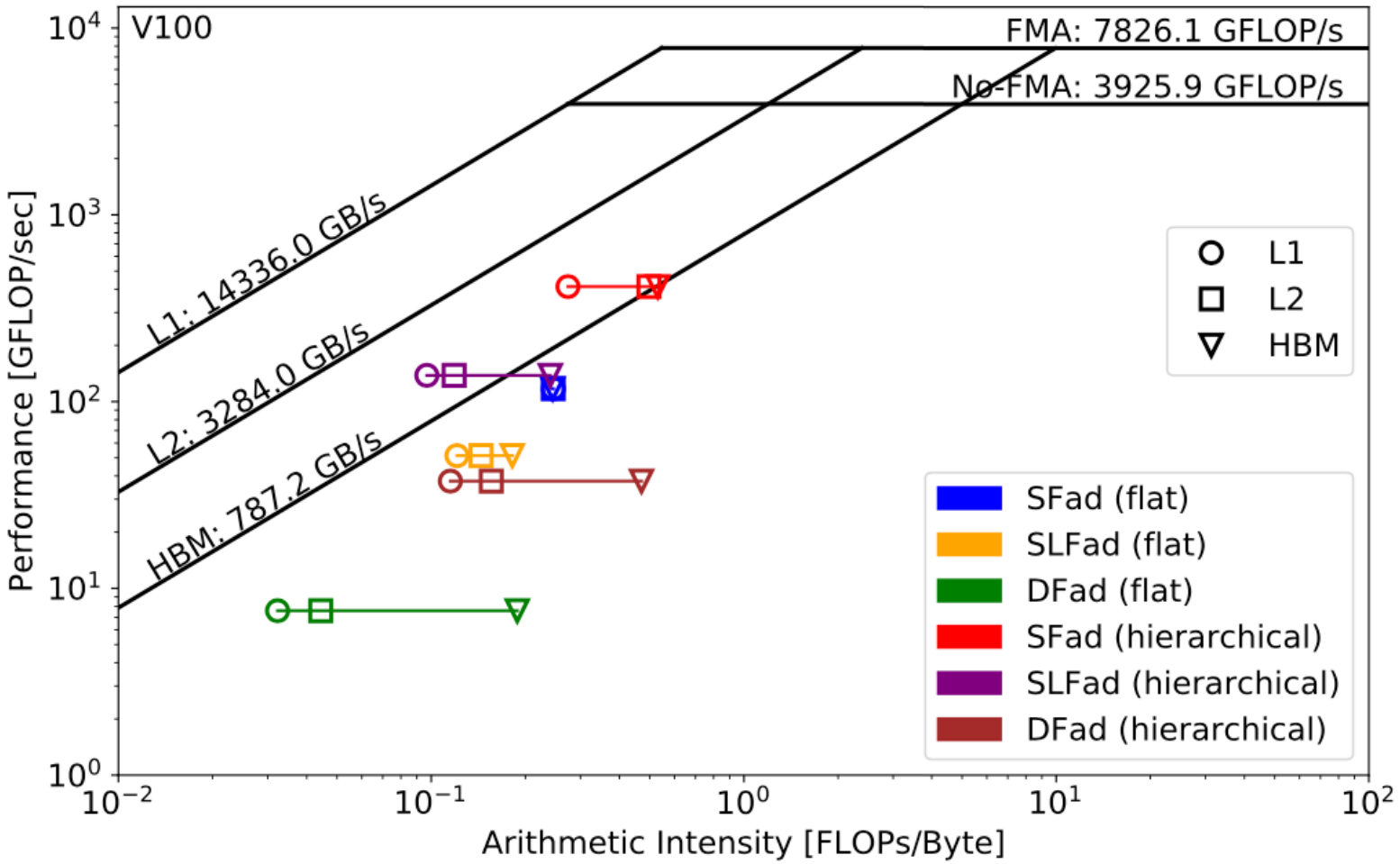
Flat vs Hierarchic Parallelism

- Hierarchic is significantly more performant for our use case
- Static FAD implementations are more performant



Hierarchic Roofline (Nvidia V100)

- Roofline:
  - Triangles: HBM bound
  - Squares: L2 bound
  - Circles: L1 bound
- SFad and SLFad are memory bound



Empirical Roofline Toolkit: C. Yang, et. al., 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC , 2018.
https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/software/ert
NOTE: L1 is theoretical limit – not in ERT tool at time of runs

Example: Mixed-Basis Poisson Finite Element Assembly

- Same operators used in plasma code
- Least squares Finite Element discretization
- Mixed H-Div and H-Grad formulation
- Assess volumetric operator performance

$$\nabla^2 \phi = f$$

$$\nabla \phi = \mathbf{g}$$

$$\phi \in \mathcal{H}_{(\nabla)}$$

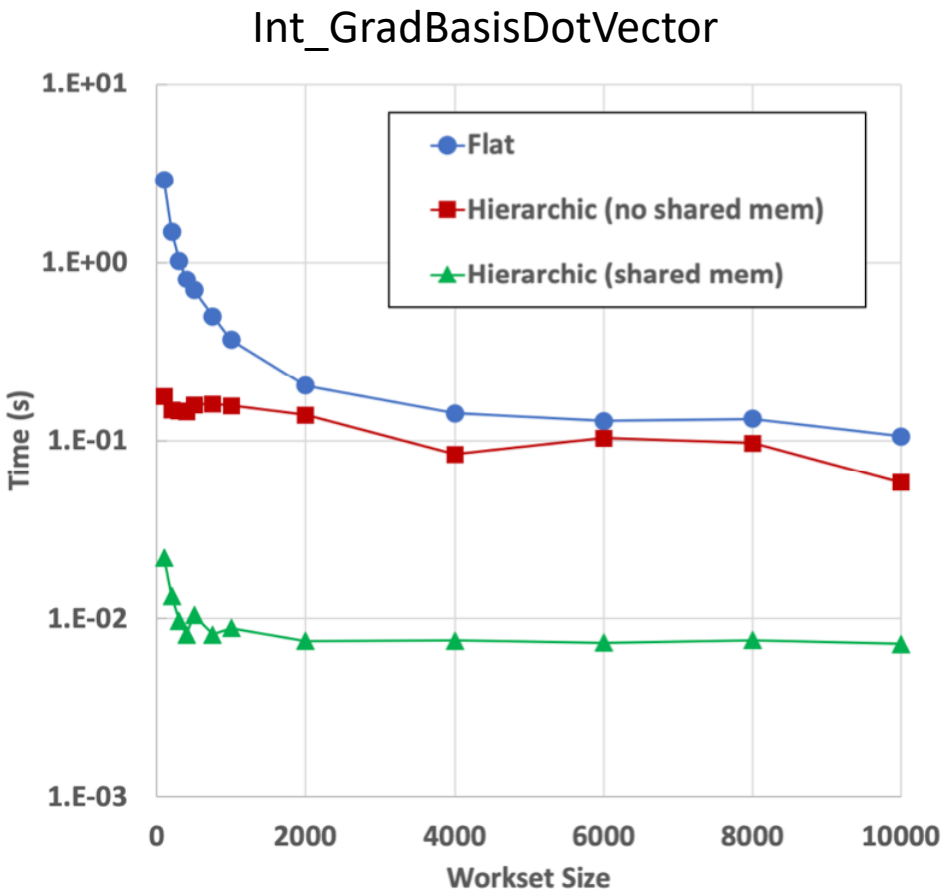
$$\mathbf{g} \in \mathcal{H}_{(\nabla \cdot)}$$

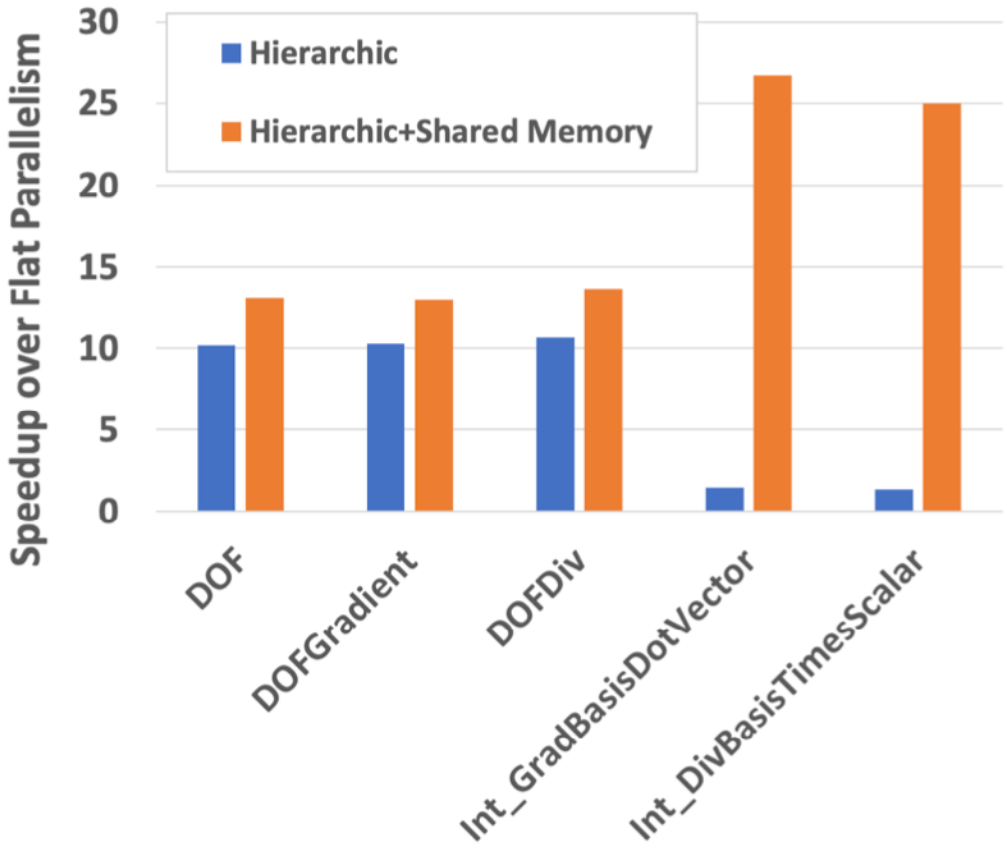
$$\int_\Omega (\nabla \cdot \mathbf{g} - f)(\nabla \cdot \mathbf{w}) d\Omega = 0 \quad \forall \mathbf{w} \in \mathcal{H}_{(\nabla \cdot)}$$

$$\int_\Omega (\nabla \phi - \mathbf{g}) \cdot (\nabla q) d\Omega = 0 \quad \forall q \in \mathcal{H}_{(\nabla)}$$

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

Performance: Nvidia V100

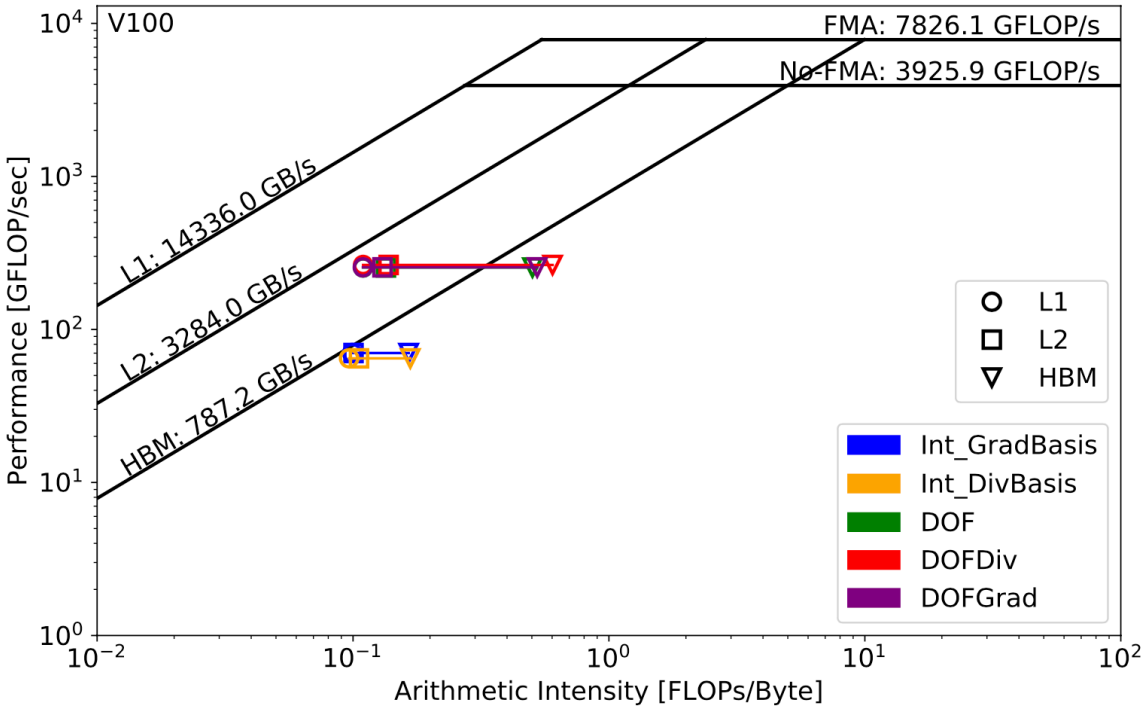




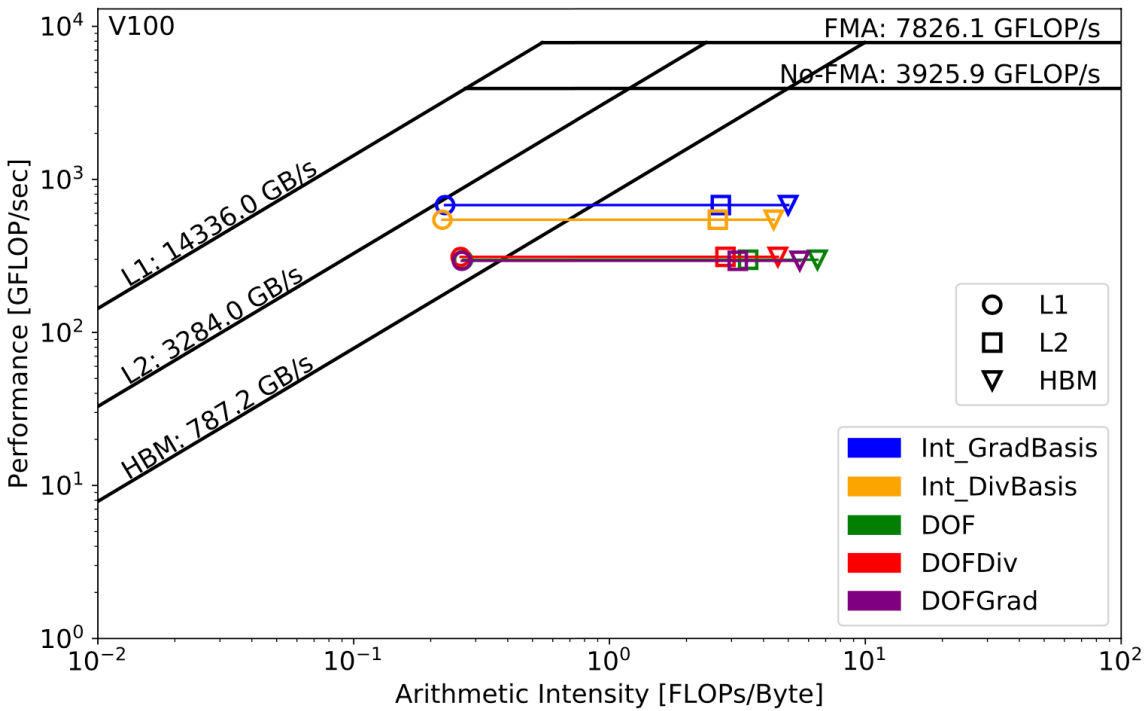
- Shared memory can have a significant impact on performance
- Shared memory size is very limited: example limited to 2nd order bases

Hierarchic Roofline

No Shared Memory



With Shared Memory



NOTE: all runs use the DFad AD type

Significant speedup, but shared memory not perfectly performance bound due to mixed scalar type.

Empirical Roofline Toolkit: C. Yang, et. al., 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC , 2018.
https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/software/ert
NOTE: L1 is theoretical limit – not in ERT tool at time of runs

Complexities

- References to individual values have to be handled in a specific way
- Shared memory requires type-specialization for FAD sizing
- Specializing FAD-only for hierarchic layouts in a general kernel
  - MPL to choose layout based on scalar type
- Kokkos supports use of lambdas, but bug in gcc 5/6 caused compiler failure (fixed in gcc7) in nested lambdas.
- Explicit template instantiation for reduced compile times.

```
View<Scalar**,...> a;
...
parallel_for(...
  ...
  const Scalar& tmp = a(i,j); // compiler error for AD types!
  ...
});
```

```
View<Scalar**,...> a;
using ref_type = View<Scalar**,...>::reference_type;
...
parallel_for(...
  ...
  const ref_type tmp1 = a(i,j);
  const auto tmp2 = a(i,j);
  ...
});
```

```
using scratch_view = View<ScalarT*,
                          typename DevLayout<ScalarT>::type,
                          typename PHX::exec_space::scratch_memory_space,
                          Kokkos::MemoryUnmanaged>;
scratch_view tmp_field;
if (Sacado::IsADType<ScalarT>::value) {
  const int fadSize = Kokkos::dimension_scalar(field_.get_view());
  tmp_field = scratch_view(team.team_shmem(),numBases,fadSize);
}
else {
  tmp_field = scratch_view(team.team_shmem(),numBases);
}
```

Summary

- AD is a powerful tool for writing a flexible code base for using operators in both explicit and implicit modes
- Performance portability requires careful attention to memory layouts
  - Hidden dimensions in AD makes this more complex
- Hierarchic parallelism significantly improves performance for our use case
  - Comes with extra complexity, potential for race conditions
- Comment: Converting code to be performance portable is application specific.
  - Kokkos team experience: 50% can do simple flat parallelism, 30 % need hierarchic, 20% need algorithmic specialization.