Exceptional service in the national interest

Sandia National Laboratories

June 29, 2023 EuroTUG

Intrepid2 for Fast High-Order Assembly

Nathan V. Roberts
nvrober@sandia.gov
Sandia National Laboratories
SAND 2023-05417C

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

Outline

Intrepid2

Intrepid2 provides tools for finite element (/volume) computations:

- high-order basis functions computed on a reference element for the whole exact sequence: H^1, $\mathrm{H}(\text{curl})$, $\mathrm{H}(\text{div})$, L^2
- Jacobians of the reference-to-physical transformation
- pullbacks from reference to physical element
- projections into finite element function spaces

Structure Preservation

Sandia National Laboratories

Typical high-level FEM codes ignore or discard structure in order to maintain generality.

Structure Preservation

Typical high-level FEM codes ignore or discard structure in order to maintain generality.

Example: using the standard Intrepid2 interface, if you want Jacobians on an affine grid, you compute and store these at each quadrature point, in a multi-dimensional array (a Kokkos View) with shape (C,P,D,D). This is wasteful, and waste grows with polynomial order and number of spatial dimensions.

Structure Preservation

Typical high-level FEM codes ignore or discard structure in order to maintain generality.

Example: using the standard Intrepid2 interface, if you want Jacobians on an affine grid, you compute and store these at each quadrature point, in a multi-dimensional array (a Kokkos View) with shape (C,P,D,D). This is wasteful, and waste grows with polynomial order and number of spatial dimensions.

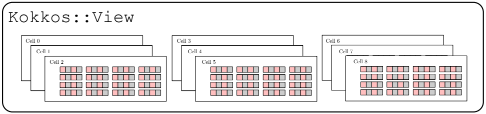
By contrast, a custom implementation could store the same Jacobians in a (C,D,D) array. For a uniform grid, this reduces to an array of length (D).

Structure Preservation

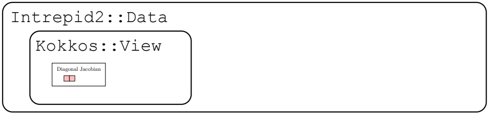
The new Intrepid2 Data class is a starting point for addressing this. It stores just the unique data, but presents the same functor interface as the standard View.

Suppose we have a uniform, affine 2D mesh:

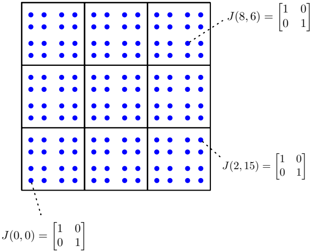
Old way: 4 doubles per Jacobian per point per cell.



New way: 2 doubles.



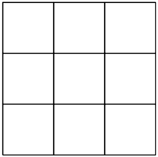
Same access pattern for both old and new:



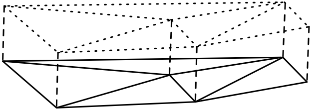
Our interest is not primarily in reducing storage costs, but in enabling structure-aware algorithms, such as sum factorization.

CellGeometry Mesh Types

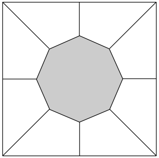
We added a `CellGeometry` class to Intrepid2, supporting the following:



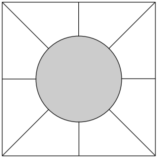
(a) Uniform Grid



(b) Extruded Mesh[1]



(c) First-Order Mesh



(d) Higher-Order Mesh

Among other features, `CellGeometry` supports computing Jacobians in an efficient, structure-preserving way.

[1] Implementation incomplete.

Design Goal: Accessible, Simple API

- The new interfaces should look like the old interfaces.
- The user should not generally need to invoke particular algorithms directly; these should be implied by e.g. the geometric context that is provided.

Old code:

```
using DRV = Kokkos::DynRankView<double,DeviceType>;
DRV points(P,D);
DRV weights(P);
cubature->getCubature(points, weights);
```

New code:

```
auto points  = cubature->allocateCubaturePoints();  // TensorPoints<double,DeviceType>
auto weights = cubature->allocateCubatureWeights(); // TensorData<double,DeviceType>
cubature->getCubature(points, weights);
```

Motivation: Sum Factorization

Assembly/Evaluation Costs[1]

	Storage	Assembly	Evaluation
Full Assembly + matvec	$O(p^{2d})$	$O(p^{3d})$	$O(p^{2d})$
Sum-Factorized Full Assembly + matvec	$O(p^{2d})$	$O(p^{2d+1})$	$O(p^{2d})$
Partial Assembly + matrix-free action	$O(p^{d})$	$O(p^{d})$	$O(p^{d+1})$

For hexahedral elements in 3D:

- standard assembly: $O(p^9)$ flops
- sum factorization: $O(p^7)$ flops in general; $O(p^6)$ flops in special cases.
- partial assembly: $O(p^4)$ flops (but need matrix-free solver)

Savings increase for higher dimensions...
Basic idea: save flops by factoring sums.

	Adds	Multiplies	Total Ops
$\sum_{i=1}^{N}\sum_{j=1}^{N} a_i b_j$	$N^2 - 1$	N^2	$2N^2 - 1$
$\sum_{i=1}^{N} a_i \sum_{j=1}^{N} b_j$	$2N - 2$	N	$3N - 2$

[1] Table 1 in Anderson et al, MFEM: A modular finite element methods library. doi: 10.1016/j.camwa.2020.06.009.

Intrepid2's Basis Class

- Principal method: `getValues()` — arguments: points, operator, Kokkos View for values
- Fills View with shape (P) or (P,D) with basis values at each ref. space quadrature point.

Structure has been lost:

- points: flat container discards tensor structure of points.
- values: each basis value is the product of tensorial component bases; we lose that by storing the value of the product.

Both points and values will generally require (a lot) more storage than a structure-preserving data structure would allow.

But our major interest is in supporting algorithms that take advantage of structure: we add a `getValues()` variant that accepts a `BasisValues` object (see next slide).

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.
- `TensorData`: tensor product of `Data` containers; allows storage of tensor-product basis evaluations such as those from H^1 value basis evaluation.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.
- `TensorData`: tensor product of `Data` containers; allows storage of tensor-product basis evaluations such as those from H^1 value basis evaluation.
- `VectorData`: vectors of `TensorData`, possibly with multiple families defined within one object. Allows storage of vector-valued basis evaluations.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.
- `TensorData`: tensor product of `Data` containers; allows storage of tensor-product basis evaluations such as those from H^1 value basis evaluation.
- `VectorData`: vectors of `TensorData`, possibly with multiple families defined within one object. Allows storage of vector-valued basis evaluations.
- `TensorPoints`: tensor point container defined in terms of component points.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.
- `TensorData`: tensor product of `Data` containers; allows storage of tensor-product basis evaluations such as those from H^1 value basis evaluation.
- `VectorData`: vectors of `TensorData`, possibly with multiple families defined within one object. Allows storage of vector-valued basis evaluations.
- `TensorPoints`: tensor point container defined in terms of component points.
- `BasisValues`: abstraction from `TensorData` and `VectorData`; allows arbitrary reference-space basis values to be stored.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.
- `TensorData`: tensor product of `Data` containers; allows storage of tensor-product basis evaluations such as those from H^1 value basis evaluation.
- `VectorData`: vectors of `TensorData`, possibly with multiple families defined within one object. Allows storage of vector-valued basis evaluations.
- `TensorPoints`: tensor point container defined in terms of component points.
- `BasisValues`: abstraction from `TensorData` and `VectorData`; allows arbitrary reference-space basis values to be stored.
- `TransformedBasisValues`: `BasisValues` object alongside a transformation matrix, stored in a `Data` object, that maps it to physical space.

Sample Code

See `intrepid2/assembly-examples` for sample implementations of assembly on hexahedral meshes:

- Assembly of norm matrices for $\mathrm{H}^1, \mathrm{H}(\text{curl}), \mathrm{H}(\text{div}), \mathrm{L}^2$.
- Examples for both old and new data structures.
- Invoked by `StructuredIntegrationPerformance` test driver, which we used to generate timings we'll discuss later.

New Basis Implementations

`DerivedBasisFamily` is so named because tensor-product element bases are *derived* from bases on lower-dimensional geometries.

- New nodal bases that can output to `BasisValues` for quads, hexahedra, and wedges. High-order wedges are available for the first time.
- New family of high-order, hierarchical bases taken from work by Leszek Demkowicz's group at UT Austin; these also output to `BasisValues`. Simplices, quads, hexahedra, and wedges implemented; pyramids planned.
- Support for hyper-dimensional (up to 7D) hypercube H^1 and L^2 bases:
  - `getHypercubeBasis_HGRAD(polyOrder, spaceDim)`
  - `getHypercubeBasis_HVOL(polyOrder, spaceDim)`
- Support for Serendipity Bases: sub-bases of the hierarchical bases.
- Tensor-product bases support *anisotropic* polynomial order.

New Basis Implementations

```
auto basis = getBasis< BasisFamily >(cellTopo, fs, polyOrder);
```

- New `BasisFamily` pattern allows basis construction from cell topology, function space, and poly. order.
- Included `BasisFamily`'s:
  - `NodalBasisFamily` (classic Intrepid2 bases)
  - `DerivedNodalBasisFamily` (structure-supporting variant of nodal bases)
  - `HierarchicalBasisFamily`
  - `DGHierarchicalBasisFamily` (all dofs interior; for H^1, there is a constant member)
  - `SerendipityBasisFamily`
  - `DGSerendipityBasisFamily`

Sum Factorization Implementation

- Sum factorization takes advantage of tensor-product structure in finite element bases to reduce the cost of FE assembly in N dimensions from $\mathrm{O}(\mathrm{p}^{3\mathrm{N}})$ to $\mathrm{O}(\mathrm{p}^{2\mathrm{N}+1})$.
- Theoretical speedup for hexahedra (3D): $\mathrm{O}(\mathrm{p}^2)$.
- We implement sum factorized `integrate()` with two core kernels: one generic to the dimension, and one $\mathrm{N} = 3$ specialization.
- Both implementations are agnostic to architecture as well as function space.

Sum Factorization Performance Comparison

Performance comparison between standard Intrepid2 and sum-factorized assembly:

- We assemble the so-called *Gram matrix* for $\mathrm{H}^1, \mathrm{H}(\mathrm{curl}), \mathrm{H}(\mathrm{div}), \mathrm{L}^2$ function spaces, with hexahedral element counts from 16 (for $p = 10$) up to 32,768 (for $p = 1$).
- Workset sizes are determined experimentally; we use the best choice for each algorithm.
- We estimate flop counts for each algorithm, and use timings to derive a throughput estimate.

Intrepid2 Sum Factorization: Serial Speedups

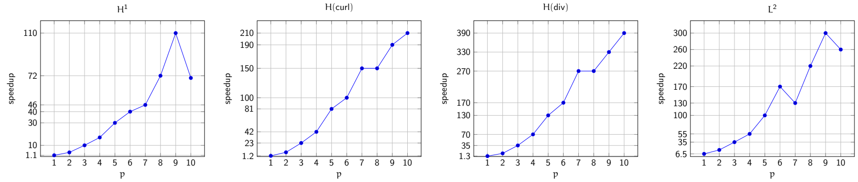


Figure: Serial (28-core 2.5 GHz Xeon W) speedups compared to standard assembly for H^1, $H(\text{curl})$, $H(\text{div})$, and L^2 norms on hexahedra. For $p = 2$, speedups are 3.7, 7.2, 10, and 16, respectively. (First y tick indicates the $p = 1$ speedup/slowdown.)

Intrepid2 Sum Factorization: Serial Est. Throughput

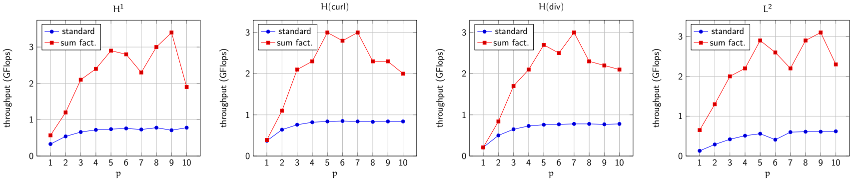


Figure: Serial (28-core 2.5 GHz Xeon W), estimated throughput for standard and sum-factorized assembly for H^1, $\mathrm{H}(\mathrm{curl})$, $\mathrm{H}(\mathrm{div})$, and L^2 norms on hexahedra.

Intrepid2 Sum Factorization: OpenMP Speedups

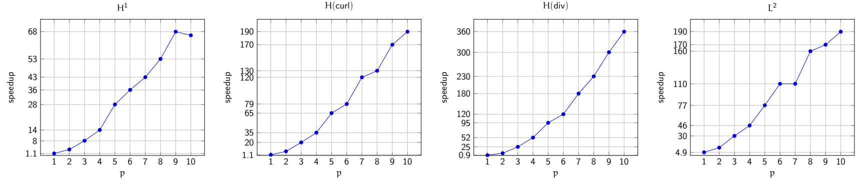


Figure: OpenMP (28-core 2.5 GHz Xeon W, 16 threads) speedups compared to standard assembly for H^1, $\mathrm{H}(\mathrm{curl})$, $\mathrm{H}(\mathrm{div})$, and L^2 norms on hexahedra. For $p = 2$, speedups are 3.3, 6.3, 6.5, and 12, respectively. (First y tick indicates the $p = 1$ speedup/slowdown.)

Intrepid2 Sum Factorization: OpenMP Est. Throughput

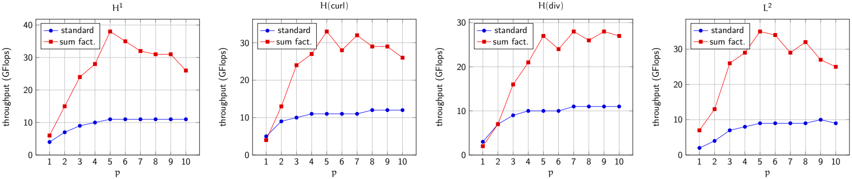


Figure: OpenMP (28-core 2.5 GHz Xeon W, 16 threads), estimated throughput for standard and sum-factorized assembly for H^1, $\mathrm{H}(\mathrm{curl})$, $\mathrm{H}(\mathrm{div})$, and L^2 norms on hexahedra.

Intrepid2 Sum Factorization: CUDA Speedups

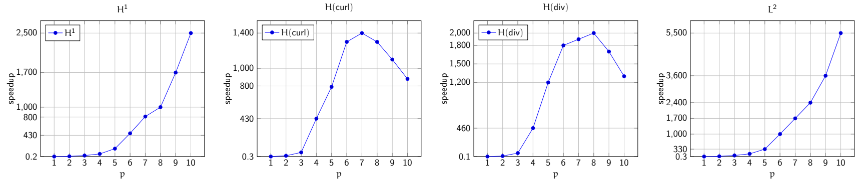


Figure: CUDA (V100) speedups compared to standard assembly for H^1, $H(\mathrm{curl})$, $H(\mathrm{div})$, and L^2 norms on hexahedra. For $p = 2$, speedups are 4.8, 8.6, 8.0, and 11.0, respectively. (First y tick indicates the $p = 1$ speedup/slowdown.)

Intrepid2 Sum Factorization: CUDA Est. Throughput

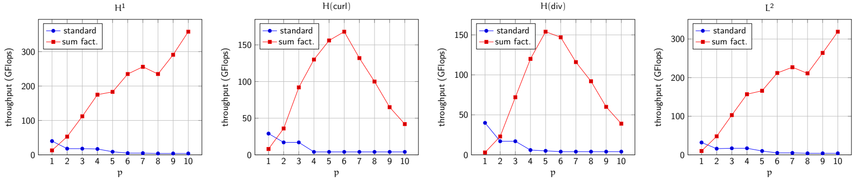


Figure: CUDA (V100), estimated throughput for standard and sum-factorized assembly for H^1, $\mathrm{H}(\mathrm{curl})$, $\mathrm{H}(\mathrm{div})$, and L^2 norms on hexahedra.

Conclusion and Future Work

Future work:

- Support for orientations with structured integration (imminent: PR 11667).
- Soon: high-order pyramids.
- Support for MFEM-style matrix-free/partial assembly.
- libCEED benchmark comparisons.
- Sum factorization for simplices?[2]

Please do contact me (`nvrober@sandia.gov`) with questions and/or feature requests.

Thanks for your attention!

[2] Ainsworth, et al. Bernstein–Bézier Finite Elements of Arbitrary Order and Optimal Assembly Procedures. https://doi.org/10.1137/11082539X.