EMPIRE: Sandia’s Next Generation Plasma Tool — Kokkosifying EMPIRE-Fluid —

April 24, 2019

EMPIRE Code Suite

- EMPIRE: ElectroMagnetic Plasma In Realistic Environments
- New code, started in 2015
- Designed to meet Advanced Technology Demonstration and Mitigation (ATDM) requirement for next-generation architectures

- Focus today: EMPIRE-Fluid
EMPIRE-Fluid Code Structure

EMPIRE is built on top of Trilinos. EMPIRE-Fluid makes extensive use of Panzer, which provides FEM discretization support:

- simplifies assembly of linear, nonlinear FEM problems
- assembles both residual and Jacobian systems
- for Jacobian support, use Sacado AD scalar types
- Panzer builds a directed acyclic graph (DAG) using Phalanx
EMPIRE-Fluid Code Structure

EMPIRE is built on top of Trilinos. EMPIRE-Fluid makes extensive use of Panzer, which provides FEM discretization support:

- simplifies assembly of linear, nonlinear FEM problems
- assembles both residual and Jacobian systems
- for Jacobian support, use Sacado AD scalar types
- Panzer builds a directed acyclic graph (DAG) using Phalanx

A node in the Phalanx DAG evaluates one or more fields, and may depend on other fields.

- specifies which fields it depends on
- specifies which fields it evaluates (provides)
- implements method to perform evaluation
Phalanx DAG

Advantages of the DAG:

- Graph expresses the dependency logic of the algorithm.
- Graph can be (and is!) pruned to avoid unnecessary computations.
- Other optimizations are also possible (e.g. parallel execution of nodes).

Challenges:

- “Event-driven” software model
  - what’s being executed?
  - in what order?
- debugging can be harder
- art of defining reasonable kernel granularity

\[ R_u^i = \int_{\Omega} [\phi_u^i \nabla - \phi_u^i \cdot q + \phi_u^i s] \, d\Omega \]
EMPIRE-Fluid: Kokkosification

Initial implementation of EMPIRE-Fluid focused on the physics, deferring thread-parallel design decisions (no Kokkos). Goal: get representative problems running on CUDA.

- Almost all the work is in nodes of the DAG
- In Fluid (and Panzer), we call these Evaluators
- We have a lot of Evaluators (about 130 distinct implementations)
EMPIRE-Fluid: Kokkosification

Initial implementation of EMPIRE-Fluid focused on the physics, deferring thread-parallel design decisions (no Kokkos). Goal: get representative problems running on CUDA.

- Almost all the work is in nodes of the DAG
- In Fluid (and Panzer), we call these Evaluators
- We have a lot of Evaluators (about 130 distinct implementations)

Note: we depend on UVM. This makes reasoning about performance harder:

- converting some (but not all) code to run on device may cause page faults / data migration from host to device
- we want to have all Evaluators running on device
- but 130 is a lot of Evaluators!
EMPIRE-Fluid: Kokkosification

Initial implementation of EMPIRE-Fluid focused on the physics, deferring thread-parallel design decisions (no Kokkos). Goal: get representative problems running on CUDA.

- Almost all the work is in nodes of the DAG
- In Fluid (and Panzer), we call these Evaluators
- We have a lot of Evaluators (about 130 distinct implementations)

Note: we depend on UVM. This makes reasoning about performance harder:

- converting some (but not all) code to run on device may cause page faults / data migration from host to device
- we want to have all Evaluators running on device
- but 130 is a lot of Evaluators!

We use the thermalization problem as an intermediate goal. (What Evaluators does it use?)
Hierarchic Parallelism is Critical to Performance

- Single level parallelism is insufficient
- Does not expose enough parallelism

Drekar Incompressible CFD Kernel

\[ \int_{e} c \left( \bar{f}(x) \cdot \nabla \varphi_i(x) + s(x) \varphi_i(x) \right) \, dx \]
Hierarchic Parallelism is Critical to Performance

- Single level parallelism is insufficient
- Does not expose enough parallelism
- 3-level hierarchical parallelism shows significant improvement
- Hand coded sensitivity array outside libraries
- **Key is to parallelize over FAD derivative dimension**
Hierarchic Parallelism is Required!

- Where is the `parallel_for(ThreadVectorRange(...));`?
  - Hidden in AD Scalar type – run vector loop over the AD derivative hidden dimension

```cpp
template<typename EvalT, typename Traits>
KOKKOS_INLINE_FUNCTION
void IntegrateDiffusionTerm<EvalT,Traits>::operator() (const TeamPolicy<exec_space>::member_type& team) const{
    const int cell = team.league_rank();
    parallel_for(TeamThreadRange(team,0,grad_basis.extent(2)), 
      [&] (const int& basis) {
        for (int qp = 0; qp < static_cast<int>(grad_basis.extent(1)); ++qp) {
          for (int dim = 0; dim < static_cast<int>(grad_basis.extent(3)); ++dim) {
            residual(cell,basis) += -weighted_grad_basis(cell,qp,basis,dim) * flux(cell,qp,dim);
          }
        });
    }
}
```

- Hierarchic parallelism might require multiple implementations for non-AD performance
  - The inner-most loop (vector loop) is implemented in Sacado
  - For non-AD types, this loop is missing (run with vector size 1 to reuse impl.)
  - Not usually a big deal as AD kernels are usually the bottleneck

- So how can we reuse this code for all evaluation types?
  - Either run with vector size=1 or...
  - SIMD!!!
  - Upcoming SIMD data type can add this inner loop back for non-AD types!
Speedup from Hierarchic

- Projection kernels sensitive to thread teams
- Integration kernels sensitive to shared memory
  - Cached basis values and accumulations in shared memory

\[ \nabla u = \sum_{i=0}^{N} u_i \nabla \phi_i \]

\[ \int_{\Omega} \nabla \phi \cdot u \, d\Omega \]

\[ \int_{\Omega} (\nabla \cdot \phi) u \, d\Omega \]
A Key Tool: Teuchos::StackedTimer

How do we determine what’s getting run by Phalanx? Teuchos::StackedTimer can help!

- Call TEUCHOS_FUNC_TIME_MONITOR("my method") in the method you’d like to track.

- We do this for every Evaluator.

StackedTimer can output timing data that looks like this:

```plaintext
*** Teuchos::StackedTimer::report() - Remainder for a block will be ***
*** incorrect if a timer in the block does not exist on every rank ***
*** of the MPI Communicator. ***

Teuchos::StackedTimer: 100.0 [1]
| panzer::SquareQuadMeshFactory::buildMesh(): 50.0 [1]
| | panzer::SquareQuadMeshFactory::buildUncommittedMesh(): 30.0 - 60.0% [1]
| | panzer::SquareQuadMeshFactory::completeMeshConstruction(): 5.0 - 10.0% [1]
| | Remainder: 15.0 - 30.0%
| panzer_stk::generateLocalMeshInfo: 10.0 [2]
| TaskGraph::run: 40.0 [1]
| | Phalanx::SortAndOrderEvaluators: 3.0e-01 - 0.75% [18]
| | Phalanx: Evaluator 543: [panzer::Traits::Residual] InvertMatrix: 30.0 - 75.0% [1]
| | | utilities::InvertMatrix::evaluate: 3.0 - 10.0% [1]
| | | Remainder: 27.0 - 90.0%
| | Phalanx: Evaluator 534: [panzer::Traits::Residual] GenerateInternalFluxIntegrationArray: 8.0 - 20.0% [1]
| | | utilities::GenerateInternalFluxIntegrationArray::evaluateFields: 6.0 - 75.0% [1]
| | | Remainder: 2.0 - 25.0%
| | | Remainder: 1.7 - 4.25%
| | Remainder: 0.0 - 0.00%
```
A Useful Python Script

Note that Phalanx gives each Evaluator instance a unique name. For concise timer summary, aggregation is useful. We’ve implemented `aggregateTimings.py`, which can be run in two modes:

- timing report mode
- names only: alphabetized list

Timing report:

<table>
<thead>
<tr>
<th>Name</th>
<th>Time (seconds)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>panzer::SquareQuadMeshFactory::buildUncomittedMesh()</td>
<td>3.00e+01</td>
<td>30.0</td>
</tr>
<tr>
<td>InvertMatrix (Residual)</td>
<td>2.70e+01</td>
<td>27.0</td>
</tr>
<tr>
<td>panzer::SquareQuadMeshFactory::buildMesh()</td>
<td>1.50e+01</td>
<td>15.0</td>
</tr>
<tr>
<td>panzer_stk::generateLocalMeshInfo</td>
<td>1.00e+01</td>
<td>10.0</td>
</tr>
<tr>
<td>utilities::GenerateInternalFluxIntegrationArray::evaluateFields</td>
<td>6.00e+00</td>
<td>6.0</td>
</tr>
<tr>
<td>panzer::SquareQuadMeshFactory::completeMeshConstruction()</td>
<td>5.00e+00</td>
<td>5.0</td>
</tr>
<tr>
<td>utilities::InvertMatrix::evaluate</td>
<td>3.00e+00</td>
<td>3.0</td>
</tr>
<tr>
<td>GenerateInternalFluxIntegrationArray (Residual)</td>
<td>2.00e+00</td>
<td>2.0</td>
</tr>
<tr>
<td>TaskGraph::run</td>
<td>1.70e+00</td>
<td>1.7</td>
</tr>
<tr>
<td>Phalanx::SortAndOrderEvaluators</td>
<td>3.00e-01</td>
<td>0.3</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>----------------</td>
<td>----</td>
</tr>
<tr>
<td>Total</td>
<td>1.00e+02</td>
<td>100.0</td>
</tr>
</tbody>
</table>

For our purposes, the “names only” mode is what we want...
Kokkos Conversion Approach

We identified 21 Evaluators (of 130) that get used by thermalization. Once those are converted, everything should be on the device. What do we do for the conversion?

- Use hierarchical parallelism
  - Kokkos supports three levels: team, thread, vector
  - Recall Sacado handles vector parallelism for AD types
  - Panzer lets us use common code for the two types (vector length 1 for POD scalar)

```cpp
#include "Panzer_HierarchicParallelism.hpp"
auto policy = panzer::HP::inst().teamPolicy<Scalar,PHX::Device>(num_outer_loops);
Kokkos::parallel_for( policy , my_functor, "my_functor" );
```

- Once we have a SIMD POD scalar type, we can vectorize for those, too...

- Use shared memory when it makes sense
Measuring Performance

Now that we have converted the thermalization problem, we can measure its performance, comparing CUDA performance with serial performance on a 2.3 GHz Xeon W CPU (as a baseline). (This is a 2D problem with quartic polynomials for the FEM basis.)

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Cell Count</th>
<th>Time Per Timestep (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon W</td>
<td>1600</td>
<td>0.064</td>
</tr>
<tr>
<td>Pascal</td>
<td>1600</td>
<td>0.050</td>
</tr>
<tr>
<td>Xeon W</td>
<td>10000</td>
<td>0.50</td>
</tr>
<tr>
<td>Pascal</td>
<td>10000</td>
<td>0.19</td>
</tr>
<tr>
<td>Volta</td>
<td>10000</td>
<td>0.18</td>
</tr>
<tr>
<td>Xeon W</td>
<td>40000</td>
<td>1.93</td>
</tr>
<tr>
<td>Pascal</td>
<td>40000</td>
<td>0.81</td>
</tr>
<tr>
<td>Volta</td>
<td>40000</td>
<td>0.67</td>
</tr>
</tbody>
</table>
Performance Profiling

We used **nvprof** on a 1600-element run (Pascal) to profile performance. We determined:

- spend about 12 ms per timestep in GPU computations
- sustained 6 ms per timestep with GPU completely idle (CPU busy – doing what?)
- spend 56% of GPU compute in *DistributedArrayScatterFromFunctor* (can refactor away)
- for the remaining top 3 kernels, have thread occupancy of 40%, 34%, and 84%

Next steps:

- Account for CPU time (and reduce)
- Refactor to eliminate *DistributedArrayScatterFromFunctor*
- Improve occupancy of top 2 kernels, perhaps by adjusting team sizes.
Summary

- Panzer facilitates rapid development of CUDA-capable code
- Can be challenging to know where to start in converting existing code
- Teuchos::StackedTimer can help in figuring out what kernels are being used (we could have used Kokkos Tools as well!)
- Shared memory, hierarchical parallelism are particularly important to get good performance.