OAK RIDGE National Laboratory

CFD Simulations with Panzer

Trilinos User-Developer Group Meeting 2023

Steven Hamilton (ORNL)
Stuart Slattery (ORNL)
Taylor Erwin (ORNL)
Roger Pawlowski (SNL)
Bryan Reuter (SNL)

ORNL is managed by UT-Battelle LLC for the US Department of Energy

U.S. DEPARTMENT OF ENERGY

Outline

- Overview and code summary
- Optimizing GPU Jacobian construction
- Linear solvers and preconditioning
- Conclusions and wishlist

Overview

- 2D/3D time-dependent, unstructured mesh compressible Navier-Stokes
  - Cartesian and RZ coordinate systems
- Continuous finite element
  - SUPG and entropy viscosity stabilization
- Multithreaded CPU and GPU (Nvidia and AMD) are all important
- Heavily built on Trilinos framework
  - Panzer/Phalanx packages for finite element construction
  - Tempus for time integration
  - NOX for nonlinear solvers
  - Epetra/Tpetra for parallel linear algebra
  - Stratimikos for linear solver construction (many packages underneath)
  - Sacado for automatic differentiation
  - Kokkos for performance portability

Flow around cylinder in tunnel

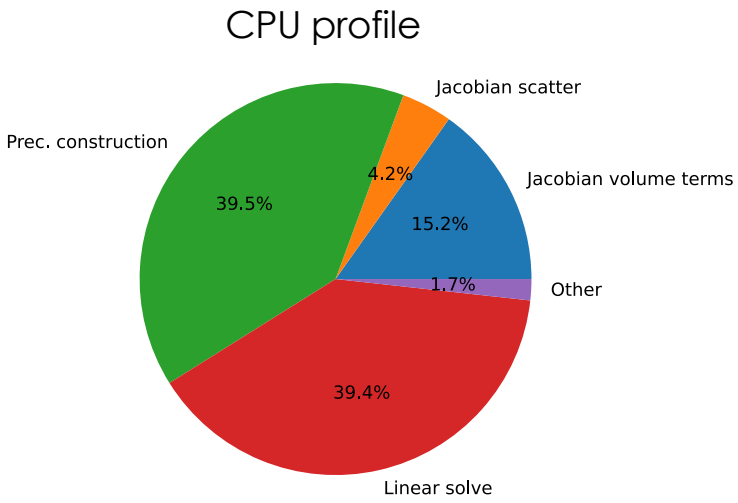
Jacobian matrix construction

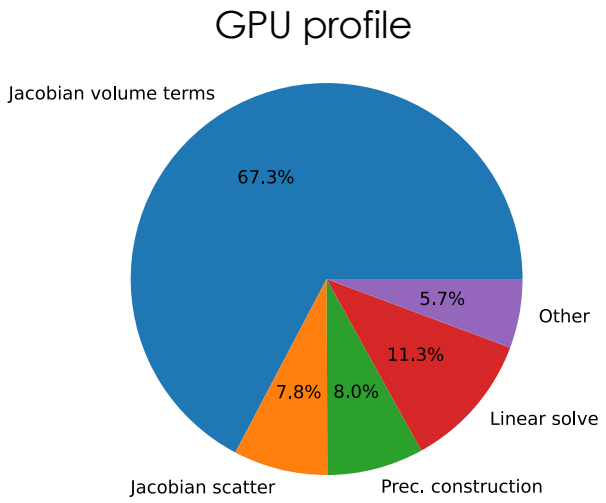
- Automatic differentiation allows analytic Jacobians to be computed for use with Newton's method
  - No hand derivation
  - No finite difference approximations
  - Only requires function (residual) evaluation

- Sacado package provides AD types
  - C++ templates allow single function implementation for either standard floating point or AD types
  - Kokkos support allows AD operations to be performed on GPUs

AD Jacobian on GPUs

- Initial CUDA port showed huge bottleneck in Jacobian construction
- Profiling turned up several hot spots
  - Teuchos stacked timer is awesome!





Primary culprit: memory management

- Objects in the Panzer evaluator graph are templated to evaluation type
  - `scalar_type` is `double` when evaluating the residual
  - `scalar_type` is AD type when computing the Jacobian
- Developers are in the habit of treating `scalar_type` like POD
  - Standard math operations are overloaded
  - Generally works fine on the CPU
- Panzer uses `Sacado::DFad<double>`
  - Derivative dimension is not known at compile time
  - Thread-local scalars may trigger on-device memory allocations

Basic evaluator

- Consider simple evaluator with dependent (input) and evaluated (output) fields

```
template<class EvalType, class Traits>
void MyEvaluator<EvalType, Traits>::operator()(
    const Kokkos::TeamPolicy<PHX::exec_space>::member_type& team) const
{
    int cell = team.league_rank();
    int num_points = _input1.extent(1);

    Kokkos::parallel_for(
        Kokkos::TeamThreadRange(team, 0, num_points), [&](const int point)
    {
        // Extract input data
        scalar_type in1 = _input1(cell, point);
        scalar_type in2 = _input2(cell, point);

        // Perform local manipulations
        scalar_type tmp = in1 * in1 + in2 * in2;

        // Compute properties from data
        _output1(cell, point) = _properties.compute_out1(in1);
        _output2(cell, point) = _properties.compute_out2(in1, in2);
        _output3(cell, point) = _properties.compute_out3(tmp);
    }
}
```

Perfectly reasonable if `scalar_type` is `double`

Basic evaluator

- Consider simple evaluator with dependent (input) and evaluated (output) fields

```
template<class EvalType, class Traits>
void MyEvaluator<EvalType, Traits>::operator()(
    const Kokkos::TeamPolicy<PHX::exec_space>::member_type& team) const
{
    int cell = team.league_rank();
    int num_points = _input1.extent(1);

    Kokkos::parallel_for(
        Kokkos::TeamThreadRange(team, 0, num_points), [&](const int point)
    {
        // Extract input data
        scalar_type in1 = _input1(cell, point);
        scalar_type in2 = _input2(cell, point);

        // Perform local manipulations
        scalar_type tmp = in1 * in1 + in2 * in2;

        // Compute properties from data
        _output1(cell, point) = _properties.compute_out1(in1);
        _output2(cell, point) = _properties.compute_out2(in1, in2);
        _output3(cell, point) = _properties.compute_out3(tmp);
    }
}
```

For `Sacado::DFad`, these are dynamic memory allocations!!!

Basic evaluator

- Consider simple evaluator with dependent (input) and evaluated (output) fields

```
template<class EvalType, class Traits>
void MyEvaluator<EvalType, Traits>::operator()(
    const Kokkos::TeamPolicy<PHX::exec_space>::member_type& team) const
{
    int cell = team.league_rank();
    int num_points = _input1.extent(1);

    Kokkos::parallel_for(
        Kokkos::TeamThreadRange(team, 0, num_points), [&](const int point)
    {
        // Extract input data
        auto in1 = _input1(cell, point);
        auto in2 = _input2(cell, point);

        // Perform local manipulations
        auto tmp = in1 * in1 + in2 * in2;

        // Compute properties from data
        _output1(cell, point) = _properties.compute_out1(in1);
        _output2(cell, point) = _properties.compute_out2(in1, in2);
        _output3(cell, point) = _properties.compute_out3(tmp);
    }
}
```

Using auto will “do the right thing”

Handling thread-local operations

- What if we want to perform extra operations on thread-locals?
  - Using `auto` doesn't work anymore

```
template<class EvalType, class Traits>
void MyEvaluator<EvalType, Traits>::operator()(
    const Kokkos::TeamPolicy<PHX::exec_space>::member_type& team) const
{
    int cell = team.league_rank();
    int num_points = _input1.extent(1);

    Kokkos::parallel_for(
        Kokkos::TeamThreadRange(team, 0, num_points), [&](const int point)
    {
        // Extract input data
        auto in1 = _input1(cell, point);
        auto in2 = _input2(cell, point);

        scalar_type tmp = in1 * in1;
        if (some_condition)
            tmp += in2 * in2;

        // Compute properties from data
        _output1(cell, point) = _properties.compute_out(tmp);
    }
}
```

Handling thread-local operations

- Create `Kokkos::View` for storing temporary values
  - Must be preallocated before kernel launch for all threads

```
template<class EvalType, class Traits>
void MyEvaluator<EvalType, Traits>::operator()(
    const Kokkos::TeamPolicy<PHX::exec_space>::member_type& team) const
{
    int cell = team.league_rank();
    int num_points = _input1.extent(1);

    Kokkos::parallel_for(
        Kokkos::TeamThreadRange(team, 0, num_points), [&](const int point)
    {
        // Extract input data
        auto in1 = _input1(cell, point);
        auto in2 = _input2(cell, point);

        auto&& tmp = _temporary(cell, point);
        tmp = in1 * in1;
        if (some_condition)
            tmp += in2 * in2;

        // Compute properties from data
        _output1(cell, point) = _properties.compute_out(tmp);
    }
}
```

Returning values from a method

- What should return type be?

```
struct Properties
{
    template<typename T1, typename T2>
    KOKKOS_INLINE_FUNCTION
    xxx compute_out(const T1& in1, const T2& in2) const
    {
        return a * in1 + b * in2;
    }
};
```

Returning values from a method

- What should return type be?

```
struct Properties
{
    template<typename T1, typename T2>
    KOKKOS_INLINE_FUNCTION
    xxx compute_out(const T1& in1, const T2& in2) const
    {
        return a * in1 + b * in2;
    }
};
```

- Possibilities:
  - `scalar_type`: same issues as thread local variables
  - `Sacado::Promote<T1, T2>`: essentially the same thing
  - `auto`: Segfault *on CPU* (performant with CUDA!)

Returning values from a method

- Our solution: move return value to function argument

```
struct Properties
{
    template<typename T1, typename T2, typename T3>
    KOKKOS_INLINE_FUNCTION
    void compute_out(const T1& in1, const T2& in2, T3&& out) const
    {
        out = a * in1 + b * in2;
    }
};
```

- The universal reference (&&) is important!
  - Allows correct behavior for both POD and AD types

Returning values, again

- What if you **really** want to return values from a function?
  - Converting return value to function argument may reduce readability
  - Can't "chain" operations together
- Solution: Create your own Sacado expression
  - Explicitly implement derivative terms rather than relying on Sacado to propagate

Use case: smooth math operations

- Non-smooth math operations can be replaced with differentiable approximations

```
template<typename T>
KOKKOS_INLINE_FUNCTION
scalar_type smooth_abs(const T& x, double tol)
{
    if (x >= tol)
        return x;
    else if (x <= -tol)
        return -x;
    else
        return 0.5 * (x * x / tol + tol);
}
```

- As before, return forces memory allocation if `T` is an AD type
  - Ideally, we should be able to use this as:

```
out = a * smooth_abs(x, tol) + b;
```

User-defined Sacado expression

```
template<typename T>
class SmoothAbsOp
{
  public:
    SmoothAbsOp(const T& x, double tol)
      : x_(x)
      , tol_(tol)
    {}

    value_type val() const
    {
        if (x_ >= tol_)
            return x_.val();
        else if (x_ <= -tol)
            return -x_.val();
        else
            return 0.5 * (x_.val() * x_.val() / tol_ + tol_);
    }

    value_type dx(int i) const
    {
        if (x_ >= tol_)
            return x_.dx(i);
        else if (x_ <= -tol)
            return -x_.dx(i)
        else
            return x_.val() * x_.dx(i) / tol_;
    }

  private:
    const T& x_;
    double tol_;
};

template<typename T>
SmoothAbsOp<Sacado::Expr<T>> smooth_abs(const Sacado::Expr<T>& x, double tol)
{
    return SmoothAbsOp<Sacado::Expr<T>>(x, tol);
}
```

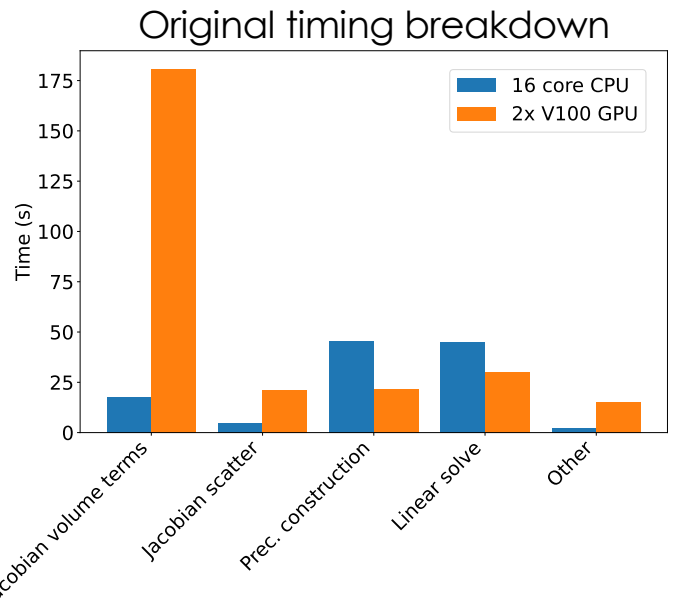
- This is not a complete example!
  - Additional templating and traits specializations required
- Multi-parameter cases are much more complicated
  - Mixing AD and POD types

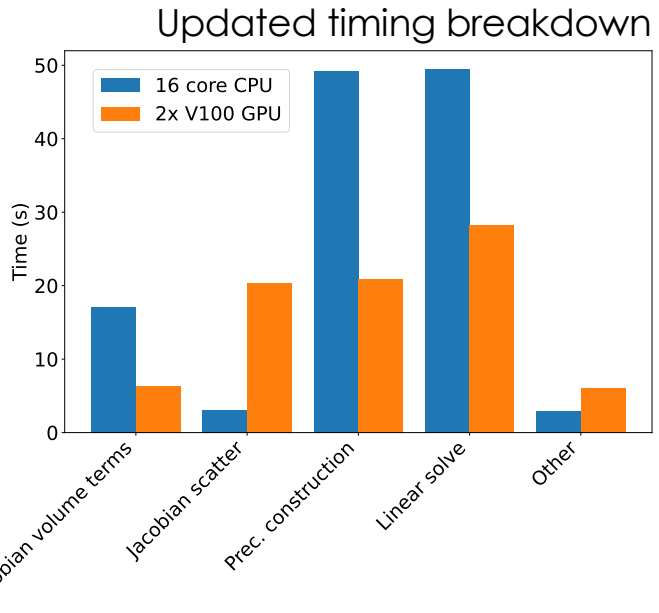
Hierarchical parallelism

- With Kokkos, Sacado can use derivative dimension for parallelism
  - Maps to vector unit (CUDA warp)
  - Automatically embedded in AD operations
- Panzer hierarchical parallelism (`Kokkos::TeamPolicy`) reserves vector unit for Sacado derivative dimension
  - Sacado parallelism not enabled by default!
  - Need configure option "`-D Sacado_ENABLE_HIERARCHICAL_DFAD=ON`"
- Only single thread per warp was active in hierarchical kernels
  - Not caught initially due to other performance issues

Performance results

- Optimizations in Jacobian construction have made significant difference on GPU
  - Gas properties kernel is 400x faster on GPU, 1.15x faster on CPU
  - Scatter operation in Tpetra::CrsMatrix has become bottleneck





Solver hierarchy

- Newton's method used as nonlinear solve
  - One or more nonlinear solves per time step
  - Typically 3-5 nonlinear iterations per nonlinear solve
  - One linear solve per nonlinear iteration
  - Anderson acceleration is also viable option through Trilinos NOX interface, but not yet evaluated

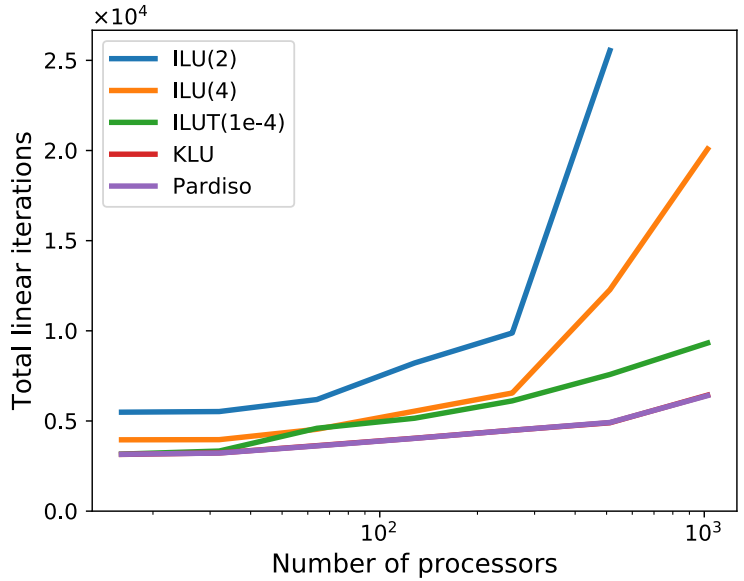
- GMRES used almost exclusively as linear solver
  - Unpreconditioned GMRES fails even for trivial problems
  - Restarting is very ineffective – large subspace size potentially needed

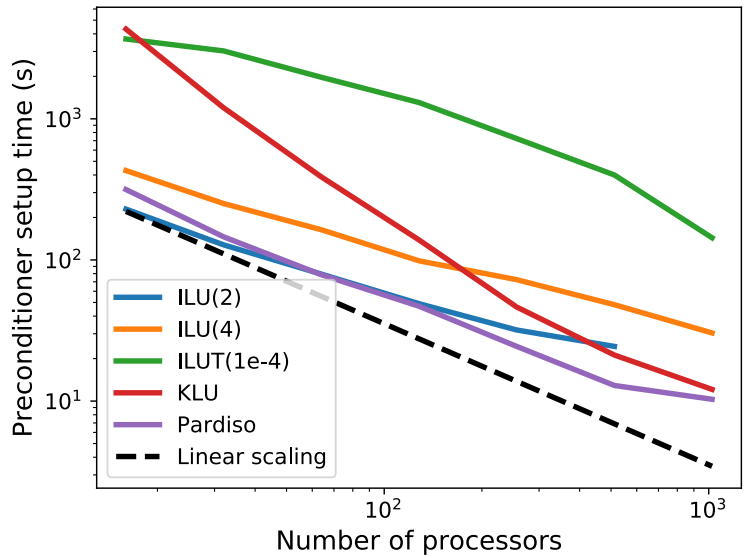
Preconditioner selection

- Availability of Jacobian matrix has led to focus on algebraic preconditioners
  - Physics-based preconditioning is an area for future consideration
- Algebraic multigrid approaches fail for all but simplest cases
  - ML, MueLu, BoomerAMG, and AMGCL have all been evaluated
- Additive Schwarz style preconditioners have shown significant promise
  - Small inter-block overlap improves robustness and parallel scalability
  - Both incomplete factorizations and sparse direct solvers have potential for local block solves

Solver performance

- MPI only (1 thread per rank), 2x 64-core AMD CPUs
- Global matrix size: 120k spatial elements, 600k DOFs



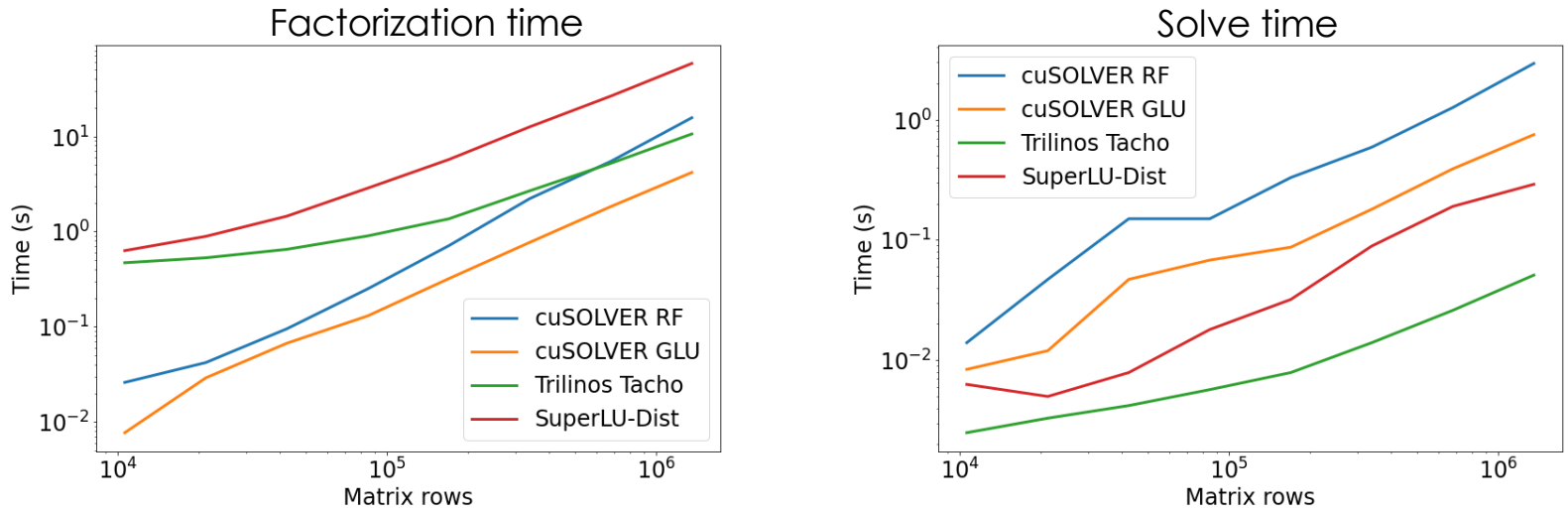


GPU local block solvers

- CuSOLVER
  - Nvidia GPUs only
  - GPU-based QR factorization
  - RF solver requires one-time host factorization with on-GPU refactorization
  - Undocumented GLU solver provides optimized alternative to RF
    - Also requires initial host factorization
- SuperLU-Dist
  - GPU kernel under active development
  - Provides distributed-memory parallel sparse direct solves
- Trilinos Tacho
  - Recent introduction
  - Implemented using Kokkos
- Pardiso
  - Non-free distribution
- Ginkgo
  - Only ILU factorizations?

Not yet evaluated

Sparse direct solver GPU (V100) performance



- cuSOLVER GLU provides best factorization performance
  - One-time CPU setup expected to be amortized across multiple solves
- Trilinos Tacho solver has best triangular solve performance

Preconditioner reuse

- Jacobian matrix frequently changes slowly for many problems
  - Repeating factorization for preconditioner every Newton iteration, every time step is overkill
- Selecting factorization frequency not available with Trilinos preconditioners
- Reducing to one factorization per time step has no impact on solver convergence!
- Actual behavior expected to be highly problem dependent
  - Automatically determine when to refactor?

Factorization frequency	Prec.	Prec. compute (s)	Linear solve (s)	Total GMRES iters
Every Newton iteration	Tacho	661.5	58.7	2859
Every Newton iteration	GLU	199.1	335.4	1681
Once per time step	GLU	76.0	335.1	1681
Every 2 time steps	GLU	45.3	349.7	1755
Every 4 time steps	GLU	30.6	363.4	1825

Conclusions and future work

- AD-generated Jacobians are extremely powerful
  - No need to compute terms by hand
  - Require careful attention to achieve performance on GPUs
- Continued evaluation of GPU sparse direct solvers
- Develop options for AMD/Intel GPUs
  - SuperLU, Tacho
  - Reevaluate ILU-based approaches (FastILU?)
- Physics-based preconditioning?
  - Use of low-order approaches leads to less work per domain – may be inefficient on GPUs

Wishlist

- Optimizations for matrix scatter operations in Tpetra
- Support for customizable preconditioner reuse
- More GPU-capable preconditioners
  - When AMG doesn't work, there aren't many alternatives...
  - AMD path is unclear (SuperLU?)
  - Is Tacho the answer?
  - Refactor-based sparse direct solvers?
- Removing `nvcc_wrapper`