SAND2020-15670

Test-driven development and incremental app integration of a GPU-enabled finite element library

J. Plews, M. Mosby, N. Belcourt, S. Miller, J. Thomas

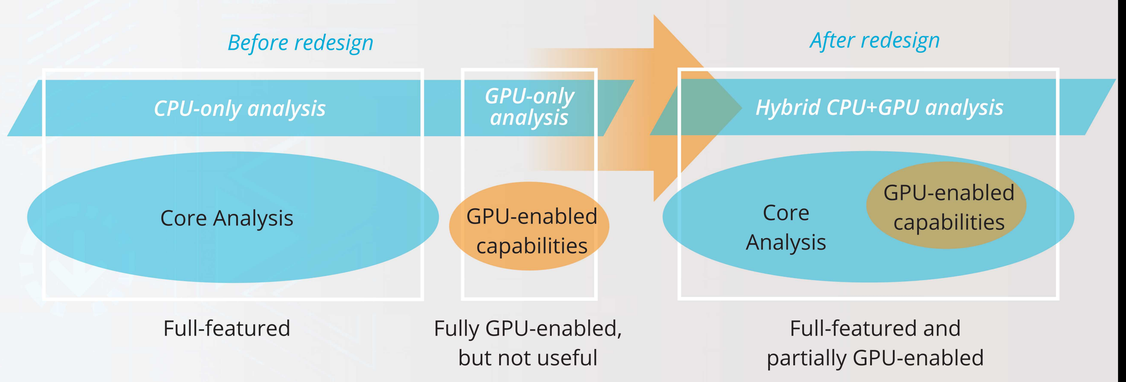
THE NEED FOR INCREMENTAL GPU PORTING

- Sierra/SolidMechanics (SM) is a large, general-purpose, legacy finite element codebase
  - Material models
  - Contact algorithms
  - Loads and boundary conditions
  - Solvers
- Analysts in a diverse userbase rely on many useful but non-GPU-enabled capabilities…
- …But the core code team has developed a small suite of GPU-enabled algorithms

GPU-enabled algorithms

Core code algorithms
- Complex dependencies
- CPU-only data structures, design patterns

- Previous GPU code development adopted a “vertical slice” approach…
- …But incremental delivery of GPU-enabled code capabilities requires incremental integration



ADAPTING LEGACY INTERFACES FOR GPU PERFORMANCE

- GPU and CPU execution utilize different mesh traversal (loop) structures
- Initial GPU code used separate data structures & execution paths
  - Unnecessary code duplication and bloat
  - Incompatible with hybrid CPU/GPU algorithms
- Initial GPU code was forced to conform to legacy interfaces
  - Undesirable dependencies
  - Difficult to unit test

Sample time integration algorithm before redesign

Evaluated only on GPU data when GPU execution enabled; static algorithm

```
void processCentDiffIntegrator(sm::Region& region, const
std::vector<const stk::mesh::Part*>& rigidBodyParts,
                               const Real dtNew, const Real dtOld) {
  const Real time_factor = 0.5 * (dtNew + dtOld);
  // ...
  if (region.getGpuMeshType() == sm::Region::GpuMeshType::STK) {
    sm::processCentDiffIntegratorGPU(region.getNgpMesh(), region,
       selector, dtNew, dtOld);
    return;
  }
  // ... Else: CPU-only legacy code here ...
}
```

- Redesigning from the high level allows interleaved CPU and GPU code execution under a uniform interface
- Clean interface enables test-driven development of underlying implementation without unnecessary dependencies on legacy data structures

Sample element kernel launch after redesign

Each object creates its own algorithm with requisite data

Generic, high-level interface may implement any underlying loop structure; GPU- or CPU-friendly

```
  std::vector<std::shared_ptr<InternalForceAlgorithm>> algorithms;
    for (auto&& e : elems) {
      algorithms.emplace_back(
        e->create_internal_force_algorithm(localErrors, active,
           colorSelectors));
    }

    InternalForceTimeSteps timesteps;
    for (auto&& alg : algorithms) {
      alg->compute(timesteps);
    }
```

- What about polymorphism?
  - Sierra/SM relies heavily on many dynamic types of material models, boundary conditions, finite element formulations…
  - Enable runtime creation of polymorphic objects through interface to allocate on CPU or GPU

Sample creation of polymorphic object for CPU or GPU execution

```
void GpuTotalLagrange::create_device_material() {
  material::AlProperties props;
  material::LinearElasticMaterial mat(props);
  material =
      mtk_ngp::device_new<material::LinearElasticMaterial>(mat);
}
```

UNIT TESTING ON THE GPU

- Sierra/SM depends upon the GoogleTest framework for unit testing, which does not execute on the GPU
  - Cannot check quantities computed on the GPU during kernel execution
- As part of GPU implementation redesign, Sierra/SM developers created a wrapper library based on Kokkos and GoogleTest to enable unit testing constructs on the GPU
  - Collect a subset of test failures on the device, if any, and report on host

```
TEST_P(DeviceMeshFixture, compute_stress) {
  auto&& mesh = make_host_mesh(get_mesh_spec(1));
  // ...
  fs->compute_gradient(*mesh, displacement, dispGrad,
                       detF);
  // ...
  device::compute_stress(*mesh, dispGrad, numIntg,
                         deviceMat, pkStress);
  test_ip_tensor_values(*mesh, pkStress,
     get_gold_pk_stress(create_host_material()),
     numIntg, 1.e-5);
}
```

Sample GPU-enabled unit test

Launch GPU kernel to test in unit test body

Call function to evaluate kernel correctness

Sample GPU-enabled unit test correctness check

Correctness check on device; if failure, collect result and report after GPU execution ends

```
meshtk::device_for_each<meshtk::Element>("test ip scalar
values", m, KOKKOS_LAMBDA(const meshtk::DeviceElement& e) {
    for (int ip = 0; ip < numIntg; ++ip) {
      NGP_EXPECT_NEAR(gold, scal(e, ip), tol);
    }
  });
```

TARGETING GPU-FRIENDLY ALGORITHMS FOR PERFORMANCE

Comparing data requirements for early GPU prototype (top) and new GPU implementation (bottom).

- First attempt at GPU kernel implementation targeted algorithms involving
  - Large data movement from host to device and back
  - Multiple unstructured field data accesses
  - Many temporary variables in GPU kernels
  - …excessive pessimism!
- Redesign for sustainability of GPU implementation targets algorithms involving
  - Less data required!
  - Synchronization of host/device data only when needed
  - Reuse of data structures static on device (e.g., interpolation spaces for finite elements)

```
class HexInternalForceCalculatorUseVelGrad {
 public:
  // ...
 private:
  ngp::Field<double> currentCoordsField;
  ngp::Field<double> velocityField;
  ngp::ConstField<double> vorticityField;
  ngp::ConstField<double> stretchingTensorField;
  ngp::Field<double> newInternalForceField;
  ngp::Field<double> newHourglassForceField;
  ngp::ConstField<double> elementMassField;
  ngp::Field<double> hourglassDecayField;
  ngp::Field<double> volumeField;
  ngp::ConstField<double> oldRotationField;
  ngp::Field<double> newRotationField;
  ngp::ConstField<double> oldStretchField;
  ngp::Field<double> newStretchField;
  ngp::ConstField<double> oldRateOfDeformationField;
  ngp::Field<double> newRateOfDeformationField;
  ngp::ConstField<double> oldUnrotatedStressField;
  ngp::Field<double> newUnrotatedStressField;
  ngp::Field<double> newRotatedStressField;
  ngp::ConstField<double> oldHourglassOpField;
  ngp::Field<double> newHourglassOpField;
  ngp::Field<double> elemTimeStepField;
  const ngp::Mesh mesh;
  Real currentTimeStep;
  const UGHexProps hexProps;
  const ElasticProps elasticProps;
};
```

```
class DeviceInternalForce {
 public:
 // ...
 private:
  const meshtk::Mesh mesh;
  const device::FunctionSpace& functionSpace;
  const int numIntgPts;
  const DevicePtr<material::MaterialLight> material;


  meshtk::NodeVecField displacement;
  meshtk::ElementPointerField dispGrad;
  meshtk::ElementPointerField detF;
  meshtk::ElementPointerField pkStress;
  meshtk::NodeVecForce force;
};
```

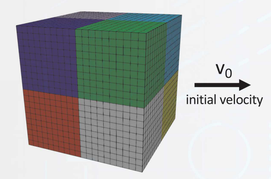
PERFORMANCE OF NEW GPU ELEMENT IMPLEMENTATION

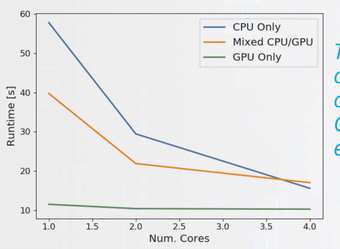
Single element block, GPU- or CPU-only execution

Analysis phase	Serial	16 MPI ranks	Vortex (GPU)
Setup	1.94	0.18	114
Internal Force (total)	174	14	4.55
Sync to device	-	-	0.94
Compute	174	14	2.97
Sync to host	-	-	0.46
Internal force speedup	**1.0**	**12.4**	**38.2**

Speedup summary of GPU vs. CPU: cube domain of 125,000 elements, 500 time steps, no output.

Multiple element blocks, hybrid CPU+GPU execution





Timing comparison of GPU vs. CPU: 8,000 elements

CONCLUSIONS

- Refactoring high-level interfaces incrementally to accommodate GPU algorithms enables incremental code integration
- Targeting algorithms and data structures which have favorable properties (e.g., low data access to flops ratio) for prototyping on the GPU avoids overly pessimistic outlooks on performance
- Enabling unit testing constructs on GPU architectures is necessary for test-driven development of new capabilities

CITATIONS

[1] M. Mosby et al. “STK NGP Test: a platform portable unit testing framework.” In DOE Performance, Portability and Productivity Annual Meeting, Denver, CO, April 2019.

Sandia National Laboratories

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.