Performant MPM Basis Operations for GPU Architectures

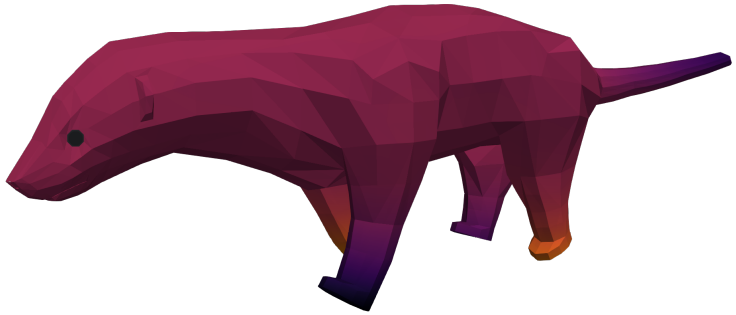
Jeremy L Thompson

University of Colorado Boulder

jeremy@jeremylt.org

11 Sept 2025

Ratel Team



Repository: https://gitlab.com/micromorph/ratel

Developers: Zach R. Atkins, Jed Brown, Fabio Di Gioacchino, Leila Ghaffari, Zach Irwin, Rezgar Shakeri, Ren Stengel, Jeremy L Thompson

The authors acknowledge support by the Department of Energy, National Nuclear Security Administration, Predictive Science Academic Alliance Program (PSAAP) under Award Number DE-NA0003962.

Overview

Ratel - high order, performance portable solid mechanics

Built on libCEED and PETSc

GPU and CPU performance

Overview

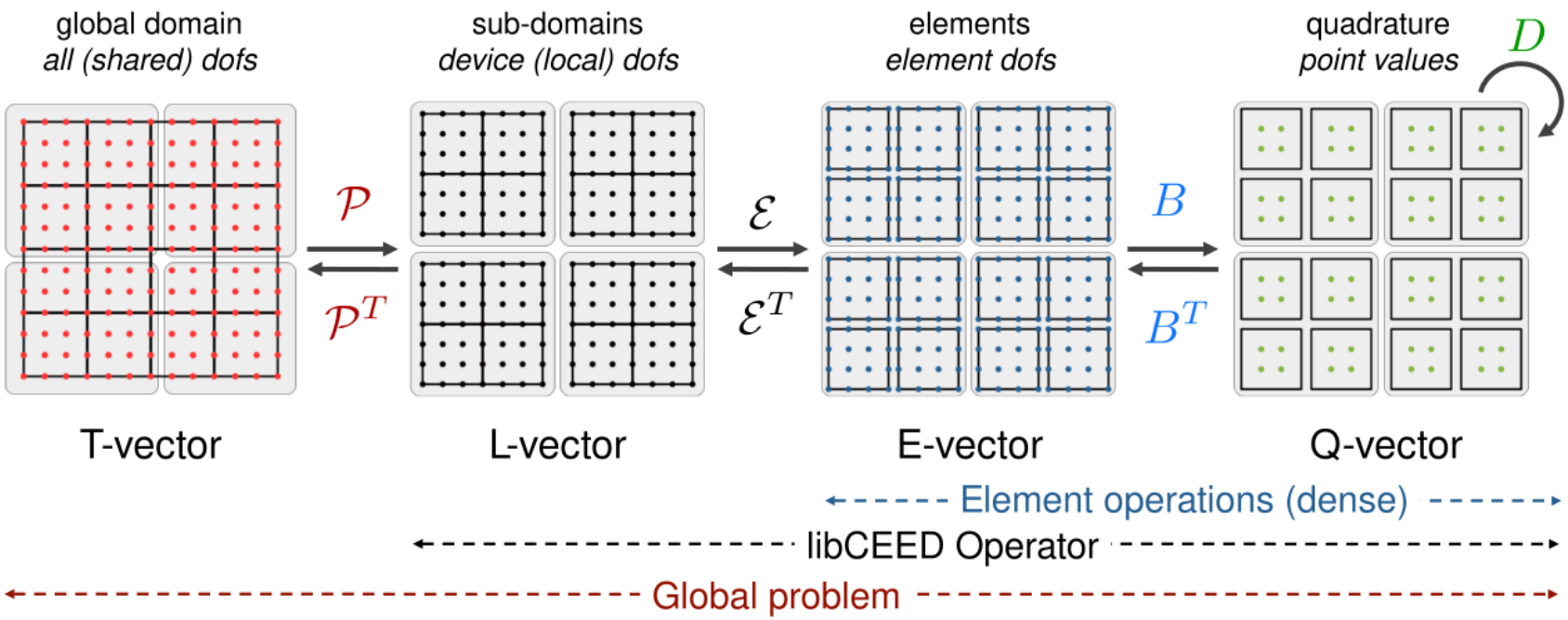
1. Ratel Background
2. AtPoints Evaluation
3. Implementation
4. Performance
5. Multigrid
6. Future Work

ECP Roots

- Ratel built directly on results from ECP CEED project
- libCEED provides high-performance operator evaluation
- PETSc provides linear/non-linear solvers and time steppers
- Ratel built from libCEED + PETSc solid mechanics demo app

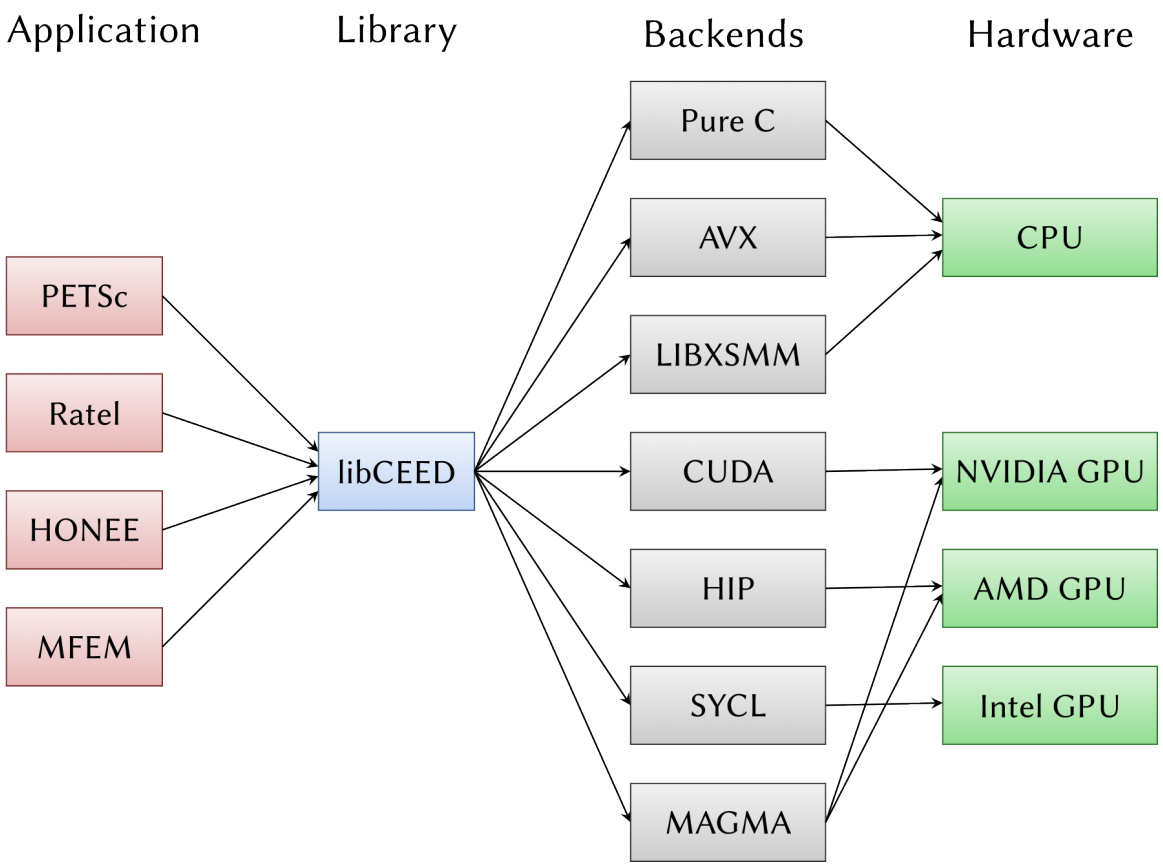
Matrix-Free Operators from libCEED

$$A = \mathcal{P}^T \mathcal{E}^T B^T D B \mathcal{E} \mathcal{P}$$



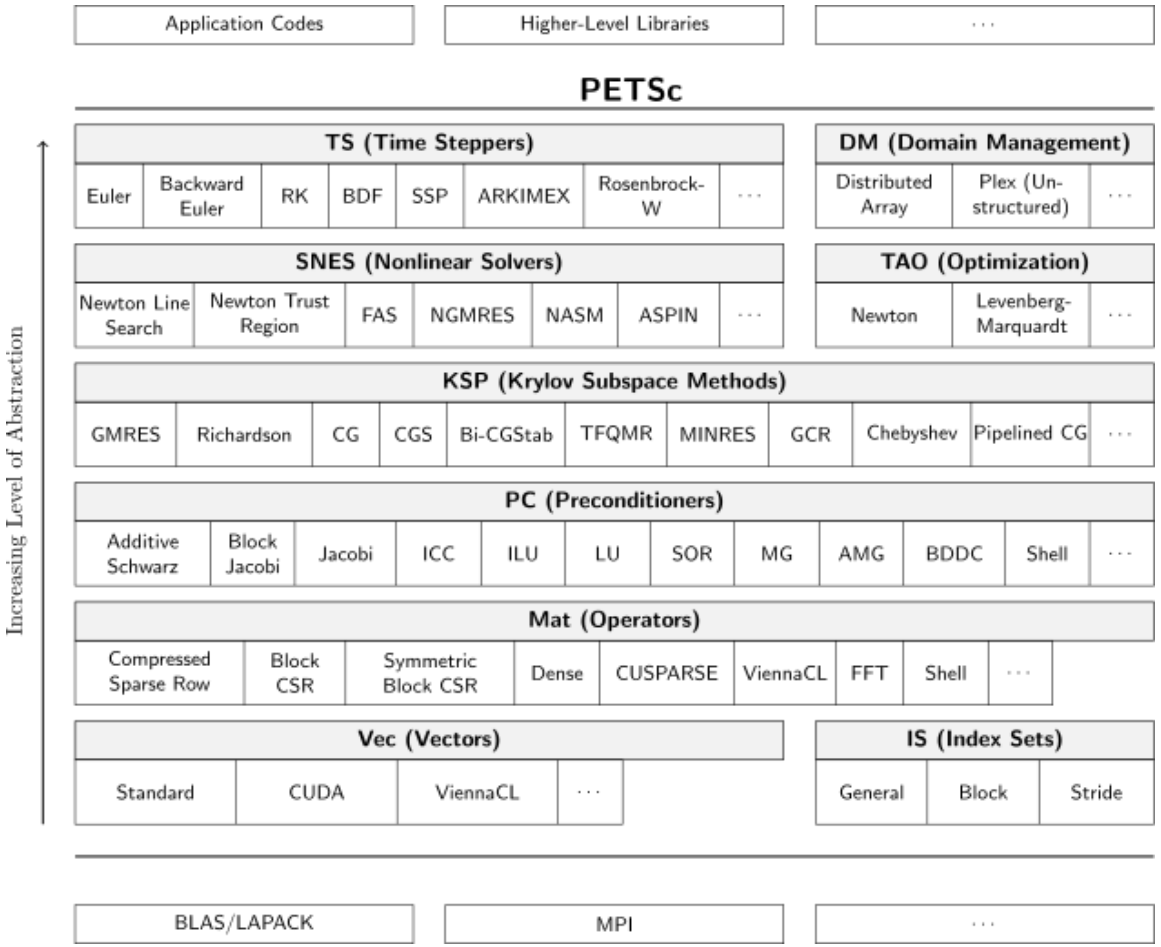
libCEED provides arbitrary order matrix-free operator evaluation

Performance Portability from libCEED



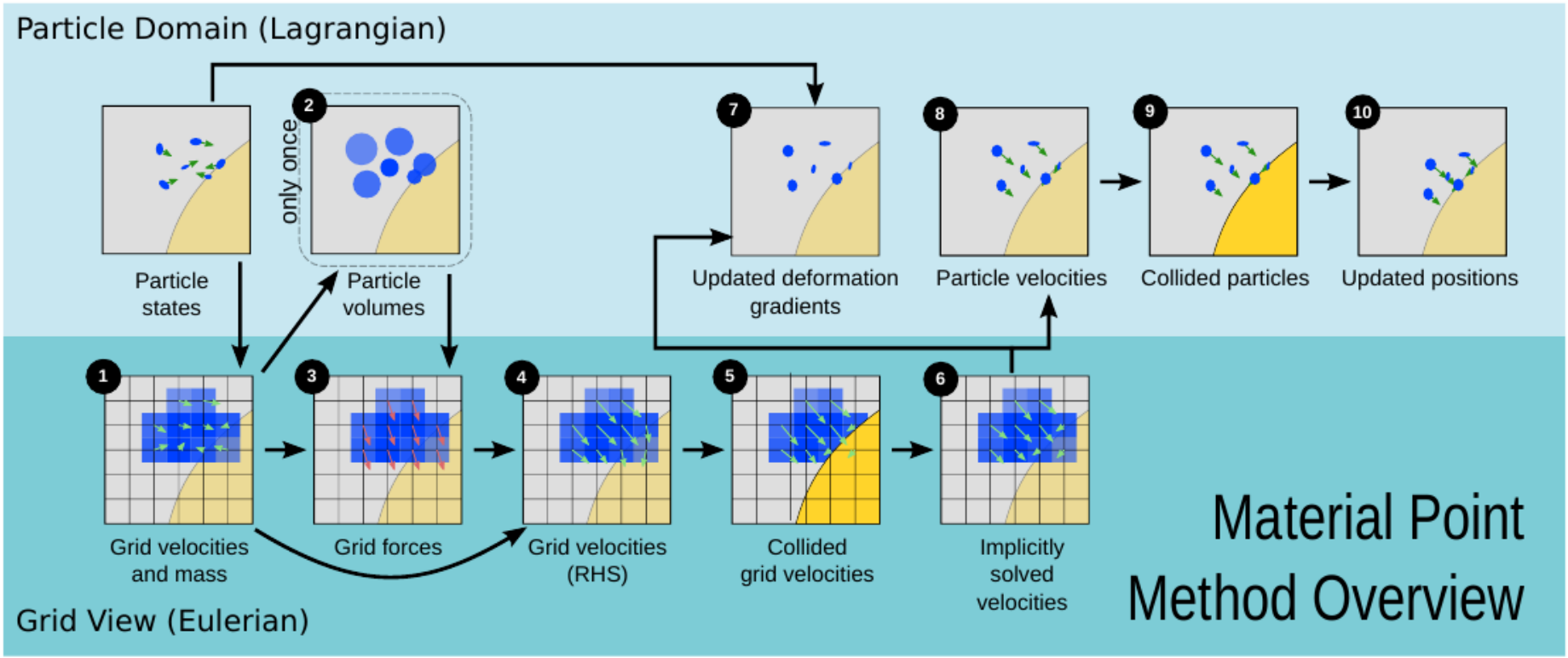
Performance portability with libCEED's matrix-free operators

Extensible Solvers from PETSc



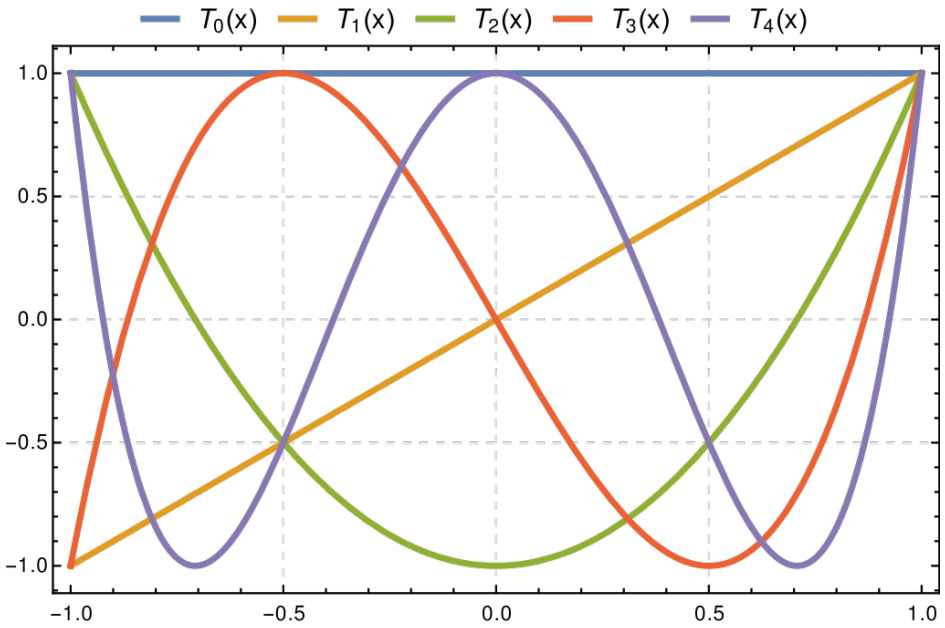
PETSc provides extensible, scalable solvers

What is MPM?



- Can be viewed as FEM with arbitrary quadrature point locations
- Natural fit for libCEED matrix-free representation
- Ratel FEM infrastructure provides fast background mesh solves

libCEED Basis Evaluation to Points



- Interpolate from primal to dual (quadrature) space
- Fit Chebyshev polynomials to values at quadrature points
- Evaluate Chebyshev polynomials at arbitrary points
- Evaluate either values or derivatives (more expensive)

libCEED Basis Evaluation to Points

Interpolation to Chebyshev has same FLOPs as FEM $\mathcal{O}\left(q^4\right)$

- Invert map C^{-1} from quadrature points to Chebyshev coeffs
- Create 1D interpolation matrix $B = CN$
- Tensor product:
  $B = (C \otimes C \otimes C)(N \otimes N \otimes N) = (CN) \otimes (CN) \otimes (CN)$
- Additional cost from evaluation to arbitrary points

libCEED Basis Evaluation to Points

Interpolation to Chebyshev has same FLOPs as FEM $\mathcal{O}\left(q^4\right)$

- Invert map C^{-1} from quadrature points to Chebyshev coeffs
- Create 1D interpolation matrix $B = CN$
- Tensor product:
  $B = (C \otimes C \otimes C)(N \otimes N \otimes N) = (CN) \otimes (CN) \otimes (CN)$
- Additional cost from evaluation to arbitrary points

libCEED Basis Evaluation to Points

Per point evaluation has higher FLOPs $\mathcal{O}\left(q^6\right)$ (same as non-tensor)

- Recurrence for Chebyshev values at point
  $f_0 = 1,\ f_1 = 2x,\ f_n = 2xf_{n-1} - f_{n-2}$
  $f_0' = 0,\ f_1' = 2,\quad f_n' = 2xf_{n-1}' + 2f_{n-1} - f_{n-2}'$
- Contract pencil of values with element coefficients
- Operation is independent per quadrature point
- $\mathcal{O}\left(q^3\right)$ FLOPs at $\mathcal{O}\left(\hat{q}^3\right)$ points (often $q \approx \hat{q}$)

AtPoints Operator

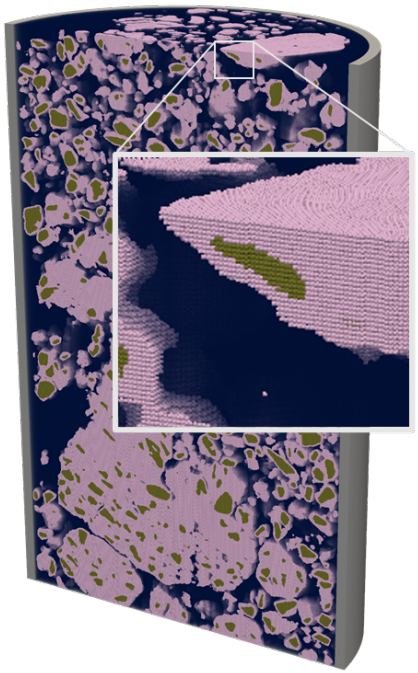
Final operator very similar to FEM

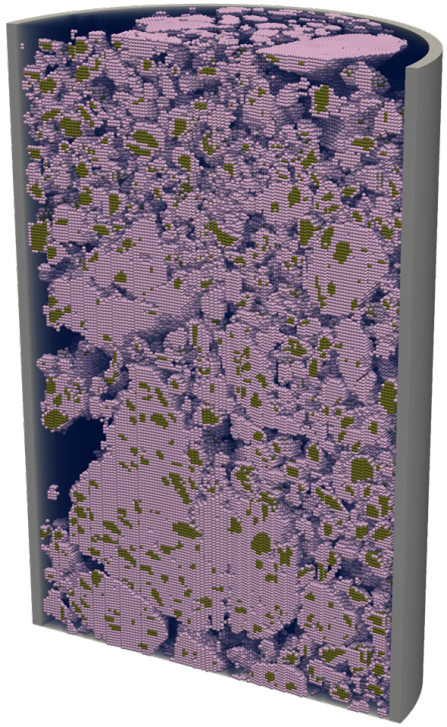
- $L = \mathcal{E}^T B^T B^{eT} D B^e B \mathcal{E}$ - CeedOperator
- All other operations identical to FEM
- libCEED gives action of local MPM operator
- PETSc responsible for communication between devices

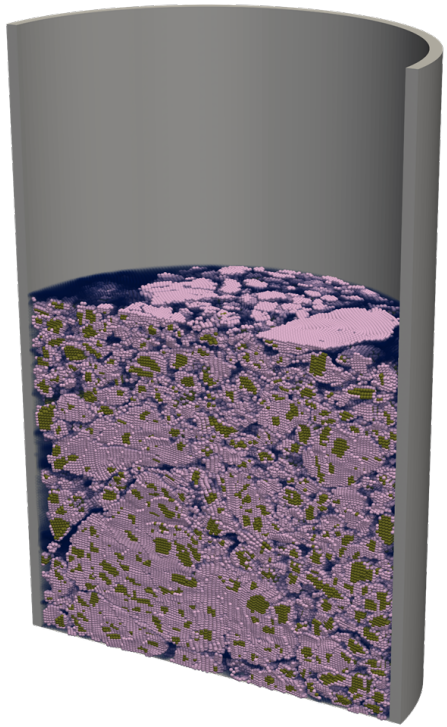
  $A = P^T L P$

Example - Press Simulation

316 MPoints







Compression of mock HE grains (gold) and binder (pink) mixture
(Reset background mesh to computational region on each timestep)

Two Families of Approaches

Three libCEED backends with two approaches to operator application

- Separate kernels
  - `/gpu/*/ref` and `/gpu/*/shared`
  - $\mathcal{E}$, B, and D all separate kernels
  - Higher overall memory usage, multiple kernel launches
- Fused kernel (focus for this talk)
  - `/gpu/*/gen`
  - Single kernel JiTed with data from $\mathcal{E}$, B, and D
  - Lower overall memory usage, single kernel launch

Operator Code

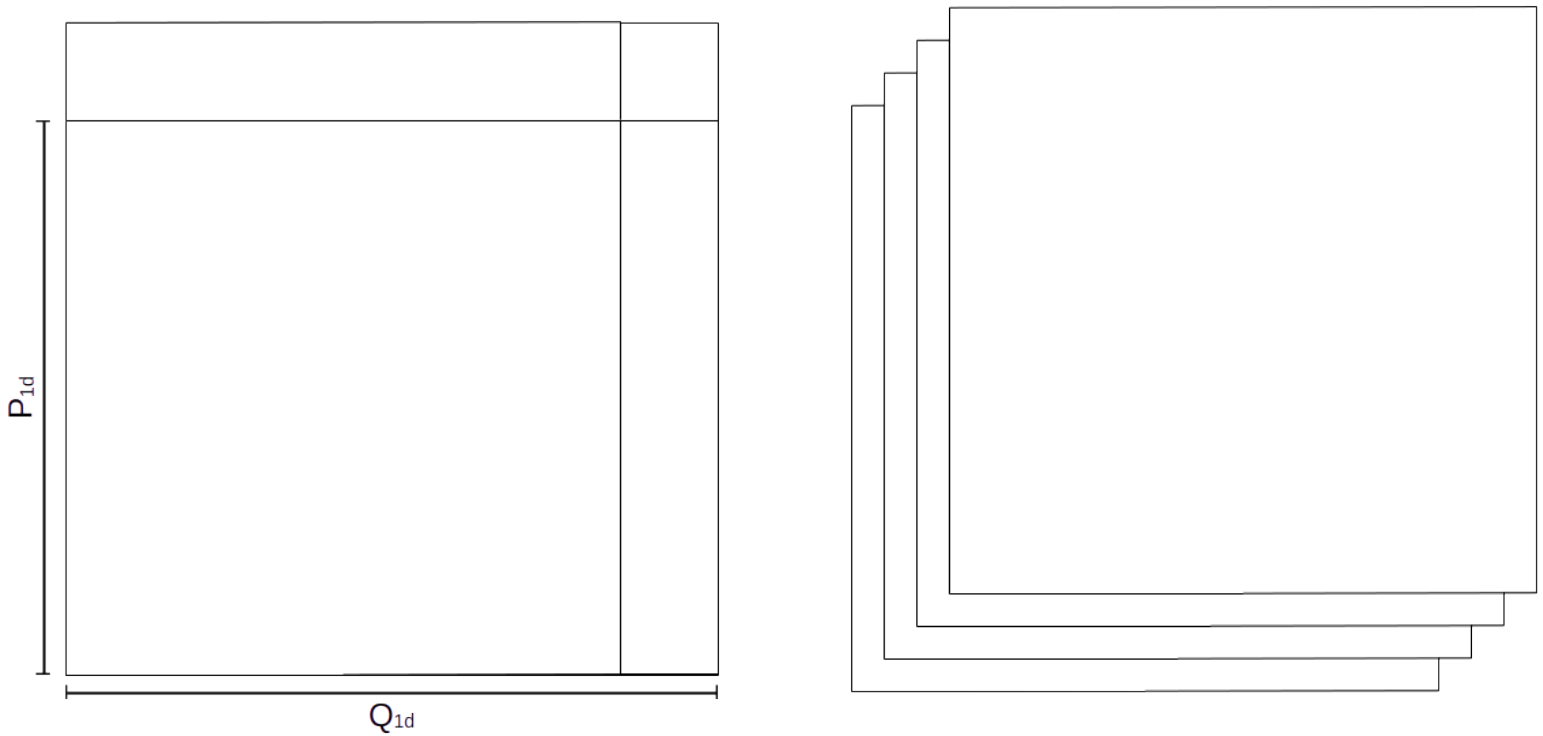
```
extern "C" __global__ void CeedKernelCudaGenOperator_mass(CeedInt nelem,
    void* ctx, FieldsInt_Cuda indices, Fields_Cuda fields, Fields_Cuda B,
    Fields_Cuda G, CeedScalar *W, Points_Cuda points) {
 // Setup kernel data

 // Input and Output field constants and basis data

 // Element loop
 __syncthreads();
 for (int e=blockIdx.x*blockDim.z+threadIdx.z; e<nelem; e+=gridDim.x*blockDim.z) {
  // -- Input field restrictions (E) and basis actions (B) to coeffs

  // -- Output field setup

  // -- Point batch loop
  for (int i=threadIdx.x+threadIdx.y*blockDim.x; i<bound; i+=blockDim.x*blockDim.y) {
   const int p=i%max_npts;
   // -- Input field basis action (B^e) to pts

   // -- Apply QFunction (D)
   mass(ctx, 1, inputs, outputs);

   // -- Output field basis action (B^eT) to coeffs
  }
  // -- Output field basis actions (B^T) and restrictions (E^T)
 } }
```

Single kernel with E, B, Q, B^T, E^T

Assembly

- Single operator kernel gives higher performance
- Higher memory usage for iMPM vs FEM
- Similar single kernels for assembly
- MPM element matrices/diagonals assembled by applying unit vectors
- MPM assembly more expensive than FEM assembly

Thread Usage



x and y thread index gives point (2D) or column of points (3D)

3D strategy works on 2D planes of points

Interp3d

```
template <int NCOMP, int P, int Q, int T>
inline __device__ void InterpTensor3d(SharedData_Cuda &dat,
    const CeedScalar *__restrict__ r_U, const CeedScalar *c_B,
 CeedScalar *__restrict__ r_V) {
 CeedScalar r_t1[T], r_t2[T]; // Memory per thread is a mild concern
 for (int c=0; c<NCOMP; c++) {
  ContractX3d<NCOMP, P, Q, T>(dat, &r_U[c*P], c_B, r_t1);
  ContractY3d<NCOMP, P, Q, T>(dat, r_t1, c_B, r_t2);
  ContractZ3d<NCOMP, P, Q, T>(dat, r_t2, c_B, &r_V[c*Q]);
 } }
```

```
template <int NCOMP, int P, int Q, int T>
inline __device__ void ContractX3d(SharedData_Cuda &dat,
    const CeedScalar *U, const CeedScalar *B, CeedScalar *V) {
 CeedScalar r_B[P_1D];
 for (int i=0; i<P; i++) r_B[i] = B[i+dat.tidx*P];
 for (int k=0; k<P; k++) {
  __syncthreads();
  dat.slice[dat.tidx+dat.tidy*T] = U[k];
  __syncthreads(); // Threads don't tend to be far off
  V[k] = 0.0;
  if (dat.tidx<Q && dat.tidy<P) {
   for (int i=0; i<P; i++) V[k]+=r_B[i]*dat.slice[i+dat.tidy*T];
  } } }
```

Interpolation mapping to the coefficients

Of Note

- Small memory usage per thread unless P/Q large
- Sync threads only once per 2D plane per contraction and comp
- Conservative shared memory usage for tensor-product elements

InterpAtPoints3d

```
template <int NCOMP, int NPTS, int P, int Q>
inline __device__ void InterpAtPoints3d(SharedData_Cuda &dat,
    const CeedInt p, const CeedScalar *__restrict__ r_C,
    const CeedScalar *r_X, CeedScalar *__restrict__ r_V) {
 for (int i=0; i<NCOMP; i++) r_V[i] = 0.0;
 for (int k=0; k<Q; k++) {
  CeedScalar buffer[Q], cheb_x[Q]; // Same size scratch space
  ChebyshevPolynomialsAtPoint<Q>(r_X[2], chebyshev_x);
  const CeedScalar z=chebyshev_x[k]; // z contraction value
  for (int c=0; c<NCOMP; c++) {
   // Load coefficients
   __syncthreads();
   if (dat.tidx<Q&&dat.tidy<Q) dat.slice[dat.tidx+dat.tidy*Q]=r_C[k+c*Q];
   __syncthreads(); // Again, threads tend not to drift
   // Contract x direction
   ChebyshevPolynomialsAtPoint<Q>(r_X[0], cheb_x);
   for (int i=0; i<Q; i++) {
    buffer[i] = 0.0;
    for (int j=0; j<Q; j++) buffer[i] += cheb_x[j]*dat.slice[j+i*Q];
   }
   // Contract y and z direction
   ChebyshevPolynomialsAtPoint<Q>(r_X[1], cheb_x);
   for (int i=0; i<Q; i++) r_V[c] += cheb_x[i]*buffer[i]*z;
  } } }
```

Interpolation to points (or grad) is straightforward

Of Note

- Faster AtPoints kernels
- Similar size thread local temp buffers
- Minor difference for grad - use Chebyshev derivative recursion
- Only sync threads once per 2D plane and comp

InterpTransposeAtPoints3d

```
template <int NCOMP, int NPTS, int P, int Q>
inline __device__ void InterpTransposeAtPoints3d(SharedData_Cuda &dat,
    const CeedInt p, const CeedScalar *__restrict__ r_U,
    const CeedScalar *r_X, CeedScalar *__restrict__ r_C) {
 for (int k=0; k<Q; k++) {
  CeedScalar buffer[Q], cheb_x[Q];
  ChebyshevPolynomialsAtPoint<Q>(r_X[2], cheb_x);
  const CeedScalar z = cheb_x[k]; // z contraction value
  for (int c=0; c<NCOMP; c++) {
   // Clear shared memory
   if (dat.tidx<Q&&dat.tidy<Q) dat.slice[dat.tidx+dat.tidy*Q] = 0.0;
   __syncthreads();
   // Contract y and z direction
   ChebyshevPolynomialsAtPoint<Q_1D>(r_X[1], cheb_x);
   const CeedScalar r_u = p<NPTS ? r_U[c] : 0.0;
   for (int i=0; i<Q; i++) buffer[i] = cheb_x[i]*r_u*z;
   // Contract x direction
   ChebyshevPolynomialsAtPoint<Q>(r_X[0], cheb_x);
   for (int i=0; i<Q; i++) {
    const int ii=(i+dat.tidy)%Q; // Note: shifting to avoid conflicts
    for (int j=0; j<Q; j++) {
     const int jj=(j+dat.tidx)%Q; // More shifting
     // HERE IS THE EXPENSIVE BIT! All points contribute to each coeff
     if(dat.tidx<Q&&dat.tidy<Q) atomicAdd(&dat.slice[jj+ii*Q],cheb_x[jj]*buffer[ii]);
    }
   }
   // Pull from shared to register
   __syncthreads();
   if (dat.tidx<Q&&dat.tidy<Q) r_C[k+c*Q] += dat.slice[dat.tidx+dat.tidy*Q];
  } } }
```

Of Note

- Slowest AtPoints kernels
- Similar size thread local temp buffers
- Only sync threads once per 2D plane and comp
- $Q * Q$ `atomicAdd` per 2D plane and comp

CEED Benchmark Problems

Performance on CEED BPs

- BP1 - Scalar projection problem
- BP2 - 3 component projection problem
- BP3 - Scalar Poisson problem
- BP4 - 3 component Poisson problem

Bulk of FLOPs are in basis evaluation

CEED Benchmark Problems

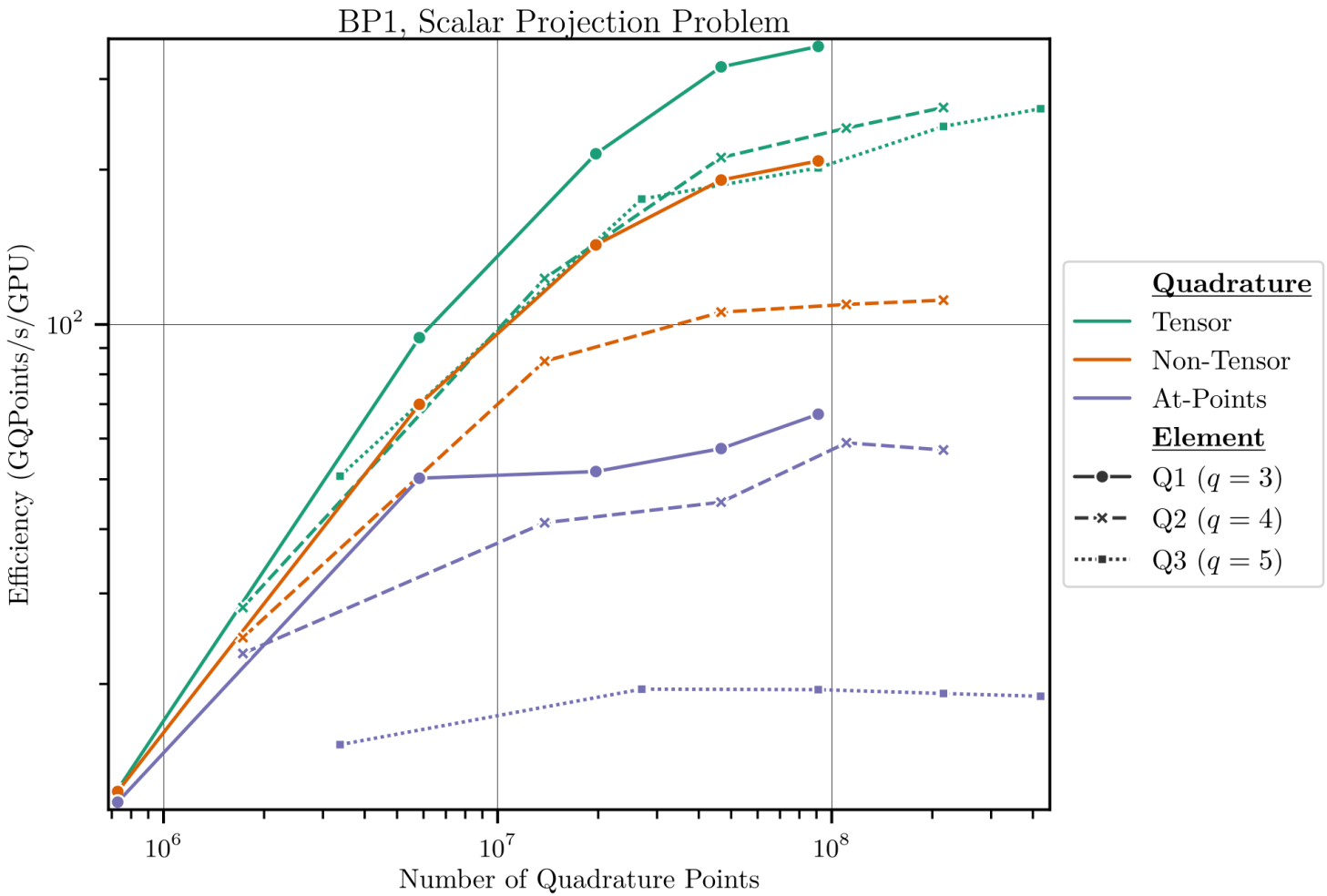
Performance on CEED BPs

- $p = 2, 3, 4$ and $\boldsymbol{q = p + 1}$
- Units cube with 30^3, 60^3, 90^3, 120^3, and 150^3 elements
- Compare tensor, non-tensor, and at-points basis evaluation
- MMS w/ partial sum of Weierstrass function, $a = 0.5$, $b = 1.5$, $N = 2$

Using 4x AMD Instinct™MI300A Accelerated Processing Units (APUs)

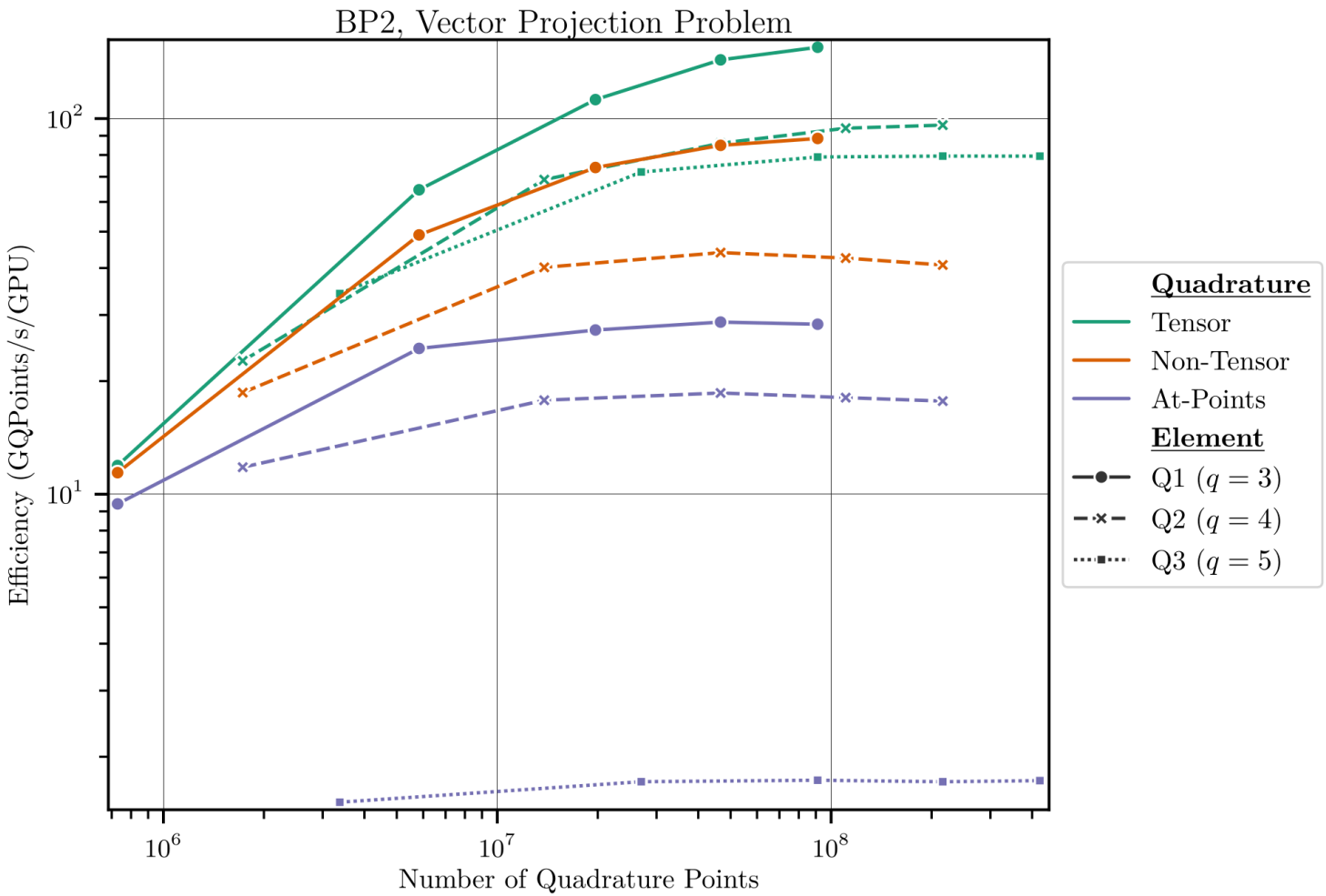
Aside - Unified memory important b/c point location/migration is on CPU

BP1



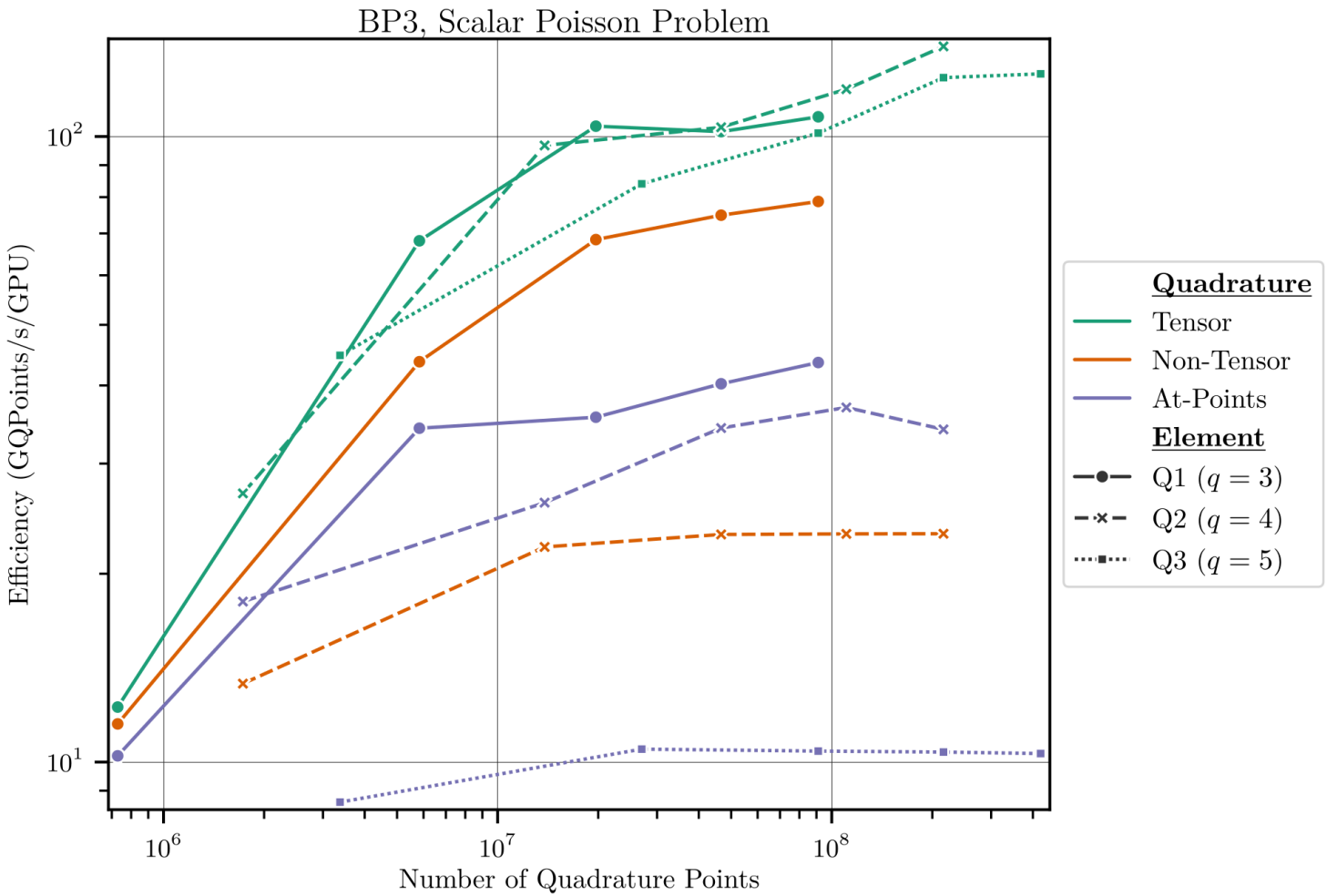
More FLOPs to do leads to lower efficiency

BP2



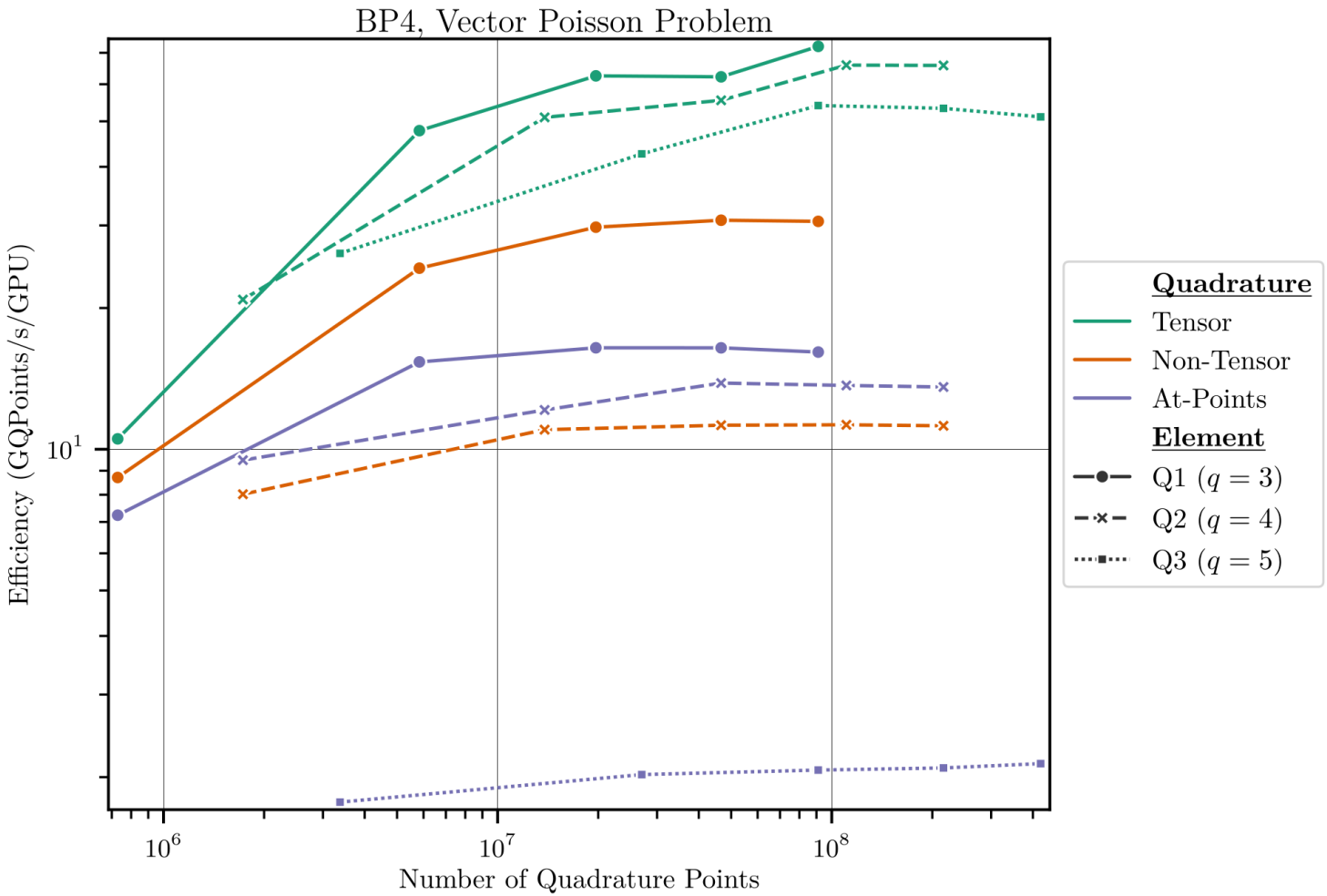
With more components, reach peak efficiency faster

BP3



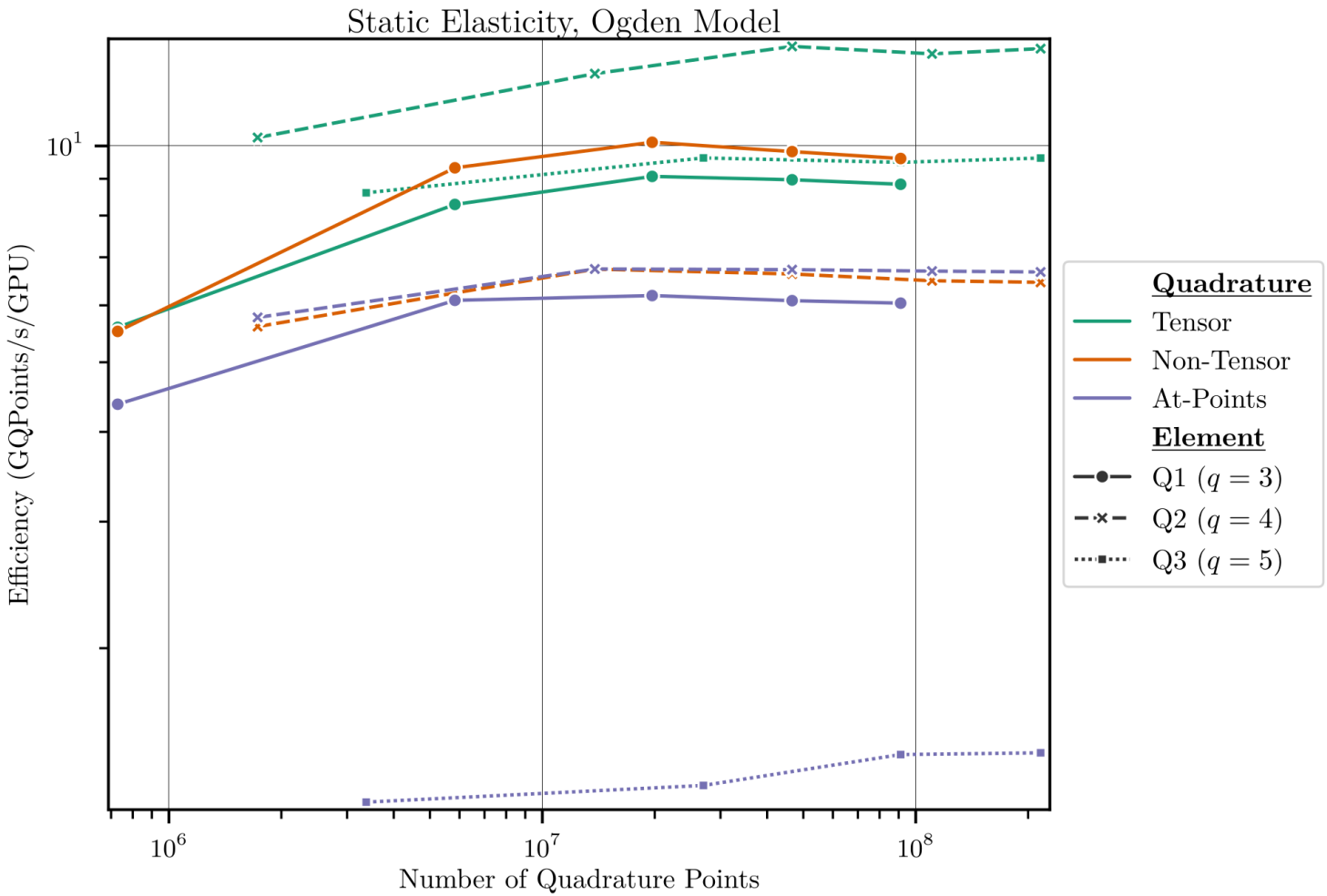
With derivatives, at-points closer to non-tensor

BP4



Closest benchmark to representative workload

Ogden



Basis cost less important with heavier QFunctions

Preconditioning

Practical problems require preconditioning

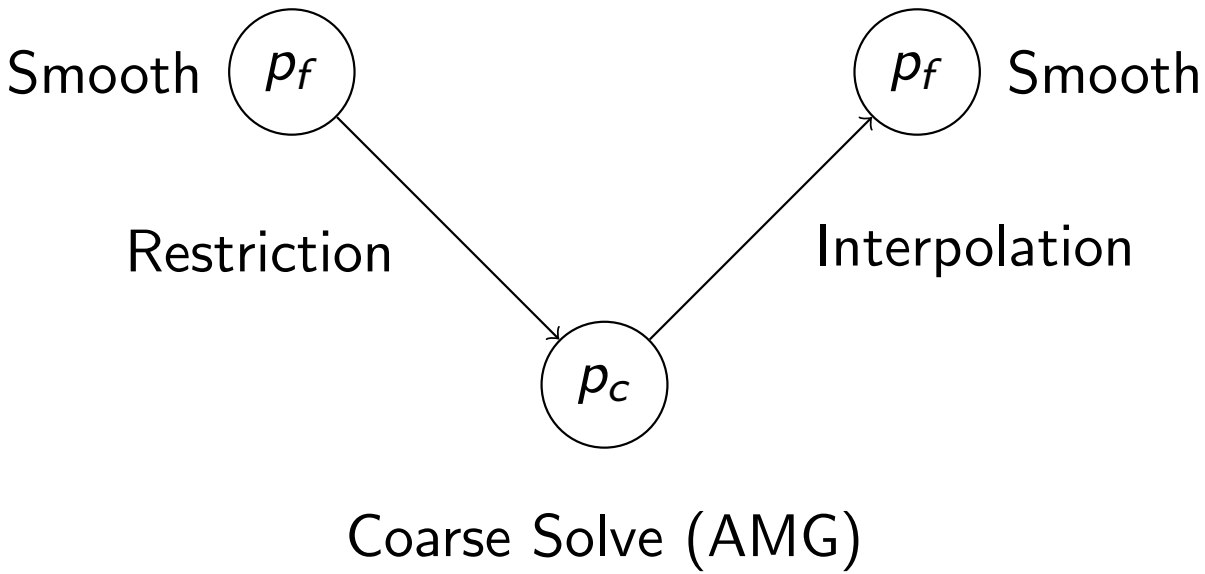
- Problems for MPM tend to be poorly conditioned
- Poor conditioning + expensive Mat-Vec = need preconditioning
- Varying structure between elements makes assembly more difficult

PETSc PCMG

- PCMG - PETSc geometric multigrid preconditioner
- Requires several operators from the user
  - Restriction operator
  - Interpolation operator
  - Smoother
  - Coarse grid solver

Ratel PCpMG

2 level multigrid with PCpMG



Ratel PCpMG

pMG giving promising initial results with GPU impl

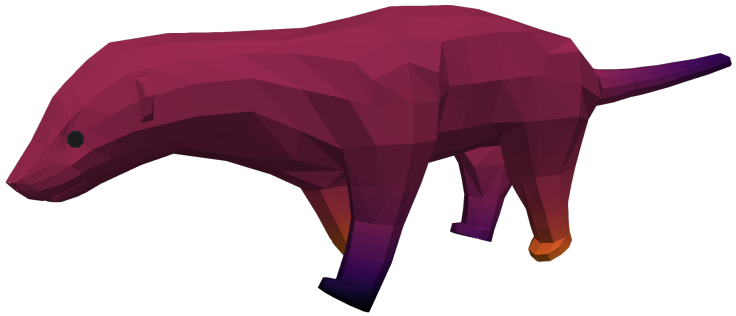
- Finite strain elasticity with damage
- Confined press of grain/binder with "sticky air" voids
- Jacobi iterations tend to double with 2x refinement
- pMG iteration counts robust with refinement

	# MPM Points	Jacobi its	pMG its
Coarse	388,800	900-1000	35-45
Fine	7,372,800	-	25-40

Future Work

- Continued iMPM development
- AtPoints basis and assembly perf tuning
- More models using Automatic Differentiation
- Further contact models development
- Rust QFunctions
- UHyper, UMat integration
- Addition of fluid dynamics models
- Upstream PETSc + libCEED integration
- We invite contributors and friendly users

Questions?



Repository: https://gitlab.com/micromorph/ratel

Ratel Team: Zach R. Atkins, Jed Brown, Fabio Di Gioacchino, Leila Ghaffari, Zach Irwin, Rezgar Shakeri, Ren Stengel, Jeremy L Thompson

Grant: Predictive Science Academic Alliance Program (DE-NA0003962)