

Meta-Meta Programming for Performance

Jeremy L Thompson

Helmholtz-Zentrum Hereon

jeremy@jeremylt.org

09 Oct 2026

Matrix-free finite elements present performance benefits

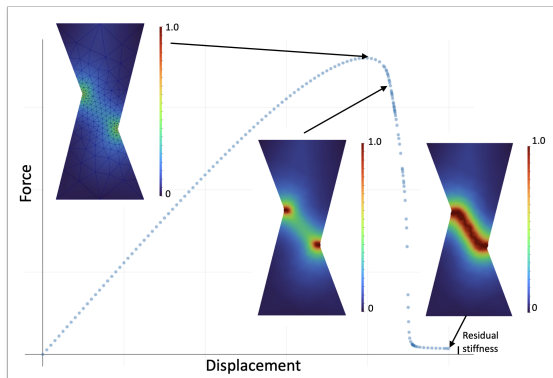
Exact parameter values allow compilers to emit more optimized code
but its impossible to compile full parameter space ahead of time

We'll see how Just in Time (JiT) compilation allows
libCEED to generate optimized code at runtime

Overview

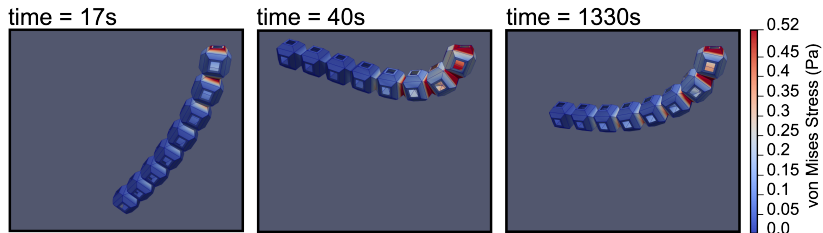
- 1 Motivating Examples
- 2 Finite Elements
- 3 Matrix-Free
- 4 Meta-Meta Programming
- 5 Questions

Example - Brittle Fracture



Quasistatic simulation of a compressive shear test for a generic brittle material

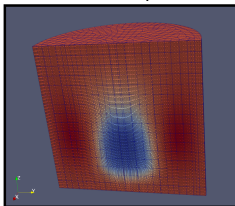
Example - Schwarz Pendulum



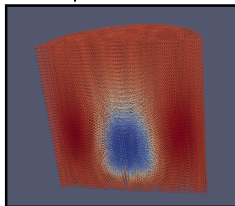
Dynamic simulation of 8 Schwarz-P cells with an applied traction force creating pendulum-like motion

Example - Sinker Simulation

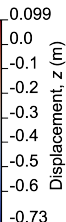
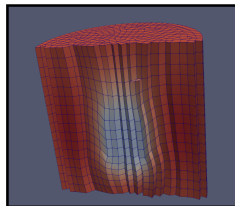
iMPM - mesh & particles



iMPM - particles



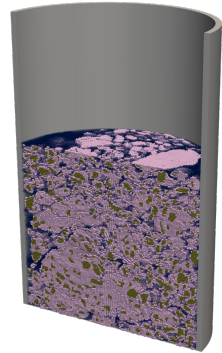
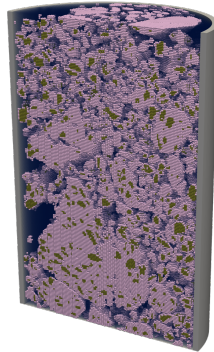
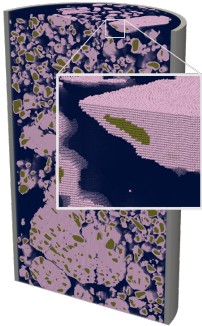
FEM - mesh



Quasistatic simulation of a cylinder with a dense high-modulus inclusion surrounded by a low-density low-modulus near-incompressible "foam"

Example - Press Simulation

316 MPoints



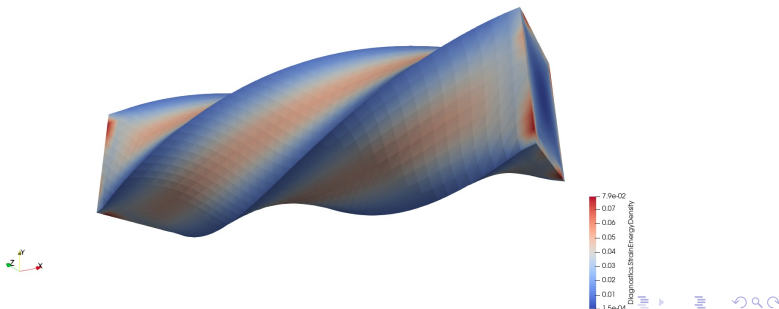
Compression of mock HE grains (gold) and binder (pink) mixture
(Reset background mesh to computational region on each timestep)

End Goal

We need to transform equations:

$$\begin{aligned} -\nabla \cdot \sigma - \rho g &= 0, \text{ in } \Omega \\ u &= \bar{u}, \text{ on } \partial\Omega^D \\ \sigma \cdot n &= \bar{t}, \text{ on } \partial\Omega^N \end{aligned}$$

into **high-performance** simulations:



Equation to Solve

Consider the general non-linear residual

$$F(u) - R = 0$$

where

- $F(u)$ represents our 'physics'
- R is some 'right hand side' or forcing term
- u is the unknown solution

Problem

Computers like discrete equations/operations

Our equation is continuous, and computers *hate* that

Weak Form

With the L^2 inner product

$$\langle v, u \rangle = \int_{\Omega} v u \, dV$$

We can reformulate to the 'weak form':

$$\langle v, f(u) \rangle = \int_{\Omega} v \cdot f_0(u, \nabla u) + \nabla v : f_1(u, \nabla u) \, dV = 0$$

Note:

- boundary terms are omitted here (integration by parts)
- v is in some homogeneous space V_0

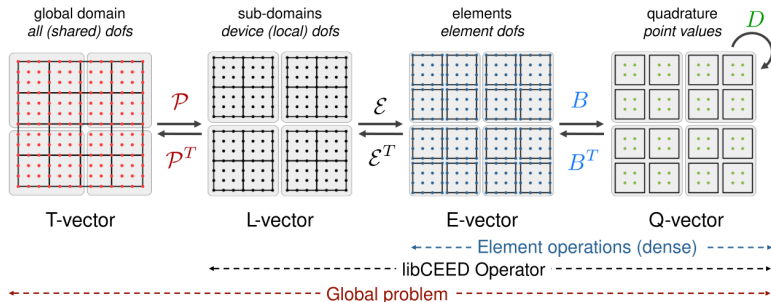
Discretization

How to discretize?

- $u, v \in V$
- Different V give different properties
- We select V to be the space of piecewise continuous polynomials
- No continuity in derivatives (not splines)

The 'Elements'

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



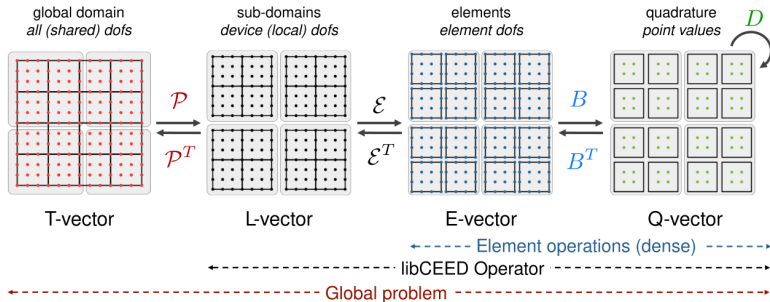
T(otal)-vector represents discretization

Want solution values on red dots (Degrees of Freedom [DoF])

Fitting piecewise polynomial to red dots gives continuous solution

The 'Basis'

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



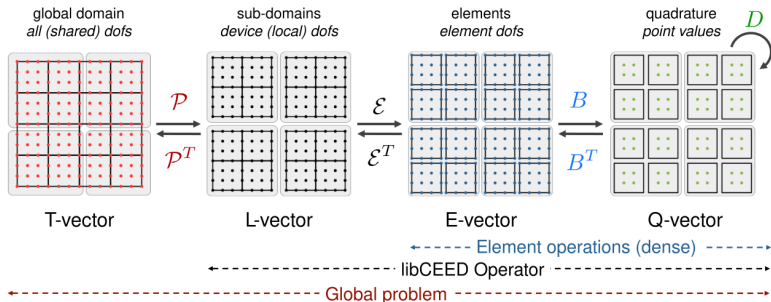
E(lement)-vector represents each 'piece' of our solution polynomial

DoFs on blue dots are red dots on a specific element

B maps to the quadrature points - green dots

The 'Physics'

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



Q(uadrature)-vector is where our physics is evaluated

Green dots represent values/derivatives in quadrature space

D evaluates f_0 , f_1 , multiply by quadrature weights

Galerkin Form

Galerkin form for an arbitrary second order PDE:

$$\sum_e \mathcal{E}^T \left[(\mathbf{N}^e)^T \mathbf{W}^e \Lambda(f_0(u^e, \nabla u^e)) + \sum_{i=0}^{d-1} (\mathbf{D}_i^e)^T \mathbf{W}^e \Lambda(f_1(u^e, \nabla u^e)) \right] = 0$$

- $\mathcal{E}$ - element assembly/restriction operator
- $\mathbf{N}^e$ - interpolation to quadrature points
- $\mathbf{D}_i^e$ - derivatives at quadrature points
- $\mathbf{W}^e$ - quadrature weights
- Λ - pointwise multiplication at quadrature points
- $u^e = \mathbf{N}^e \mathcal{E}^e u$ and $\nabla u^e = \{\mathbf{D}_i^e \mathcal{E}^e u\}_{i=0}^{d-1}$

Wait, What?!?

Non-linear problem:

$$F(u) - R = 0$$

Weak form:

$$\langle v, f(u) \rangle = \int_{\Omega} v \cdot f_0(u, \nabla u) + \nabla v : f_1(u, \nabla u) dV = 0$$

Galerkin form:

$$\sum_e \mathcal{E}^T \left[(\mathbf{N}^e)^T \mathbf{W}^e \Lambda(f_0(u^e, \nabla u^e)) + \sum_{i=0}^{d-1} (\mathbf{D}_i^e)^T \mathbf{W}^e \Lambda(f_1(u^e, \nabla u^e)) \right] = 0$$

Wait, What?!? - Linear Equation

Linearized problem:

$$F(u_k) - R + J(u_k)(u_k - u_{k+1}) = 0$$

Weak form:

$$\langle v, J(u_k) w \rangle = \int_{\Omega} \begin{bmatrix} v^T & (\nabla v)^T \end{bmatrix} \begin{bmatrix} f_{0,0} & f_{0,1} \\ f_{1,0} & f_{1,1} \end{bmatrix} \begin{bmatrix} w \\ \nabla w \end{bmatrix}$$

Galerkin form:

$$\sum_e \mathcal{E}^T \left[(\mathbf{N}^e)^T \mathbf{W}^e \Lambda(f_0^*(w^e, \nabla w^e)) + \sum_{i=0}^{d-1} (\mathbf{D}_i^e)^T \mathbf{W}^e \Lambda(f_1^*(w^e, \nabla w^e)) \right] = 0$$

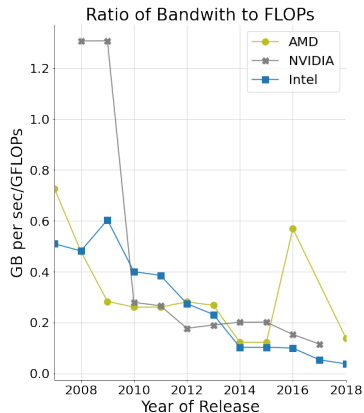
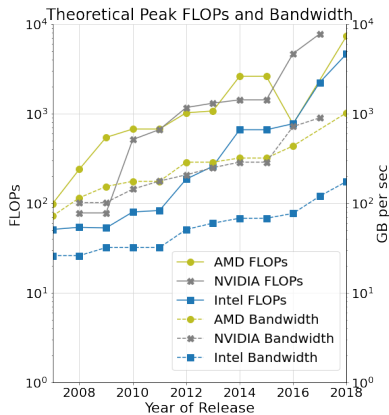
What I'm Handwaving

- Elements can have deformed shapes (unstructured meshes)
- Need 'geometric' factors for each individual element
(Change of coordinate systems from physical to reference)
- $\mathcal{P}$ is parallel communication (supercomputers!)
(T-vector $\leftrightarrow$ L-vector in diagram)
- R (and its weak analogue) can be considered part of f_0 or separate
- Boundary terms are similar but 1 dimension lower (i.e. $3D \rightarrow 2D$)
- Linearization isn't fundamentally hard but requires care

Traditional Approach

- The Galerkin form naturally creates a sparse matrix for linear $J(u_k)$
- This sparse matrix can be solved with linear algebra
- But sparse matrices don't use hardware optimally!
- Big Idea: perform the operations without forming a matrix
- libCEED provides the action of $F(u)$ and $J(u_k)$ matrix-free

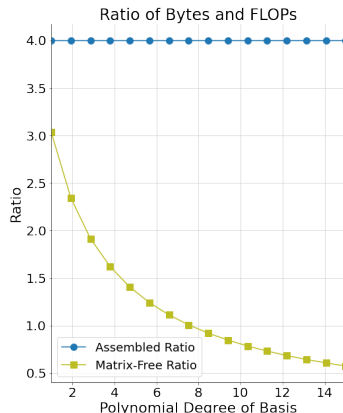
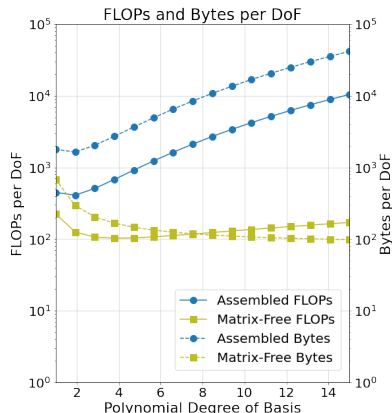
Modern Hardware



Modern hardware has lower memory bandwidth than FLOPs

Electricity can only move so fast!

Matrix-Free



Requirements for matrix-vector product with sparse matrix vs matrix-free
for screened Poisson $\nabla^2 u - \alpha^2 u = f$ in 3D

Matrix-free representations using tensor product bases
better match modern hardware limitations

Finally!

- Problem: We have many parameters and loops to optimize
- Template metaprogramming generalizes a function over some inputs
- The template parameters become compile-time constants
- Our (many) loops can be unrolled!

Example:

```
1  template <int Q>
2  int ApplyQWeights(const double u*, const double w*, double v*)
3  {
4      for (int i = 0; i < Q; i++) v[i] = w[i] * u[i];
5      return 0;
6  }
```

Cannot compile every possible combination of parameters ahead of time!

Meta-Meta Though?

`/gpu/*/gen` and `/cpu/self/gen/*` generate a single function

$A_L = \mathcal{E}^T B^T D B \mathcal{E}$ written at run-time, JiT compiled

$\mathcal{E}$ and B are templated math parameters

D source is given by the user

Generated Operator Kernel

/gpu/*/gen

```

1 extern "C" __global__ void CeedKernelCudaGenOperator_myCoolPhysics(CeedInt num_elem,
2   void* ctx, FieldsInt_Cuda indices, Fields_Cuda fields, Fields_Cuda B,
3   Fields_Cuda G, CeedScalar *W, Points_Cuda points) {
4   // Setup kernel data
5
6   // Input and Output field constants and basis data
7
8   // Element loop
9   __syncthreads();
10  for (CeedInt elem = blockIdx.x*blockDim.z + threadIdx.z; elem < num_elem;
11      elem += gridDim.x*blockDim.z) {
12    // -- Input E and B templated functions
13
14    // -- Output field setup
15
16    // -- Apply QFunction (D)
17    myCoolPhysics(ctx, 1, inputs, outputs);
18
19    // -- Output B^T and E^T templated functions
20  }
21 }
22 // -----

```

$A_L = \mathcal{E}^T B^T D B \mathcal{E}$ in a single GPU kernel

Generated Operator Function

/cpu/self/gen/*

```

1 static int CeedCpuGenOperator_myCoolPhysics(void *ctx, const FieldData_Cpu_Gen *inputs,
2       FieldData_Cpu_Gen *outputs) {
3     // Setup kernel data
4
5     // Input and Output field constants and basis data
6
7     // Loop over blocks
8     for (CeedInt block = 0; block < num_blocks; block++) {
9         // -- Input E and B templated functions
10
11        // -- Output field setup
12
13        // -- Apply QFunction (D)
14        myCoolPhysics(ctx, Q * BLOCK_SIZE, inputs, outputs);
15
16        // -- Output B^T and E^T templated functions
17    }
18    return CEED_ERROR_SUCCESS;
19 }
20 // -----

```

$A_L = \mathcal{E}^T \textcolor{blue}{B}^T \textcolor{blue}{D} \textcolor{blue}{B} \mathcal{E}$ in a single CPU function

Big Picture

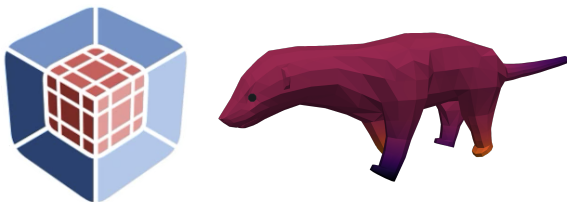
- User specifies their problem via libCEED API
- gen backends write .cpp file with function/kernel
- .cpp file compiled (Clang, NVRTC, etc)
- Kernel/function loaded and executed
- Great performance wins!

Help!

`/cpu/self/gen/*` is brand new!

- Hacked in a weekend of LLM-free joy and refined collaboratively (Protect your skill formation and joy folks!)
- Seen great gen benefits for GPU but CPU code hasn't been refined
- Several open issues in development
<https://github.com/CEED/libCEED/issues>
- Cool cutting edge work for science with a brand new feature
- I really enjoy mentoring people through contributing process

Questions?



libCEED Repo: <https://github.com/CEED/libCEED>

Ratel Repo: <https://gitlab.com/micromorph/ratel>

Meta-Meta Programming for Performance

Jeremy L Thompson

Helmholtz-Zentrum Hereon

jeremy@jeremylt.org

09 Oct 2026