Performance-Portable Implicit Scale-Resolving Compressible Flow Using libCEED

SIAM CSE 2023

James Wright

February 27, 2023

Ann and H.J. Smead Department of Aerospace Engineering Sciences

Smead Aerospace
UNIVERSITY OF COLORADO **BOULDER**

Outline

1. What is libCEED?
2. A (Very) Brief Finite Element Refresher
3. libCEED Overview
4. Fluid Simulations with libCEED

What is libCEED?

What is libCEED?

- C library for element-based discretizations

What is libCEED?

- C library for element-based discretizations
  - Bindings for Fortran, Rust, Python, and Julia available

What is libCEED?

- C library for element-based discretizations
    - Bindings for Fortran, Rust, Python, and Julia available
- Designed for matrix-free operator evaluation

What is libCEED?

- C library for element-based discretizations
  - Bindings for Fortran, Rust, Python, and Julia available
- Designed for matrix-free operator evaluation
- Capable of running on GPU/Accelerator hardware

What is libCEED?

- C library for element-based discretizations
  - Bindings for Fortran, Rust, Python, and Julia available
- Designed for matrix-free operator evaluation
- Capable of running on GPU/Accelerator hardware
  - Code that runs on CPU also runs on GPU without changes

What is libCEED?

- C library for element-based discretizations
  - Bindings for Fortran, Rust, Python, and Julia available
- Designed for matrix-free operator evaluation
- Capable of running on GPU/Accelerator hardware
  - Code that runs on CPU also runs on GPU without changes
  - Computational backend selectable at runtime via JIT compilation

What is libCEED?

- C library for element-based discretizations
  - Bindings for Fortran, Rust, Python, and Julia available
- Designed for matrix-free operator evaluation
- Capable of running on GPU/Accelerator hardware
  - Code that runs on CPU also runs on GPU without changes
  - Computational backend selectable at runtime via JIT compilation
- Geared toward high-order element discretizations

A (Very) Brief Finite Element Refresher

Galerkin Form

Given the homogenous Poisson equation in weak form: Find $u \in \mathcal{S}$ such that

$$\int_\Omega v_{,x} u_{,x} \, \mathrm{d}\Omega = \int_\Omega v f \, \mathrm{d}\Omega \quad \forall v \in \mathcal{V}$$

Galerkin Form

Given the homogenous Poisson equation in weak form: Find $u \in \mathcal{S}$ such that

$$\int_\Omega v_{,x} u_{,x} \, \mathrm{d}\Omega = \int_\Omega v f \, \mathrm{d}\Omega \quad \forall v \in \mathcal{V}$$

Approximate $\mathcal{S} \approx \mathcal{S}^h = \mathrm{span}(\{\phi^i\}_i^{n_\mathrm{dof}})$ and $\mathcal{V} \approx \mathcal{V}^h = \mathrm{span}(\{\phi^j\}_j^{n_\mathrm{dof}})$

Galerkin Form

Given the homogenous Poisson equation in weak form: Find $u \in \mathcal{S}$ such that

$$\int_\Omega v_{,x} u_{,x} \, \mathrm{d}\Omega = \int_\Omega v f \, \mathrm{d}\Omega \quad \forall v \in \mathcal{V}$$

Approximate $\mathcal{S} \approx \mathcal{S}^h = \mathrm{span}(\{\phi^i\}_i^{n_{\mathrm{dof}}})$ and $\mathcal{V} \approx \mathcal{V}^h = \mathrm{span}(\{\phi^j\}_j^{n_{\mathrm{dof}}})$

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

Galerkin Form

Given the homogenous Poisson equation in weak form: Find $u \in \mathcal{S}$ such that

$$\int_\Omega v_{,x} u_{,x} \, \mathrm{d}\Omega = \int_\Omega v f \, \mathrm{d}\Omega \quad \forall v \in \mathcal{V}$$

Approximate $\mathcal{S} \approx \mathcal{S}^h = \mathrm{span}(\{\phi^i\}_i^{n_\mathrm{dof}})$ and $\mathcal{V} \approx \mathcal{V}^h = \mathrm{span}(\{\phi^j\}_j^{n_\mathrm{dof}})$

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

$$u = \sum_i^{n_\mathrm{dof}} \phi^i x_i \in \mathcal{S}^h, \quad A_{ij} = \int_\Omega \phi^i_{,x} \phi^j_{,x} \, \mathrm{d}\Omega, \quad b_j = \int_\Omega \phi^j f \, \mathrm{d}\Omega$$

Finite Element Method

Discretize the domain Ω into elements Ω^e

$$\Omega^h = \bigcup_{e=0}^{n_{\mathrm{elm}}} \Omega^e \approx \Omega$$

Finite Element Method

Discretize the domain Ω into elements Ω^e

$$\Omega^h = \bigcup_{e=0}^{n_{\mathrm{elm}}} \Omega^e \approx \Omega$$

Approximate the integrals over elements via quadrature

$$\int g(x)\,\mathrm{d}x \approx \sum_{k}^{n_{\mathrm{quad}}} w^k g(\xi^k)$$

Finite Element Method

Discretize the domain Ω into elements Ω^e

$$\Omega^h = \bigcup_{e=0}^{n_{\mathrm{elm}}} \Omega^e \approx \Omega$$

Approximate the integrals over elements via quadrature

$$\int g(x)\,\mathrm{d}x \approx \sum_k^{n_{\mathrm{quad}}} w^k g(\xi^k)$$

Thus the final problem can be stated as: Find $\boldsymbol{x} \in \mathbb{R}^{n_{\mathrm{dof}}}$ in $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ for

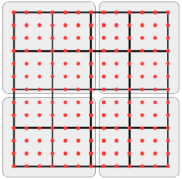
$$A_{ij} = \sum_e^{n_{\mathrm{elm}}} \sum_k^{n_{\mathrm{quad}}} \left[w^k \phi^i_{,x}(\xi^k) \phi^j_{,x}(\xi^k) \right]_{\Omega^e}, \quad b_j = \sum_e^{n_{\mathrm{elm}}} \sum_k^{n_{\mathrm{quad}}} \left[w^k \phi^j(\xi^k) f(\xi^k) \right]_{\Omega^e}$$

libCEED Overview

Finite Element Operator Decomposition

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

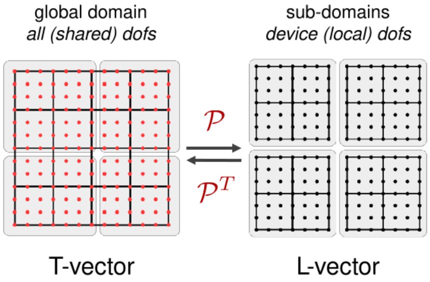
global domain
all (shared) dofs



T-vector

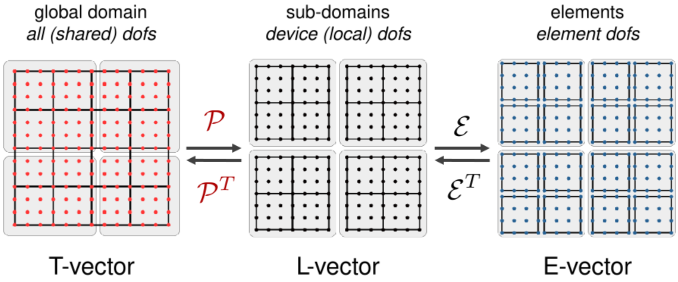
Finite Element Operator Decomposition

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



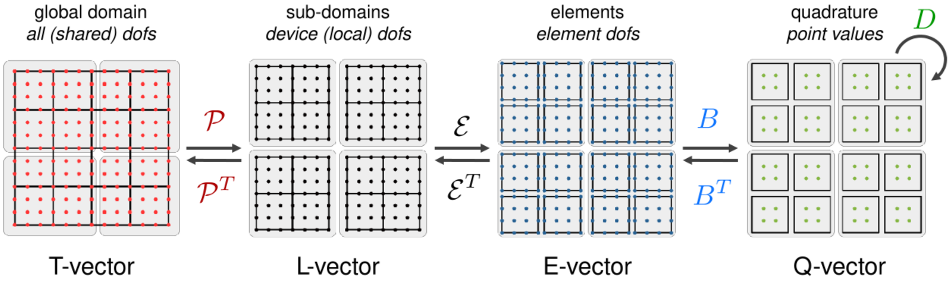
Finite Element Operator Decomposition

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



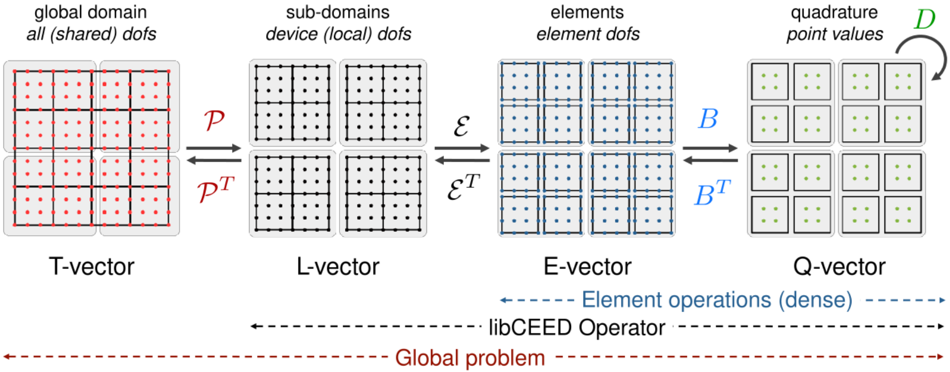
Finite Element Operator Decomposition

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



Finite Element Operator Decomposition

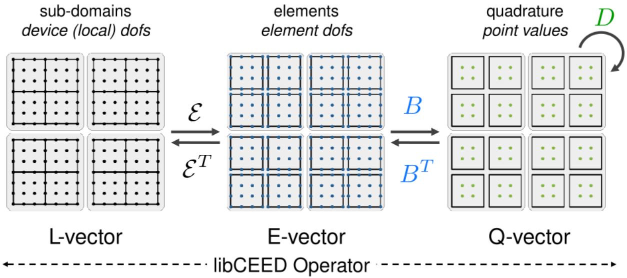
$$A = \mathcal{P}^T \mathcal{E}^T B^T D B \mathcal{E} \mathcal{P}$$



libCEED Operator

Operator	libCEED Object
$\mathcal{E}$	`CeedElemRestriction`

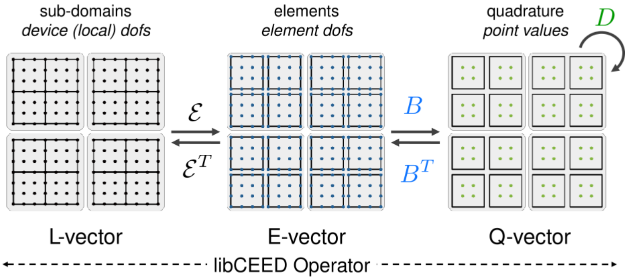
$$L = \mathcal{E}^T B^T D B \mathcal{E}$$



libCEED Operator

Operator	libCEED Object
$\mathcal{E}$	`CeedElemRestriction`
B	`CeedBasis`

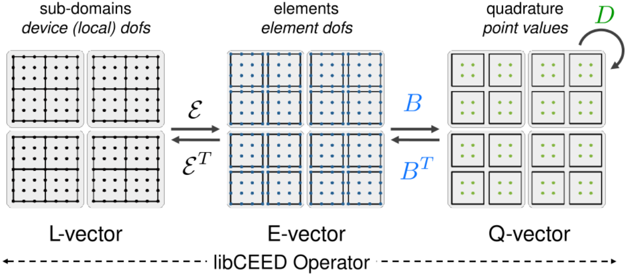
$$L = \mathcal{E}^T B^T D B \mathcal{E}$$



libCEED Operator

Operator	libCEED Object
$\mathcal{E}$	`CeedElemRestriction`
B	`CeedBasis`
D	`CeedQFunction`

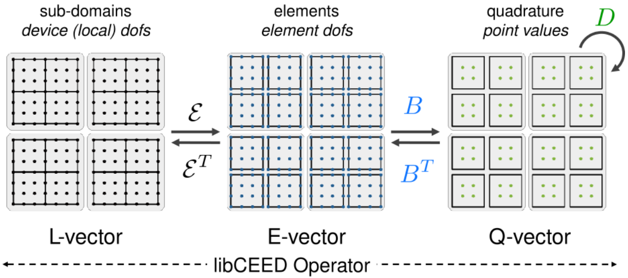
$$L = \mathcal{E}^T B^T D B \mathcal{E}$$



libCEED Operator

Operator	libCEED Object
$\mathcal{E}$	`CeedElemRestriction`
B	`CeedBasis`
D	`CeedQFunction`
$\mathcal{E}^T B^T D B \mathcal{E}$	`CeedOperator`

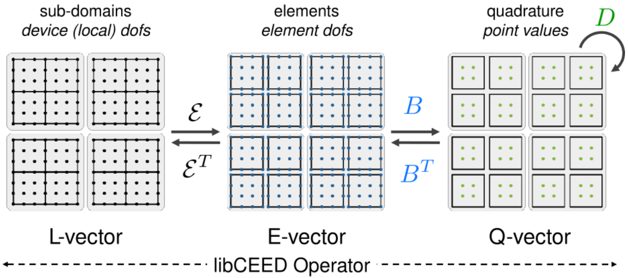
$$L = \mathcal{E}^T B^T D B \mathcal{E}$$



libCEED Operator

Operator	libCEED Object
$\mathcal{E}$	`CeedElemRestriction`
B	`CeedBasis`
D	`CeedQFunction`
$\mathcal{E}^T B^T D B \mathcal{E}$	`CeedOperator`

$$L = \mathcal{E}^T B^T D B \mathcal{E}$$



Note

libCEED implements the operators matrix-free; None of the libCEED Objects store a matrix

Defining the libCEED Operator

`CeedElemRestriction`

- Defined by element connectivity and the number of components

Defining the libCEED Operator

`CeedElemRestriction`

- Defined by element connectivity and the number of components

`CeedBasis`

- Defined by the basis functions and the quadrature rule
- Built-in options for common basis functions and quadrature rules

Defining the libCEED Operator

`CeedElemRestriction`

- Defined by element connectivity and the number of components

`CeedBasis`

- Defined by the basis functions and the quadrature rule
- Built-in options for common basis functions and quadrature rules

`CeedQFunction`

- C function
- Common operators available (mass, poisson, identity, etc.)
- User defined QFunctions also possible

Defining the libCEED Operator

`CeedElemRestriction`

- Defined by element connectivity and the number of components

`CeedBasis`

- Defined by the basis functions and the quadrature rule
- Built-in options for common basis functions and quadrature rules

`CeedQFunction`

- C function
- Common operators available (mass, poisson, identity, etc.)
- User defined QFunctions also possible

`CeedOperator`

- Computation backend defined at runtime
- Full L operator (can be) JITed to the desired backend (CUDA, HIP, CPU, etc.)

Poisson CeedQFunction

$$\sum_{k}^{n_{\text{quad}}} \left[w^k u_{,x}(\xi^k) v_{,x}(\xi^k) \right]_{\Omega^e}$$

$$B^T D B$$

```c
CEED_QFUNCTION(Poisson1DApply)(void *ctx, const CeedInt Q,
                               const CeedScalar *const *in,
                               CeedScalar *const *out) {
  // in[0] is gradient u, size (Q)
  // in[1] is quadrature data, size (Q)
  const CeedScalar *ug = in[0], *q_data = in[1];

  // out[0] is output to multiply against gradient v, size (Q)
  CeedScalar *vg = out[0];

  // Quadrature point loop
  CeedPragmaSIMD
  for (CeedInt i=0; i<Q; i++) {
    vg[i] = ug[i] * q_data[i];
  } // End of Quadrature Point Loop

  return CEED_ERROR_SUCCESS;
}
```

Poisson CeedQFunction

$$\sum_{k}^{n_{\text{quad}}} \left[w^k u_{,x}(\xi^k) v_{,x}(\xi^k) \right]_{\Omega^e}$$

$$B^T D B$$

- ug= $u_{,x}(\xi^k) = B u^e$

```c
CEED_QFUNCTION(Poisson1DApply)(void *ctx, const CeedInt Q,
                               const CeedScalar *const *in,
                               CeedScalar *const *out) {
  // in[0] is gradient u, size (Q)
  // in[1] is quadrature data, size (Q)
  const CeedScalar *ug = in[0], *q_data = in[1];

  // out[0] is output to multiply against gradient v, size (Q)
  CeedScalar *vg = out[0];

  // Quadrature point loop
  CeedPragmaSIMD
  for (CeedInt i=0; i<Q; i++) {
    vg[i] = ug[i] * q_data[i];
  } // End of Quadrature Point Loop

  return CEED_ERROR_SUCCESS;
}
```

Poisson CeedQFunction

$$\sum_{k}^{n_{\text{quad}}} \left[w^k u_{,x}(\xi^k) v_{,x}(\xi^k) \right]_{\Omega^e}$$

$$B^T D B$$

- ug= $u_{,x}(\xi^k) = B u^e$
- q_data= w^k[a]

[a] along with jacobian determinant

```c
CEED_QFUNCTION(Poisson1DApply)(void *ctx, const CeedInt Q,
                               const CeedScalar *const *in,
                               CeedScalar *const *out) {
  // in[0] is gradient u, size (Q)
  // in[1] is quadrature data, size (Q)
  const CeedScalar *ug = in[0], *q_data = in[1];

  // out[0] is output to multiply against gradient v, size (Q)
  CeedScalar *vg = out[0];

  // Quadrature point loop
  CeedPragmaSIMD
  for (CeedInt i=0; i<Q; i++) {
    vg[i] = ug[i] * q_data[i];
  } // End of Quadrature Point Loop

  return CEED_ERROR_SUCCESS;
}
```

Poisson CeedQFunction

$$\sum_{k}^{n_{\text{quad}}} \left[w^k u_{,x}(\xi^k) v_{,x}(\xi^k) \right]_{\Omega^e}$$

$$B^T D B$$

- ug= $u_{,x}(\xi^k) = B u^e$
- q_data= w^k[a]
- vg= $w^k u_{,x}(\xi^k) = D B u^e$

[a] along with jacobian determinant

```c
CEED_QFUNCTION(Poisson1DApply)(void *ctx, const CeedInt Q,
                               const CeedScalar *const *in,
                               CeedScalar *const *out) {
  // in[0] is gradient u, size (Q)
  // in[1] is quadrature data, size (Q)
  const CeedScalar *ug = in[0], *q_data = in[1];

  // out[0] is output to multiply against gradient v, size (Q)
  CeedScalar *vg = out[0];

  // Quadrature point loop
  CeedPragmaSIMD
  for (CeedInt i=0; i<Q; i++) {
    vg[i] = ug[i] * q_data[i];
  } // End of Quadrature Point Loop

  return CEED_ERROR_SUCCESS;
}
```

Poisson CeedQFunction

$$\sum_{k}^{n_{\text{quad}}} \left[w^k u_{,x}(\xi^k) v_{,x}(\xi^k) \right]_{\Omega^e}$$

$$B^T D B$$

- ug= $u_{,x}(\xi^k) = B u^e$
- q_data= w^k[a]
- vg= $w^k u_{,x}(\xi^k) = D B u^e$
- $w^k u_{,x}(\xi^k) v_{,x}(\xi^k) = B^T D B u^e$

[a] along with jacobian determinant

```c
CEED_QFUNCTION(Poisson1DApply)(void *ctx, const CeedInt Q,
                               const CeedScalar *const *in,
                               CeedScalar *const *out) {
  // in[0] is gradient u, size (Q)
  // in[1] is quadrature data, size (Q)
  const CeedScalar *ug = in[0], *q_data = in[1];

  // out[0] is output to multiply against gradient v, size (Q)
  CeedScalar *vg = out[0];

  // Quadrature point loop
  CeedPragmaSIMD
  for (CeedInt i=0; i<Q; i++) {
    vg[i] = ug[i] * q_data[i];
  } // End of Quadrature Point Loop

  return CEED_ERROR_SUCCESS;
}
```

Fluid Simulations with libCEED

Compressible Navier-Stokes

$$\boldsymbol{U}_{,t} + \boldsymbol{F}_{i,i}(\boldsymbol{U}) - S(\boldsymbol{U}) = 0$$

for

$$\boldsymbol{U} = \begin{bmatrix} \rho \\ \rho u_i \\ E \equiv \rho e \end{bmatrix}, \quad \boldsymbol{F}_i(\boldsymbol{U}) = \underbrace{\begin{pmatrix} \rho u_i \\ \rho u_i u_j + p\delta_{ij} \\ (\rho e + p) u_i \end{pmatrix}}_{\boldsymbol{F}_i^{\text{adv}}} + \underbrace{\begin{pmatrix} 0 \\ -\sigma_{ij} \\ -\rho u_i \sigma_{ij} - kT_{,i} \end{pmatrix}}_{\boldsymbol{F}_i^{\text{diff}}}, \quad S(\boldsymbol{U}) = -\begin{pmatrix} 0 \\ \rho \boldsymbol{g} \\ 0 \end{pmatrix}$$

Compressible Navier-Stokes for FEM

Find $\boldsymbol{U} \in \mathcal{S}^h$

$$\int_\Omega \boldsymbol{v} \cdot \left(\boldsymbol{U}_{,t} - \boldsymbol{S}(\boldsymbol{U})\right) \, \mathrm{d}\Omega - \int_\Omega \boldsymbol{v}_{,i} \cdot \boldsymbol{F}_i(\boldsymbol{U}) \, \mathrm{d}\Omega$$
$$+ \int_{\partial\Omega} \boldsymbol{v} \cdot \boldsymbol{F}_i(\boldsymbol{U}) \cdot \widehat{\boldsymbol{n}} \, \mathrm{d}\partial\Omega$$

Compressible Navier-Stokes for FEM

Find $\boldsymbol{U} \in \mathcal{S}^h$

$$\begin{aligned}\int_\Omega \boldsymbol{v} \cdot \left(\boldsymbol{U}_{,t} - \boldsymbol{S}(\boldsymbol{U})\right) \, \mathrm{d}\Omega - \int_\Omega \boldsymbol{v}_{,i} \cdot \boldsymbol{F}_i(\boldsymbol{U}) \, \mathrm{d}\Omega \\ + \int_{\partial\Omega} \boldsymbol{v} \cdot \boldsymbol{F}_i(\boldsymbol{U}) \cdot \widehat{\boldsymbol{n}} \, \mathrm{d}\partial\Omega \\ \underbrace{+ \int_\Omega \mathcal{P}(\boldsymbol{v})^T \, \left(\boldsymbol{U}_{,t} + \boldsymbol{F}_{i,i}(\boldsymbol{U}) - S(\boldsymbol{U})\right) \, \mathrm{d}\Omega}_{\text{SUPG}} = 0 \, , \; \forall \boldsymbol{v} \in \mathcal{V}^h\end{aligned}$$

Compressible Navier-Stokes for FEM

Find $\boldsymbol{U} \in \mathcal{S}^h$

$$\int_\Omega \boldsymbol{v} \cdot \left(\boldsymbol{U}_{,t} - \boldsymbol{S}(\boldsymbol{U})\right) \, \mathrm{d}\Omega - \int_\Omega \boldsymbol{v}_{,i} \cdot \boldsymbol{F}_i(\boldsymbol{U}) \, \mathrm{d}\Omega$$
$$+ \int_{\partial\Omega} \boldsymbol{v} \cdot \boldsymbol{F}_i(\boldsymbol{U}) \cdot \widehat{\boldsymbol{n}} \, \mathrm{d}\partial\Omega$$
$$\underbrace{+ \int_\Omega \mathcal{P}(\boldsymbol{v})^T \, \left(\boldsymbol{U}_{,t} + \boldsymbol{F}_{i,i}(\boldsymbol{U}) - S(\boldsymbol{U})\right) \, \mathrm{d}\Omega}_{\text{SUPG}} = 0 \,, \; \forall \boldsymbol{v} \in \mathcal{V}^h$$

Further simplified into residual form:

$$\mathcal{G}(\boldsymbol{U}_{,t}, \boldsymbol{U}) = 0$$

Code Architecture

- PETSc used for handling everything libCEED doesn't
  - $\mathcal{P}, \mathcal{P}^T$ (Partition global-to-local operations)
  - Time integration, linear, non-linear equation solving
  - Strong boundary conditions

Code Architecture

- PETSc used for handling everything libCEED doesn't
  - $\mathcal{P}, \mathcal{P}^T$ (Partition global-to-local operations)
  - Time integration, linear, non-linear equation solving
  - Strong boundary conditions
- PETSc calls a libCEED operator when it needs the residual evaluation

Code Architecture

- PETSc used for handling everything libCEED doesn't
  - $\mathcal{P}, \mathcal{P}^T$ (Partition global-to-local operations)
  - Time integration, linear, non-linear equation solving
  - Strong boundary conditions
- PETSc calls a libCEED operator when it needs the residual evaluation
- libCEED Operator based on user-implemented `CeedQFunctions` (D)
  - Use different `CeedQFunction`s for volume vs boundary integrals
  - Combined into a single `CeedOperator` to represent $\mathcal{G}(\boldsymbol{U}_{,t}, \boldsymbol{U})$

Time Step Loop

1. PETSc gets $\boldsymbol{U}^L = \mathcal{P}\boldsymbol{U}^G$ from current solution

Time Step Loop

1. PETSc gets $\boldsymbol{U}^L = \mathcal{P}\boldsymbol{U}^G$ from current solution
2. PETSc calls libCEED to get $\boldsymbol{G}^L = \underbrace{\mathcal{E}^T B^T D B \mathcal{E}}_{L} \boldsymbol{U}^L$

Time Step Loop

1. PETSc gets $\boldsymbol{U}^L = \mathcal{P}\boldsymbol{U}^G$ from current solution
2. PETSc calls libCEED to get $\boldsymbol{G}^L = \underbrace{\mathcal{E}^T B^T D B \mathcal{E}}_{L} \boldsymbol{U}^L$
3. PETSc gets $\boldsymbol{G}^G = \mathcal{P}^T \boldsymbol{G}^L$

Time Step Loop

1. PETSc gets $\boldsymbol{U}^L = \mathcal{P}\boldsymbol{U}^G$ from current solution
2. PETSc calls libCEED to get $\boldsymbol{G}^L = \underbrace{\mathcal{E}^T B^T D B \mathcal{E}}_{L} \boldsymbol{U}^L$
3. PETSc gets $\boldsymbol{G}^G = \mathcal{P}^T \boldsymbol{G}^L$
4. PETSc uses $\boldsymbol{G}^G$ to compute new solution value

Time Step Loop

1. PETSc gets $\boldsymbol{U}^L = \mathcal{P}\boldsymbol{U}^G$ from current solution
2. PETSc calls libCEED to get $\boldsymbol{G}^L = \underbrace{\mathcal{E}^T B^T D B \mathcal{E}}_{L} \boldsymbol{U}^L$
3. PETSc gets $\boldsymbol{G}^G = \mathcal{P}^T \boldsymbol{G}^L$
4. PETSc uses $\boldsymbol{G}^G$ to compute new solution value ...or whatever else it wants

Other Misc Things

Other features not discussed/shown that are in the libCEED-Fluids example

- Primitive variables
- Synthetic turbulence inflow boundary conditions
- Shock capturing

Questions?