Performance-Portable Implicit Scale-Resolving Compressible Flow Using libCEED

SIAM CSE 2023

James Wright, Jed Brown, Kenneth Jansen, Leila Ghaffari

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. libCEED Overview
3. Fluid Simulations with libCEED
4. Accuracy and Performance of High-Order Scale-Resolving Simulations

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
- Portable to different hardware via computational backends

What is libCEED?

- C library for element-based discretizations
  - Bindings for Fortran, Rust, Python, and Julia available
- Designed for matrix-free operator evaluation
- Portable to different hardware via computational backends
  - 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
- Portable to different hardware via computational backends
  - Code that runs on CPU also runs on GPU without changes
  - Computational backend selectable at runtime, using runtime compilation

What is libCEED?

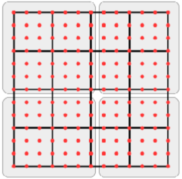
- C library for element-based discretizations
  - Bindings for Fortran, Rust, Python, and Julia available
- Designed for matrix-free operator evaluation
- Portable to different hardware via computational backends
  - Code that runs on CPU also runs on GPU without changes
  - Computational backend selectable at runtime, using runtime compilation
- Geared toward high-order element discretizations

libCEED Overview

Finite Element Operator Decomposition

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

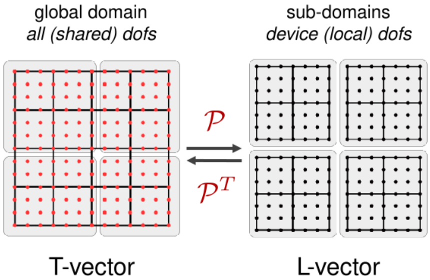
global domain
all (shared) dofs



T-vector

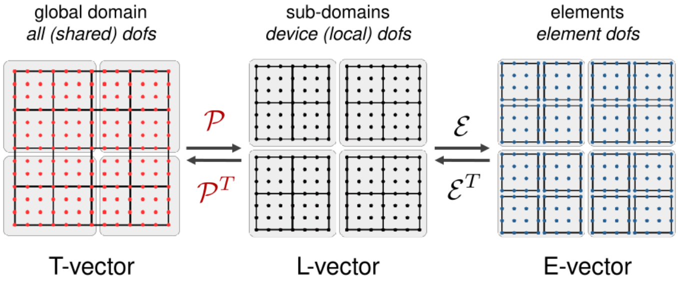
Finite Element Operator Decomposition

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



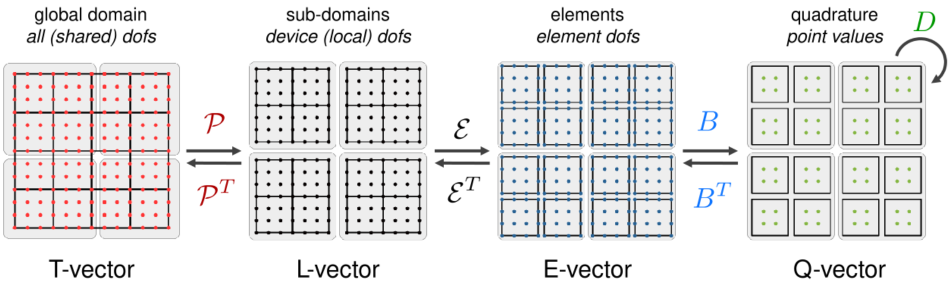
Finite Element Operator Decomposition

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



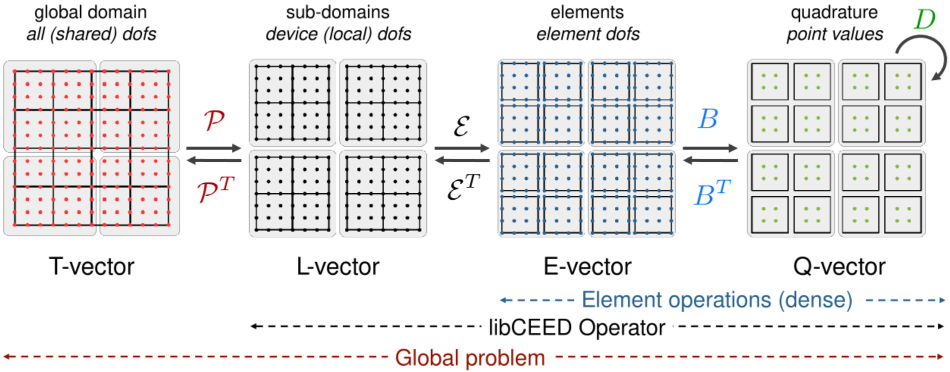
Finite Element Operator Decomposition

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



Finite Element Operator Decomposition

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



Fluid Simulations with libCEED

Compressible Navier-Stokes

$$\boldsymbol{A_0} \boldsymbol{Y}_{,t} + \boldsymbol{F}_{i,i}(\boldsymbol{Y}) - S(\boldsymbol{Y}) = 0$$

for

$$\boldsymbol{A_0} \underbrace{\begin{bmatrix} p \\ u_i \\ T \end{bmatrix}}_{\boldsymbol{Y}} = \begin{bmatrix} \rho \\ \rho u_i \\ \rho e \end{bmatrix}, \quad \boldsymbol{F}_i(\boldsymbol{Y}) = \underbrace{\begin{pmatrix} \rho u_i \\ \rho u_i u_j + p\delta_{ij} \\ (\rho e + p) u_i \end{pmatrix}}_{\boldsymbol{F}_i^{\text{adv}}} + \underbrace{\begin{pmatrix} 0 \\ -\sigma_{ij} \\ -\rho u_j \sigma_{ij} - k T_{,i} \end{pmatrix}}_{\boldsymbol{F}_i^{\text{diff}}}, \quad S(\boldsymbol{Y}) = - \begin{pmatrix} 0 \\ \rho \boldsymbol{g} \\ 0 \end{pmatrix}$$

Compressible Navier-Stokes for FEM

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

$$\int_\Omega \boldsymbol{v} \cdot \left(\boldsymbol{A_0} \boldsymbol{Y}_{,t} - \boldsymbol{S}(\boldsymbol{Y})\right) \, \mathrm{d}\Omega - \int_\Omega \boldsymbol{v}_{,i} \cdot \boldsymbol{F}_i(\boldsymbol{Y}) \, \mathrm{d}\Omega + \int_{\partial\Omega} \boldsymbol{v} \cdot \boldsymbol{F}_i(\boldsymbol{Y}) \cdot \widehat{\boldsymbol{n}}_i \, \mathrm{d}\partial\Omega$$

Compressible Navier-Stokes for FEM

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

$$\begin{aligned}
\int_\Omega \boldsymbol{v} \cdot \left(\boldsymbol{A_0} \boldsymbol{Y}_{,t} - \boldsymbol{S}(\boldsymbol{Y})\right) \, \mathrm{d}\Omega - \int_\Omega \boldsymbol{v}_{,i} \cdot \boldsymbol{F}_i(\boldsymbol{Y}) \, \mathrm{d}\Omega \\
+ \int_{\partial\Omega} \boldsymbol{v} \cdot \boldsymbol{F}_i(\boldsymbol{Y}) \cdot \widehat{\boldsymbol{n}}_i \, \mathrm{d}\partial\Omega \\
\underbrace{+ \int_\Omega P(\boldsymbol{v})^T \, \left(\boldsymbol{A_0} \boldsymbol{Y}_{,t} + \boldsymbol{F}_{i,i}(\boldsymbol{Y}) - S(\boldsymbol{Y})\right) \, \mathrm{d}\Omega}_{\text{SUPG}} = 0 \,, \; \forall \boldsymbol{v} \in \mathcal{V}^h
\end{aligned}$$

Compressible Navier-Stokes for FEM

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

$$\begin{aligned}\int_\Omega \boldsymbol{v} \cdot \left(\boldsymbol{A_0}\boldsymbol{Y}_{,t} - \boldsymbol{S}(\boldsymbol{Y})\right) \, \mathrm{d}\Omega - \int_\Omega \boldsymbol{v}_{,i} \cdot \boldsymbol{F}_i(\boldsymbol{Y}) \, \mathrm{d}\Omega \\ + \int_{\partial\Omega} \boldsymbol{v} \cdot \boldsymbol{F}_i(\boldsymbol{Y}) \cdot \widehat{\boldsymbol{n}}_i \, \mathrm{d}\partial\Omega \\ \underbrace{+ \int_\Omega P(\boldsymbol{v})^T \, \left(\boldsymbol{A_0}\boldsymbol{Y}_{,t} + \boldsymbol{F}_{i,i}(\boldsymbol{Y}) - S(\boldsymbol{Y})\right) \, \mathrm{d}\Omega}_{\text{SUPG}} = 0 \,, \ \forall \boldsymbol{v} \in \mathcal{V}^h\end{aligned}$$

Further simplified into residual form:

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

Compressible Navier-Stokes for FEM

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

$$
\begin{aligned}
\int_\Omega \boldsymbol{v} \cdot \left(\boldsymbol{A_0} \boldsymbol{Y}_{,t} - \boldsymbol{S}(\boldsymbol{Y})\right) \, \mathrm{d}\Omega - \int_\Omega \boldsymbol{v}_{,i} \cdot \boldsymbol{F}_i(\boldsymbol{Y}) \, \mathrm{d}\Omega \\
+ \int_{\partial\Omega} \boldsymbol{v} \cdot \boldsymbol{F}_i(\boldsymbol{Y}) \cdot \widehat{\boldsymbol{n}}_i \, \mathrm{d}\partial\Omega \\
\underbrace{+ \int_\Omega P(\boldsymbol{v})^T \left(\boldsymbol{A_0} \boldsymbol{Y}_{,t} + \boldsymbol{F}_{i,i}(\boldsymbol{Y}) - S(\boldsymbol{Y})\right) \, \mathrm{d}\Omega}_{\mathrm{SUPG}} = 0 \,, \ \forall \boldsymbol{v} \in \mathcal{V}^h
\end{aligned}
$$

Further simplified into residual form:

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

$$
\Rightarrow \quad \left[{\color{red}\mathcal{P}^T} \mathcal{E}^T {\color{blue}B^T} {\color{green}G} {\color{blue}B} \mathcal{E} {\color{red}\mathcal{P}} \right] \begin{bmatrix} \boldsymbol{Y}_{,t} \\ \boldsymbol{Y} \end{bmatrix} = 0
$$

Implicit Timestepping

Implicit timestepping requires solving:

$$\frac{\mathrm{d}\mathcal{G}(\boldsymbol{Y}_{,t}, \boldsymbol{Y})}{\mathrm{d}\boldsymbol{Y}} \Delta \boldsymbol{Y} = \mathcal{G}(\boldsymbol{Y}_{,t}, \boldsymbol{Y})$$

Implicit Timestepping

Implicit timestepping requires solving:

$$\frac{\mathrm{d}\mathcal{G}(\boldsymbol{Y}_{,t},\boldsymbol{Y})}{\mathrm{d}\boldsymbol{Y}}\Delta\boldsymbol{Y} = \mathcal{G}(\boldsymbol{Y}_{,t},\boldsymbol{Y})$$

- Store $\frac{\mathrm{d}\mathcal{G}}{\mathrm{d}\boldsymbol{Y}}$ directly
  - Pros: Opens up preconditioning options
  - Cons: Is large, expensive to store

Implicit Timestepping

Implicit timestepping requires solving:

$$\frac{\mathrm{d}\mathcal{G}(\boldsymbol{Y}_{,t}, \boldsymbol{Y})}{\mathrm{d}\boldsymbol{Y}} \Delta \boldsymbol{Y} = \mathcal{G}(\boldsymbol{Y}_{,t}, \boldsymbol{Y})$$

- Store $\frac{\mathrm{d}\mathcal{G}}{\mathrm{d}\boldsymbol{Y}}$ directly
    - Pros: Opens up preconditioning options
    - Cons: Is large, expensive to store
- Finite difference matrix-free approximation:

$$\frac{\mathrm{d}\mathcal{G}(\boldsymbol{Y}_{,t}, \boldsymbol{Y})}{\mathrm{d}\boldsymbol{Y}} \Delta \boldsymbol{Y} \approx \frac{\mathcal{G}(\boldsymbol{Y}_{,t}, \boldsymbol{Y} + \epsilon \Delta \boldsymbol{Y}) - \mathcal{G}(\boldsymbol{Y}_{,t}, \boldsymbol{Y})}{\epsilon}$$

    - Pros: Just need a residual evaluation, cheap (in programming and computation)
    - Cons: Approximation, accuracy limited to $\sqrt{\epsilon_{\text{machine}}}$

Exact Matrix-Free Jacobian

$$\frac{\mathrm{d}\mathcal{G}}{\mathrm{d}\boldsymbol{Y}}\Delta\boldsymbol{Y} = \frac{\mathrm{d}}{\mathrm{d}\boldsymbol{Y}}\left[\mathcal{G}(\boldsymbol{Y}_{,t}, \boldsymbol{Y})\right]\Delta\boldsymbol{Y}$$

Exact Matrix-Free Jacobian

$$
\begin{aligned}
\frac{\mathrm{d}\mathcal{G}}{\mathrm{d}\boldsymbol{Y}}\Delta\boldsymbol{Y} &= \frac{\mathrm{d}}{\mathrm{d}\boldsymbol{Y}}\left[\mathcal{G}(\boldsymbol{Y}_{,t},\boldsymbol{Y})\right]\Delta\boldsymbol{Y} \\
&= \frac{\mathrm{d}}{\mathrm{d}\boldsymbol{Y}}\left[\mathcal{P}^T\mathcal{E}^T B^T G B \mathcal{E}\mathcal{P}\right]\Delta\boldsymbol{Y}
\end{aligned}
$$

Exact Matrix-Free Jacobian

$$\begin{aligned}
\frac{\mathrm{d}\mathcal{G}}{\mathrm{d}\boldsymbol{Y}}\Delta\boldsymbol{Y} &= \frac{\mathrm{d}}{\mathrm{d}\boldsymbol{Y}}\left[\mathcal{G}(\boldsymbol{Y}_{,t},\boldsymbol{Y})\right]\Delta\boldsymbol{Y} \\
&= \frac{\mathrm{d}}{\mathrm{d}\boldsymbol{Y}}\left[\mathcal{P}^T\mathcal{E}^T B^T G B\mathcal{E}\mathcal{P}\right]\Delta\boldsymbol{Y} \\
&= \left[\mathcal{P}^T\mathcal{E}^T B^T \frac{\mathrm{d}G}{\mathrm{d}\boldsymbol{Y}} B\mathcal{E}\mathcal{P}\right]\Delta\boldsymbol{Y}
\end{aligned}$$

Exact Matrix-Free Jacobian

$$
\begin{aligned}
\frac{\mathrm{d}\mathcal{G}}{\mathrm{d}\boldsymbol{Y}}\Delta\boldsymbol{Y} &= \frac{\mathrm{d}}{\mathrm{d}\boldsymbol{Y}}\left[\mathcal{G}(\boldsymbol{Y}_{,t}, \boldsymbol{Y})\right]\Delta\boldsymbol{Y} \\
&= \frac{\mathrm{d}}{\mathrm{d}\boldsymbol{Y}}\left[\mathcal{P}^T\mathcal{E}^T B^T G B \mathcal{E}\mathcal{P}\right]\Delta\boldsymbol{Y} \\
&= \left[\mathcal{P}^T\mathcal{E}^T B^T \frac{\mathrm{d}G}{\mathrm{d}\boldsymbol{Y}} B \mathcal{E}\mathcal{P}\right]\Delta\boldsymbol{Y}
\end{aligned}
$$

- Store intermediary data at quadrature points to improve efficiency (“taping”)
  - We store $\boldsymbol{Y}$, viscous stress, and stabilization perturbation ($P(\boldsymbol{v})$)
- Pros: Exact Jacobian matrix-vector product (potentially faster convergence)
- Cons: More expensive than residual evaluation (but not by too much)

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 $\mathcal{G}(\boldsymbol{Y}_{,t}, \boldsymbol{Y})$

Time Step Loop

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

Time Step Loop

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

Time Step Loop

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

Time Step Loop

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

Accuracy and Performance of High-Order Scale-Resolving Simulations

Flat Plate Boundary Layer, Zero Pressure Gradient

Problem Description:

- $Re_\theta \approx 970$ boundary layer at inflow, $M \approx 0.1$
- Synthetic turbulence generation (STG) used for inflow structures
- Internal damping layer (IDL) used in STG development region to prevent pressure wave growth
- asdf
- Domain size of $\{27 \times 24 \times 4\}\delta_0$

High-Order Approach

- Test 3 different order elements, Q_1, Q_2, Q_3 tensor-product hexes
- Maintain *DOF resolution* (DOFs per physical length)
- DOF resolution for streamwise and spanwise was $\Delta x^+ = 30$ and $\Delta z^+ = 12$
    - For Q_1, this is about half the resolution required for DNS resolution

Support and References

This work was supported by.... Add in sponsor support (DOE, ECP, etc)