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. libCEED Overview
2. Compressible Fluid Equations in libCEED
3. Efficient Implicit Timestepping
4. Performance and Results of Flat Plate Boundary Layer Simulation

libCEED Overview

What is libCEED?

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

What is libCEED?

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

What is libCEED?

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

What is libCEED?

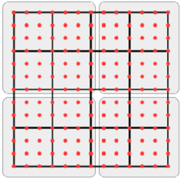
- C library for element-based discretizations
    - Bindings available for Fortran, Rust, Python, and Julia
- Designed for matrix-free operator evaluation
- Portable to different hardware via computational backends
    - Code that runs on CPU also runs on GPU without changes
    - Computational backend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

Finite Element Operator Decomposition

Finite Element Operator Decomposition

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

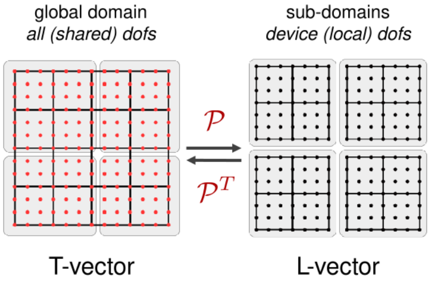
global domain
all (shared) dofs



T-vector

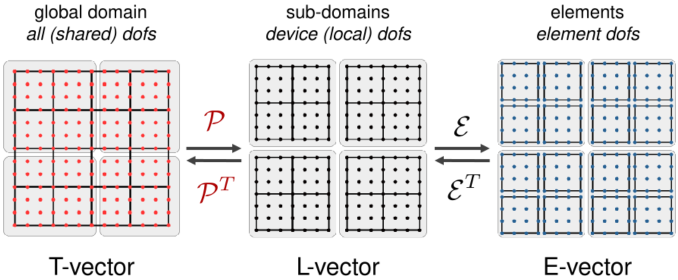
Finite Element Operator Decomposition

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



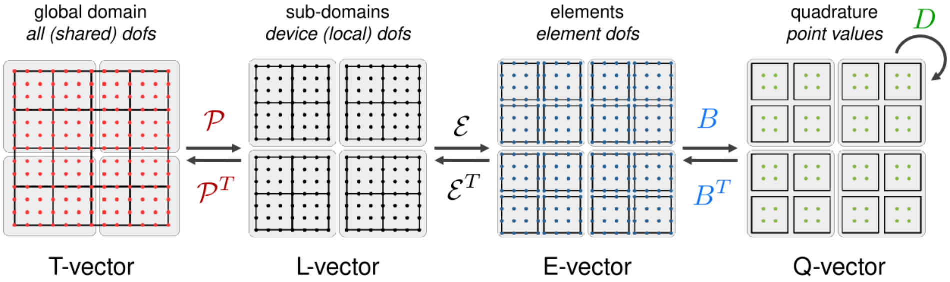
Finite Element Operator Decomposition

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



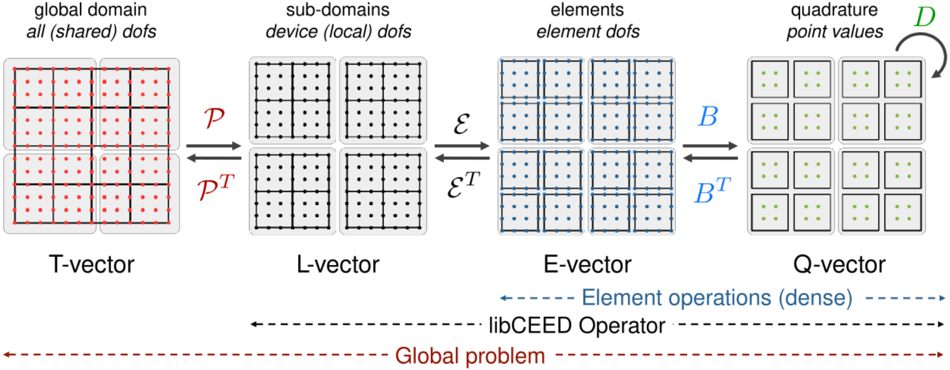
Finite Element Operator Decomposition

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



Finite Element Operator Decomposition

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



Compressible Fluid Equations in libCEED

Compressible Navier-Stokes

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

for

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

Compressible Navier-Stokes for Continuous-Galerkin FEM

Find $\boldsymbol{Y} \in \mathcal{S}^h$, $\forall \boldsymbol{v} \in \mathcal{V}^h$

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

Compressible Navier-Stokes for Continuous-Galerkin FEM

Find $\boldsymbol{Y} \in \mathcal{S}^h$, $\forall \boldsymbol{v} \in \mathcal{V}^h$

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

Compressible Navier-Stokes for Continuous-Galerkin FEM

Find $\boldsymbol{Y} \in \mathcal{S}^h$, $\forall \boldsymbol{v} \in \mathcal{V}^h$

$$\int_\Omega \boldsymbol{v} \cdot \left[\boldsymbol{A_0} \boldsymbol{Y}_{,t} - \boldsymbol{S}(\boldsymbol{Y})\right] \,\mathrm{d}\Omega - \int_\Omega \boldsymbol{v}_{,i} \cdot \boldsymbol{F}_i(\boldsymbol{Y}) \,\mathrm{d}\Omega + \int_{\partial\Omega} \boldsymbol{v} \cdot \boldsymbol{F}_i(\boldsymbol{Y}) \cdot \widehat{\boldsymbol{n}}_i \,\mathrm{d}\partial\Omega$$
$$\underbrace{+ \int_\Omega \mathscr{L}^{\mathrm{adv}}(\boldsymbol{v}) \boldsymbol{\tau} \left[\boldsymbol{A_0} \boldsymbol{Y}_{,t} + \boldsymbol{F}_{i,i}(\boldsymbol{Y}) - S(\boldsymbol{Y})\right] \,\mathrm{d}\Omega}_{\text{SUPG}} = 0$$

Compressible Navier-Stokes for Continuous-Galerkin FEM

Find $\boldsymbol{Y} \in \mathcal{S}^h$, $\forall \boldsymbol{v} \in \mathcal{V}^h$

$$\int_\Omega \boldsymbol{v} \cdot \left[\boldsymbol{A_0} \boldsymbol{Y}_{,t} - \boldsymbol{S}(\boldsymbol{Y})\right] \, \mathrm{d}\Omega - \int_\Omega \boldsymbol{v}_{,i} \cdot \boldsymbol{F}_i(\boldsymbol{Y}) \, \mathrm{d}\Omega + \int_{\partial\Omega} \boldsymbol{v} \cdot \boldsymbol{F}_i(\boldsymbol{Y}) \cdot \widehat{\boldsymbol{n}}_i \, \mathrm{d}\partial\Omega \\ \underbrace{+ \int_\Omega \mathscr{L}^{\mathrm{adv}}(\boldsymbol{v}) \boldsymbol{\tau} \left[\boldsymbol{A_0} \boldsymbol{Y}_{,t} + \boldsymbol{F}_{i,i}(\boldsymbol{Y}) - S(\boldsymbol{Y})\right] \, \mathrm{d}\Omega}_{\mathrm{SUPG}} = 0$$

Simplify into residual form:

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

Compressible Navier-Stokes for Continuous-Galerkin FEM

Find $\boldsymbol{Y} \in \mathcal{S}^h$, $\forall \boldsymbol{v} \in \mathcal{V}^h$

$$\int_\Omega \boldsymbol{v} \cdot \left[\boldsymbol{A_0}\boldsymbol{Y}_{,t} - \boldsymbol{S}(\boldsymbol{Y})\right] \,\mathrm{d}\Omega - \int_\Omega \boldsymbol{v}_{,i} \cdot \boldsymbol{F}_i(\boldsymbol{Y}) \,\mathrm{d}\Omega + \int_{\partial\Omega} \boldsymbol{v} \cdot \boldsymbol{F}_i(\boldsymbol{Y}) \cdot \widehat{\boldsymbol{n}}_i \,\mathrm{d}\partial\Omega$$
$$\underbrace{+ \int_\Omega \mathscr{L}^{\mathrm{adv}}(\boldsymbol{v})\boldsymbol{\tau} \left[\boldsymbol{A_0}\boldsymbol{Y}_{,t} + \boldsymbol{F}_{i,i}(\boldsymbol{Y}) - S(\boldsymbol{Y})\right] \,\mathrm{d}\Omega}_{\text{SUPG}} = 0$$

Simplify into residual form:

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

$$\Rightarrow \quad \mathcal{P}^T \mathcal{E}^T B^T G B \mathcal{E} \mathcal{P} \begin{bmatrix} \boldsymbol{Y}_{,t} \\ \boldsymbol{Y} \end{bmatrix} = 0$$

Efficient Implicit Timestepping

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:

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

- System too large for direct solve $\longrightarrow$ iterative solve

Implicit Timestepping

Implicit timestepping requires solving:

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

- System too large for direct solve $\longrightarrow$ iterative solve
- Krylov iterative solvers used most commonly

Implicit Timestepping

Implicit timestepping requires solving:

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

- System too large for direct solve $\longrightarrow$ iterative solve
- Krylov iterative solvers used most commonly
- Krylov solvers form solution basis from $\mathrm{span}\left\{ \left[\frac{\mathrm{d}\mathcal{G}(\boldsymbol{Y}_{,t}, \boldsymbol{Y})}{\mathrm{d}\boldsymbol{Y}} \right]^n \Delta \boldsymbol{Y} \right\}_{n=0}$

Implicit Timestepping

Implicit timestepping requires solving:

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

- System too large for direct solve $\longrightarrow$ iterative solve
- Krylov iterative solvers used most commonly
- Krylov solvers form solution basis from $\operatorname{span}\left\{ \left[\frac{\mathrm{d}\mathcal{G}(\boldsymbol{Y}_{,t}, \boldsymbol{Y})}{\mathrm{d}\boldsymbol{Y}} \right]^n \Delta \boldsymbol{Y} \right\}_{n=0}$

Bottom Line

Cost of $\frac{\mathrm{d}\mathcal{G}(\boldsymbol{Y}_{,t}, \boldsymbol{Y})}{\mathrm{d}\boldsymbol{Y}} \Delta \boldsymbol{Y}$ dominates implicit timestepping cost

Jacobian Matrix-Vector Multiply Options

How to compute $\dfrac{\mathrm{d}\mathcal{G}(\boldsymbol{Y}_{,t}, \boldsymbol{Y})}{\mathrm{d}\boldsymbol{Y}} \Delta \boldsymbol{Y}$?

Jacobian Matrix-Vector Multiply Options

How to compute $\dfrac{\mathrm{d}\mathcal{G}(\boldsymbol{Y}_{,t}, \boldsymbol{Y})}{\mathrm{d}\boldsymbol{Y}} \Delta \boldsymbol{Y}$?

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

Jacobian Matrix-Vector Multiply Options

How to compute $\dfrac{\mathrm{d}\mathcal{G}(\boldsymbol{Y}_{,t}, \boldsymbol{Y})}{\mathrm{d}\boldsymbol{Y}} \Delta \boldsymbol{Y}$?

- Store $\frac{\mathrm{d}\mathcal{G}}{\mathrm{d}\boldsymbol{Y}}$ directly (sparse matrix representation)
  - **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:** Accuracy limited to $\sqrt{\epsilon_{\text{machine}}}$, preconditiong require partial assembly

Exact Matrix-Free Jacobian via CeedOperator

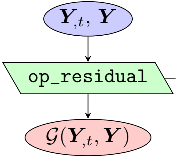
$$\frac{\mathrm{d}\mathcal{G}}{\mathrm{d}\boldsymbol{Y}}\Delta\boldsymbol{Y} = \frac{\mathrm{d}}{\mathrm{d}\boldsymbol{Y}} \overbrace{\left[\mathcal{P}^T \mathcal{E}^T B^T G B \mathcal{E} \mathcal{P}\right]}^{\mathcal{G}(\boldsymbol{Y}_{,t},\boldsymbol{Y})} \Delta\boldsymbol{Y}$$

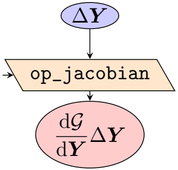
$$= \left[\mathcal{P}^T \mathcal{E}^T B^T \frac{\mathrm{d}G}{\mathrm{d}\boldsymbol{Y}} B \mathcal{E} \mathcal{P}\right] \Delta\boldsymbol{Y}$$

Exact Matrix-Free Jacobian via CeedOperator

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

$$= \left[\mathcal{P}^T\mathcal{E}^T B^T \frac{\mathrm{d}G}{\mathrm{d}\boldsymbol{Y}} B\mathcal{E}\mathcal{P}\right]\Delta\boldsymbol{Y}$$

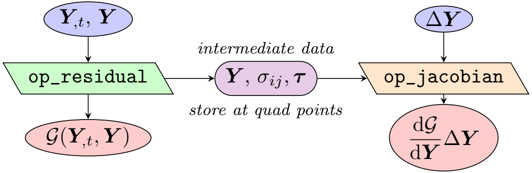




Exact Matrix-Free Jacobian via CeedOperator

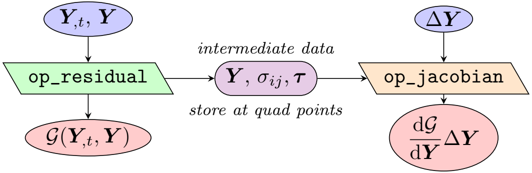
$$\frac{\mathrm{d}\mathcal{G}}{\mathrm{d}\boldsymbol{Y}}\Delta\boldsymbol{Y} = \frac{\mathrm{d}}{\mathrm{d}\boldsymbol{Y}} \overbrace{\left[\mathcal{P}^T\mathcal{E}^T B^T G B \mathcal{E}\mathcal{P}\right]}^{\mathcal{G}(\boldsymbol{Y}_{,t},\boldsymbol{Y})} \Delta\boldsymbol{Y}$$

$$= \left[\mathcal{P}^T\mathcal{E}^T B^T \frac{\mathrm{d}G}{\mathrm{d}\boldsymbol{Y}} B \mathcal{E}\mathcal{P}\right]\Delta\boldsymbol{Y}$$



Exact Matrix-Free Jacobian via CeedOperator

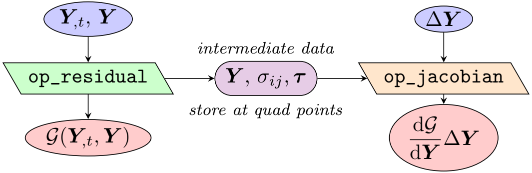
$$
\begin{aligned}
\frac{\mathrm{d}\mathcal{G}}{\mathrm{d}\boldsymbol{Y}}\Delta\boldsymbol{Y} &= \frac{\mathrm{d}}{\mathrm{d}\boldsymbol{Y}}\overbrace{\left[\mathcal{P}^T\mathcal{E}^T B^T G B\mathcal{E}\mathcal{P}\right]}^{\mathcal{G}(\boldsymbol{Y}_{,t},\boldsymbol{Y})}\Delta\boldsymbol{Y} \\
&= \left[\mathcal{P}^T\mathcal{E}^T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



- **Pros:** Exact Jacobian matrix-vector product[1] (potentially faster convergence)
- **Cons:** Preconditioning requires partial assembly, requires coding Jacobian (automatic differentiation helps though)

Exact Matrix-Free Jacobian via CeedOperator

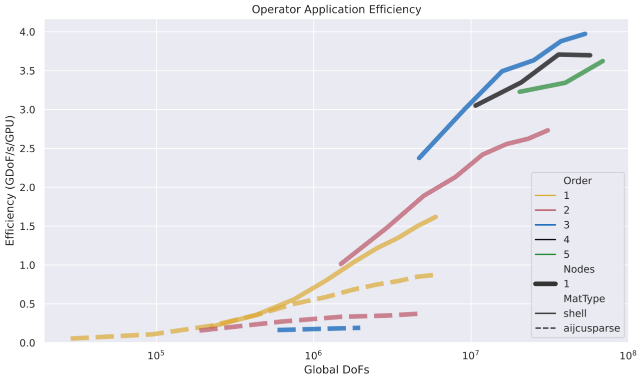
$$\frac{\mathrm{d}\mathcal{G}}{\mathrm{d}\boldsymbol{Y}}\Delta\boldsymbol{Y} = \frac{\mathrm{d}}{\mathrm{d}\boldsymbol{Y}}\overbrace{\left[\mathcal{P}^T\mathcal{E}^T B^T G B \mathcal{E}\mathcal{P}\right]}^{\mathcal{G}(\boldsymbol{Y}_{,t},\boldsymbol{Y})}\Delta\boldsymbol{Y}$$
$$= \left[\mathcal{P}^T\mathcal{E}^T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



- **Pros:** Exact Jacobian matrix-vector product[1] (potentially faster convergence)
- **Cons:** Preconditioning requires partial assembly, requires coding Jacobian (automatic differentiation helps though)

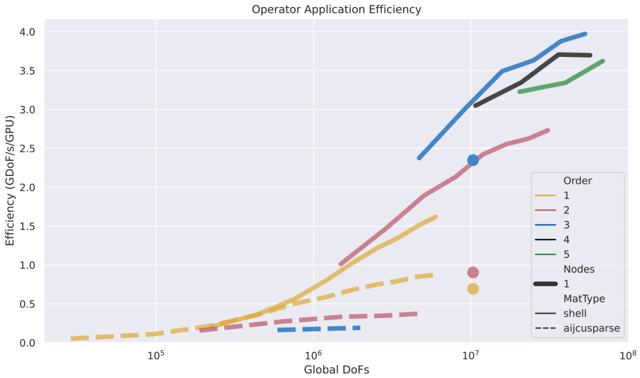
[1] Affect of specific terms may be ignored from the Jacobian. This is done for $\mathrm{d}\boldsymbol{\tau}/\mathrm{d}\boldsymbol{Y}$

Performance Exact Matrix-Free Jacobian via CeedOperator



Matrix-Free Application Comparison (Reproduced from Brown *et al.* 2022). Not from fluids code, but representative of libCEED matrix-free

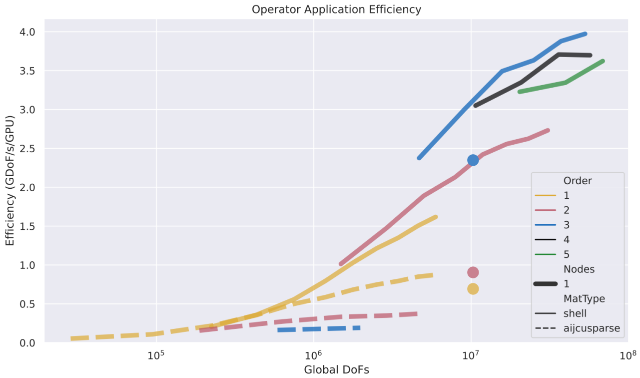
Performance Exact Matrix-Free Jacobian via CeedOperator



Matrix-Free Application Comparison (Reproduced from Brown *et al.* 2022). Not from fluids code, but representative of libCEED matrix-free

- Significant differences with performance implications
  - Fluids run over 2 nodes ⟶ network latency present
  - Solids code lower in FLOPs and storage per DoF

Performance Exact Matrix-Free Jacobian via CeedOperator

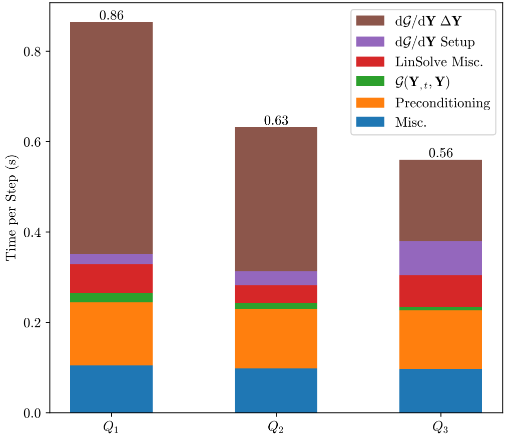


Matrix-Free Application Comparison (Reproduced from Brown *et al.* 2022). Not from fluids code, but representative of libCEED matrix-free

- Significant differences with performance implications
  - Fluids run over 2 nodes $\longrightarrow$ network latency present
  - Solids code lower in FLOPs and storage per DoF
- Significant increase in efficiency for $Q_1, Q_2 \rightarrow Q_3$

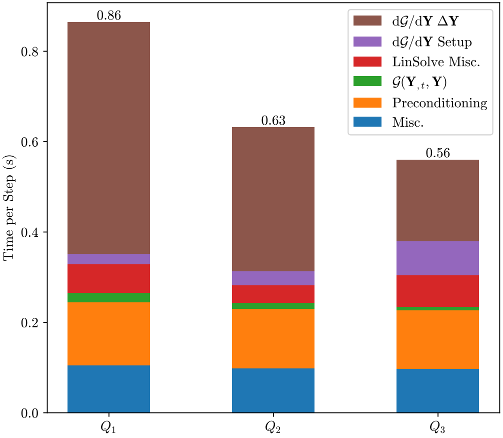
Performance and Results of Flat Plate Boundary Layer Simulation

Fluids Performance Analysis



Run on ALCF's Polaris on two nodes (8× NVIDIA A100, global DoF count fixed

Fluids Performance Analysis

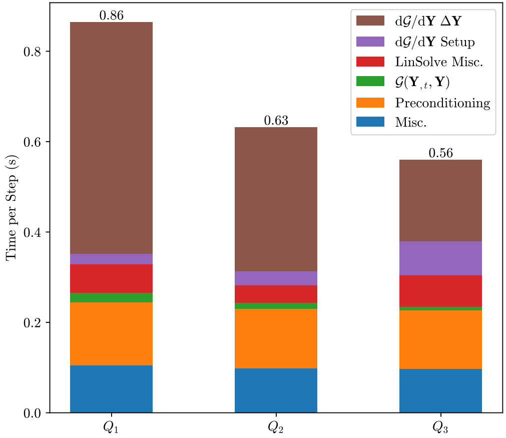


Run on ALCF's Polaris on two nodes (8× NVIDIA A100, global DoF count fixed

- Time of $\mathrm{d}\mathcal{G}/\mathrm{d}\boldsymbol{Y}\,\Delta\boldsymbol{Y}$ decreases as order increases
  - Despite increase in $\mathrm{d}\mathcal{G}/\mathrm{d}\boldsymbol{Y}\,\Delta\boldsymbol{Y}$ calculations per timestep:

Q_1	Q_2	Q_3
250	221	360

Fluids Performance Analysis



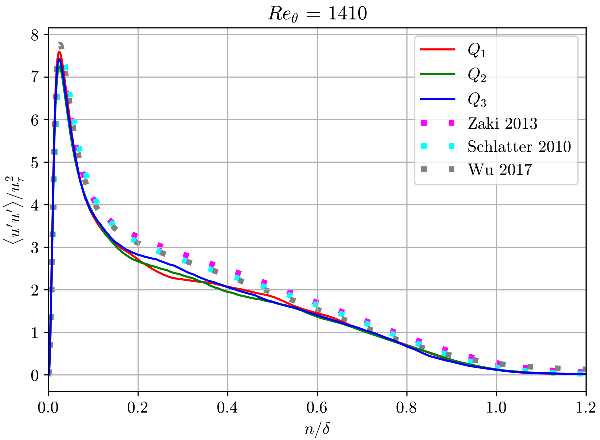
Run on ALCF's Polaris on two nodes (8× NVIDIA A100, global DoF count fixed

- Time of $\mathrm{d}\mathcal{G}/\mathrm{d}\boldsymbol{Y}\Delta\boldsymbol{Y}$ decreases as order increases
  - Despite increase in $\mathrm{d}\mathcal{G}/\mathrm{d}\boldsymbol{Y}\Delta\boldsymbol{Y}$ calculations per timestep:

Q_1	Q_2	Q_3
250	221	360

- $\mathrm{d}\mathcal{G}/\mathrm{d}\boldsymbol{Y}$ setup time increases with order
  - Dominant cost is partial matrix assembly for preconditioning
  - For Q_3, eats up gains from faster $\mathrm{d}\mathcal{G}/\mathrm{d}\boldsymbol{Y}\Delta\boldsymbol{Y}$ calculations

Results of Flat Plate Boundary Layer



- Spanwise statistics implemented to verify scale-resolving results
- Results *not* converged, but show realistic stress profiles

Support and References

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