Automatic Generation of Matrix-Free Routines for PDE Solvers with Devito via PETSc

Z. Leibowitz[1], R. Nelson[1], F. Luporini[2], M. Louboutin[2], M. Knepley[3], L. Mitchell, J. Betteridge[2], E. Caunt[2], G. Bisbas[1], M. Piggott[1], G. Gorman[1,2]

[1] Imperial College London
[2] Devito Codes
[3] University at Buffalo

PETSc User Meeting, May 20-21, 2025

Talk Outline

- Why do people care about DSLs?
- What is Devito?
- What are some things Devito is currently lacking?
- What are my new contributions to the software? What are my updates since last year?
- Results
- Next steps

Traditional approach to solving PDEs

$$m\frac{\partial^2 u}{\partial t^2} + \eta\frac{\partial u}{\partial t} - \Delta u = 0$$

```
void kernel(...) {
  ...
  <impenetrable code with aggressive performance optimizations
written by rockstars, gurus, ninjas, unicorns and celestial
beings>    ...
}
```

DEVITO

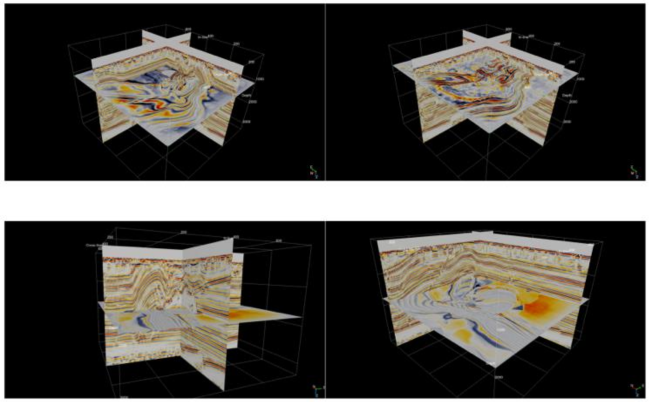
IMPERIAL

Domain Specific Languages (DSLs) Manifesto

- Decoupling mathematics from HPC
- Productivity, portability, performance
- Write once, run everywhere
- Do in O(days) what would normally take O(months)

Introduction to Devito

- A DSL and compiler to express explicit finite difference operators.
- Embedded in Python – easy to learn, symbolic API.
- Generates optimised C code for multiple architectures:
  - CPUs
    - OpenMP + MPI
  - GPUs
    - OpenMP/OpenACC + MPI
    - CUDA/HIP/SYCL (proprietary)
- Many users in both academia and industry.
- Open source platform – MIT license.



Louboutin et al.: Synthetic 3D anisotropic subsurface model simulating a realistic industry-scale TTI problem

The Devito Vision

$$m\frac{\partial^2 u}{\partial t^2}(\boldsymbol{x},t) + \eta\frac{\partial u}{\partial t}(\boldsymbol{x},t) - \nabla^2 u(\boldsymbol{x},t) = 0$$

User writes …

```
eqn = m * u.dt2 + eta * u.dt – u.laplace
```

Devito creates …

```
void kernel (...) ...
```

Example – simple 2D diffusion operator

Example – simple 2D diffusion operator

Step 1 - Import Devito:

```
from devito import Grid, TimeFunction, Eq, solve, Operator
```

Example – simple 2D diffusion operator

Step 1 - Import Devito:

```
from devito import Grid, TimeFunction, Eq, solve, Operator
```

Step 2 - Create computational domain:

```
grid = Grid(shape=(11, 11), extent=(1., 1.))
```

Example – simple 2D diffusion operator

Step 1 - Import Devito:

```
from devito import Grid, TimeFunction, Eq, solve, Operator
```

Step 2 - Create computational domain:

```
grid = Grid(shape=(11, 11), extent=(1., 1.))
```

Step 3 - Define the functions present in the PDE:

```
f = TimeFunction(name='f', grid=grid, space_order=so, ...)
```

Example – simple 2D diffusion operator

Step 1 - Import Devito:

```
from devito import Grid, TimeFunction, Eq, solve, Operator
```

Step 2 - Create computational domain:

```
grid = Grid(shape=(11, 11), extent=(1., 1.))
```

Step 3 - Define the functions present in the PDE:

```
f = TimeFunction(name='f', grid=grid, space_order=so, …)
```

Step 4 - Define the PDE, explicit FD scheme and boundary conditions:

```
eqn = Eq(f.dt, 0.5 * f.laplace)
update = Eq(f.forward, solve(eqn, f.forward))
bcs = … # BCs expressed in symbolic form
```

Example – simple 2D diffusion operator

Step 5 - Generate C kernel through a sequence of compilation passes:

```
op = Operator(update, bcs, …)
```

This will generate OpenMP + MPI code!

Example – simple 2D diffusion operator

Step 5 - Generate C kernel through a sequence of compilation passes:

```
op = Operator(update, bcs, …)
```

This will generate OpenMP + MPI code!

Step 6 - Just-in-time (JIT) compile and execute:

```
op(t=timesteps, dt=dt)
```

Example – simple 2D diffusion operator

Step 5 - Generate C kernel through a sequence of compilation passes:

```
op = Operator(update, bcs, …)
```

This will generate OpenMP + MPI code!

Step 6 - Just-in-time (JIT) compile and execute:

```
op(t=timesteps, dt=dt)
```

Low level C loop structure automatically generated:

```
for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M; time += 1, t0 = (time)%(2), t1 = (time + 1)%(2))
{
  for (int x = x_m; x <= x_M; x += 1)
  {
    for (int y = y_m; y <= y_M; y += 1)
    {
      f[t1][x + 2][y + 2] = dt*(r2*f[t0][x + 2][y + 2] + (-1.0F)*(r0*f[t0][x + 2][y + 2] + r1*f[t0][x + 2][y + 2]) + ... );
    }
  }
}
```

DEVITO

IMPERIAL

Flexibility in terms of the numerical scheme

f = TimeFunction(name=‘f’, grid=grid, space_order=so, …)

so = 2

```
for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); ...)
{
  for (int x = x_m; x <= x_M; x += 1)
  {
    for (int y = y_m; y <= y_M; y += 1)
    {
      f[t1][x + 2][y + 2] = dt*(r2*f[t0][x + 2][y + 2] +
      (-1.0F)*(r0*f[t0][x + 2][y + 2] + r1*f[t0][x + 2][y + 2]) +
      5.0e-1F*(r0*f[t0][x + 1][y + 2] + r0*f[t0][x + 3][y + 2] +
      r1*f[t0][x + 2][y + 1] + r1*f[t0][x + 2][y + 3]));
    }
  }
}
```

so = 12

```
for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); ...)
{
  for (int x = x_m; x <= x_M; x += 1)
  {
    for (int y = y_m; y <= y_M; y += 1)
    {
      float r3 = -2.982777780F*f[t0][x + 12][y + 12];
      f[t1][x + 12][y + 12] = dt*(r2*f[t0][x + 12][y + 12] +
      5.0e-1F*(r0*(r3 + (-6.01250601e-5F)*(f[t0][x + 6][y + 12] +
      f[t0][x + 18][y + 12]) + 1.03896104e-3F*(f[t0][x + 7][y + 12] +
      f[t0][x + 17][y + 12]) + (-8.92857143e-3F)*(f[t0][x + 8][y + 12] +
      f[t0][x + 16][y + 12]) + 5.29100529e-2F*(f[t0][x + 9][y + 12] +
      f[t0][x + 15][y + 12]) + (-2.67857143e-1F)*(f[t0][x + 10][y + 12] +
      f[t0][x + 14][y + 12]) + 1.714285710F*(f[t0][x + 11][y + 12] +
      f[t0][x + 13][y + 12])) + r1*(r3 + (-6.01250601e-5F)*(f[t0][x + 12][y + 6] +
      f[t0][x + 12][y + 18]) + 1.03896104e-3F*(f[t0][x + 12][y + 7] +
      f[t0][x + 12][y + 17]) + (-8.92857143e-3F)*(f[t0][x + 12][y + 8] +
      f[t0][x + 12][y + 16]) + 5.29100529e-2F*(f[t0][x + 12][y + 9] +
      f[t0][x + 12][y + 15]) + (-2.67857143e-1F)*(f[t0][x + 12][y + 10] +
      f[t0][x + 12][y + 14]) + 1.714285710F*(f[t0][x + 12][y + 11] +
      f[t0][x + 12][y + 13]))));
    }
  }
}
```

Limitations to certain application domains

Nicholas Doherty from unsplash

My work: Automatically generate matrix-free routines in Devito to interact with PETSc's scalable solvers.

- Devito compiler pass

Main update since PETSc Meeting 2024

- Automation!!!

PETSc lowering (*PETScSolve*)

- Generate sets of equations required by each callback (matrix, residual, preconditioning...)
- Time dependence handling
- Boundary condition treatment
- Process solver parameters

→

Equations lowering

Input equations -> Lowered Equations

→

Clustering

Lowered Equations -> Clusters

- Pre-process clusters for PETSc

→

Symbolic optimisation

Clusters -> Clusters

↓

Schedule tree lowering

Clusters -> ScheduleTree

→

Iteration/Expression tree lowering

Schedule Tree -> IET

- Build various IET routines to generate PETSc callbacks (problem-specific)
- Build solver objects e.g DMDA, Mat, Vecs
- Gather data for callback functions by populating the required structs
- Create PetscCalls to run the solvers
- Time dependence handling

→

Synthesis

IET -> CGen AST-> C/C++ string

- PETSc declarations, headers

→

JIT Compilation

C/C++ string -> kernel.c -> kernel.so

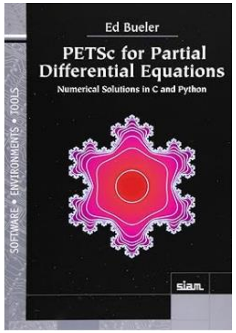
Example: The biharmonic equation as a coupled system

$$\nabla^4 u = f$$

$$u = 0, \quad \nabla^2 u = 0 \quad \text{on } \partial\Omega$$

Substitute $v = -\nabla^2 u$, then system has triangular block form:

$$\begin{aligned} -\nabla^2 v &= f \\ -\nabla^2 u &= v \end{aligned} \quad \Leftrightarrow \quad \begin{bmatrix} -\nabla^2 & 0 \\ -I & -\nabla^2 \end{bmatrix} \begin{bmatrix} v \\ u \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix}$$



Thank you Ed Bueler!!

We manufacture an exact solution $u(x, y) = c(x)c(y)$, where $c(x)$ is a 6th-degree polynomial that satisfies the boundary conditions.

IMPERIAL

DEVITO

User level code: The biharmonic equation

```
# Build computational domain
grid = Grid(shape=(n,n), extent=(1.,1.))

# Define functions present in PDEs
u = Function(name='u', grid=grid, space_order=2)
v = Function(name='v', grid=grid, space_order=2)
f = Function(name='f', grid=grid, space_order=2)

# Set up RHS
f.data[:] = ...

# Symbolically express the equations
eq_v = Eq(-v.laplace, f, subdomain=grid.interior)
eq_u = Eq(-u.laplace, v, subdomain=grid.interior)

# Employ new API object to trigger the lowering to PETSc
solver = PETScSolve({v: [eq_v], u: [eq_u]},
                   solver_parameters={'ksp_type':'gmres',
                                      'ksp_rtol': 1e-7,
                                      'ksp_atol': 1e-30, ...})

# Automatically generate the low level C kernel
op = Operator(solver, language='petsc')

# JIT compile and execute the code
op.apply()
```

DEVITO

IMPERIAL

Automatically generated PETSc code: The biharmonic equation

```
...
PetscCall(DMDACreate2d(...,2,...,&da));
...
PetscCall(DMSetMatType(da,MATSHELL));
PetscCall(DMCreateMatrix(da,&(J)));
PetscCall(SNESSetJacobian(snes,J,J,MatMFFDComputeJacobian,NULL));
PetscCall(SNESSetType(snes,SNESKSPONLY));
PetscCall(KSPSetType(ksp,KSPGMRES));
PetscCall(MatShellSetOperation(J,MATOP_MULT,...WholeMatMult0));
PetscCall(SNESSetFunction(snes,NULL,WholeFormFunc0,(void*)(da)));
PetscCall(DMSetApplicationContext(da,&(ctx)));
...
PetscCall(DMCreateFieldDecomposition(da,...,&fields,&subdms));
PetscCall(MatShellSetOperation(J,...MatCreateSubMatrices0));
...
PetscCall(SNESSolve(snes,NULL,xglobal));
...
```

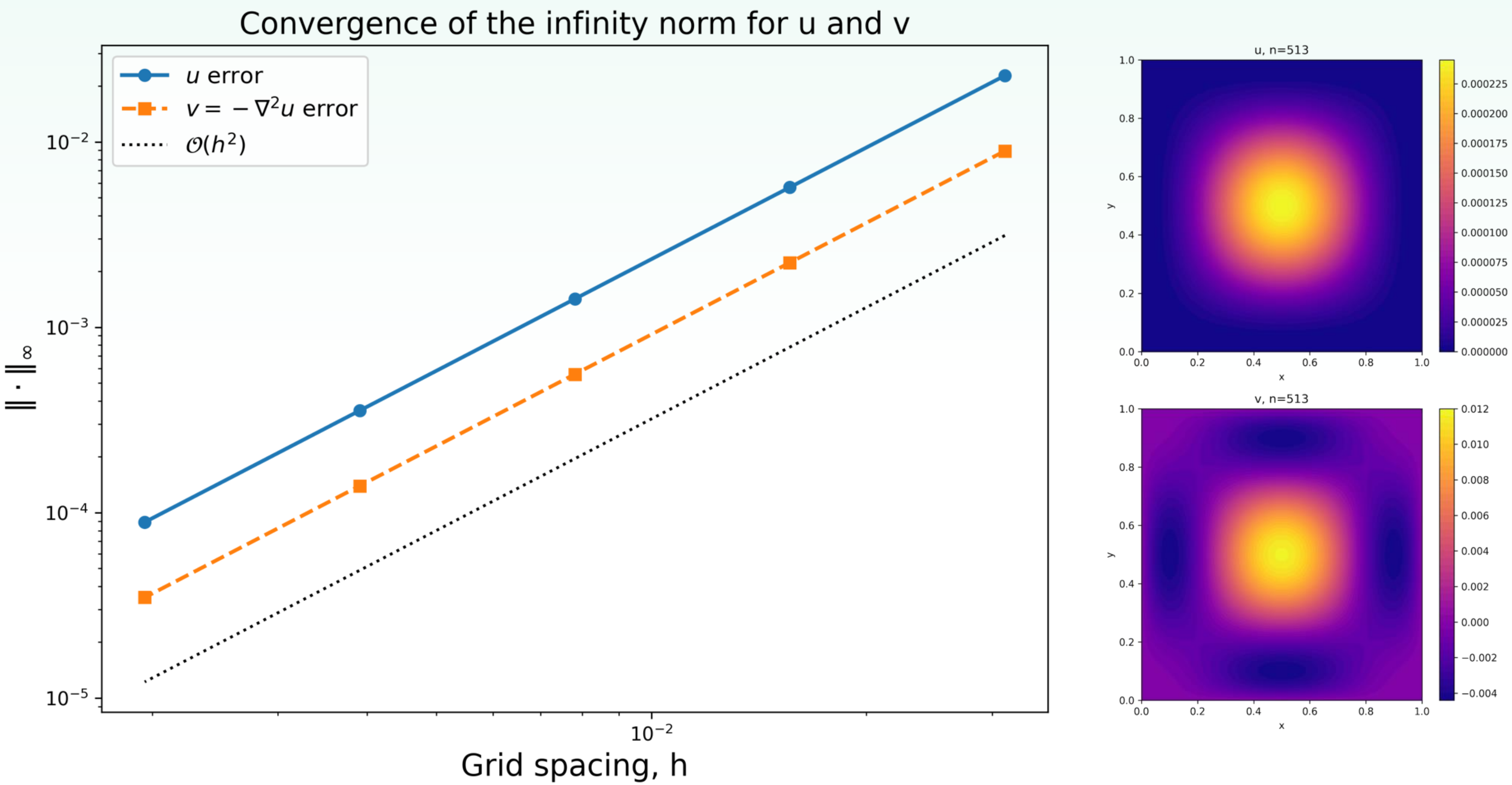
```
PetscErrorCode WholeFormFunc0(SNES snes, Vec X, Vec F, void* ptr)
{
  ...
  for (int ix = ctx0->x_m + ctx0->x_ltkn0; ...)
  {
    for (int iy = ctx0->y_m + ctx0->y_ltkn0; ...)
    {
      f_vu[ix + 2][iy + 2].v = 2.0*(r4*x_vu[ix + 2][iy + 2].v + ... ;
      f_vu[ix + 2][iy + 2].u = 2.0*(r4*x_vu[ix + 2][iy + 2].u + ... ;
    }
  }
  ...
  PetscCall(DMLocalToGlobalBegin(dm,floc,ADD_VALUES,F));
  PetscCall(DMLocalToGlobalEnd(dm,floc,ADD_VALUES,F));
  ...

  PetscFunctionReturn(0);
}
```

```
PetscErrorCode MatCreateSubMatrices0(Mat J, PetscInt nfields, IS *irow, ...)
{
  ...
  PetscCall(PetscCalloc1(nsubmats,submats));
  for (int i = 0; i <= nsubmats - 1; i += 1)
  {
    PetscCall(MatCreate(PETSC_COMM_WORLD,&(block)));
    PetscCall(MatSetSizes(block,...));
    PetscCall(MatSetType(block,MATSHELL));
    ...
    subctx->rows = &(irow[rowidx]);
    subctx->cols = &(icol[colidx]);
    PetscCall(MatSetUp(block));
    submat_arr[i] = block;
  }
  PetscCall(MatShellSetOperation(submat_arr[0],MATOP_MULT,...J00_MatMult0));
  PetscCall(MatShellSetOperation(submat_arr[2],MATOP_MULT,...J10_MatMult0));
  PetscCall(MatShellSetOperation(submat_arr[3],MATOP_MULT,...J11_MatMult0));
  PetscFunctionReturn(0);
}
```

```
PetscErrorCode J00_MatMult0(Mat J, Vec X, Vec Y)
{
  ...
  for (int ix = ctx0->x_m + ctx0->x_ltkn0; ...)
  {
    for (int iy = ctx0->y_m + ctx0->y_ltkn0; ...)
    {
      y_v[ix + 2][iy + 2] = 2.0*(r0*x_v[ix + 2][iy + 2] + r1*... ;
    }
  }
  ...
  PetscCall(VecRestoreArray(yloc,&y_v_vec));
  PetscCall(VecRestoreArray(xloc,&x_v_vec));
  PetscCall(DMLocalToGlobalBegin(dm,yloc,ADD_VALUES,Y));
  PetscCall(DMLocalToGlobalEnd(dm,yloc,ADD_VALUES,Y));
  ...
  PetscFunctionReturn(0);
}
```

Convergence plot: The biharmonic equation



Flexibility

```
solver = PETScSolve(..., solver_parameters={'ksp_type': 'cg', 'pc_type': 'jacobi'})
```

```
solver = PETScSolve([eqn, bc1, bc2, bc3, ...], target=f, ...)
```

```
solver = PETScSolve({f: [eq_f, bc_f1, bc_f2,...], g: [eq_g, bc_g1, bc_g2,...]}, ...)
```

```
solver1 = PETScSolve(eq_u, target=u, ...)
solver2 = PETScSolve(eq_v, target=v, ...)
op = Operator([solver1, solver2], language='petsc')
```

And more ...

Next steps

- Stress testing compiler -> checking for correctness on large range of 2D+ 3D problems
- Additional solver options: Implement geometric multigrid and other solver and preconditioner options
- Further efficiency and QoL improvements: e.g remove constrained degrees of freedom from global solve with DMDA
- Staggered grids -> DMSTAG?

Summary

- New compiler passes in Devito now enable iterative solver support via PETSc
- Goal is to produce a flexible, portable, high-level framework
- Enables us to tackle new problem classes, such as incompressible flow in CFD
- Very much in development phase – actively exploring the best ways to interface with PETSc – if you have suggestions or ideas, I'd love to hear them!

GitHub Website

Summary

- New compiler passes in Devito now enable iterative solver support via PETSc
- Goal is to produce a flexible, portable, high-level framework
- Enables us to tackle new problem classes, such as incompressible flow in CFD
- Very much in development phase – actively exploring the best ways to interface with PETSc – if you have suggestions or ideas, I'd love to hear them!

Thank you!

GitHub

Website