Devito
Automated fast finite difference computation

Navjot Kukreja, Mathias Louboutin, Felippe Vieira, Fabio Luporini, Michael Lange, Gerard Gorman

WolfHPC 2016

November 13, 2016
Finite Difference Methods

\[ f'(a) \approx \frac{f(a + h) - f(a)}{h} \quad (1) \]

- Taylor series expansion
- Calculate derivatives of any order with relative simplicity
- Mathematically simple method for solving PDEs

Figure: Discretizing a function on a grid (Wikipedia)
Introduction

Devito Example - Seismic Imaging

Why FD?

The acoustic wave equation

$$\frac{\partial^2 u}{\partial x^2} - \frac{1}{c^2} \frac{\partial^2 u}{\partial t^2} = 0$$  \hspace{1cm} (2)

Discretized:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$  \hspace{1cm} (3)

**Figure:** Mesh in space and time for a 1D wave equation
Introduction
Devito
Example - Seismic Imaging

Why the wave equation?

Seismic imaging
  - RTM
    - Inputs: Velocity Model, Seismic Data
    - Output: Seismic image
  - FWI
    - Inputs: Initial Velocity Model, Seismic Data
    - Output: Improved Velocity Model

Figure: Offshore seismic survey
for ti in range(0, nt):
    for a in range(1, nx-1):
        for b in range(1, ny-1):
            if ti == 0:
                u[ti, a, b] = self.ts(0, 0, 0, 0, 0, 0,
                                      m[a, b], dt, h, damp[a, b])
            elif ti == 1:
                u[ti, a, b] = self.ts(0, u[ti - 1, a - 1, b],
                                      u[ti - 1, a, b],
                                      u[ti - 1, a + 1, b],
                                      u[ti - 1, a, b - 1],
                                      u[ti - 1, a, b + 1],
                                      m[a, b], dt, h, damp[a, b])
            else:
                u[ti, a, b] = self.ts(u[ti - 2, a, b],
                                      u[ti - 1, a - 1, b],
                                      u[ti - 1, a, b],
                                      u[ti - 1, a + 1, b],
                                      u[ti - 1, a, b - 1],
                                      u[ti - 1, a, b + 1],
                                      m[a, b], dt, h, damp[a, b])
Why does it need to be fast?

- Large number of operations: $\approx 5000$ FLOPs per loop iteration of a 16th order TTI Kernel
- Realistic problems have large grids: $1580 \times 1580 \times 1130 \approx 2.82$ Billion points (SEAM Benchmark)
- $2.82 \times 10^9 \times 5000 \times 3000(t) \times 2$ (forward-reverse) $\approx 8.5 \times 10^{16}$ per iteration of FWI
- Typically $\approx 30000$ FWI iterations ($\approx 2.5 \times 10^{21} = 2.5 \times 10^9$ TFLOPs)

$\approx 135$ wall-clock hours on the TACC Stampede (ideally)
Why automated

Computer Science

- Fast code is complex
  - Loop blocking
  - OpenMP clauses
  - Vectorisation - intrinsics
  - Memory - alignment, NUMA
  - Factorization
  - FMA
- Fast code is platform dependent
  - Intrinsics
  - CUDA
  - OpenCL
- Fast code is error prone

Geophysics

- Change of discretizations
- Change of physics
  - Anisotropy - VTI/TTI
  - Elastic equation
- Boundary conditions
- Continuous acquisition
SymPy - Symbolic computation in Python

- Symbolic computer algebra system written in pure Python
- Features
  - Complex symbolic expressions as Python object trees
  - Symbolic manipulation routines and interfaces
  - Convert symbolic expressions to numeric functions
    - Python or NumPy functions
    - C or Fortran kernels

For specialised domains generating C code is not enough!
Devito - a prototype Finite Difference DSL

Devito - A Finite Difference DSL for seismic imaging
- Aimed at creating fast high-order inversion kernels
- Development is driven by ”real world” problems
Based on SymPy expressions
- The acoustic wave equation:

\[
m\frac{\partial^2 u}{\partial t^2} + \eta \frac{\partial u}{\partial t} - \nabla u = 0 \quad (4)
\]

can be written as
\[
eqn = m * u.dt2 + \eta * u.dt - u.laplace
\]
Devito auto-generates optimised C kernel code
- OpenMP threading and vectorisation pragmas
- Cache blocking and auto-tuning
- Symbolic stencil optimisation (eg. CSE, hoisting)
Devito

Devito Data Objects
- `TimeData('u', shape=())`
- `DenseData('m', shape=())`

Stencil Equation
\[ eqn = m \times u.dt2 - u.laplace \]

Devito Operator
\[ Operator(eqn) \]

Devito Propagator

Devito Compiler
- GCC — Clang — Intel — Intel® Xeon Phi™

Act as symbols in expression
+ numpy arrays

Expands to symbolic kernel (finite-difference)

Transforms stencil in indexed format

Autogenerates C code

Compiles and loads platform specific executable function

**Figure**: An overview of Devito’s architecture
Real-world applications need more than PDE solvers

- File I/O and support for large datasets
- Non PDE kernel code e.g. sparse point interpolation
- Ability to easily interface with complex outer code

Devito follows the principle of graceful degradation

- Circumvent restrictions to the high-level API by customisation
- Devito translates high-level PDE-based stencils into ”matrix index” format in steps

  # High-level expression equivalent to f.dx2
  (-2*f(x, y) + f(x - h, y) + f(x + h, y)) / h**2

  # Low-level expression with explicit indexing
  (-2*f[x, y] + f[x - 1, y] + f[x + 1, y]) / h**2

- Allows custom functionality in auto-generated kernels
Seismic Imaging

Full waveform inversion

- Acoustic and TTI wave equations of varying spatial order
- Numerically verified against Industrial standard software on standard datasets
- Achieved performance is also comparable to industrial standard software
FWI
Example code for forward propagation

def forward(model, nt, dt, h, order=2):
    shape = model.shape
    m = DenseData(name="m", shape=shape,
                   space_order=order)
    m.data[:] = model
    u = TimeData(name="u", shape=shape,
                 time_dim=nt, time_order=2,
                 space_order=order, save=True)
    eta = DenseData(name="eta", shape=shape,
                     space_order=order)
    # Derive stencil from symbolic equation
    eqn = m * u.dt2 - u.laplace + eta * u.dt
    stencil = solve(eqn, u.forward)[0]
    op = Operator(stencils=Eq(u.forward, stencil),
                  nt=nt, subs={s: dt, h: h}, shape=shape,
                  forward=True)
    # Source injection code omitted for brevity
    op.apply()
// #include directives omitted for brevity
extern "C" int ForwardOperator(double *u_vec, double *damp_vec, double *m_vec, double *src_vec,
{
    double (*u)[130][130][130] = (double (*)[130][130][130]) u_vec;
    double (*damp)[130][130] = (double (*)[130][130]) damp_vec;
    double (*m)[130][130] = (double (*)[130][130]) m_vec;
    double (*src)[1] = (double (*)[1]) src_vec;
    float (*src_coords)[3] = (float (*)[3]) src_coords_vec;
    double (*rec)[101] = (double (*)[101]) rec_vec;
    float (*rec_coords)[3] = (float (*)[3]) rec_coords_vec;
{
    #pragma omp parallel
    for (int i4 = 0; i4 < 149; i4++)
    {
    }
    #pragma omp for schedule(static)
    for (int i1b = 1; i1b < 129 - (128 % i1block); i1b++)
    for (int i2b = 1; i2b < 129 - (128 % i2block); i2b++)
    for (int i1 = i1b; i1 < i1b + i1block; i1++)
    for (int i2 = i2b; i2 < i2b + i2block; i2++)
    {
        #pragma omp simd aligned(damp, m, u:64)
        for (int i3 = 1; i3 < 129; i3++)
        {
            double temp1 = damp[i1][i2][i3];
            double temp2 = m[i1][i2][i3];
            double temp4 = u[i4 - 1][i1][i2][i3];
            double temp5 = u[i4 - 2][i1][i2][i3];
            u[i4][i1][i2][i3] = ...
        }
    }
    for (int i1 = 129 - (128 % i1block); i1 < 129; i1++)
    for (int i2 = 1; i2 < 129 - (128 % i2block); i2++)
    {
```python
def adjoint(model, nt, dt, h, spc_order=2):
    m = DenseData("m", model.shape)
    m.data[:] = model
    v = TimeData(name='v', shape=model.shape, time_dim=nt,
                  time_order=2, space_order=spc_order, save=False)
    damp = DenseData("damp", model.shape)

    # Derive stencil from symbolic equation
    eqn = m * v.dt2 - v.laplace - damp * v.dt
    stencil = solve(eqn, v.backward)[0]

    # Add spacing substitutions
    subs = {s: dt, h: h}
    op = Operator(stencils=Eq(u.backward, stencil), nt=nt,
                  shape=model.shape, subs=subs, forward=False)
    op.apply()
```

Performance of acoustic forward operator

- Intel Xeon E5-2690v2 10C 3GHz and a Intel® Xeon Phi™ Knightscorner
- Model size 201 x 201 x 70 + 40 ABC
- Grid size 15m 10Hz Ricker wavelet source

<table>
<thead>
<tr>
<th>Arithmetic Intensity (Flops/Byte)</th>
<th>Performance of acoustic forward operator</th>
<th>Max Achievable</th>
<th>2nd Order</th>
<th>4th Order</th>
<th>6th Order</th>
<th>8th Order</th>
<th>10th Order</th>
<th>12th Order</th>
<th>14th Order</th>
<th>16th Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0625</td>
<td>451 Gflops/s</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>0.125</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>0.25</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>0.5</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>128</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>512</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
</tbody>
</table>

Graphs showing single precision performance (GFlops/s) and arithmetic intensity (Flops/Byte) for different orders and grid sizes.
Performance

2D Diffusion equation on a single core

<table>
<thead>
<tr>
<th></th>
<th>Wall time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy</td>
<td>5</td>
</tr>
<tr>
<td>lambda</td>
<td>15</td>
</tr>
<tr>
<td>devito</td>
<td>0.1</td>
</tr>
</tbody>
</table>
Performance Optimizations

Automated code optimizations:
- OpenMP and vectorization pragmas
- Loop blocking and auto-tuning for block size
- Automated roofline plotting for performance analysis

Symbolic Optimizations
- Common Subexpression elimination
  - Reduces compilation time from hours to seconds for large stencils
  - Enables further factorization techniques to reduce flops

Potential future optimizations
- Polyhedral compilation (time blocking)
- Automated data layout optimizations
Verification

Adjoint Test and Gradient Test

![Graph showing error vs. h for different orders of Taylor error: Zeroth order Taylor Error, O(h), First order Taylor Error, O(h^2).]
Conclusions

- **Devito**: A finite difference DSL for seismic imaging
  - Symbolic problem description (PDEs) via SymPy
  - Low-level API for kernel customisation
  - Automated performance optimisation

- Devito is driven by real-world scientific problems
  - Not yet another stencil compiler
  - Bridge the gap between stencil compilers and real world applications

- **Future work:**
  - Extend feature range to facilitate more science
  - MPI parallelism for larger models
  - Integrate stencil or polyhedral compiler backends
  - Additional symbolic optimisation (factorisation, hoisting, etc.)
  - Integrate automated verification tools to catch compiler bugs
Thank you

Publications

