TITLE:  
Automatic code generation - developing high performance propagators better, faster and cheaper

Abstract:  
Devito is an open-source domain-specific framework for solving partial differential equations from symbolic problem definitions by the finite difference method. In this tutorial we will show how Devito makes it possible for a non-HPC specialist to develop highly optimised and verified solvers, such as Tilted Transverse Isotropic (TTI) propagators and their adjoints, within hours rather than months.

We show how a domain specific language for finite differences can be built on top of the symbolic mathematics engine SymPy and Python, enabling finite difference solvers to be written in just a few lines of code. The combination of symbolic reasoning and cutting edge compiler technologies enables Devito to generate highly optimized and parallelised finite difference code in a low level language and bespoke for the target architecture.

Presenter’s Bio:  
Gerard J Gorman, PhD – Reader in Computational Science
Dr. Gorman is a Reader in Computational Science at Imperial College London and leads an Intel Parallel Computing Centre (http://www.opesci.org) focused on developing high productivity software technologies for seismic imaging. The tutorial will be co-presented with members of his team including Dr Michael Lange and Dr Fabio Luporini.

Dr. Gorman holds a PhD in Computational Physics from Imperial College London, a MSc in High Performance Computing and BSc in Physics from the National University of Ireland, Galway. He has over 20 years experience in high performance computing and modernizing large-scale legacy multi-physics codes for massively parallel computers.
Today his primary research interest is in code generation based software technologies that enable revolutionary rather than evolutionary approaches to software development.
Automatic code generation - developing high performance propagators better, faster and cheaper.

G. Gorman¹  M. Lange¹  F. Luporini¹  M. Louboutin²  
N. Kukreja¹  P. Witte²  F. Hermann²

June 13, 2017

¹Department of Earth Science and Engineering, Imperial College London, UK  
²Seismic Lab. for Imaging and Modeling, The University of British Columbia, Canada
Something is rotten in the state of Denmark...

Seismic inversion is extremely computationally demanding!
Yet new models are built around bespoke operators...

- Discretization and numerical methods are chosen a priori
- Performance optimization repeated for each architecture
- Requires many person-months (years) to develop new algorithms

Complex algorithms need end-to-end optimization

- Optimization at various levels of expertise
- Domain-specialists, numericists and compiler experts...
- But we can’t all be polymaths: We need separation of concerns!

---

Symbolic computation is a powerful tool!

- **FEniCS / Firedrake** - Finite element DSL packages

Velocity-stress formulation of elastic wave equation, with isotropic stress:

\[
\rho \frac{\partial \mathbf{u}}{\partial t} = \nabla \cdot \mathbf{T}
\]

\[
\frac{\partial \mathbf{T}}{\partial t} = \lambda (\nabla \cdot \mathbf{u}) \mathbb{I} + \mu \left( \nabla \mathbf{u} + \nabla^T \mathbf{u} \right)
\]

Weak form of equations written in UFL\(^1\):

\[
F_\_u = \text{density} \times \text{inner}(w, (u - u0)/dt) \times dx - \text{inner}(w, \text{div}(s0)) \times dx
\]

\[
solve(lhs(F_\_u) == rhs(F_\_u), u)
\]

---

Symbolic computation is a powerful tool!

**Dolfin-Adjoint**: Symbolic adjoints from symbolic PDEs

- Solves complex optimisation problems
- 2015 Wilkinson prize winner

Below is the optimal design of a double pipe that minimises the dissipated power in the fluid.

---

For Seismic imaging we need to solve inversion problems

- Finite Difference solvers for forward and adjoint runs
- Different types of wave equations with large complicated stencils

Many stencil languages exist, but few are practical

- Stencil still written by hand!
Devito - Automated finite difference propagators

- **SymPy** - Symbolic computer algebra system in pure Python\(^1\)

- 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 a great overview see A. Meuer's talk at SciPy 2016

For specialised domains generating C code is not enough!

Devito - Automated finite difference propagators

Devito: a finite difference DSL for seismic imaging

- Generates highly optimized stencil code
  - OpenMP threading and vectorisation pragmas
  - Cache blocking and auto-tuning
  - Symbolic stencil optimisation

- From concise mathematical syntax

Acoustic wave equation:

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

can be written as

\[
eqn = m * u.dt2 + eta * u.dt - u.laplace
\]
Development is driven by real-world problems!

- Productivity through code generation
  - Variable numerical discretisation stencil size
  - Individual operators in 10s of lines of code
  - Complete problem setups in a few 100 lines

- Fast high-order operators for inversion problems
  - Automated performance optimisation
  - Customization through hierarchical API

Devito - Automated finite difference propagators
Devito - Automated finite difference propagators

Development is driven by real-world problems!

**Devito Data Objects**

\[ u = \text{TimeData}(\text{\textasciitilde}u, \text{shape}=(\text{nx, ny})) \]
\[ m = \text{DenseData}(\text{\textasciitilde}m, \text{shape}=(\text{nx, ny})) \]

**Stencil Equation**

\[ \text{eqn} = m \ast u.\text{dt2} - u.\text{laplace} \]

**Devito Operator**

\[ \text{op} = \text{Operator}(\text{eqn}) \]

**Devito Propagator**

\[ u = \text{op.apply}(u.\text{data}, m.\text{data}) \]

**Devito Compiler**

GCC — Clang — Intel® — Intel® Xeon Phi™

\[ \text{op.compiler} = \text{IntelMIC} \]

- High-level function symbols associated with user data
- Symbolic equations that expand Finite Difference stencils
- Transform stencil expressions into explicit array accesses
- Generate low-level optimized kernel code and apply to data
- Compiles and loads Platform specific executable function
Devito - Automated finite difference propagators

Wave propagators in less than 100 lines

def forward(model, m, eta, src, rec, order=2, save=True):
    # Create the wavefield function
    u = TimeData(name='u', shape=model.shape, save=save,
                 time_order=2, space_order=order)

    # Derive stencil from symbolic equation
    eqn = m * u.dt2 - u.laplace + eta * u.dt
    stencil = solve(eqn, u.forward)[0]
    update_u = [Eq(u.forward, stencil)]

    # Inject wave as source term
    src_term = src.inject(field=u, expr=src * dt**2 / m)

    # Interpolate wavefield onto receivers
    rec_term = rec.interpolate(expr=u)

    # Create operator with source and receiver terms
    return Operator(update_u + src_term + rec_term,
                    subs={s: dt, h: model.spacing})
def adjoint(model, m, eta, src, rec, order=2):
    # Create the adjoint wavefield function
    v = TimeData(name='v', shape=model.shape,
                  time_order=2, space_order=order)

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

    # Inject the previous receiver readings
    rec_term = rec.inject(field=v, expr=rec * dt**2 / m)

    # Interpolate the adjoint-source
    src_term = src.interpolate(expr=v)

    # Create operator with source and receiver terms
    return Operator(update_v + rec_term + src_term,
                    subs={s: dt, h: model.spacing},
                    time_axis=Backward)
Devito - Automated finite difference propagators

Wave propagators in less than 100 lines

def gradient(model, m, eta, srca, rec, order=2):
    # Create the adjoint wavefield function
    v = TimeData(name='v', shape=model.shape,
                  time_order=2, space_order=order)

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

    # Inject the previous receiver readings
    rec_term = rec.inject(field=v, expr=rec * dt**2 / m)

    # Gradient update terms
    grad = DenseData(name='grad', shape=model.shape)
    grad_update = Eq(grad, grad - u.dt2 * v)

    # Create operator with source and receiver terms
    return Operator(update_v + [grad_update] + rec_term
                    subs={s: dt, h: model.spacing},
                    time_axis=Backward)
Devito - Automated finite difference propagators

Reverse time migration in less than 100 lines

```python
# Create the true and a smoothed model
m_true = Model(...)  
m_smooth = Model(...)

# Create operators for forward and gradient
op_forward = forward(...)  
op_gradient = forward(...)

# Create gradient field and loop over shots
grad = DenseData(name='grad', shape=model.shape)
for shot in shots:
    # Create receiver data from true model
    src = PointData(shot.source, ...)  
    rec_true = PointData(shot.receiver.coordinates, ...)  
    op_forward(src=src, rec=rec_true, m=m_true)

    # Run forward modelling operator with smooth model
    u = TimeData(name='u', shape=model.shape,     
                 time_order=2, space_order=order)  
    rec_smooth = PointData(shot.receiver.coordinates, ...)  
    op_forward(u=u, src=src, rec=rec_smooth, m=m_smooth)

    # Compute gradient update from the residual
    v = TimeData(name='v', shape=model.shape,     
                 time_order=2, space_order=order)  
    residual = rec_true.data[:] - rec_smooth.data[:]  
    op_gradient(u=u, v=v, grad=grad, rec=residual, m=m_smooth)
```
Devito - Automated finite difference propagators

Rapid propagator development and integration

- Test and verify in Python
- Operators in < 20 lines
- RTM loop in < 100 lines
- Variable stencil order
Devito - Automated finite difference propagators

From math to tuned HPC code in a few lines:

\[
\frac{m}{\rho} \frac{d^2 p(x, t)}{dt^2} - (1 + 2\epsilon)(G_{xx} + G_{yy})p(x, t) - \sqrt{(1 + 2\delta)}G_{zz}r(x, t) = q,
\]

\[
\frac{m}{\rho} \frac{d^2 r(x, t)}{dt^2} - \sqrt{(1 + 2\delta)}(G_{xx} + G_{yy})p(x, t) - G_{zz}r(x, t) = q,
\]

\[p(\cdot, 0) = 0,\]

\[\left. \frac{dp(x, t)}{dt} \right|_{t=0} = 0,\]

\[r(\cdot, 0) = 0,\]

\[\left. \frac{dr(x, t)}{dt} \right|_{t=0} = 0,\]

\[
D_{x1} = \cos(\theta)\cos(\phi) \frac{d}{dx} + \cos(\theta)\sin(\phi) \frac{d}{dy} - \sin(\theta) \frac{d}{dz}
\]

\[
D_{x2} = \cos(\theta)\cos(\phi) \frac{d}{dx} + \cos(\theta)\sin(\phi) \frac{d}{dy} - \sin(\theta) \frac{d}{dz}
\]

\[
G_{xx} = \frac{1}{2} \left( D_{x1}^T \frac{1}{\rho} D_{x1} + D_{x2}^T \frac{1}{\rho} D_{x2} \right)
\]

(incomplete) specification of a TTI (Tilted Transverse Isotropy) forward operator

rotated second order differential operators
Devito - Automated finite difference propagators

From math to tuned HPC code in a few lines:

```
ang0, ang1 = cos(theta), sin(theta)
ang2, ang3 = cos(phi), sin(phi)
Gyp = (ang3 * u.dx - ang2 * u.dyr)
Gyy = (first_derivative(Gyp * ang3, dim=x, side=centered, order=space_order, matvec=transpose) - 
      first_derivative(Gyp * ang2, dim=y, side=right, order=space_order, matvec=transpose))
Gyp2 = (ang3 * u.dxr - ang2 * u.dy)
Gyy2 = (first_derivative(Gyp2 * ang3, dim=x, side=right, order=space_order, matvec=transpose) - 
       first_derivative(Gyp2 * ang2, dim=y, side=centered, order=space_order, matvec=transpose))
Gxp = (ang0 * ang2 * u.dx + ang0 * ang3 * u.dyr - ang1 * u.dzr)
Gzr = (ang1 * ang2 * v.dx + ang1 * ang3 * v.dyr + ang0 * v.dzr)
Gxx = (first_derivative(Gxp * ang0 * ang2, dim=x, side=centered, order=space_order, matvec=transpose) + 
       first_derivative(Gxp * ang0 * ang3, dim=y, side=right, order=space_order, matvec=transpose) - 
       first_derivative(Gxp * ang1, dim=z, side=right, order=space_order, matvec=transpose))
Gzz = (first_derivative(Gzr * ang1 * ang2, dim=x, side=centered, order=space_order, matvec=transpose) + 
       first_derivative(Gzr * ang1 * ang3, dim=y, side=right, order=space_order, matvec=transpose) + 
       first_derivative(Gzr * ang0, dim=z, side=right, order=space_order, matvec=transpose))
Gxp2 = (ang0 * ang2 * u.dxr + ang0 * ang3 * u.dyr - ang1 * u.dz)
Gzr2 = (ang1*ang2*v.dxr+ang1*ang3*v.dy+ang0*v.dz) dim=x, side=right, order=space_order, matvec=transpose) + 
      first_derivative(Gxp2 * ang0 * ang3, dim=y, side=centered, order=space_order, matvec=transpose) - 
      first_derivative(Gxp2 * ang1, dim=z, side=centered, order=space_order, matvec=transpose))
Gzz2 = (first_derivative(Gzr2 * ang1 * ang2, dim=x, side=right, order=space_order, matvec=transpose) + 
       first_derivative(Gzr2 * ang1 * ang3, dim=y, side=centered, order=space_order, matvec=transpose) + 
       first_derivative(Gzr2 * ang0, dim=z, side=centered, order=space_order, matvec=transpose))

Hp = -(.5*Gxx + .5*Gxx2 + .5 * Gyy + .5*Gyy2)
Hzr = -(.5*Gzz + .5 * Gzz2)
```

```
Stencilp = 1.0 / (2.0 * m + s * damp) * (4.0 * m * u + (s * damp − 2.0 * m) * u.backward 
       + 2.0 * s**2 * (epsilon * Hp + delta * Hzr))
Stencilr = 1.0 / (2.0 * m + s * damp) * (4.0 * m * v + (s * damp − 2.0 * m) * v.backward 
       + 2.0 * s**2 * (delta * Hp + Hzr))
```
Devito - Automated finite difference propagators

From math to tuned HPC code in a few lines:

```python
def forward(model, m, eta, epsilon, delta, theta, phi, src, rec, order=2):
    # Create two wavefields
    u = TimeData(name='u', shape=model.shape, time_order=2, space_order=order)
    v = TimeData(name='v', shape=model.shape, time_order=2, space_order=order)

    # Create update expressions from stencil
    stencilp, stencilr = ...
    update_u = Eq(u.forward, stencilp)
    update_v = Eq(v.forward, stencilr)

    # Inject wave as source term
    src_term = src.inject(field=u, expr=src * dt**2 / m)
    src_term += src.inject(field=v, expr=src * dt**2 / m)

    # Interpolate wavefield onto receivers
    rec_term = rec.interpolate(expr=u)

    # Create operator with source and receiver terms
    return Operator([update_u, update_v] + src_term + rec_term,
                     subs={s: dt, h: model.spacing})
```
Devito - Automated finite difference propagators

Summary:

- **Productivity through code generation**
  - Acoustic operators in < 20 lines
  - TTI operators in < 100 lines
  - Variable discretization and stencil order
  - Fully executable Python code, easy to experiment
  - Complete problem setups in < 1000 lines

- **Fast wave propagators for inversion problems**
  - Highly efficient development through automation
  - Interoperability: Generated code is low-level C
  - **Automated performance optimisation**
The compilation flow: from symbolics to HPC code

Symbolic equations → DSE - Devito Symbolic Engine → Analysis → Loop scheduler → DLE - Devito Loop Engine → Declarations, headers, … → Code generation → C, MPI, OpenMP

Data objects → python → NumPy
The compilation flow: from symbolics to HPC code

Symbolic equations
- SymPy

Data objects
- NumPy

Analysis
- DSE - Devito Symbolic Engine
- Loop scheduler
- DLE - Devito Loop Engine
- Declarations, headers, …

Code generation

“FLOPS” OPTIMIZATIONS

C, MPI, OpenMP
The compilation flow: from symbolics to HPC code

Symbolic equations

Data objects

Analysis

DSE - Devito Symbolic Engine

Loop scheduler

DLE - Devito Loop Engine

Declarations, headers, …

Code generation

“FLOPS” OPTIMIZATIONS

“MEMORY” OPTIMIZATIONS

C, MPI, OpenMP
Devito Symbolic Engine

A sequence of compiler passes to reduce FLOPS (no loops at this stage!)
Devito Symbolic Engine

A sequence of compiler passes to reduce FLOPS (no loops at this stage!)

- Common sub-expressions elimination
  - C compilers do it already… but necessary for symbolic processing and compilation speed
Devito Symbolic Engine

A sequence of compiler passes to reduce FLOPS (no loops at this stage!)

- Common sub-expressions elimination
  - C compilers do it already… but necessary for symbolic processing and compilation speed

- Heuristic re-factorization of recurrent terms
  - E.g., finite difference weights: 0.3*a + ... + 0.3*b => 0.3*(a+b)
  - Many possibilities (doesn’t leverage domain properties yet!)
Devito Symbolic Engine

A sequence of compiler passes to reduce FLOPS (no loops at this stage!)

- Common sub-expressions elimination
  - C compilers do it already… but necessary for symbolic processing and compilation speed

- Heuristic re-factorization of recurrent terms
  - E.g., finite difference weights: $0.3a + \ldots + 0.3b \Rightarrow 0.3(a+b)$
  - Many possibilities (doesn’t leverage domain properties yet!)

Factorization impact:

<table>
<thead>
<tr>
<th>TTI, space order</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1100</td>
<td>950</td>
</tr>
<tr>
<td>8</td>
<td>2380</td>
<td>2120</td>
</tr>
<tr>
<td>12</td>
<td>4240</td>
<td>3760</td>
</tr>
<tr>
<td>16</td>
<td>6680</td>
<td>5760</td>
</tr>
</tbody>
</table>
Devito Symbolic Engine

A sequence of compiler passes to reduce FLOPS (no loops at this stage!)

- Common sub-expressions elimination
  - C compilers do it already… but necessary for symbolic processing and compilation speed

- Heuristic re-factorization of recurrent terms
  - E.g., finite difference weights: \(0.3*a + \ldots + 0.3*b \Rightarrow 0.3*(a+b)\)
  - Many possibilities (doesn’t leverage domain properties yet!)
Devito Symbolic Engine

A sequence of compiler passes to reduce FLOPS (no loops at this stage!)

- Common sub-expressions elimination
  - C compilers do it already… but necessary for symbolic processing and compilation speed

- Heuristic re-factorization of recurrent terms
  - E.g., finite difference weights: $0.3*a + ... + 0.3*b \Rightarrow 0.3*(a+b)$
  - Many possibilities (doesn’t leverage domain properties yet!)

- Fundamental in compute-bound stencil codes (e.g., TTI)
  - E.g., $\sin(\phi[i,j,k]), \sin(\phi[i-1,j-1,k-1])$

CSE

Factorization

Alias detection

WIP
DSE’s aliases detection algorithms

Alias detection

Fundamental in compute-bound stencil codes (e.g., TTI)

tmpl = \ldots \times \sin(\phi[i, j, k]) + \ldots + 0.4 \times \sin(\phi[i-1, j-1, k-1]) + \ldots +
\ldots 0.1 \times \sin(\phi[i+2, j+2, k+2]) + \ldots

Observations (focus on underlined sub-expressions)
- Same operators (\sin)
- Same operands (\phi)
- Same indices (i, j, k)
- Linearly dependent index vectors ([i, j, k], [i-1, j-1, k-1], [i+2, j+2, k+2])
DSE’s aliases detection algorithms

Alias detection

Fundamental in compute-bound stencil codes (e.g., TTI)

tmp1 = \ldots \times \sin(\phi_{i,j,k}) + \ldots + 0.4 \times \sin(\phi_{i-1,j-1,k-1}) + \ldots + \ldots 0.1 \times \sin(\phi_{i+2,j+2,k+2}) + \ldots

Observations (focus on underlined sub-expressions)
- Same operators (\sin)
- Same operands (\phi)
- Same indices (i, j, k)
- Linearly dependent index vectors ([i, j, k], [i-1, j-1, k-1], [i+2, j+2, k+2])

B_{i,j,k} = \sin(\phi_{i,j,k})

tmp1 = \ldots \times B_{i,j,k} + \ldots + 0.4 \times B_{i-1,j-1,k-1} + \ldots + \ldots + 0.1 \times B_{i+2,j+2,k+2} + \ldots
Devito Symbolic Engine

A sequence of compiler passes to reduce FLOPS (no loops at this stage!)

- Common sub-expressions elimination
  - C compilers do it already… but necessary for symbolic processing and compilation speed

- Heuristic re-factorization of recurrent terms
  - E.g., finite difference weights: \(0.3a + \ldots + 0.3b \Rightarrow 0.3(a+b)\)
  - Many possibilities (doesn’t leverage domain properties yet!)

- Fundamental in compute-bound stencil codes (e.g., TTI)
  - E.g., \(\sin(\phi[i,j,k]), \sin(\phi[i-1,j-1,k-1])\)

- Heuristic hoisting of time-invariant quantities
  - Currently, only (expensive) trigonometric functions applied to space-varying quantities
Devito Loop Engine

A sequence of compiler passes to introduce parallelism, SIMD vectorization and to improve data locality
Devito Loop Engine

A sequence of compiler passes to introduce parallelism, SIMD vectorization and to improve data locality

- Cache optimizations (mostly L1 cache)
  - Loop fission + elemental functions (register locality)
  - Padding + data alignment (split loads)
Devito Loop Engine

A sequence of compiler passes to introduce parallelism, SIMD vectorization and to improve data locality

- Cache optimizations (mostly L1 cache)
- Loop fission + elemental functions (register locality)
- Padding + data alignment (split loads)

Intel VTune, Broadwell E5-2620 v4, TTI space orders 4-8-12
Devito Loop Engine

A sequence of compiler passes to introduce parallelism, SIMD vectorization and to improve data locality

- Cache optimizations (mostly L1 cache)
- Loop fission + elemental functions (register locality)
- Padding + data alignment (split loads)

Intel VTune, Broadwell E5-2620 v4, TTI space orders 4-8-12
Devito Loop Engine

A sequence of compiler passes to introduce parallelism, SIMD vectorization and to improve data locality

- Cache optimizations (mostly L1 cache)
  - Loop fission + elemental functions (register locality)
  - Padding + data alignment (split loads)
Devito Loop Engine

A sequence of compiler passes to introduce parallelism, SIMD vectorization and to improve data locality

- Cache optimizations (mostly L1 cache)
  - Loop fission + elemental functions (register locality)
  - Padding + data alignment (split loads)

- DRAM optimizations: loop blocking
  - 1D, 2D, 3D supported (but no time loop)
  - Auto-tuning supported
Devito Loop Engine

A sequence of compiler passes to introduce parallelism, SIMD vectorization and to improve data locality

- Cache optimizations (mostly L1 cache)
  - Loop fission + elemental functions (register locality)
  - Padding + data alignment (split loads)

- DRAM optimizations: loop blocking
  - 1D, 2D, 3D supported (but no time loop)
  - Auto-tuning supported

- SIMD vectorization
  - Through compiler auto-vectorization
  - Why should I bother using intrinsics?
  - Various #pragmas introduced (e.g., ivdep, alignment, …)
Devito Loop Engine

A sequence of compiler passes to introduce parallelism, SIMD vectorization and to improve data locality

- Cache optimizations (mostly L1 cache)
  - Loop fission + elemental functions (register locality)
  - Padding + data alignment (split loads)

- DRAM optimizations: loop blocking
  - 1D, 2D, 3D supported (but no time loop)
  - Auto-tuning supported

- SIMD vectorization
  - Through compiler auto-vectorization
  - Why should I bother using intrinsics?
  - Various `#pragma` introduced (e.g., ivdep, alignment, …)

- OpenMP
  - `#pragma collapse` clause on the Xeon Phi
Devito Loop Engine

A sequence of compiler passes to introduce parallelism, SIMD vectorization and to improve data locality

- SIMD vectorization
  - Through compiler auto-vectorization
- Why should I bother using intrinsics?
  - Various #pragma s introduced (e.g., ivdep, alignment, …)
- DRAM optimizations: loop blocking
  - 1D, 2D, 3D supported (but no time loop)
  - Auto-tuning supported
- Cache optimizations (mostly L1 cache)
  - Loop fission + elemental functions (register locality)
  - Padding + data alignment (split loads)

Other optimizations:
- Cache opts
- DRAM opts
- SIMD
- Parallelism
Acoustic on Broadwell

Acoustic[(512, 512, 512), TO=[2]], with varying <DSE,DLE>, on bdwb_ss

Performance (GFlops/s)

Operational intensity (Flops/Byte)
Acoustic on Broadwell

64% of attainable peak (best case)
TTI on Broadwell (8 threads, single socket)

$\text{Tti}[(512, 512, 512), \text{TO}=[2]]$, with varying $<\text{DSE, DLE}>$, on bdwb_ss
TTI on Broadwell (8 threads, single socket)

Quite far from attainable peak!
TTI on Xeon Phi (64 threads, cache mode, quadrant)

Tti[(512, 512, 512),TO=[2]], with varying <DSE,DLE>, on ekf_1

Performance (GFlops/s)

Operational intensity (Flops/Byte)
It’s extremely difficult (only a few examples in the literature) reaching such a high TTI space order.
Conclusions and resources

• Devito: an efficient and sustainable finite difference DSL
• Driven/inspired by real-world seismic imaging
• Interdisciplinary research effort
• Based on actual compiler technology

Useful links
• http://www.opesci.org
• https://github.com/opesci/devito
Vertical Integration

Verification of the generated code:

- Comparison with a reference implementation - IWAVE
- Adjoint test
  - For any $x \in \text{span}(P_s A^T P_r^T)$, $y \in \text{span}(P_r A^T P_s^T)$
  - $< P_r A^T P_s^T x, y > - < x P_s A^T P_r^T y > = 0$
  - Passes with at-least 8 matching significant digits for 2D and 3D with 2, 4, 6, 8, 10, 12th order discretization
- Gradient test
  - For a small model perturbation $dm$, $\phi_s(m + hdm) = \phi_s(m) + O(h)$ and $\phi_s(m + hdm) = \phi_s(m) + h(J[m]^T \delta d)dm + O(h^2)$
  - Passes at the level of the machine’s accuracy
- Automatic formal code verification being implemented \(^1\)

---