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

G. Gorman\textsuperscript{1} M. Lange\textsuperscript{1} F. Luporini\textsuperscript{1} M. Louboutin\textsuperscript{2} \\
N. Kukreja\textsuperscript{1} P. Witte\textsuperscript{2} C. Yount\textsuperscript{3} F. Hermann\textsuperscript{2}

June 13, 2017

\textsuperscript{1}Department of Earth Science and Engineering, Imperial College London, UK \\
\textsuperscript{2}Seismic Lab. for Imaging and Modeling, The University of British Columbia, Canada \\
\textsuperscript{3}Intel Corporation
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 ¹
- 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:

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

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

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

\[
F_u = \text{density} \times \text{inner}(\mathbf{w}, (\mathbf{u} - \mathbf{u}_0)/\partial t) \times dx - \text{inner}(\mathbf{w}, \text{div}(\mathbf{s}_0)) \times dx
\]

\[
\text{solve}(\text{lhs}(F_u) == \text{rhs}(F_u), \mathbf{u})
\]

---

Symbolic computation is a powerful tool!

**Dolfin-Adjoint**: Symbolic adjoints from symbolic PDEs\(^1\)

- 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.

---

Devito - Automated finite difference propagators

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 (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!

- Compiler-level optimization to leverage performance
- Stencil optimization is a research field of its own

---

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

  \[
  eqn = m \ast u.dt2 + eta \ast u.dt - u.laplace
  \]
Devito - Automated finite difference propagators

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

Development is driven by real-world problems!

- **Devito Data Objects**
  
  $u = \text{TimeData}(\text{'u'}, \text{shape}=(nx, ny))$
  
  $m = \text{DenseData}(\text{'m'}, \text{shape}=(nx, ny))$

- **Stencil Equation**
  
  $eqn = m \ast u.\text{dt2} - u.\text{laplace}$

- **Devito Operator**
  
  $op = \text{Operator}(eqn)$

- **Devito Compiler**
  
  GCC — Clang — Intel® — Intel® Xeon Phi™
  
  $op.\text{compiler} = \text{IntelMIC}$

- High-level function symbols associated with user data
- Symbolic equations that expand Finite Difference stencils
- Transform stencil expressions into explicit array accesses
- Compiles and loads Platform specific executable function
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})
Wave propagators in less than 100 lines

```python
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:

\[
\begin{align*}
\frac{m}{\rho} \frac{d^2 p(x, t)}{dt^2} & - (1 + 2\epsilon)(G_{xx} + G_{yy}) p(x, t) - \sqrt{1 + 2\delta} G_{zz} r(x, t) = q, \\
\frac{m}{\rho} \frac{d^2 r(x, t)}{dt^2} & - \sqrt{1 + 2\delta} (G_{xx} + G_{yy}) p(x, t) - G_{zz} r(x, t) = q, \\
p(., 0) &= 0, \\
\frac{dp(x, t)}{dt} \bigg|_{t=0} &= 0, \\
r(., 0) &= 0, \\
\frac{dr(x, t)}{dt} \bigg|_{t=0} &= 0,
\end{align*}
\]

\begin{align*}
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)
\end{align*}

(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:

\[
\text{ang0, ang1} = \cos(\theta), \sin(\theta) \\
\text{ang2, ang3} = \cos(\phi), \sin(\phi) \\
\text{Gyp} = (\text{ang3} \times u.d_x - \text{ang2} \times u.d_y) \\
\text{Gyy} = \left( \text{first_derivative}(\text{Gyp} \times \text{ang3}, \text{dim}=x, \text{side}=\text{centered}, \text{order}=\text{space_order}, \text{matvec}=\text{transpose}) - \right. \\
\left. \text{first_derivative}(\text{Gyp} \times \text{ang2}, \text{dim}=y, \text{side}=\text{right}, \text{order}=\text{space_order}, \text{matvec}=\text{transpose}) \right)
\]

\[
\text{Gyp2} = (\text{ang3} \times u.d_x - \text{ang2} \times u.d_y) \\
\text{Gyy2} = \left( \text{first_derivative}(\text{Gyp2} \times \text{ang3}, \text{dim}=x, \text{side}=\text{right}, \text{order}=\text{space_order}, \text{matvec}=\text{transpose}) - \right. \\
\left. \text{first_derivative}(\text{Gyp2} \times \text{ang2}, \text{dim}=y, \text{side}=\text{centered}, \text{order}=\text{space_order}, \text{matvec}=\text{transpose}) \right)
\]

\[
\text{Gxp} = (\text{ang0} \times \text{ang2} \times u.d_x + \text{ang0} \times \text{ang3} \times u.d_y - \text{ang1} \times u.d_z) \\
\text{Gzr} = (\text{ang1} \times \text{ang2} \times v.d_x + \text{ang1} \times \text{ang3} \times v.d_y + \text{ang0} \times v.d_z)
\]

\[
\text{Gxx} = \left( \text{first_derivative}(\text{Gxp} \times \text{ang0} \times \text{ang2}, \text{dim}=x, \text{side}=\text{centered}, \text{order}=\text{space_order}, \text{matvec}=\text{transpose}) + \right. \\
\left. \text{first_derivative}(\text{Gxp} \times \text{ang0} \times \text{ang3}, \text{dim}=y, \text{side}=\text{right}, \text{order}=\text{space_order}, \text{matvec}=\text{transpose}) - \right. \\
\left. \text{first_derivative}(\text{Gxp} \times \text{ang1}, \text{dim}=z, \text{side}=\text{right}, \text{order}=\text{space_order}, \text{matvec}=\text{transpose}) \right)
\]

\[
\text{Gzz} = \left( \text{first_derivative}(\text{Gzr} \times \text{ang1} \times \text{ang2}, \text{dim}=x, \text{side}=\text{centered}, \text{order}=\text{space_order}, \text{matvec}=\text{transpose}) + \right. \\
\left. \text{first_derivative}(\text{Gzr} \times \text{ang1} \times \text{ang3}, \text{dim}=y, \text{side}=\text{right}, \text{order}=\text{space_order}, \text{matvec}=\text{transpose}) + \right. \\
\left. \text{first_derivative}(\text{Gzr} \times \text{ang0}, \text{dim}=z, \text{side}=\text{right}, \text{order}=\text{space_order}, \text{matvec}=\text{transpose}) \right)
\]

\[
\text{Gxp2} = (\text{ang0} \times \text{ang2} \times u.d_x + \text{ang0} \times \text{ang3} \times u.d_y - \text{ang1} \times u.d_z) \\
\text{Gzr2} = (\text{ang1} \times \text{ang2} \times v.d_x + \text{ang1} \times \text{ang3} \times v.d_y + \text{ang0} \times v.d_z)
\]

\[
\text{Gxx2} = \left( \text{first_derivative}(\text{Gxp2} \times \text{ang0} \times \text{ang2}, \text{dim}=x, \text{side}=\text{centered}, \text{order}=\text{space_order}, \text{matvec}=\text{transpose}) + \right. \\
\left. \text{first_derivative}(\text{Gxp2} \times \text{ang0} \times \text{ang3}, \text{dim}=y, \text{side}=\text{centered}, \text{order}=\text{space_order}, \text{matvec}=\text{transpose}) - \right. \\
\left. \text{first_derivative}(\text{Gxp2} \times \text{ang1}, \text{dim}=z, \text{side}=\text{centered}, \text{order}=\text{space_order}, \text{matvec}=\text{transpose}) \right)
\]

\[
\text{Gzz2} = \left( \text{first_derivative}(\text{Gzr2} \times \text{ang1} \times \text{ang2}, \text{dim}=x, \text{side}=\text{right}, \text{order}=\text{space_order}, \text{matvec}=\text{transpose}) + \right. \\
\left. \text{first_derivative}(\text{Gzr2} \times \text{ang1} \times \text{ang3}, \text{dim}=y, \text{side}=\text{centered}, \text{order}=\text{space_order}, \text{matvec}=\text{transpose}) + \right. \\
\left. \text{first_derivative}(\text{Gzr2} \times \text{ang0}, \text{dim}=z, \text{side}=\text{centered}, \text{order}=\text{space_order}, \text{matvec}=\text{transpose}) \right)
\]

\[
\text{Hp} = -(.5 \times \text{Gxx} + .5 \times \text{Gxx2} + .5 \times \text{Gyy} + .5 \times \text{Gyy2}) \\
\text{Hzr} = -(.5 \times \text{Gzz} + .5 \times \text{Gzz2})
\]

\[
\text{stencilp} = 1.0 / \left( (2.0 \times m + s \times \text{damp}) \times (4.0 \times m \times u + (s \times \text{damp} - 2.0 \times m) \times u.\text{backward} \right. \\
\left. + 2.0 \times s^{*2} \times (\text{epsilon} \times \text{Hp} + \text{delta} \times \text{Hzr}) \right)
\]

\[
\text{stencilr} = 1.0 / \left( (2.0 \times m + s \times \text{damp}) \times (4.0 \times m \times v + (s \times \text{damp} - 2.0 \times m) \times v.\text{backward} \right. \\
\left. + 2.0 \times s^{*2} \times (\text{delta} \times \text{Hp} + \text{Hzr}) \right)
\]
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
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

Data objects

Analysis

DSE - Devito Symbolic Engine

Loop scheduler

DLE - Devito Loop Engine

Declarations, headers, …

Code generation

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

C, MPI, OpenMP
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

“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 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 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:**

TTI, space order 4: $1100 \rightarrow 950$
TTI, space order 8: $2380 \rightarrow 2120$
TTI, space order 12: $4240 \rightarrow 3760$
TTI, space order 16: $6680 \rightarrow 5760$
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 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!)
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 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., \(\sin(\phi[i,j,k]), \sin(\phi[i-1,j-1,k-1])\)
DSE’s aliases detection algorithms

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

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

**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 \cdot \sin(\phi_{i,j,k}) + \ldots + 0.4 \cdot \sin(\phi_{i-1,j-1,k-1}) + \ldots +
\ldots 0.1 \cdot \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])

$\mathbf{B}_{i,j,k} = \sin(\phi_{i,j,k})$

$\text{tmp1} = \ldots \cdot \mathbf{B}_{i,j,k} + \ldots + 0.4 \cdot \mathbf{B}_{i-1,j-1,k-1} + \ldots + \ldots + 0.1 \cdot \mathbf{B}_{i+2,j+2,k+2} + \ldots$
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 factorization of recurrent terms**
  - E.g., finite difference weights: $0.3\ast a + \ldots + 0.3\ast b \Rightarrow 0.3\ast(a+b)$
  - Many possibilities (doesn’t leverage domain properties yet!)

- **Fundamental in compute-bound stencil codes (e.g., TTI)**
  - E.g., $\sin(\phi[i,j,k]), \sin(\phi[i-1,j-1,k-1])$
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 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 opts**
  - Cache optimizations (mostly L1 cache)
    - Loop fission + elemental functions (register locality)
    - Padding + data alignment (split loads)

- **DRAM opts**
  - 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 opts

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

DRAM opts

Parallelism

SIMD

Yet Another Stencil Kernel

Y*A*S*K

WIP
Acoustic on Broadwell

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

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

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

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
TTI on Xeon Phi (64 threads, cache mode, quadrant)

It's extremely difficult (only a few examples in the literature) reaching such a high TTI space order.

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

Performance (GFlops/s)

Operational intensity (Flops/Byte)
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