



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

---

G. Gorman<sup>1</sup>   M. Lange<sup>1</sup>   F. Luporini<sup>1</sup>   M. Louboutin<sup>2</sup>  
N. Kukreja<sup>1</sup>   P. Witte<sup>2</sup>   F. Hermann<sup>2</sup>

June 13, 2017

<sup>1</sup>Department of Earth Science and Engineering, Imperial College London, UK

<sup>2</sup>Seismic Lab. for Imaging and Modeling, The University of British Columbia, Canada



# Devito - Automated finite difference propagators

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<sup>1</sup>
- 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!

---

<sup>1</sup>M. Louboutin, M. Lange, F. J. Herrmann, N. Kukreja, and G. Gorman. Performance prediction of finite-difference solvers for different computer architectures. *Computers and Geosciences*, 105:148 – 157, 2017

# Devito - Automated finite difference propagators

Symbolic computation is a powerful tool!

- FEniCS / Firedrake - Finite element DSL packages

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

$$\rho \frac{\partial \mathbf{u}}{\partial t} = \nabla \cdot \mathbb{T}$$

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

Weak form of equations written in UFL<sup>1</sup>:

```
F_u = density*inner(w, (u - u0)/dt)*dx - inner(w, div(s0))*dx
solve(lhs(F_u) == rhs(F_u), u)
```

---

<sup>1</sup>Anders Logg, Kent-Andre Mardal, and Garth Wells. *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Springer Publishing Company, Incorporated, 2012

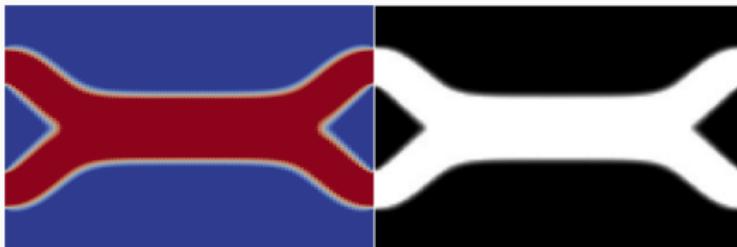
# Devito - Automated finite difference propagators

**Symbolic computation is a powerful tool!**

**Dolfin-Adjoint:** Symbolic adjoints from symbolic PDEs<sup>1</sup>

- 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.



---

<sup>1</sup>P. E. Farrell, D. A. Ham, S. W. Funke, and M. E. Rognes. Automated derivation of the adjoint of high-level transient finite element programs. *SIAM Journal on Scientific Computing*, 35(4):C369–C393, 2013

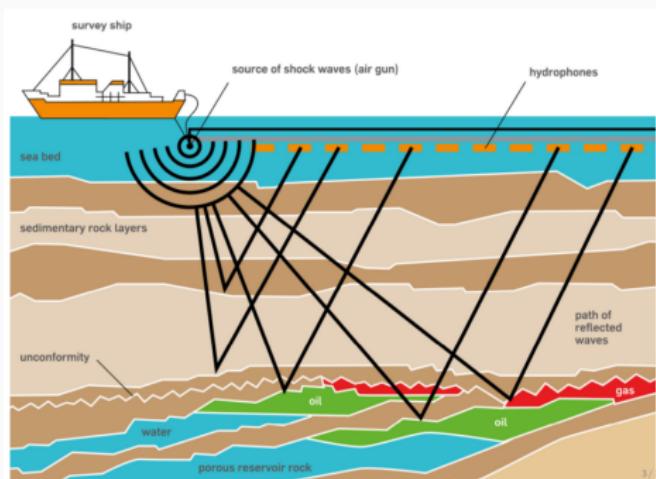
# Devito - Automated finite difference propagators

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<sup>1</sup>
- 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!**

---

<sup>1</sup>Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, Thilina Rathnayake, et al. Sympy: Symbolic computing in python. Technical report, PeerJ Preprints, 2016

# Devito - Automated finite difference propagators

## Devito: a finite difference DSL for seismic imaging

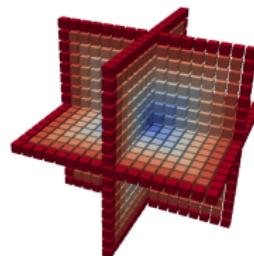
- Generates highly optimized stencil code
  - OpenMP threading and vectorisation pragmas
  - Cache blocking and auto-tuning
  - Symbolic stencil optimisation
- From concise mathematical syntax

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
```



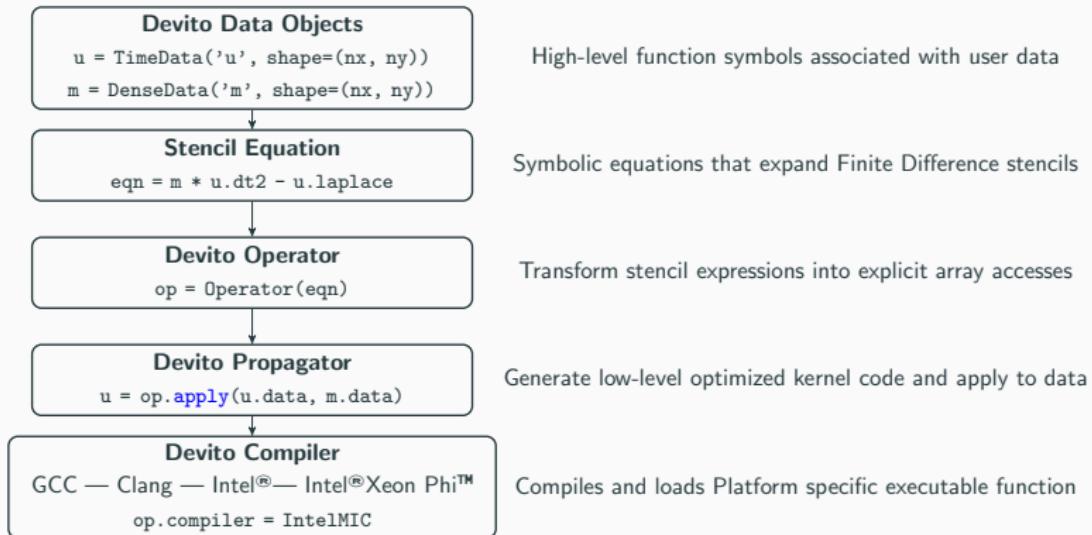
# Devito - Automated finite difference propagators

**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 - 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})
```

# Devito - Automated finite difference propagators

## Wave propagators in less than 100 lines

```
def adjoint(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)

    # Interpolate the adjoint-source
    srca_term = srca.interpolate(expr=v)

    # Create operator with source and receiver terms
    return Operator(update_v + rec_term + srca_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

```
# 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)

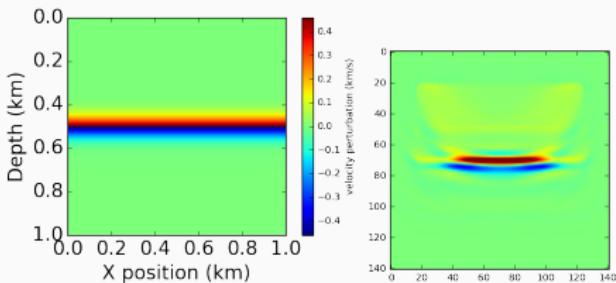
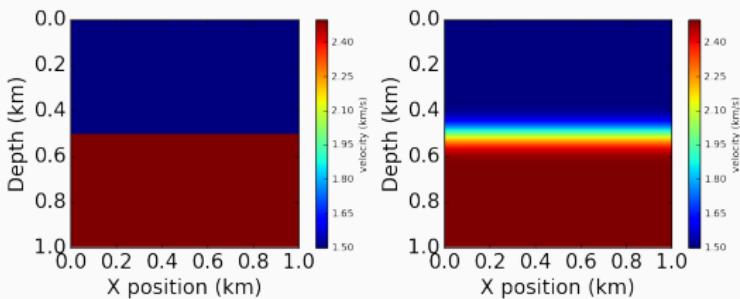
    # Run forward modelling operator with smooth model
    u = TimeData(name='u', shape=model.shape,
                 time_order=2, space_order=order)
    rec_smooth = PointData(shot.receiver.coordinates, ...)
    op_forward(u=u, src=src, rec=rec_smooth, m=m_smooth)

    # 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:

$$\frac{m}{\rho} \frac{d^2 p(x, t)}{dt^2} - (1 + 2\epsilon)(G_{\bar{x}\bar{x}} + G_{\bar{y}\bar{y}})p(x, t) - \sqrt{(1 + 2\delta)}G_{\bar{z}\bar{z}}r(x, t) = q,$$

$$\frac{m}{\rho} \frac{d^2 r(x, t)}{dt^2} - \sqrt{(1 + 2\delta)}(G_{\bar{x}\bar{x}} + G_{\bar{y}\bar{y}})p(x, t) - G_{\bar{z}\bar{z}}r(x, t) = q,$$

$$p(., 0) = 0,$$

$$\frac{dp(x, t)}{dt}|_{t=0} = 0,$$

$$r(., 0) = 0,$$

$$\frac{dr(x, t)}{dt}|_{t=0} = 0,$$

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

$$D_{x1} = \cos(\theta)\cos(\phi)\frac{d}{dx}\Big|_l + \cos(\theta)\sin(\phi)\frac{d}{dy} - \sin(\theta)\frac{d}{dz}$$

$$D_{x2} = \cos(\theta)\cos(\phi)\frac{d}{dx} + \cos(\theta)\sin(\phi)\frac{d}{dy}\Big|_l - \sin(\theta)\frac{d}{dz}|_l$$

$$G_{\bar{x}\bar{x}} = \frac{1}{2} \left( D_{x1}^T \left( \frac{1}{\rho} \right) D_{x1} + D_{x2}^T \left( \frac{1}{\rho} \right) D_{x2} \right) \quad \text{← rotated second order differential operators}$$

# 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.dy - 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)
Hrz = -(.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 * Hrz))
stencirl = 1.0 / (2.0 * m + s * damp) * (4.0 * m * v + (s * damp - 2.0 * m) * v.backward
    + 2.0 * s**2 * (delta * Hp + Hrz))
```

# Devito - Automated finite difference propagators

From math to tuned HPC code in a few lines:

```
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})
```

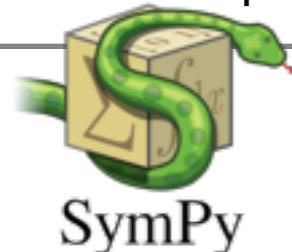
# Devito - Automated finite difference propagators

## 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



Data objects



Analysis

DSE - Devito Symbolic Engine

Loop scheduler

DLE - Devito Loop Engine

Declarations, headers, ...

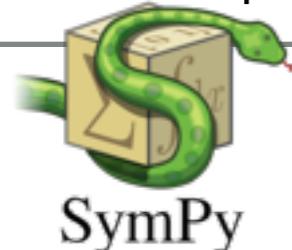


Code generation

**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, ...

**“FLOPS”  
OPTIMIZATIONS**

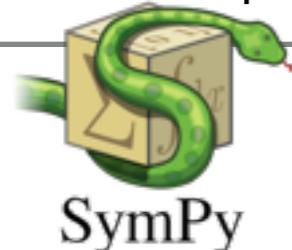


Code generation

**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**

**“MEMORY”  
OPTIMIZATIONS**

**C, MPI, OpenMP**

# Devito Symbolic Engine

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

# Devito Symbolic Engine

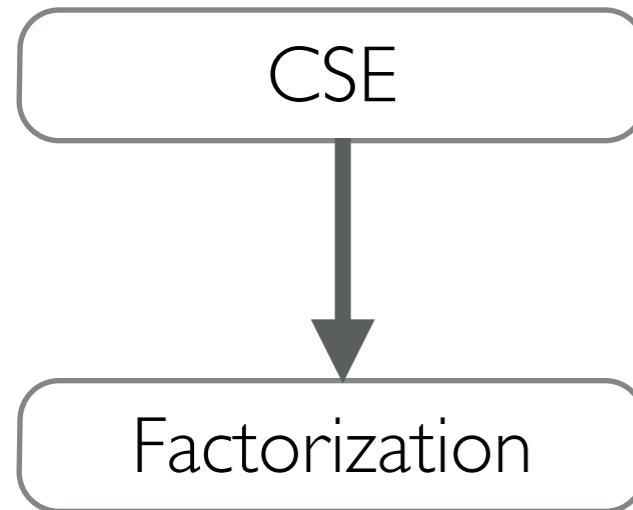
A sequence of compiler passes to reduce FLOPS (no loops at this stage!)

CSE

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

# Devito Symbolic Engine

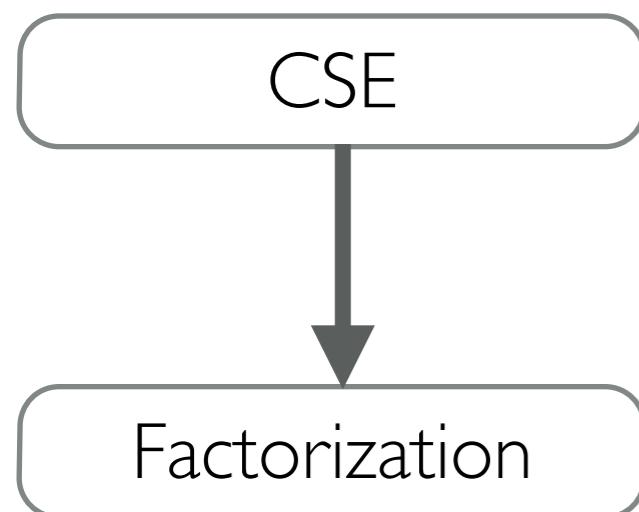
A sequence of compiler passes to reduce FLOPS (no loops at this stage!)



- 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:  $0.3*a + \dots + 0.3*b \Rightarrow 0.3*(a+b)$
  - 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:  $0.3*a + \dots + 0.3*b \Rightarrow 0.3*(a+b)$
  - Many possibilities (doesn't leverage domain properties yet!)

## Factorization impact:

TTI, space order 4: 1100 → 950

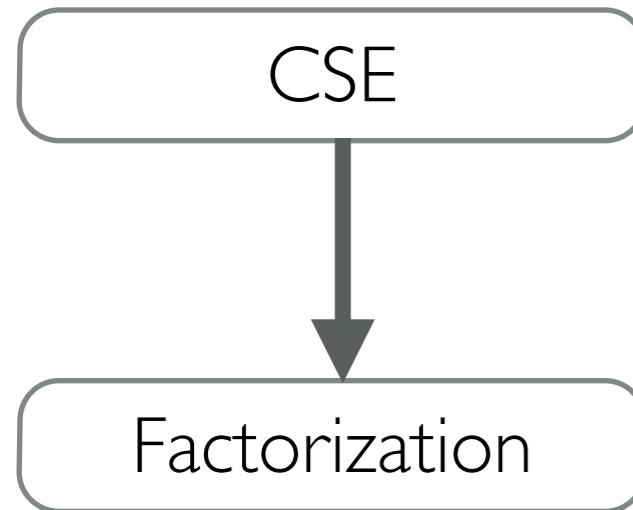
TTI, space order 8: 2380 → 2120

TTI, space order 12: 4240 → 3760

TTI, space order 16: 6680 → 5760

# Devito Symbolic Engine

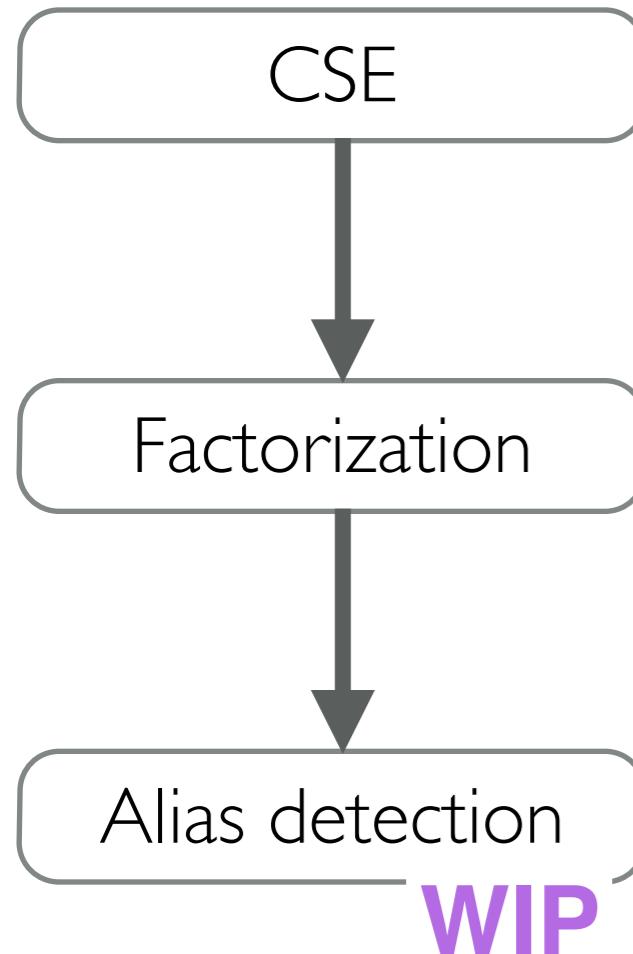
A sequence of compiler passes to reduce FLOPS (no loops at this stage!)



- Common sub-expressions elimination
  - C compilers do it already... but necessary for symbolic processing and compilation speed
- Heuristic re-factorization of recurrent terms
  - E.g., finite difference weights:  $0.3*a + \dots + 0.3*b \Rightarrow 0.3*(a+b)$
  - 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:  $0.3*a + \dots + 0.3*b \Rightarrow 0.3*(a+b)$
  - 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])`

# DSE's aliases detection algorithms



Alias detection

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

```
tmp1 = ...*sin(phii,j,k) + ... + 0.4*sin(phii-1,j-1,k-1) + ... +  
...0.1*sin(phii+2,j+2,k+2) + ...
```

## 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 = ...*sin(phii,j,k) + ... + 0.4*sin(phii-1,j-1,k-1) + ... +  
...0.1*sin(phii+2,j+2,k+2) + ...
```

## Observations (focus on underlined sub-expressions)

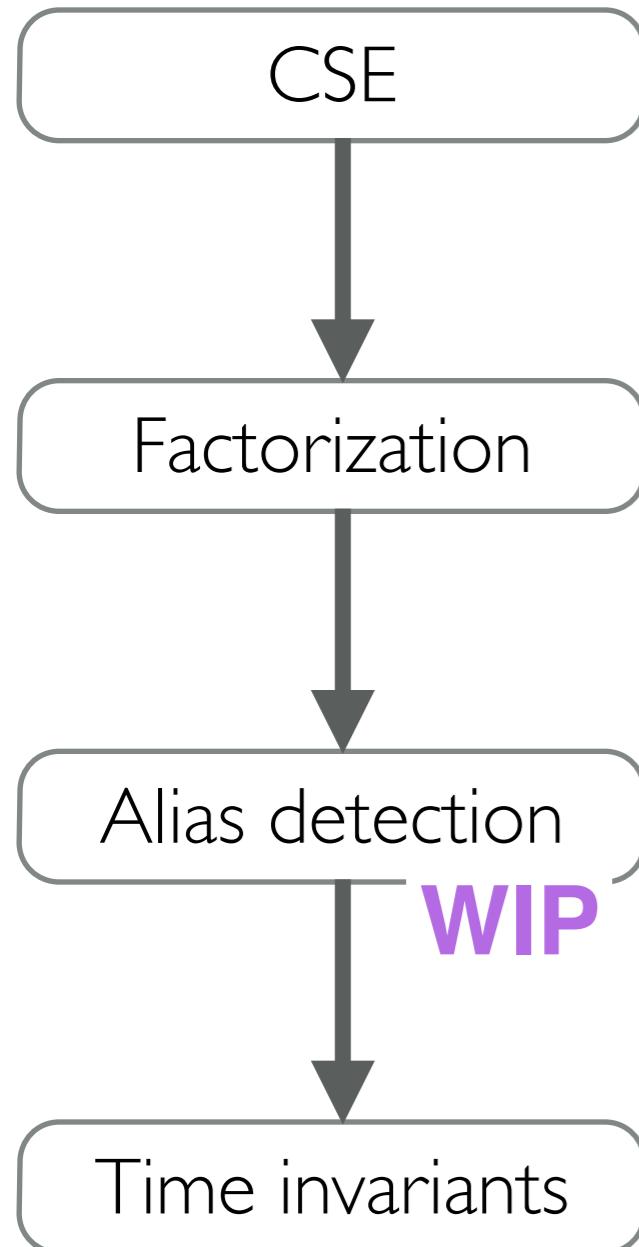
- Same operators (`sin`)
- Same operands (`phi`)
- Same indices (`i, j, k`)
- Linearly dependent index vectors (`[i, j, k], [i-1, j-1, k-1], [i+2, j+2, k+2]`)

```
B[i,j,k] = sin(phi[i,j,k])
```

```
tmp1 = ...*B[i,j,k] + ... + 0.4*B[i-1,j-1,k-1] + ... + ... + 0.1*B[i+2,j+2,k+2] + ...
```

# 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:  $0.3*a + \dots + 0.3*b \Rightarrow 0.3*(a+b)$
  - 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])`
- 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 opts

- 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

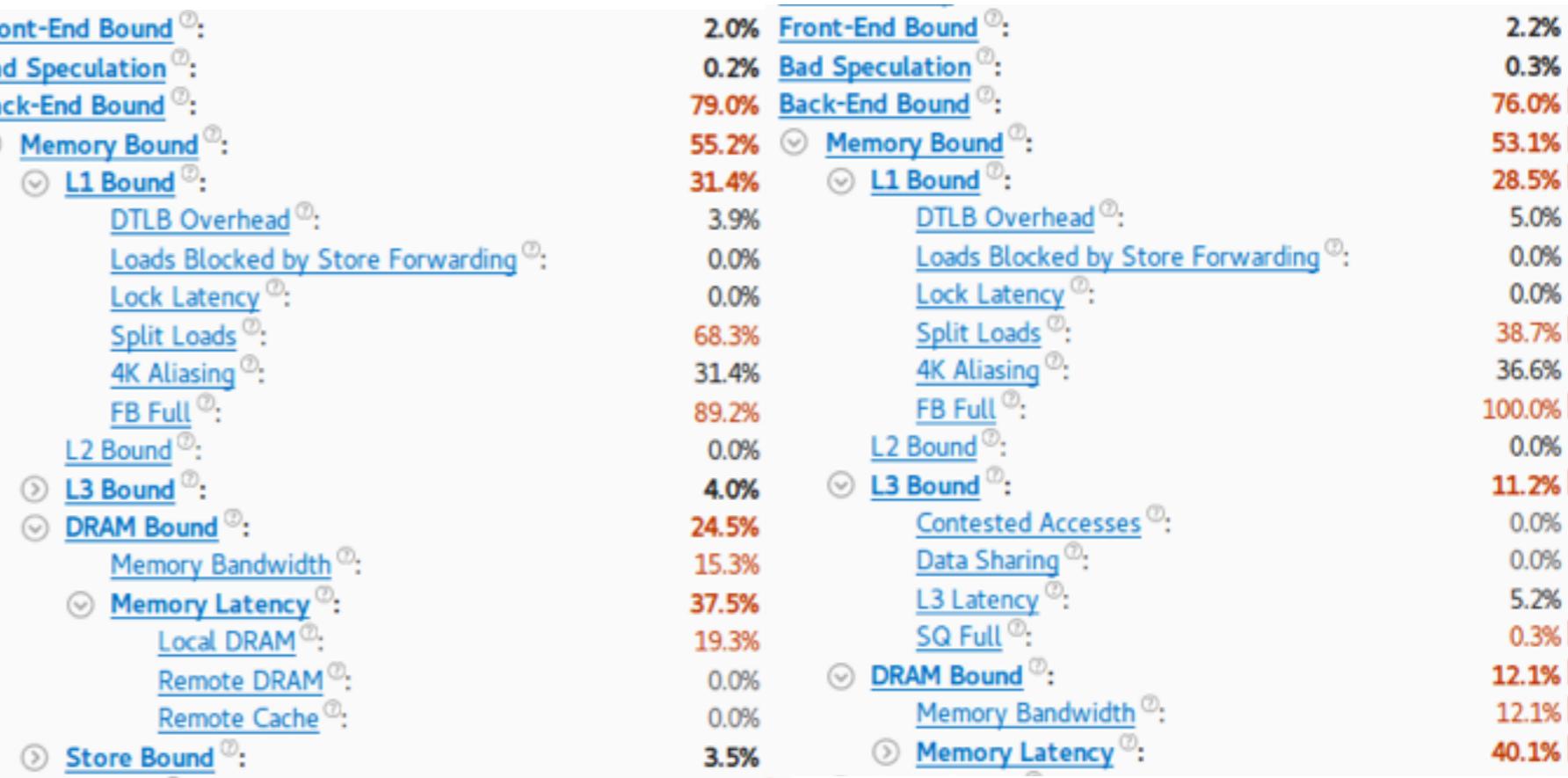
- Cache optimizations (mostly L1 cache)
  - Loop fission + elemental functions (register locality)
  - Padding + data alignment (split loads)

General Exploration General Exploration viewpoint  
Collection Log Analysis Target Analysis Type Summary

Elapsed Time <sup>?</sup>: 487.651s

Clockticks:	5,099,766,000,000
Instructions Retired:	4,963,350,000,000
CPI Rate <sup>?</sup> :	1.027
MUX Reliability <sup>?</sup> :	0.754
Front-End Bound <sup>?</sup> :	1.7%
Bad Speculation <sup>?</sup> :	0.3%
Back-End Bound <sup>?</sup> :	77.5%
Memory Bound <sup>?</sup> :	54.7%
L1 Bound <sup>?</sup> :	31.0%
DTLB Overhead <sup>?</sup> :	3.0%
Loads Blocked by Store Forwarding <sup>?</sup> :	0.0%
Lock Latency <sup>?</sup> :	0.0%
Split Loads <sup>?</sup> :	13.5%
4K Aliasing <sup>?</sup> :	22.8%
FB Full <sup>?</sup> :	22.1%
L2 Bound <sup>?</sup> :	0.0%
L3 Bound <sup>?</sup> :	3.3%
DRAM Bound <sup>?</sup> :	24.7%
Memory Bandwidth <sup>?</sup> :	24.1%
Memory Latency <sup>?</sup> :	34.7%
Local DRAM <sup>?</sup> :	95.0%
Remote DRAM <sup>?</sup> :	0.0%
Remote Cache <sup>?</sup> :	0.0%
Store Bound <sup>?</sup> :	1.8%
Core Bound <sup>?</sup> :	22.7%
Retiring <sup>?</sup> :	20.5%
Total Thread Count:	205

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

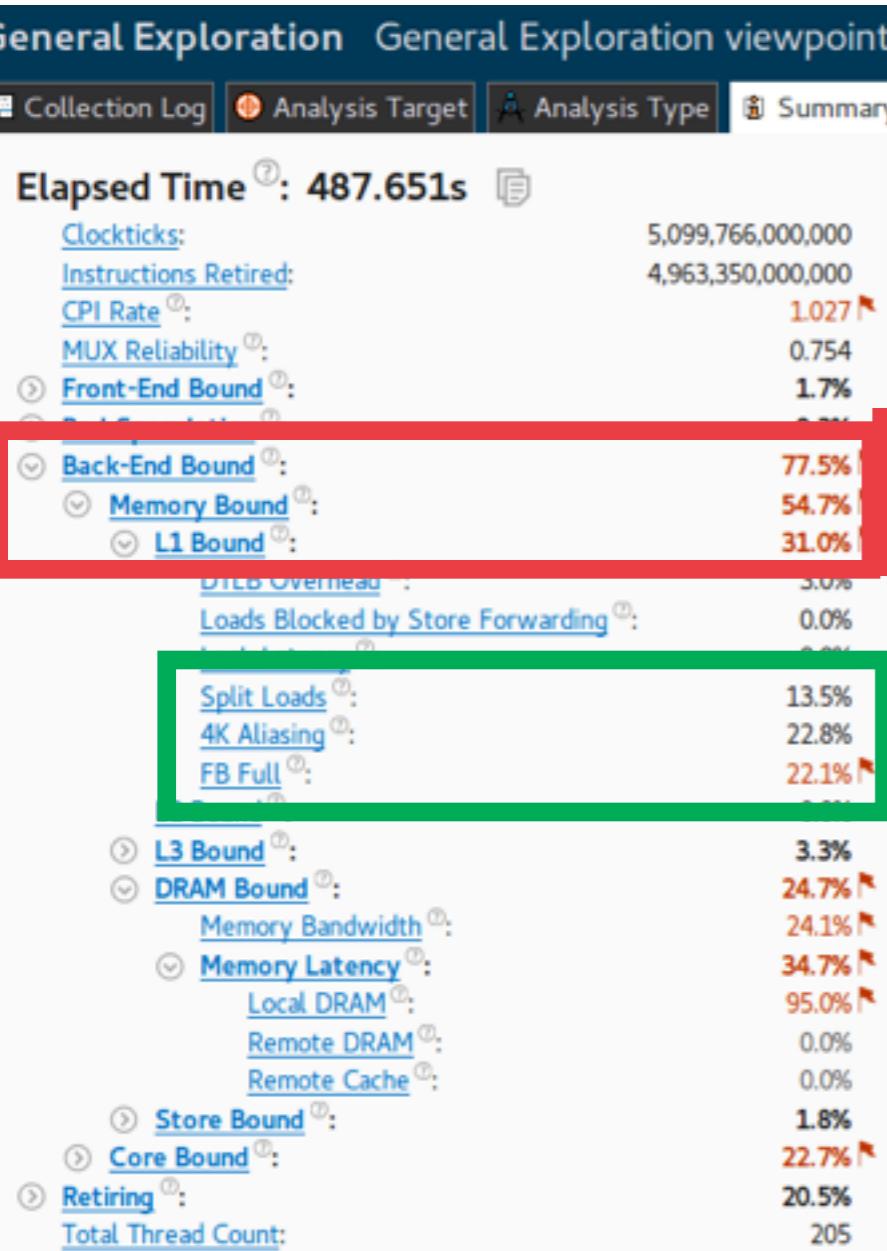


# Devito Loop Engine

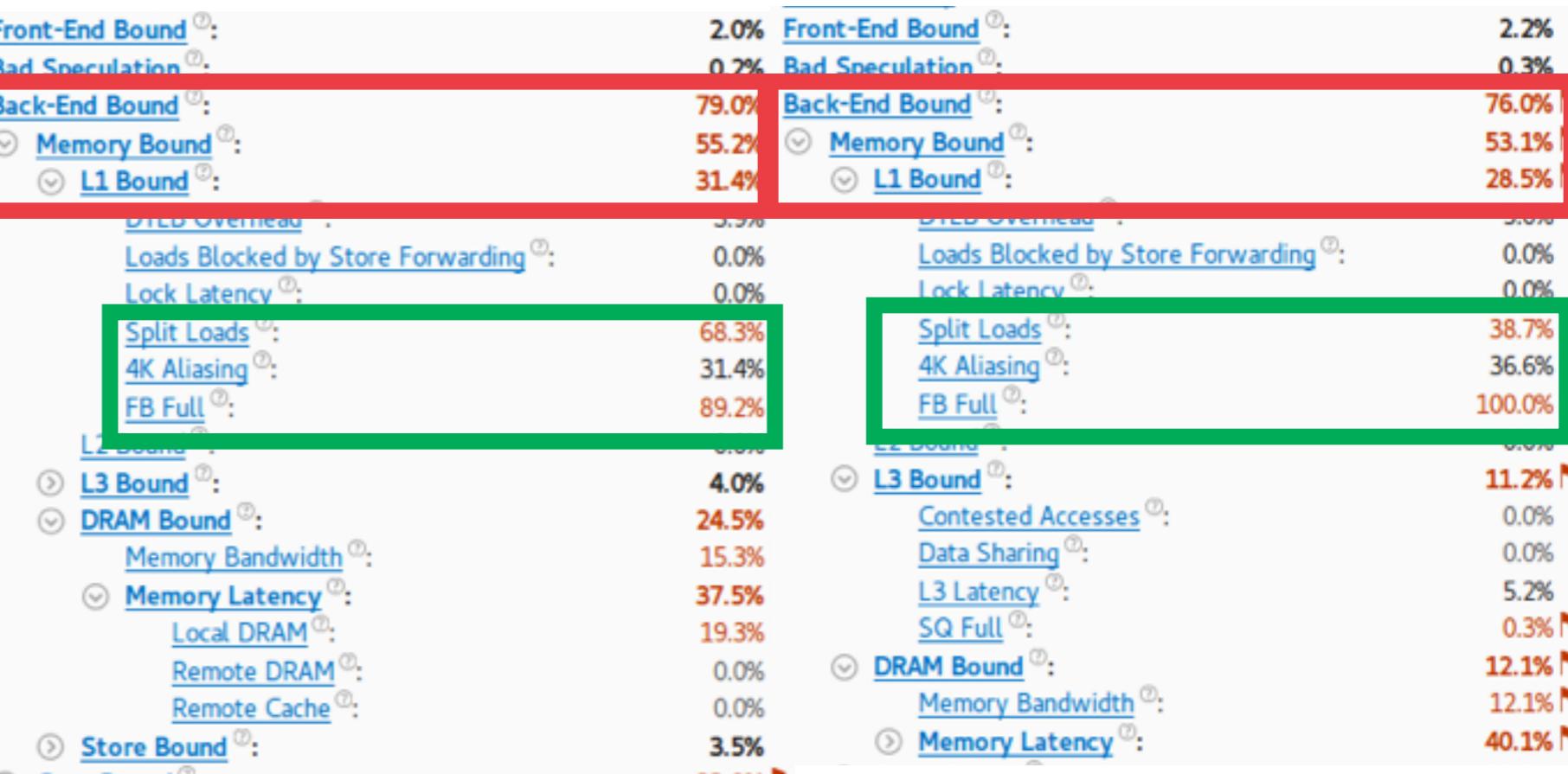
A sequence of compiler passes to introduce parallelism, SIMD vectorization and to improve data locality

Cache opts

- 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

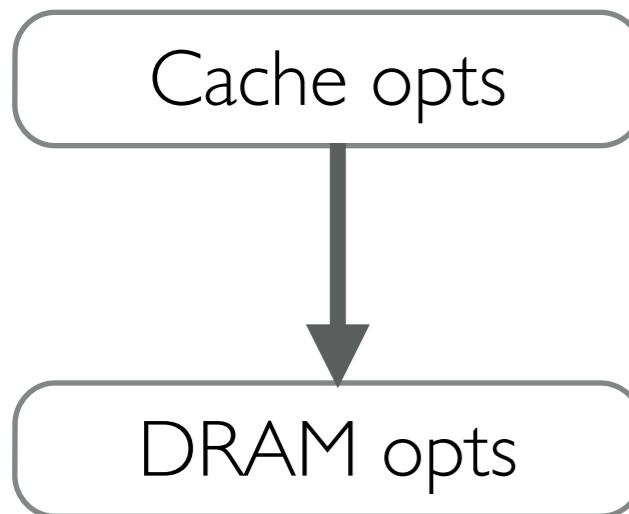
A sequence of compiler passes to introduce parallelism, SIMD vectorization and to improve data locality

Cache opts

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

# Devito Loop Engine

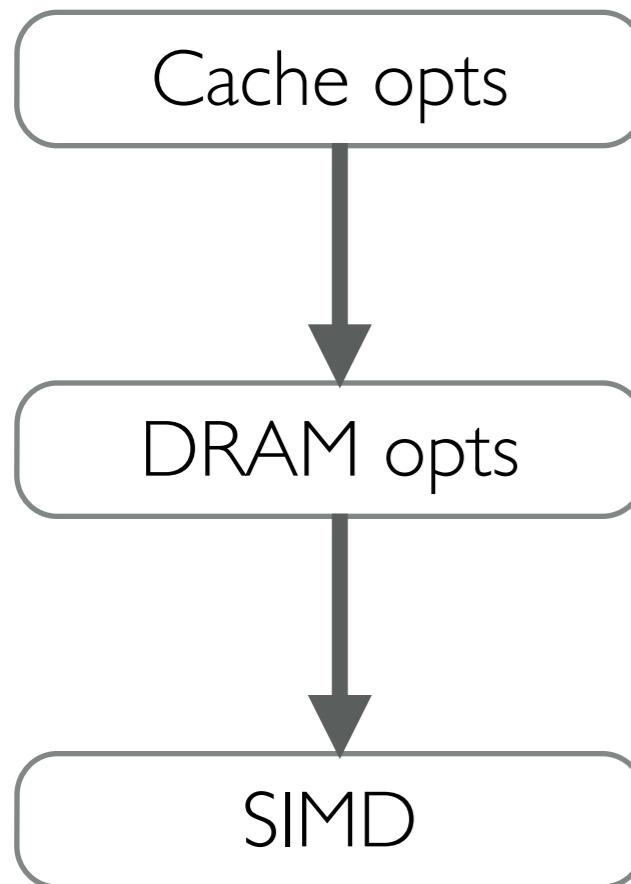
A sequence of compiler passes to introduce parallelism, SIMD vectorization and to improve data locality



- 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

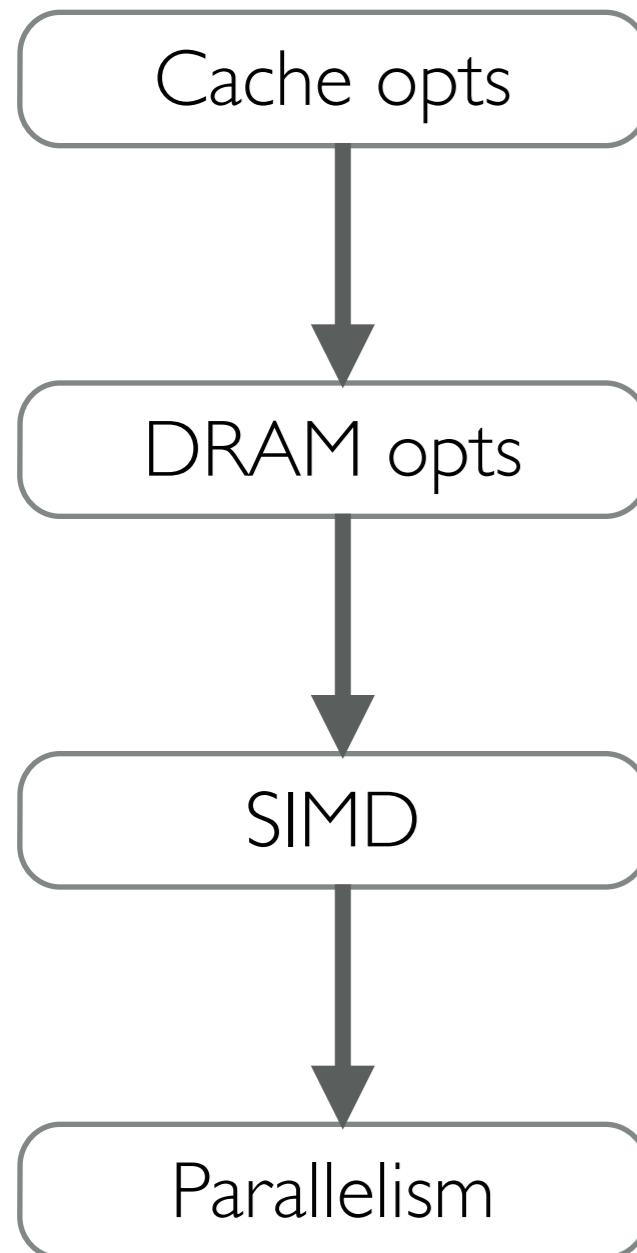
A sequence of compiler passes to introduce parallelism, SIMD vectorization and to improve data locality



- 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

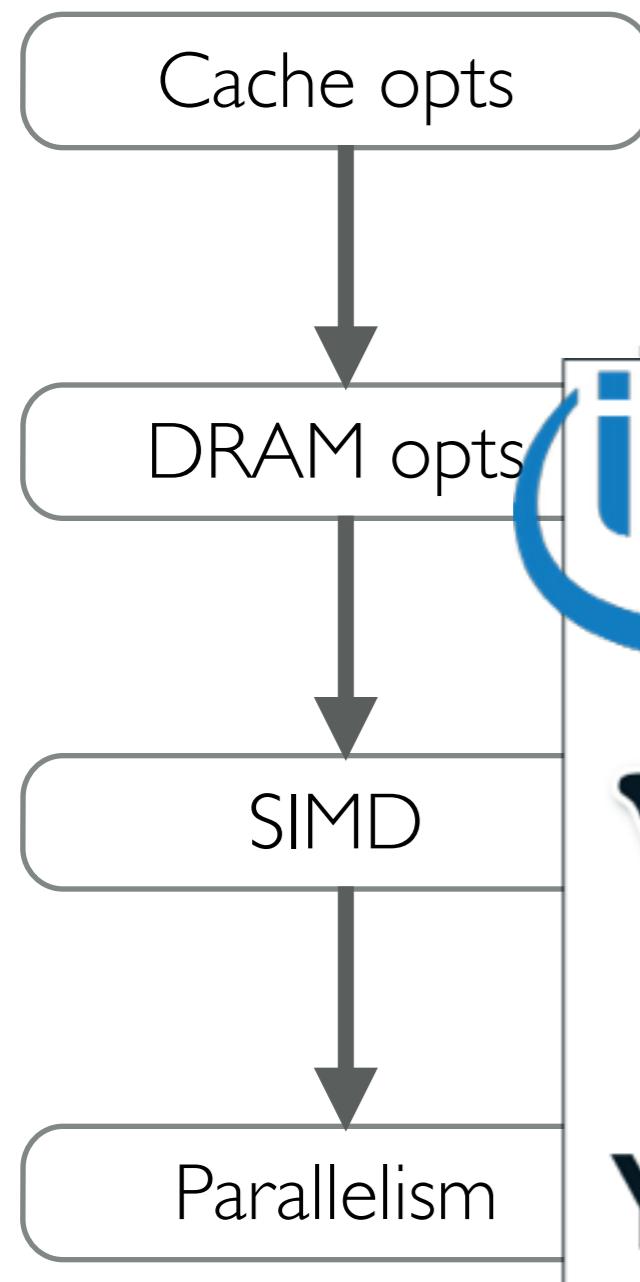
A sequence of compiler passes to introduce parallelism, SIMD vectorization and to improve data locality



- 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 #pragmas introduced (e.g., ivdep, alignment, ...)
- OpenMP
  - #pragma collapse clause on the Xeon Phi

# Devito Loop Engine

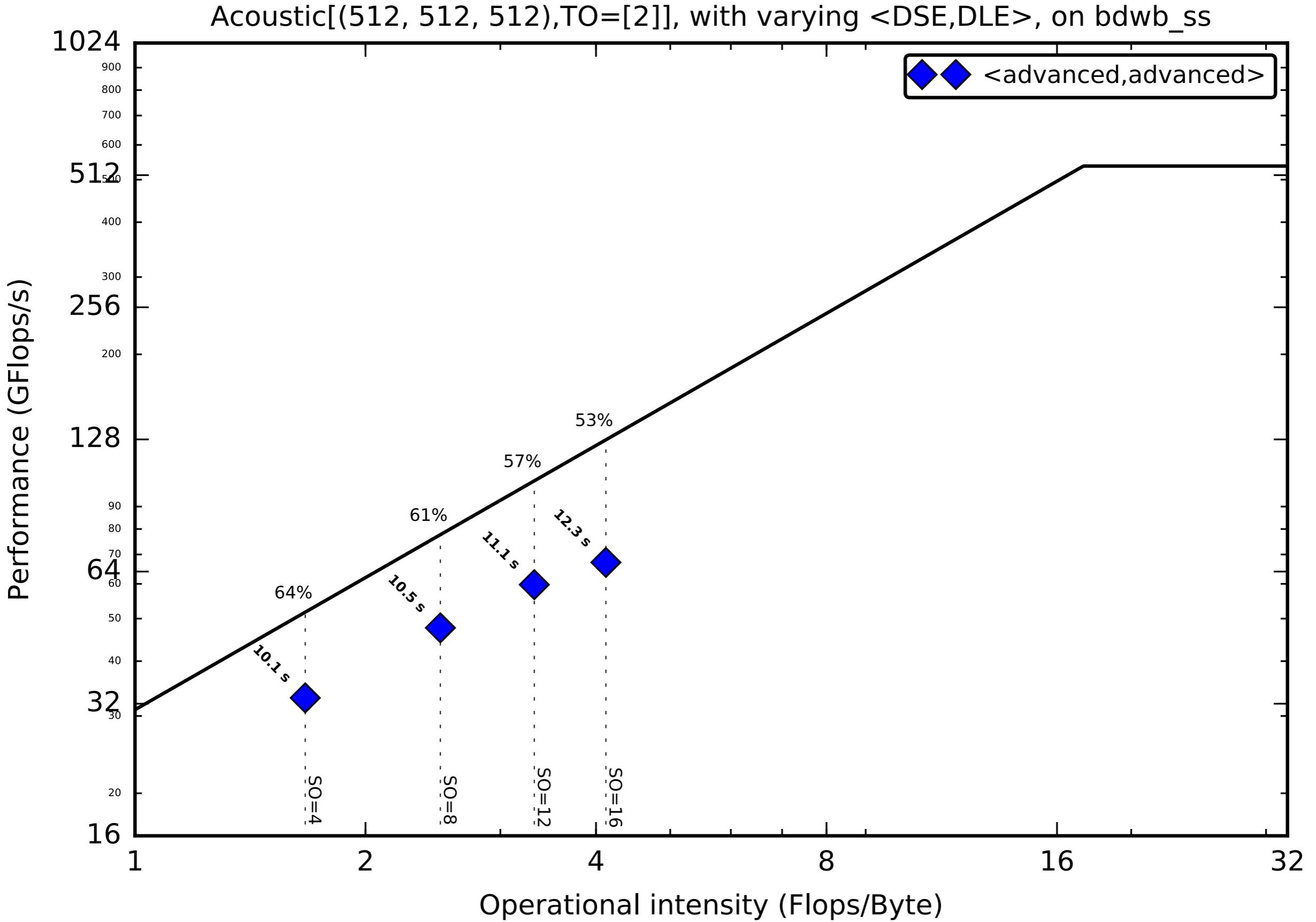
A sequence of compiler passes to introduce parallelism, SIMD vectorization and to improve data locality



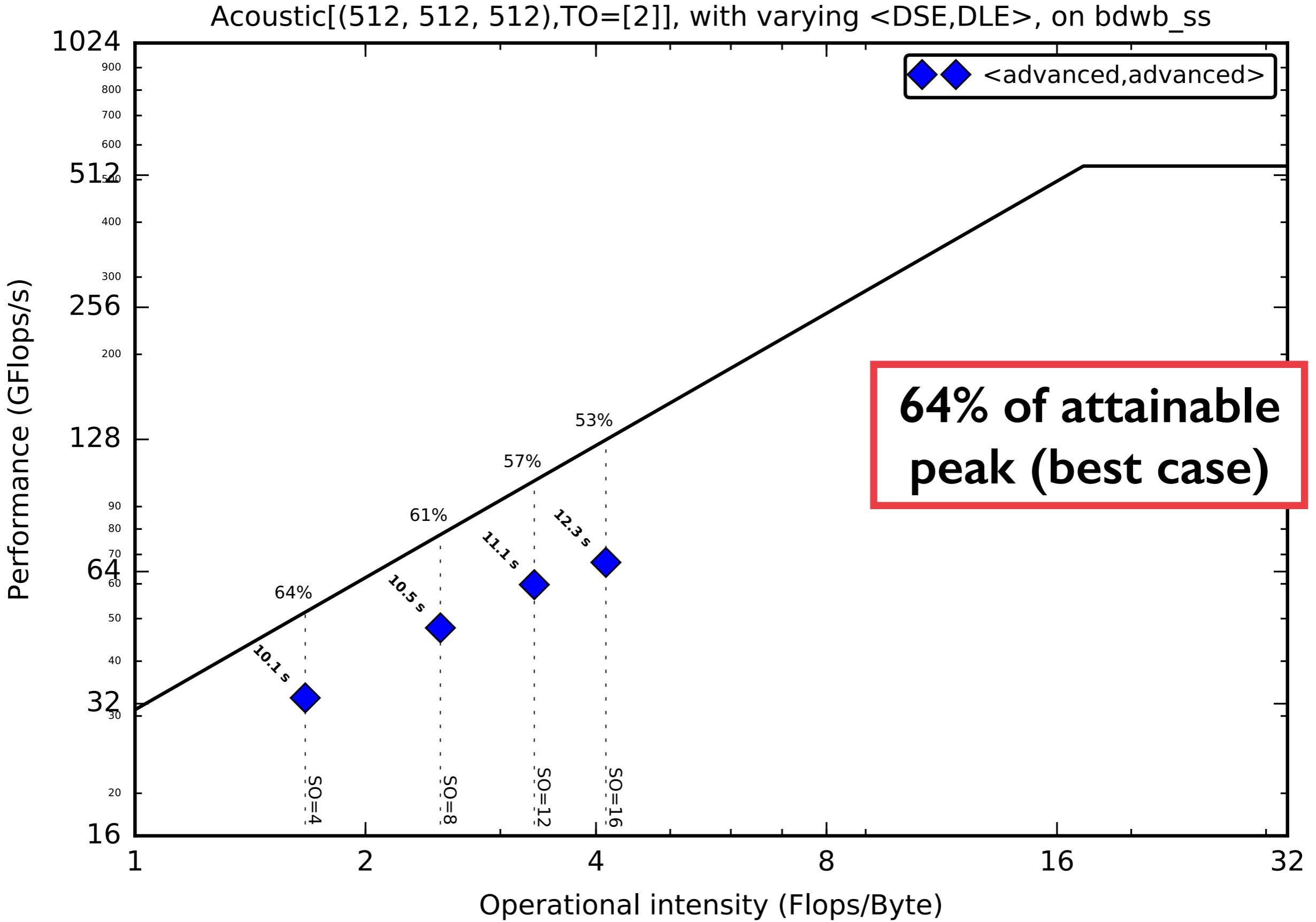
- Cache optimizations (mostly L1 cache)
  - Loop fission + elemental functions (register locality)
  - Padding + data alignment (split loads)



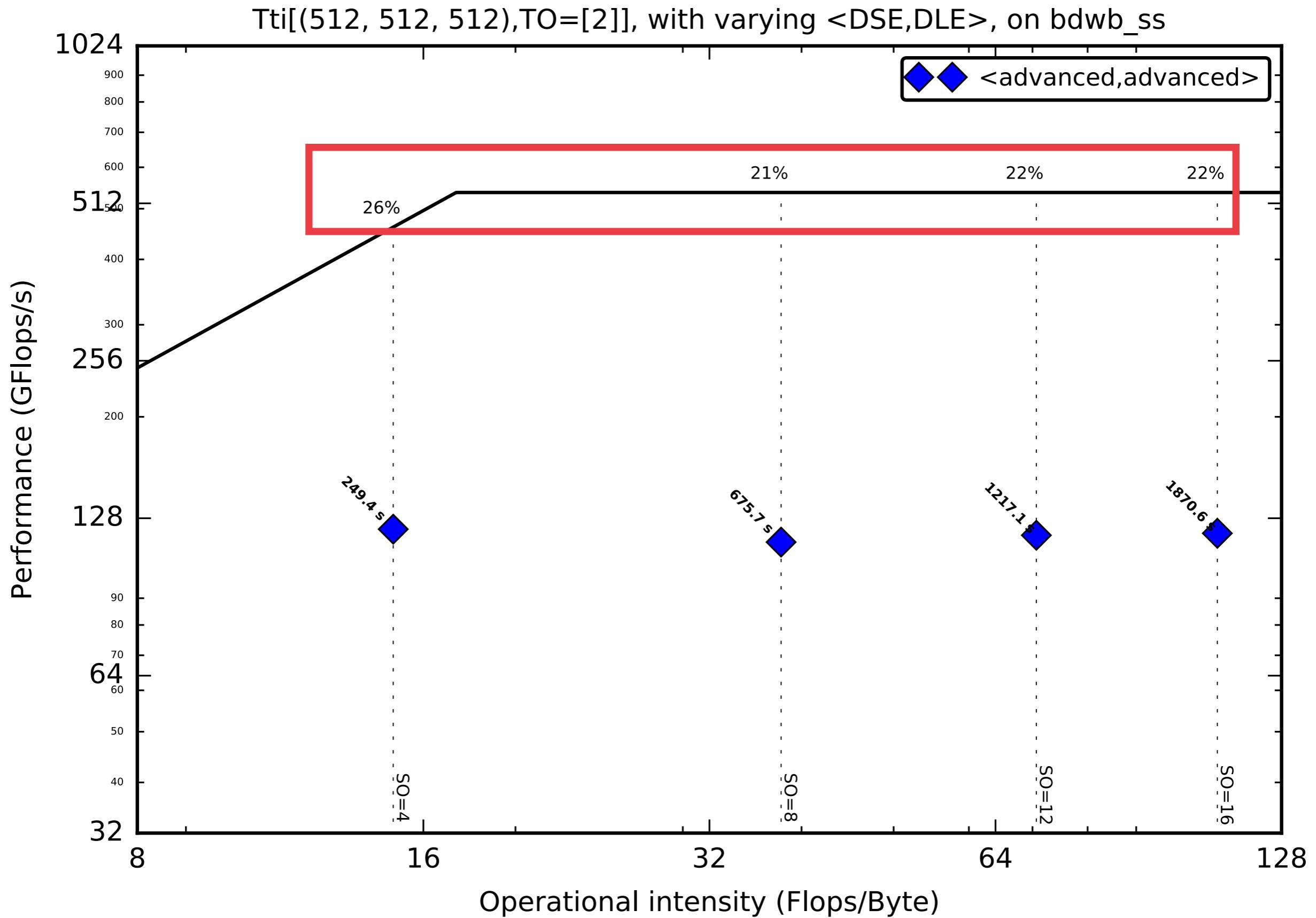
# Acoustic on Broadwell



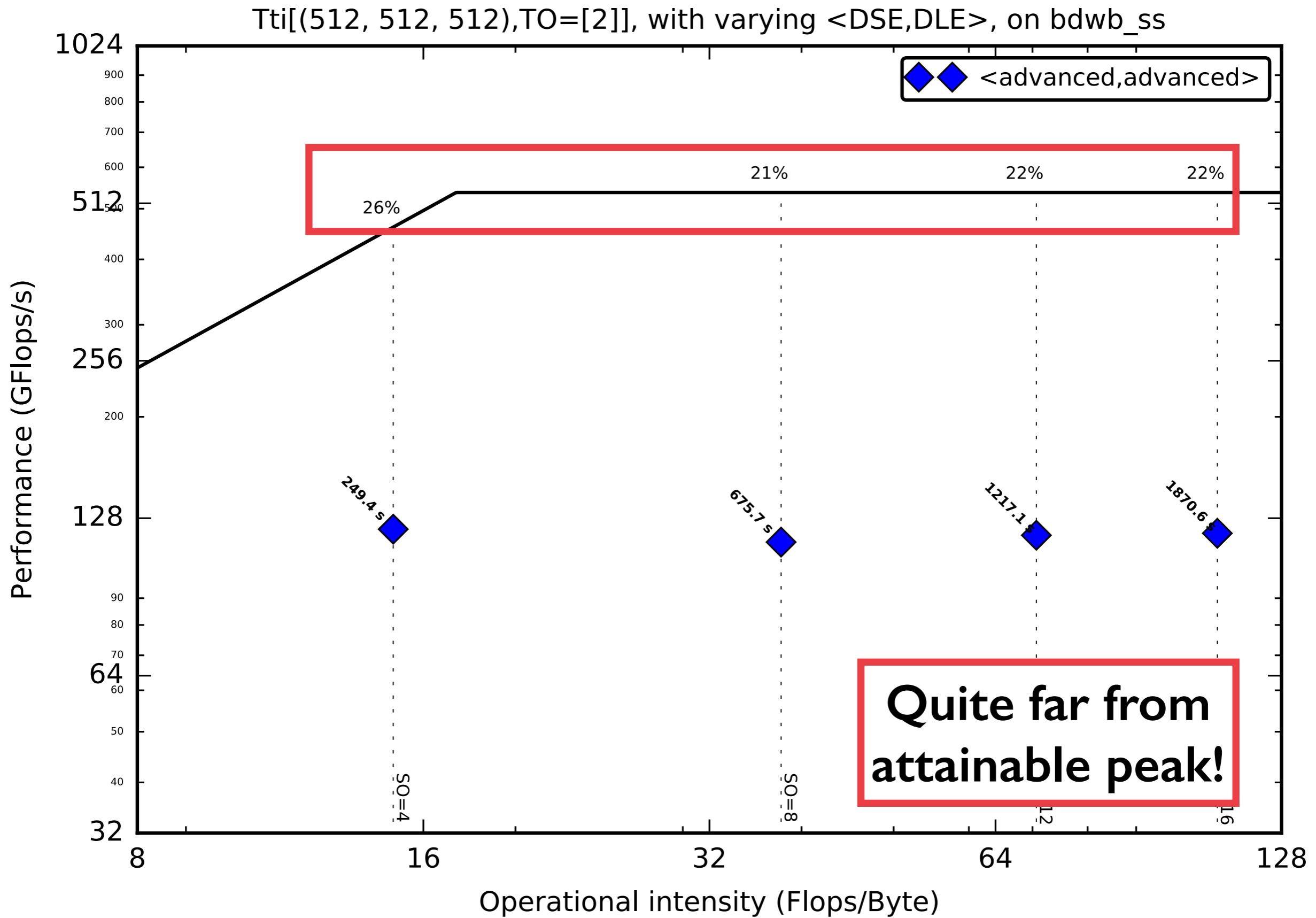
# Acoustic on Broadwell



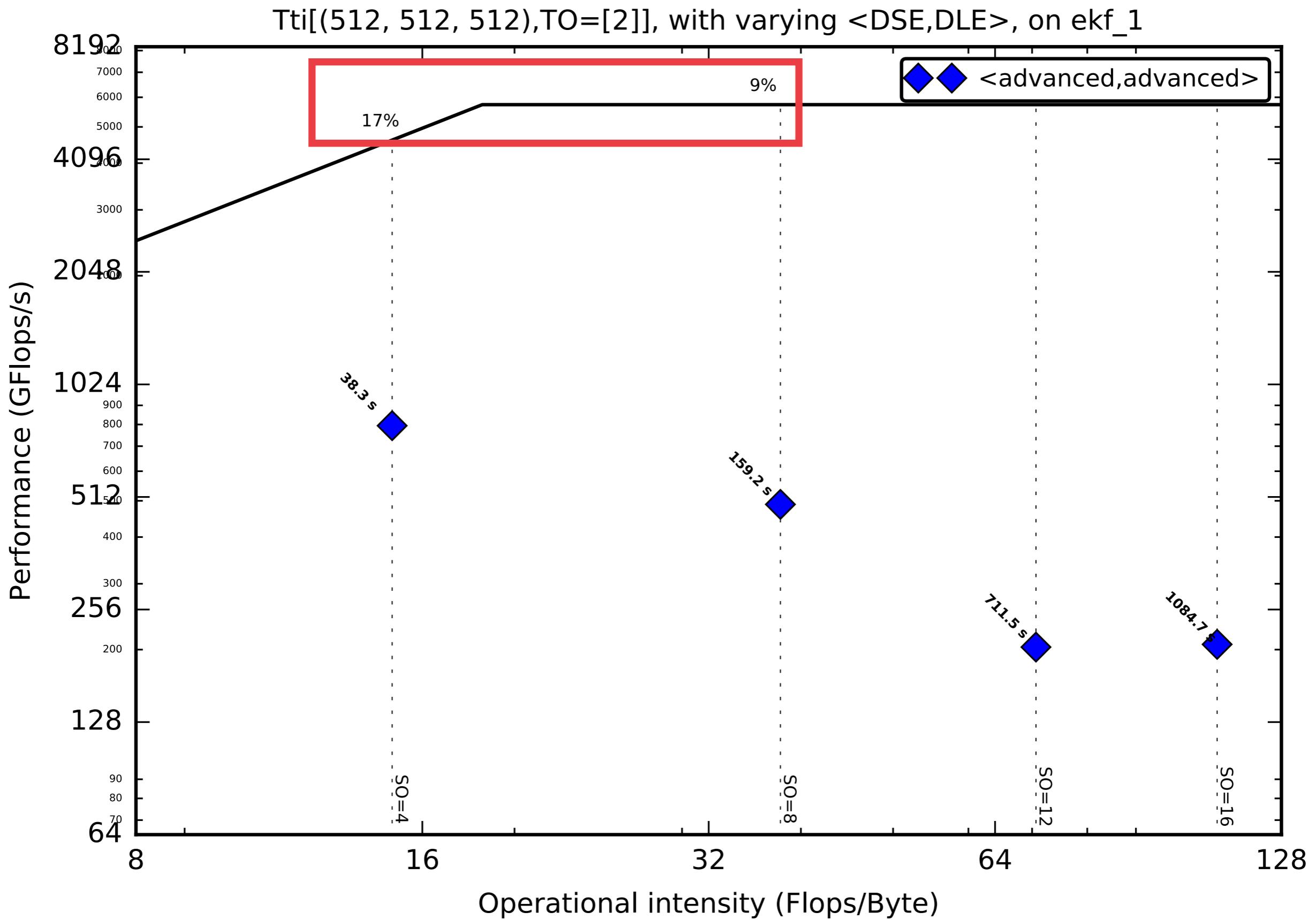
# TTI on Broadwell (8 threads, single socket)



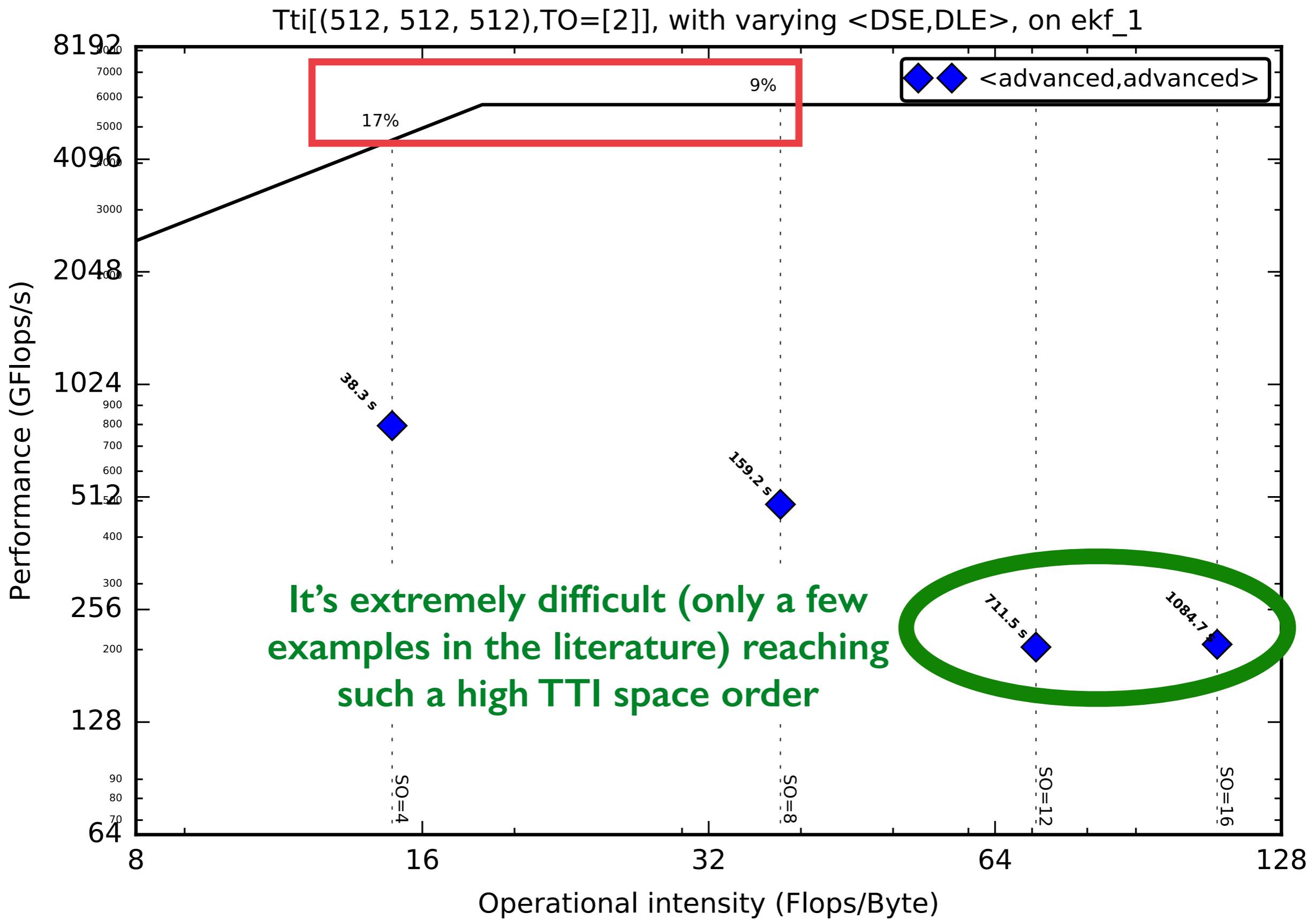
# TTI on Broadwell (8 threads, single socket)



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



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



# Where are we going

## PERFORMANCE

WIP: DSE aliases detection (“flops level”); DLE YASK integration; “domain tricks” (e.g., mixed data type arithmetic, skip computation over 0-valued regions)

## BIG RUNS

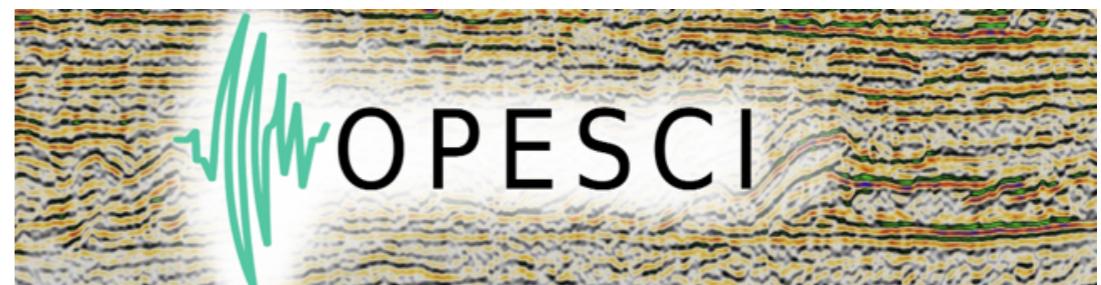
WIP: checkpointing, domain decomposition via MPI

# Conclusions and resources

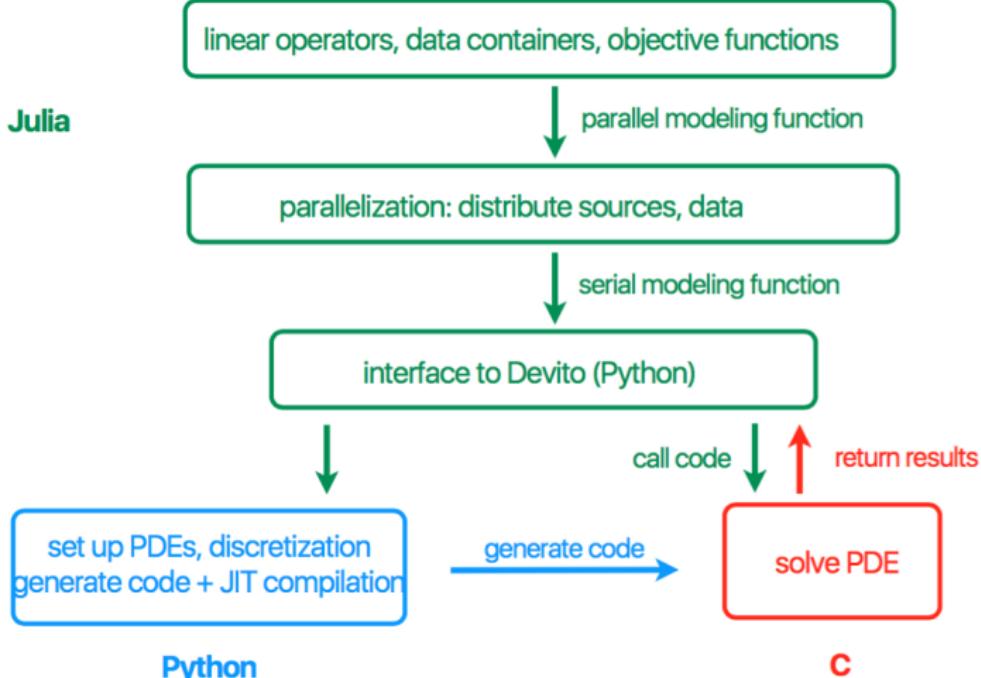
- Devito: an efficient and sustainable finite difference DSL
- Driven/inspired by **real-world seismic imaging**
- **Interdisciplinary research effort**
- Based on **actual compiler technology**

## Useful links

- <http://www.opesci.org>
- <https://github.com/opesci/devito>



# Vertical Integration



<sup>1</sup> Philipp Witte, Mathias Louboutin, and Felix J. Herrmann. Large-scale workflows for wave-equation based inversion in julia. In *Domain-Specific Abstractions for Full-Waveform Inversion at SIAM CSE*, 2017

# Verification

Verification of the generated code:

- Comparison with a reference implementation - IWAVE
- Adjoint test
  - For any  $x \in \text{span}(P_s A^T P_r^T)$ ,  $y \in \text{span}(P_r A^T P_s^T)$
  - $\langle P_r A^T P_s^T x, y \rangle - \langle x P_s A^T P_r^T y \rangle = 0$
  - 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  $dm$ ,  $\phi_s(m + hdm) = \phi_s(m) + \mathcal{O}(h)$  and  $\phi_s(m + hdm) = \phi_s(m) + h(J[m]^T \delta d) dm + \mathcal{O}(h^2)$
  - Passes at the level of the machine's accuracy
- Automatic formal code verification being implemented <sup>1</sup>

---

<sup>1</sup>Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. Many-core compiler fuzzing. In *ACM SIGPLAN Notices*, volume 50, pages 65–76. ACM, 2015