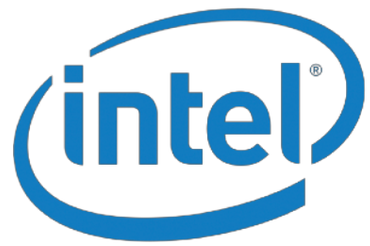

DEVITO

AUTOMATIC GENERATION OF PRODUCTION-GRADE HYBRID MPI-OPENMP PARALLEL WAVE PROPAGATORS USING DEVITO

F. Luporini[1], R. Nelson[1], M. Louboutin[2], N. Kukreja[1], G. Bisbas[1], P. Witte[2], Amik St-Cyr[5], C. Yount[3], T. Burgess[4], F. Herrmann[2], G. Gorman[1]

[1]Imperial College London
[2]Georgia Institute of Technology
[3]Intel Corporation
[4]DUG - DownUnder Geosolutions
[5]Shell



Devito: a DSL and compiler for explicit finite differences

- Embedded in Python – easy to learn.

- Devito is a compiler:
  - Generates optimized parallel C code: SIMD, OpenMP, MPI.

  - Support for multiple architectures.
    - Xeon and Xeon Phi (KNL) – fully supported.
      ARM64 – experimental.
      GPU's – in development.

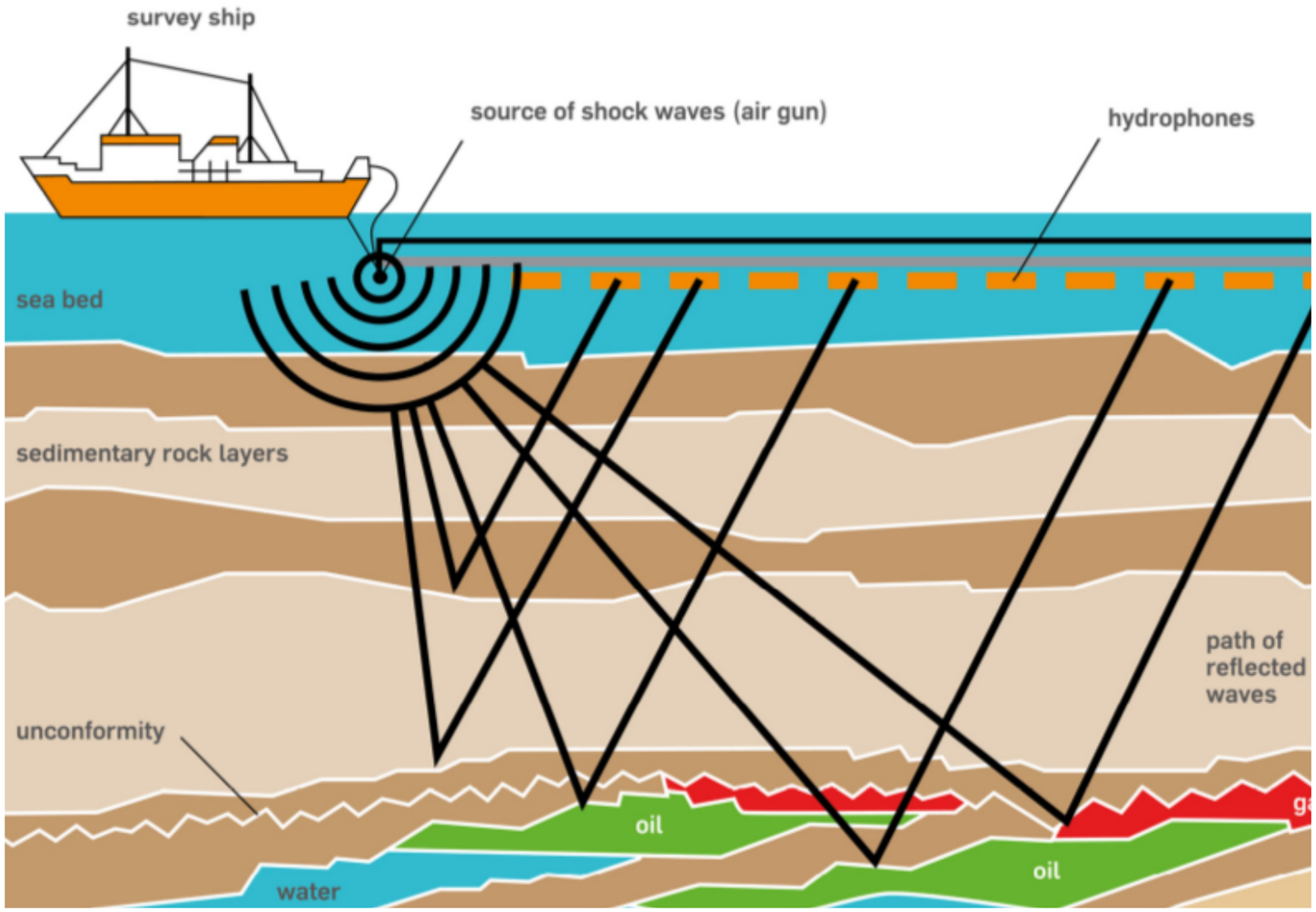
- Straightforward to integrate with existing codes in other languages.

- Open source platform – MIT license.

- **Used commercially – easy to play with, but not a toy...**

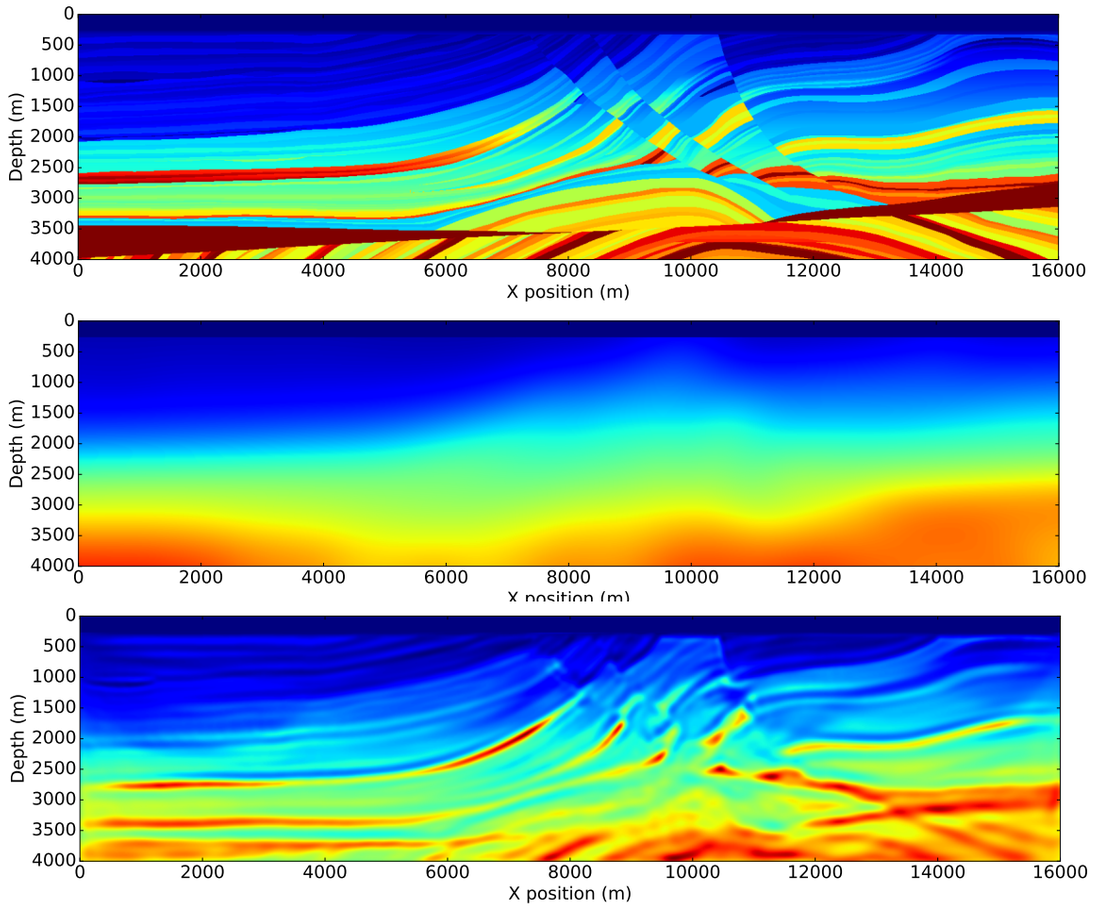
- Testing (continuous integration), code reviewing, examples, …

Overarching target application: inversion for seismic imaging



http://www.open.edu/openlearn/science-–maths-–technology/science/environmental-–science/earths-–physical-–resources-–petroleum/content-–section-–3.2.1

Real-life



Full-waveform inversion

Raising the level of abstraction

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

```
void kernel(…) {
  …
  <impenetrable code with aggressive
   performance optimizations>
  …
}
```

Raising the level of abstraction

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

```
void kernel(…) {
  …
  <impenetrable code with aggressive
   performance optimizations>
  …
}
```

Raising the level of abstraction

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

Raising the level of abstraction

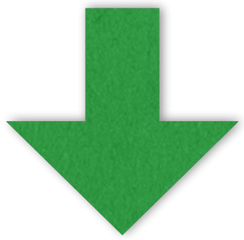
$$m\frac{\partial^2 u}{\partial t^2} + \eta\frac{\partial u}{\partial t} - \Delta u = 0$$

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

Raising the level of abstraction

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

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



```
void kernel(…) { … }
```

Raising the level of abstraction

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

Devito

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

```
void kernel(…) { … }
```

The code really needs to fly

Realistic full-waveform inversion (FWI) scenario

- **O(10³)** FLOPs per loop iteration or high memory pressure
- 3D grids with **>10⁹** grid points
- Often more than **3000** time steps
- Two operators: forward + adjoint, to be executed **~15** times
- Usually **30000** shots

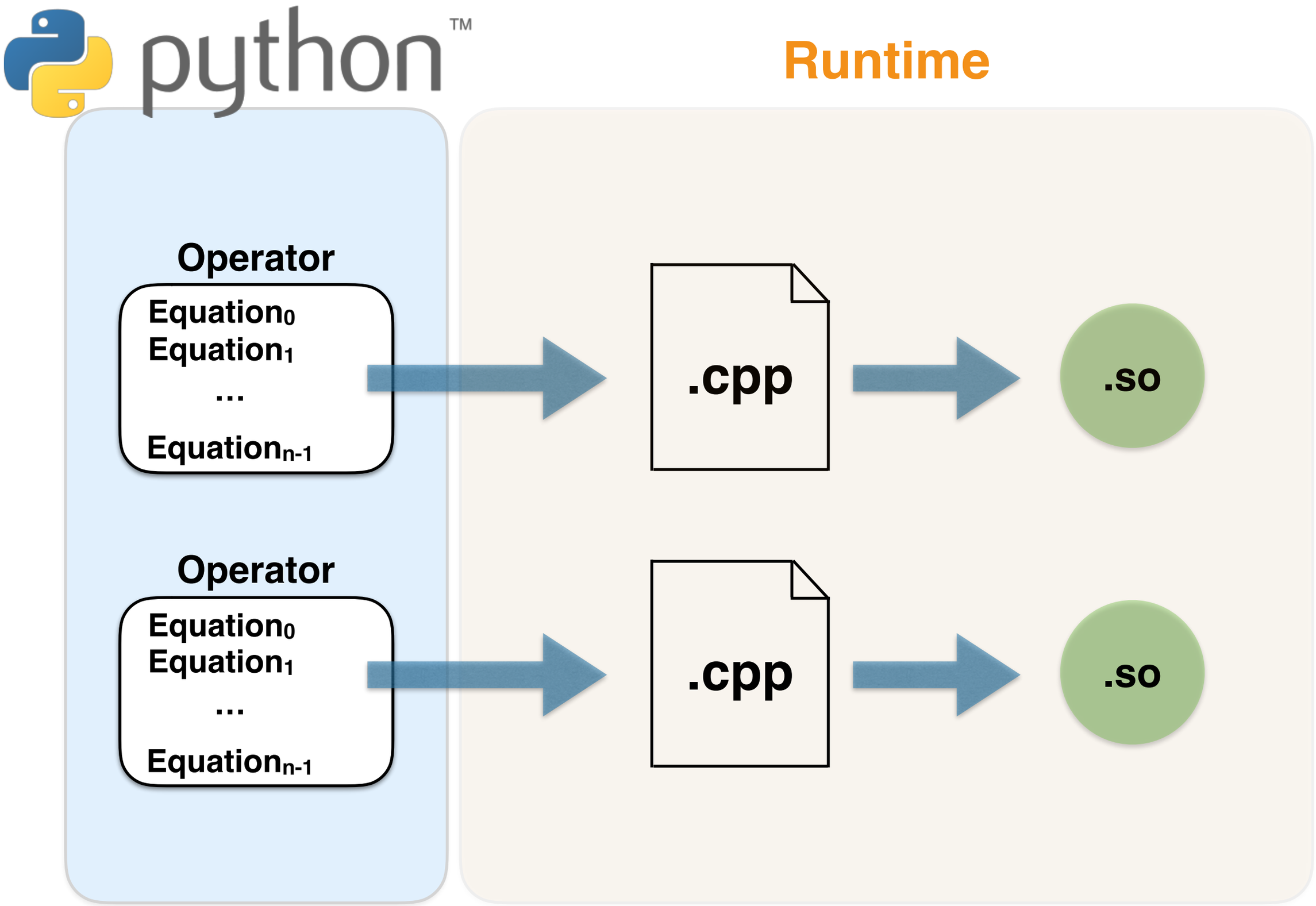
≈ **O(billions) TFLOPs**

Which means days, or weeks, or months on supercomputers!

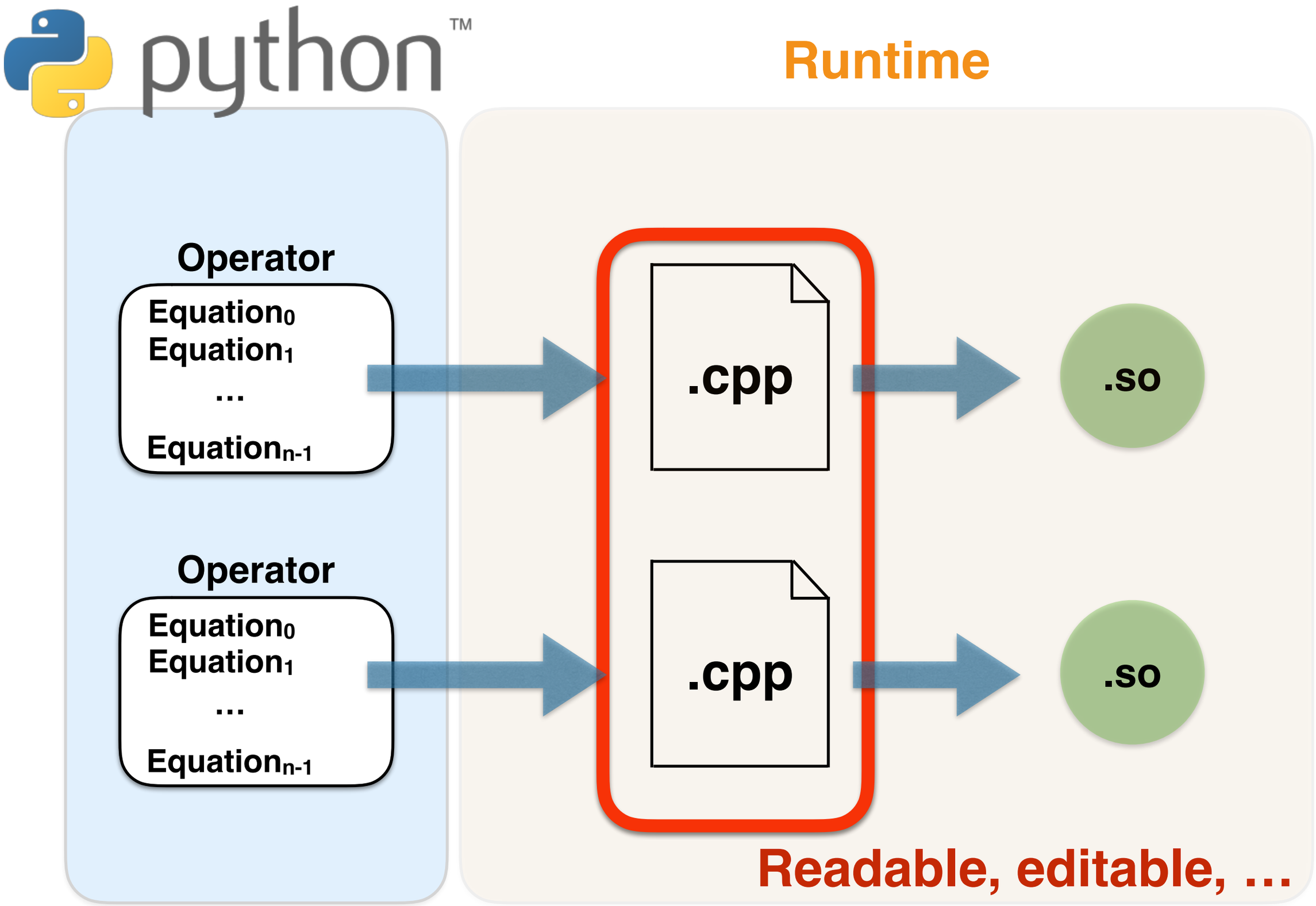
Lots of variations in physics, mathematics, platforms, …

- Many formulations of wave equations (R&D still super active)
- Many space and time discretizations
- Many types of boundary conditions in finite differences (too many)
- Unstructured computation (e.g., interpolation for sparse data)
- Proliferation of computer architectures (functional and performance portability)
- …

So, what does it look like?



So, what does it look like?



Production-Grade Hybrid MPI-OpenMP Parallel Wave Propagators using Devito — the key steps

Production-Grade Hybrid MPI-OpenMP Parallel Wave Propagators using Devito — the key steps

Create a grid

```
>>> grid = Grid(shape=(…), extent=(…))
```

Production-Grade Hybrid MPI-OpenMP Parallel Wave Propagators using Devito — the key steps

Create a grid

```
>>> grid = Grid(shape=(…), extent=(…))
```

Define the functions present in the PDE

```
>>> m = Function(name="m", grid=grid, space_order=12)
>>> u = TimeFunction(name=“wavefield”, grid=grid, …)
```

Production-Grade Hybrid MPI-OpenMP Parallel Wave Propagators using Devito — the key steps

Create a grid

```
>>> grid = Grid(shape=(…), extent=(…))
```

Define the functions present in the PDE

```
>>> m = Function(name="m", grid=grid, space_order=12)
>>> u = TimeFunction(name=“wavefield”, grid=grid, …)
```

Write out the PDE

```
>>> pde = m*u.dt2 - u.laplace
>>> eqn = Eq(u.forward, solve(pde, u.forward))
```

Production-Grade Hybrid MPI-OpenMP Parallel Wave Propagators using Devito — the key steps

Create a grid

```
>>> grid = Grid(shape=(…), extent=(…))
```

Define the functions present in the PDE

```
>>> m = Function(name="m", grid=grid, space_order=12)
>>> u = TimeFunction(name="wavefield", grid=grid, …)
```

Write out the PDE

```
>>> pde = m*u.dt2 - u.laplace
>>> eqn = Eq(u.forward, solve(pde, u.forward))
```

Write out the boundary conditions (or use Absorbing layers, or…)

```
>>> … # "Just" other Eqs or Functions
```

Production-Grade Hybrid MPI-OpenMP Parallel Wave Propagators using Devito — the key steps

Production-Grade Hybrid MPI-OpenMP Parallel Wave Propagators using Devito — the key steps

Define the source/receivers to inject/record seismic waves

```
>>> src = SparseTimeFunction(name="source", grid=grid, …)
>>> rec = SparseTimeFunction(name="receivers", grid=grid, …)
```

Production-Grade Hybrid MPI-OpenMP Parallel Wave Propagators using Devito — the key steps

Define the source/receivers to inject/record seismic waves

```
>>> src = SparseTimeFunction(name="source", grid=grid, …)
>>> rec = SparseTimeFunction(name="receivers", grid=grid, …)
```

Create an Operator

```
>>> op = Operator(eqn, bcs, src.inject(…), rec.interpolate(…))
```

Production-Grade Hybrid MPI-OpenMP Parallel Wave Propagators using Devito — the key steps

Define the source/receivers to inject/record seismic waves

```
>>> src = SparseTimeFunction(name="source", grid=grid, …)
>>> rec = SparseTimeFunction(name="receivers", grid=grid, …)
```

Create an Operator

```
>>> op = Operator(eqn, bcs, src.inject(…), rec.interpolate(…))
```

This will generate OpenMP + MPI code !

Production-Grade Hybrid MPI-OpenMP Parallel Wave Propagators using Devito — the key steps

Define the source/receivers to inject/record seismic waves

```
>>> src = SparseTimeFunction(name="source", grid=grid, …)
>>> rec = SparseTimeFunction(name="receivers", grid=grid, …)
```

Create an Operator

```
>>> op = Operator(eqn, bcs, src.inject(…), rec.interpolate(…))
```

This will generate OpenMP + MPI code !

If you want (…), run the Operator

```
>>> op.apply(…)
```

Flexibility in space/time discretization

Define the functions used in the PDE

```
>>> m = Function(name="m", grid=grid, space_order=so)
>>> u = TimeFunction(name="wavefield", grid=grid, space_order=so)
```

so=4

```
  for (int time = time_m, t0 = (time)%(3), t1 = (time + 1)%(3), t2 = (time + 2)%(3);
time <= time_M; time += 1, t0 = (time)%(3), t1 = (time + 1)%(3), t2 = (time + 2)%(3))
    for (int x = x_m; x <= x_M; x += 1)
      for (int y = y_m; y <= y_M; y += 1)
        for (int z = z_m; z <= z_M; z += 1)
          u[t1][x + 4][y + 4][z + 4] = 2*pow(dt, 3)*(-2.08333333333333e-4F*u[t0][x + 2]
[y + 4][z + 4] + 3.33333333333333e-3F*u[t0][x + 3][y + 4][z + 4] -
2.08333333333333e-4F*u[t0][x + 4][y + 2][z + 4] + 3.33333333333333e-3F*u[t0][x + 4][y +
3][z + 4] - 2.08333333333333e-4F*u[t0][x + 4][y + 4][z + 2] + 3.33333333333333e-3F*u[t0]
[x + 4][y + 4][z + 3] - 1.875e-2F*u[t0][x + 4][y + 4][z + 4] +
3.33333333333333e-3F*u[t0][x + 4][y + 4][z + 5] - 2.08333333333333e-4F*u[t0][x + 4][y +
4][z + 6] + 3.33333333333333e-3F*u[t0][x + 4][y + 5][z + 4] - 2.08333333333333e-4F*u[t0]
[x + 4][y + 6][z + 4] + 3.33333333333333e-3F*u[t0][x + 5][y + 4][z + 4] -
2.08333333333333e-4F*u[t0][x + 6][y + 4][z + 4])/(pow(dt, 2)*damp[x + 1][y + 1][z + 1] +
2*dt*m[x + 4][y + 4][z + 4]) + pow(dt, 2)*damp[x + 1][y + 1][z + 1]*u[t2][x + 4][y + 4]
[z + 4]/(pow(dt, 2)*damp[x + 1][y + 1][z + 1] + 2*dt*m[x + 4][y + 4][z + 4]) + 4*dt*m[x
+ 4][y + 4][z + 4]*u[t0][x + 4][y + 4][z + 4]/(pow(dt, 2)*damp[x + 1][y + 1][z + 1] +
2*dt*m[x + 4][y + 4][z + 4]) - 2*dt*m[x + 4][y + 4][z + 4]*u[t2][x + 4][y + 4][z + 4]/
(pow(dt, 2)*damp[x + 1][y + 1][z + 1] + 2*dt*m[x + 4][y + 4][z + 4]);
```

so=12

```
  for (int time = time_m, t0 = (time)%(3), t1 = (time + 1)%(3), t2 = (time + 2)%(3);
time <= time_M; time += 1, t0 = (time)%(3), t1 = (time + 1)%(3), t2 = (time + 2)%(3))
    for (int x = x_m; x <= x_M; x += 1)
      for (int y = y_m; y <= y_M; y += 1)
        for (int z = z_m; z <= z_M; z += 1)
          u[t1][x + 12][y + 12][z + 12] = 2*pow(dt, 3)*(-1.5031265031265e-7F*u[t0][x +
6][y + 12][z + 12] + 2.5974025974026e-6F*u[t0][x + 7][y + 12][z + 12] -
2.23214285714286e-5F*u[t0][x + 8][y + 12][z + 12] + 1.32275132275132e-4F*u[t0][x + 9][y
+ 12][z + 12] - 6.69642857142857e-4F*u[t0][x + 10][y + 12][z + 12] +
4.28571428571429e-3F*u[t0][x + 11][y + 12][z + 12] - 1.5031265031265e-7F*u[t0][x + 12]
[y + 6][z + 12] + 2.5974025974026e-6F*u[t0][x + 12][y + 7][z + 12] -
2.23214285714286e-5F*u[t0][x + 12][y + 8][z + 12] + 1.32275132275132e-4F*u[t0][x + 12]
[y + 9][z + 12] - 6.69642857142857e-4F*u[t0][x + 12][y + 10][z + 12] +
4.28571428571429e-3F*u[t0][x + 12][y + 11][z + 12] - 1.5031265031265e-7F*u[t0][x + 12]
[y + 12][z + 6] + 2.5974025974026e-6F*u[t0][x + 12][y + 12][z + 7] -
2.23214285714286e-5F*u[t0][x + 12][y + 12][z + 8] + 1.32275132275132e-4F*u[t0][x + 12]
[y + 12][z + 9] - 6.69642857142857e-4F*u[t0][x + 12][y + 12][z + 10] +
4.28571428571429e-3F*u[t0][x + 12][y + 12][z + 11] - 2.23708333333333e-2F*u[t0][x + 12]
[y + 12][z + 12] + 4.28571428571429e-3F*u[t0][x + 12][y + 12][z + 13] -
6.69642857142857e-4F*u[t0][x + 12][y + 12][z + 14] + 1.32275132275132e-4F*u[t0][x + 12]
[y + 12][z + 15] - 2.23214285714286e-5F*u[t0][x + 12][y + 12][z + 16] +
2.5974025974026e-6F*u[t0][x + 12][y + 12][z + 17] - 1.5031265031265e-7F*u[t0][x + 12][y
+ 12][z + 18] + 4.28571428571429e-3F*u[t0][x + 12][y + 13][z + 12] -
6.69642857142857e-4F*u[t0][x + 12][y + 14][z + 12] + 1.32275132275132e-4F*u[t0][x + 12]
[y + 15][z + 12] - 2.23214285714286e-5F*u[t0][x + 12][y + 16][z + 12] +
2.5974025974026e-6F*u[t0][x + 12][y + 17][z + 12] - 1.5031265031265e-7F*u[t0][x + 12][y
+ 18][z + 12] + 4.28571428571429e-3F*u[t0][x + 13][y + 12][z + 12] -
6.69642857142857e-4F*u[t0][x + 14][y + 12][z + 12] + 1.32275132275132e-4F*u[t0][x + 15]
[y + 12][z + 12] - 2.23214285714286e-5F*u[t0][x + 16][y + 12][z + 12] +
2.5974025974026e-6F*u[t0][x + 17][y + 12][z + 12] - 1.5031265031265e-7F*u[t0][x + 18][y
+ 12][z + 12])/(pow(dt, 2)*damp[x + 1][y + 1][z + 1] + 2*dt*m[x + 12][y + 12][z + 12])
+ pow(dt, 2)*damp[x + 1][y + 1][z + 1]*u[t2][x + 12][y + 12][z + 12]/(pow(dt, 2)*damp[x
+ 1][y + 1][z + 1] + 2*dt*m[x + 12][y + 12][z + 12]) + 4*dt*m[x + 12][y + 12][z +
12]*u[t0][x + 12][y + 12][z + 12]/(pow(dt, 2)*damp[x + 1][y + 1][z + 1] + 2*dt*m[x +
12][y + 12][z + 12]) - 2*dt*m[x + 12][y + 12][z + 12]*u[t2][x + 12][y + 12][z + 12]/
(pow(dt, 2)*damp[x + 1][y + 1][z + 1] + 2*dt*m[x + 12][y + 12][z + 12]);
```

Data dependence analysis

- Represents data (array) accesses as (labelled) vectors in $\mathbb{Z}^n$

$$u[t+2, x-3] \rightarrow \begin{bmatrix} t+2 \\ x-3 \end{bmatrix}$$

- Currently uses a simple in-house framework based on Lamport theory
- The Devito compiler relies on data dependence analysis for many tasks
  - Inferring the iteration direction (`i++` or `i--` ?)
  - Loop scheduling (convert list of equations into a tree of loops)
  - Loop optimizations (e.g., discovery of parallel loops)
  - MPI optimizations (e.g., reshuffling/merging communications)
- Might one day move to an external tool (ISL?) if all of our use cases are supported

Example: inferring the propagation direction

```
>>> pde = …
>>> eqn = Eq(u.forward, solve(pde, u.forward))
```

```
u[t+1, …] = f(u[t, …], u[t-1, …], …)
```

versus

```
>>> pde = …
>>> eqn = Eq(u.backward, solve(pde, u.backward))
```

```
u[t-1, …] = f(u[t, …], u[t+1, …], …)
```

In the first case, the t loop iterates forward (t++),
in the latter case it iterates backwards (t--)

Code generation for Higdon BCs is based on this exact framework

Example: topological sorting for maximal “loop fusion”

```
u[t+1,x] = F0(u[t,x], u[t-1,x], v[t,x], …)

u[t+1,s] = F1(u[t+1,u_coords[s]], …)

v[t+1,x] = F2(v[t,x], v[t-1,x], u[t,x], …)

v[t+1,s] = F3(u[t+1,v_coords[s]], …)
```

Example: topological sorting for maximal “loop fusion”

```
u[t+1,x] = F0(u[t,x], u[t-1,x], v[t,x], …)

u[t+1,s] = F1(u[t+1,u_coords[s]], …)

v[t+1,x] = F2(v[t,x], v[t-1,x], u[t,x], …)

v[t+1,s] = F3(u[t+1,v_coords[s]], …)
```

Only flow-dependences in the time (t) dimension!

Example: topological sorting for maximal “loop fusion”

```
u[t+1,x] = F0(u[t,x], u[t-1,x], v[t,x], …)

u[t+1,s] = F1(u[t+1,u_coords[s]], …)
```

```
v[t+1,x] = F2(v[t,x], v[t-1,x], u[t,x], …)

v[t+1,s] = F3(u[t+1,v_coords[s]], …)
```

Only flow-dependences in the time (t) dimension!

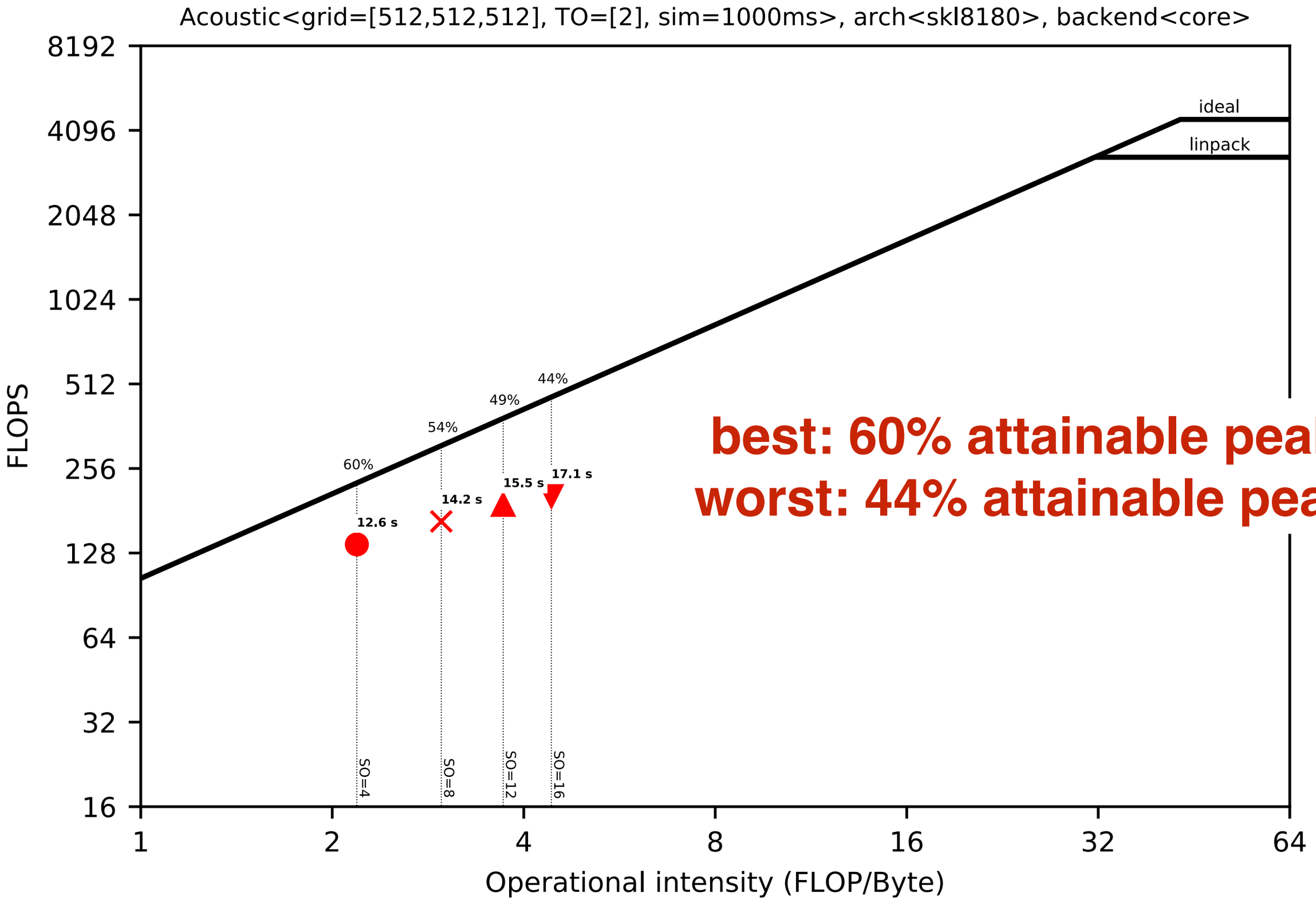
Example: topological sorting for maximal “loop fusion”

```
u[t+1,x] = F0(u[t,x], u[t-1,x], v[t,x], …)

v[t+1,x] = F2(v[t,x], v[t-1,x], u[t,x], …)

u[t+1,s] = F1(u[t+1,u_coords[s]], …)

v[t+1,s] = F3(u[t+1,v_coords[s]], …)
```

Example: topological sorting for maximal “loop fusion”

```
for t = t_m to t_M, t++
  for x = x_m to x_M, x++
    u[t+1,x] = F0(u[t,x], u[t-1,x], v[t,x], …)
    v[t+1,x] = F2(v[t,x], v[t-1,x], u[t,x], …)
for s = s_m to s_M, s++
  u[t+1,s] = F1(u[t+1,u_coords[s]], …)
  v[t+1,s] = F3(u[t+1,v_coords[s]], …)
```

Single-socket — Isotropic acoustic on Skylake 8180



best: 60% attainable peak
worst: 44% attainable peak

Generalized Common Sub-expressions Elimination

```
a = sin(phi[i,j]) + sin(phi[i-1,j-1]) + sin(phi[i+2,j+2])
```

Observations:

Same operators (`sin`), same operands (`phi`), same indices (`i, j`)

Linearly dependent index vectors (`[i, j], [i-1, j-1], [i+2, j+2]`)

Generalized Common Sub-expressions Elimination

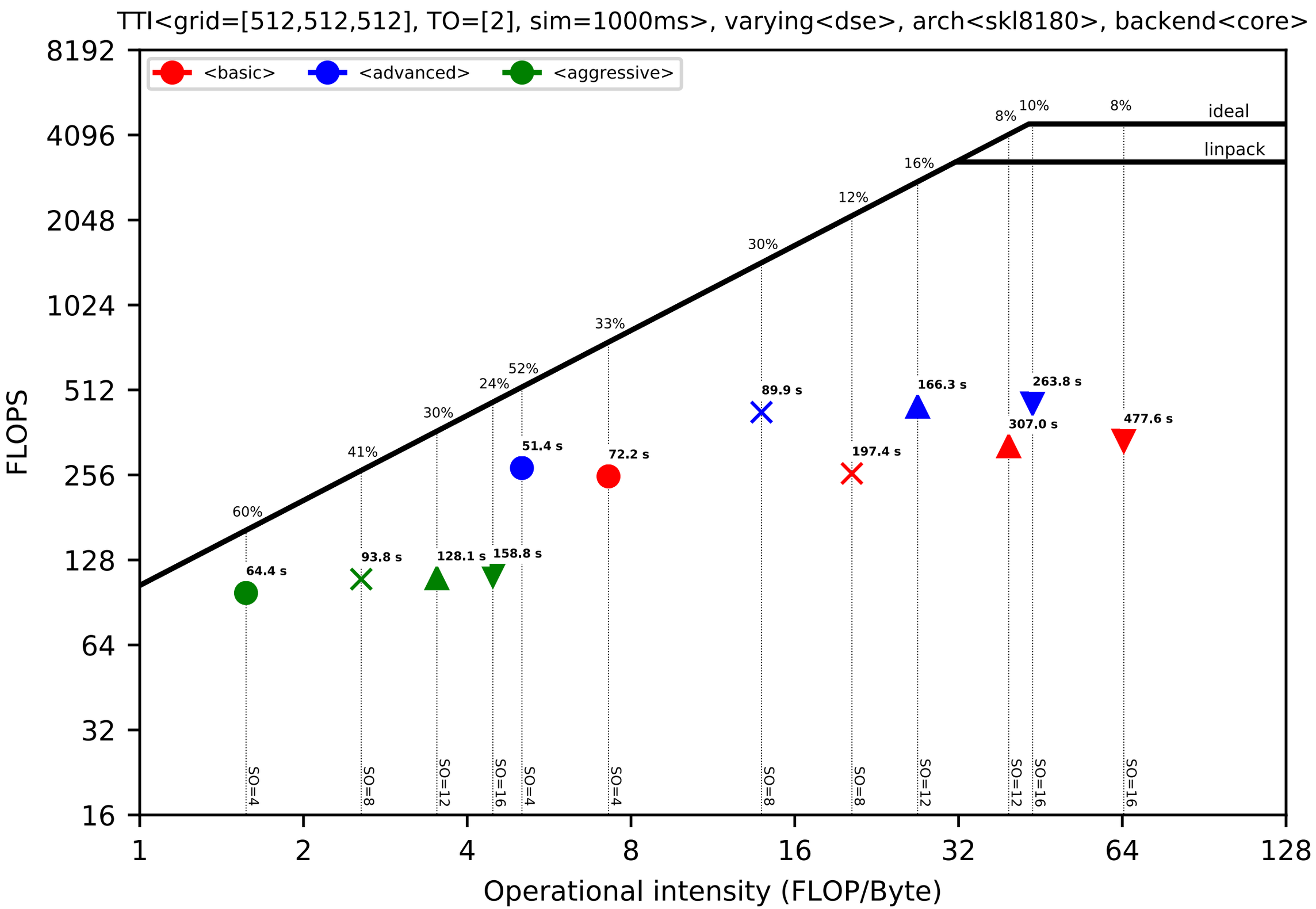
```
a = sin(phi[i,j]) + sin(phi[i-1,j-1]) + sin(phi[i+2,j+2])
```

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

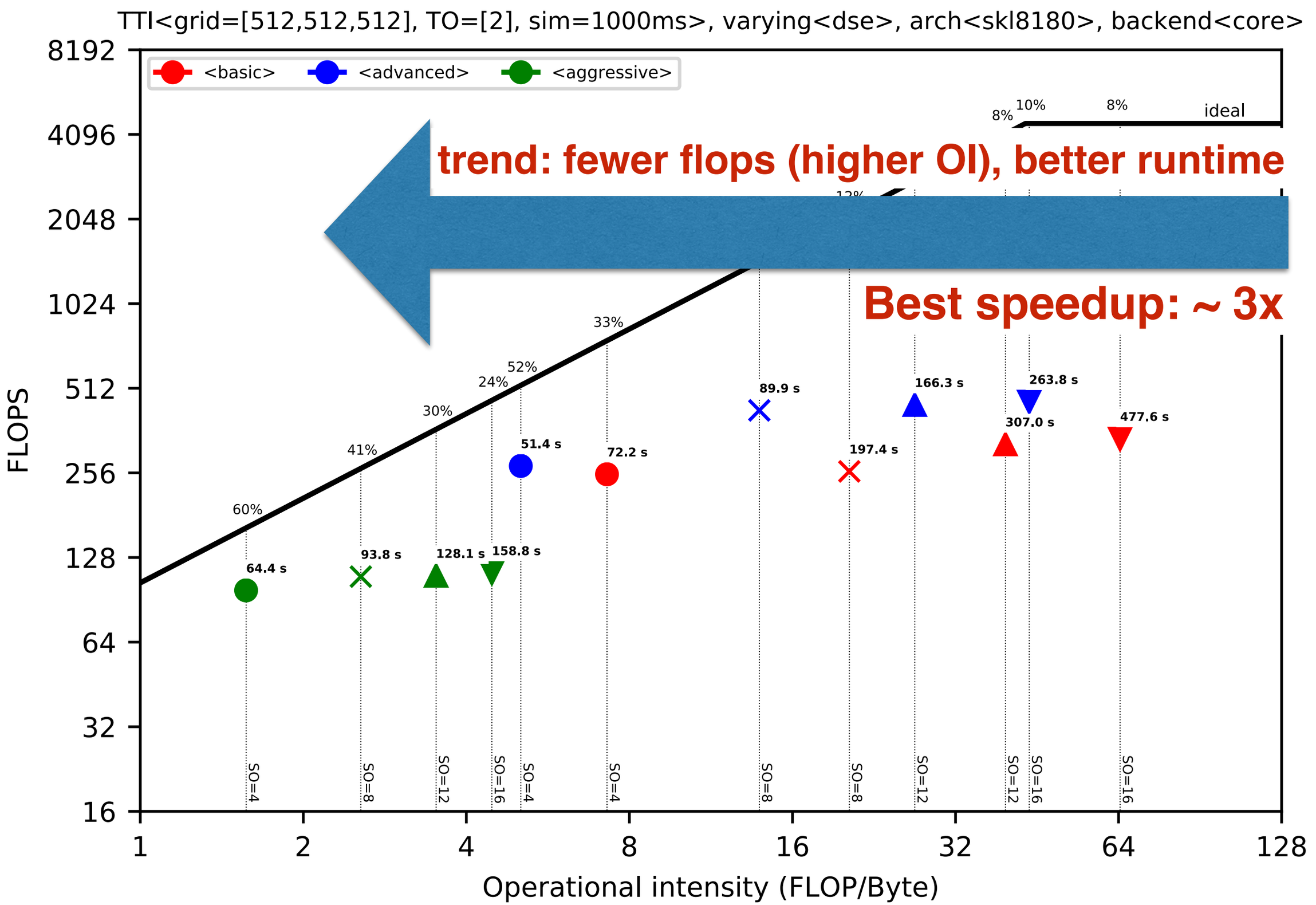
```
B[i,j] = sin(phi[i,j])

a = B[i,j] + B[i-1,j-1] + B[i+2,j+2]
```

Trade-off FLOPs/storage

Single-socket — TTI on Skylake 8180



Single-socket — TTI on Skylake 8180



MPI support — for free

```
mpirun <mpi args> python app.py
```

And that's it.

MPI support — for free

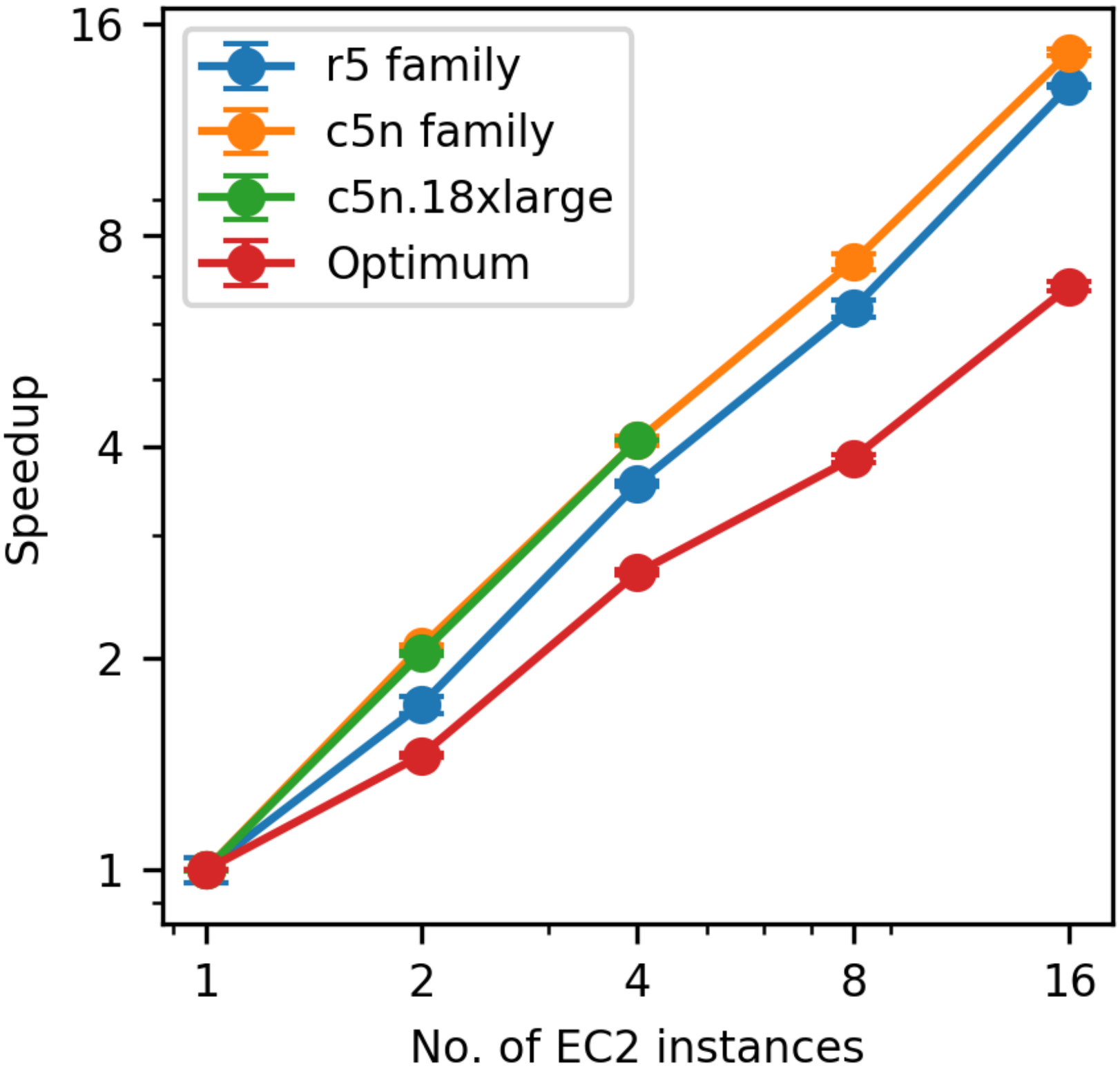
Python-level features:

- Domain decomposition based on MPI Cartesian grid abstraction.
- New package *numpy4mpi*: NumPy arrays automatically split and distributed according to domain decomposition.
  - Parallel data slicing dealt with efficiently under the hood
- Sources/receivers (i.e. "sparse data") distributed automatically.

C-level features:

- Generated code contains the MPI halo exchanges. Required halo exchanges identified through data dependence analysis.
- Optimizations (e.g., reshuffling halo updates, computation/communication overlap) also exploit data dependence analysis.
- Data packing/unpacking is threaded for performance.

Cluster/cloud — TTI strong scaling on AWS and Optimum



These are best-case scenario experiments: one process/thread per **node** (i.e., the whole bandwidth at full disposal)

Conclusions

- Devito is an open-source high-productivity and high-performance Python framework for finite-differences.
- Driven by real-world seismic imaging, inspired by projects such as FEniCS/Firedrake.
- Based on actual compiler technology.
- Interdisciplinary, interinstitutional, international research effort.
- Growing community and user base
  - Academic: e.g. used by SLIM team to develop JUDI, […]
  - Commercial: Production code by DUG, […]
  - Many other R&D teams […]

Imperial College London

EPSRC Engineering and Physical Sciences Research Council

Georgia Tech

dug

intel

Useful links and resources

Website: http://www.devitoproject.org

GitHub: https://github.com/opesci/devito

Slack: https://opesci-slackin.now.sh

Appendix

Experimentation details

- Architectures
  - Intel® Xeon® Platinum 8180 Processor ("Skylake", 28 cores)
  - Intel® XeonPhi® 7250 (68 cores)
    - Quadrant mode (still no support for NUMA)
    - Tried 1, 2, 4 threads per core. Shown best.

- Compiler
  - ICC 18 -xHost -O3
  - -xMIC-AVX512 on Xeon Phi
  - -qopt-zmm-usage=high on Skylake

- Runs
  - Single socket
  - Pinning via Numactl
  - On the XeonPhi®, data fits in MCDRAM

- Roofline calculations:
  - Memory bandwidth: STREAM
  - CPU peak: pen & paper
  - Operational intensity: source-level analysis (automated through Devito)

Experimentation details

- Experiments on AWS and Optimum

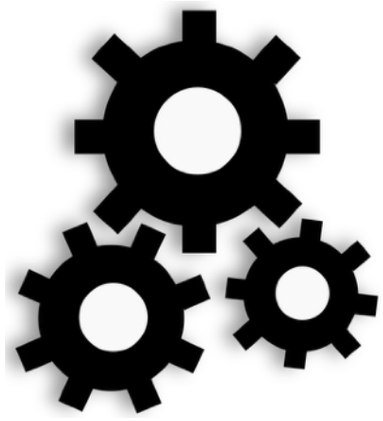
Instance	Architecture	(v)CPUs	Cores/s
m4.4xlarge	Intel Xeon E5-2686 v4 @ 2.30GHz	16	8
r5.12xlarge	Intel Xeon Platinum 8175M @ 2.50 GHz	48	24
r5.24xlarge	Intel Xeon Platinum 8175M @ 2.50 GHz	96	24
c5n.9xlarge	Intel Xeon Platinum 8124M @ 3.00 GHz	36	18
c5n.12xlarge	Intel Xeon Platinum 8142M @ 3.00 GHz	72	18
r5.metal	Intel Xeon Platinum 8175M @ 2.50 GHz	96	24
Optimum	Intel Xeon E5-2680 v2 @ 2.80GHz	20	10

What Devito does **NOT** (and probably will never) do

- Parallel I/O
- (Obviously) the overarching application (though we provide examples)
- Implicit solvers
- Many other things

Many optimizations BEFORE creating a tree of loops…

- "Loop fusion" => Equation clustering
- "Loop-invariant code motion" => Cluster lifting
- Common sub-expressions elimination (both cross- and intra-iteration), factorization, … they all happen at the cluster level

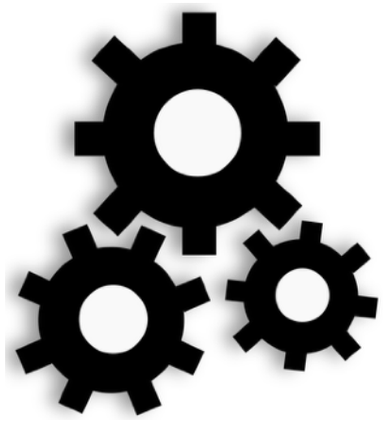


Many optimizations BEFORE creating a tree of loops…

- "Loop fusion" => Equation clustering
- "Loop-invariant code motion" => Cluster lifting
- Common sub-expressions elimination (both cross- and intra-iteration), factorization, … they all happen at the cluster level

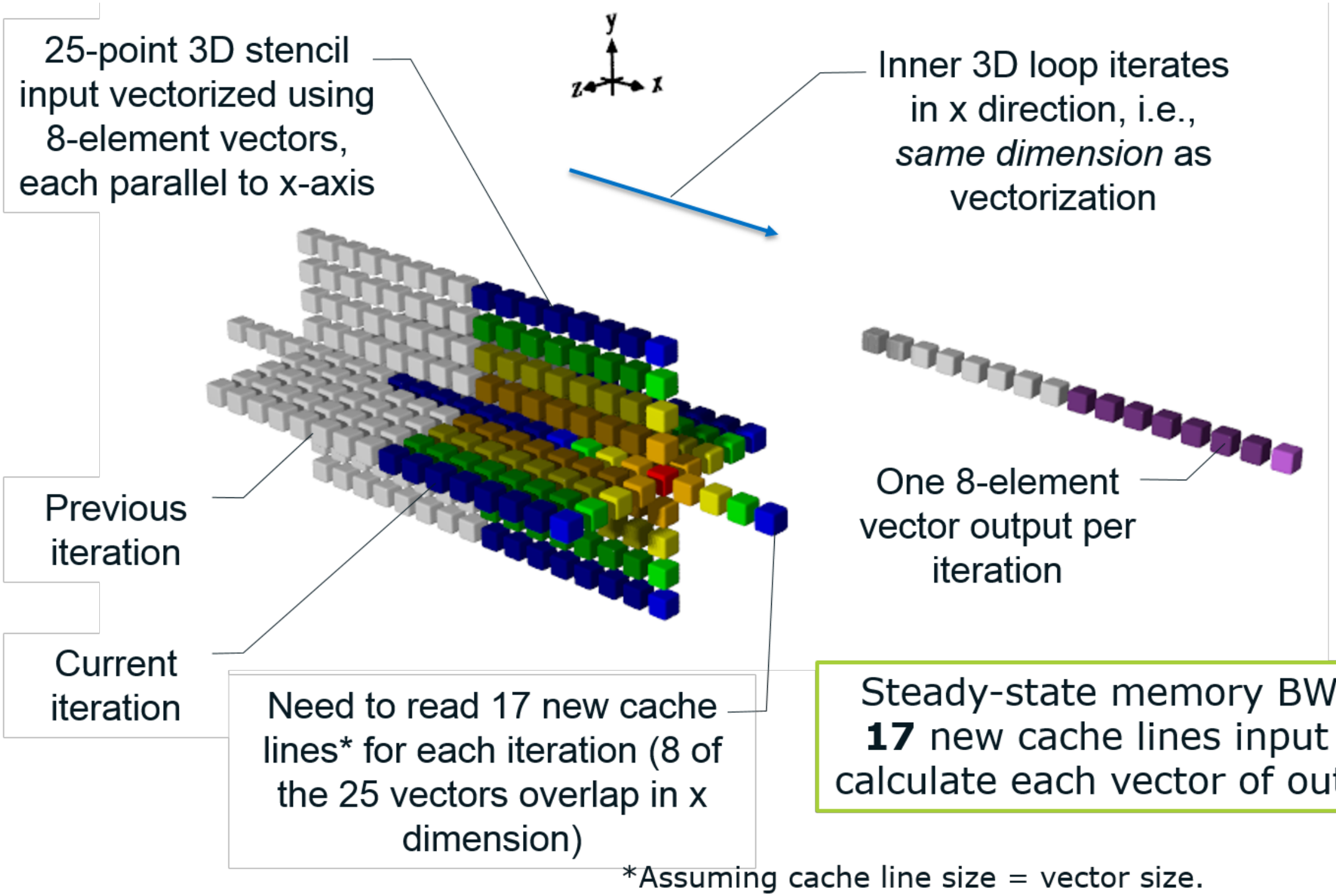
While others are applied on the tree

- OpenMP (with nested parallelism)
- SIMD-ization
- Loop blocking
- …



Beyond 1D vectorization …

Traditional "1D" vectorization requires lots of bandwidth

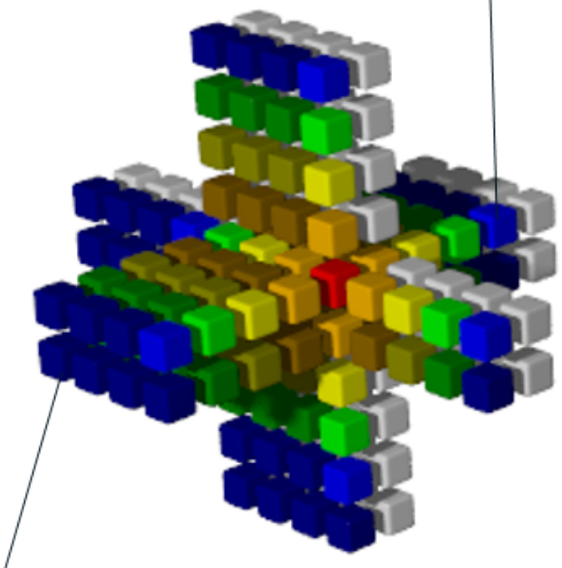


*Assuming cache line size = vector size.

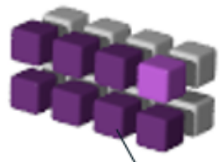
Vector folding via YASK (a Devito backend)

Data layout transformation + cross-loop vectorization to optimize bandwith usage

25-point 3D stencil input vectorized using 8-element vectors, *each containing a 4x2 grid in the x-y plane*



Inner 3D loop iterates in z direction, i.e., *perpendicular* to 2D vector



One 8-element (4x2) vector output per iteration

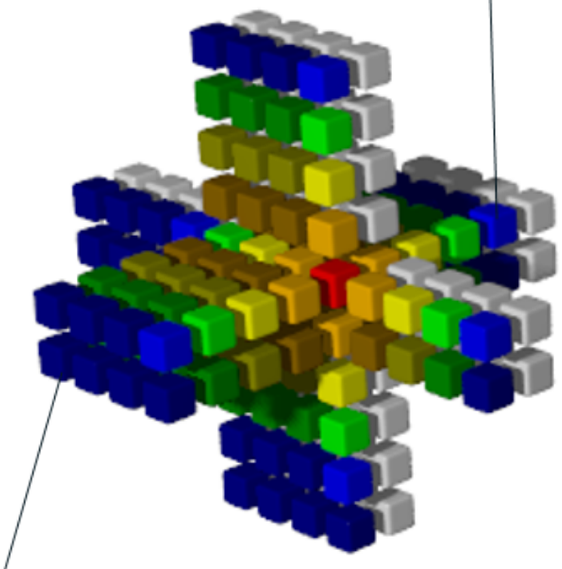
Need to read only 7 new cache lines for each iteration (vectors overlap in x-y dimensions within an iteration and in z dimension between iterations)

Steady-state memory BW = **7** new cache lines input to calculate each vector of output: **2.4x lower** than in-line

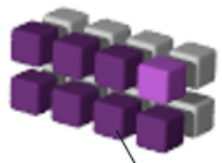
Vector folding via YASK (a Devito backend)

Data layout transformation + cross-loop vectorization to optimize bandwith usage

25-point 3D stencil input vectorized using 8-element vectors, *each containing a 4x2 grid in the x-y plane*



Inner 3D loop iterates in z direction, i.e., *perpendicular* to 2D vector



One 8-element (4x2) vector output per iteration

Need to read only 7 new cache lines for each iteration (vectors overlap in x-y dimensions within an iteration and in z dimension between iterations)

There's actually much more:
multi-level tiling
software prefetching
temporal wavefront blocking

Dual-socket — TTI on Skylake 8175

grid = 512 x 512 x 1024

single-socket, OMP only: 783 s (± 12 s)

dual-socket, OMP only: 626 s (± 45 s)

dual-socket, OMP+MPI: 319 s (± 8 s)

Loop scheduling

The problem: **convert a list of equations into a tree of loops**

```
>>> op = Operator(eqn, bcs, src.inject(…), rec.interpolate(…))
```

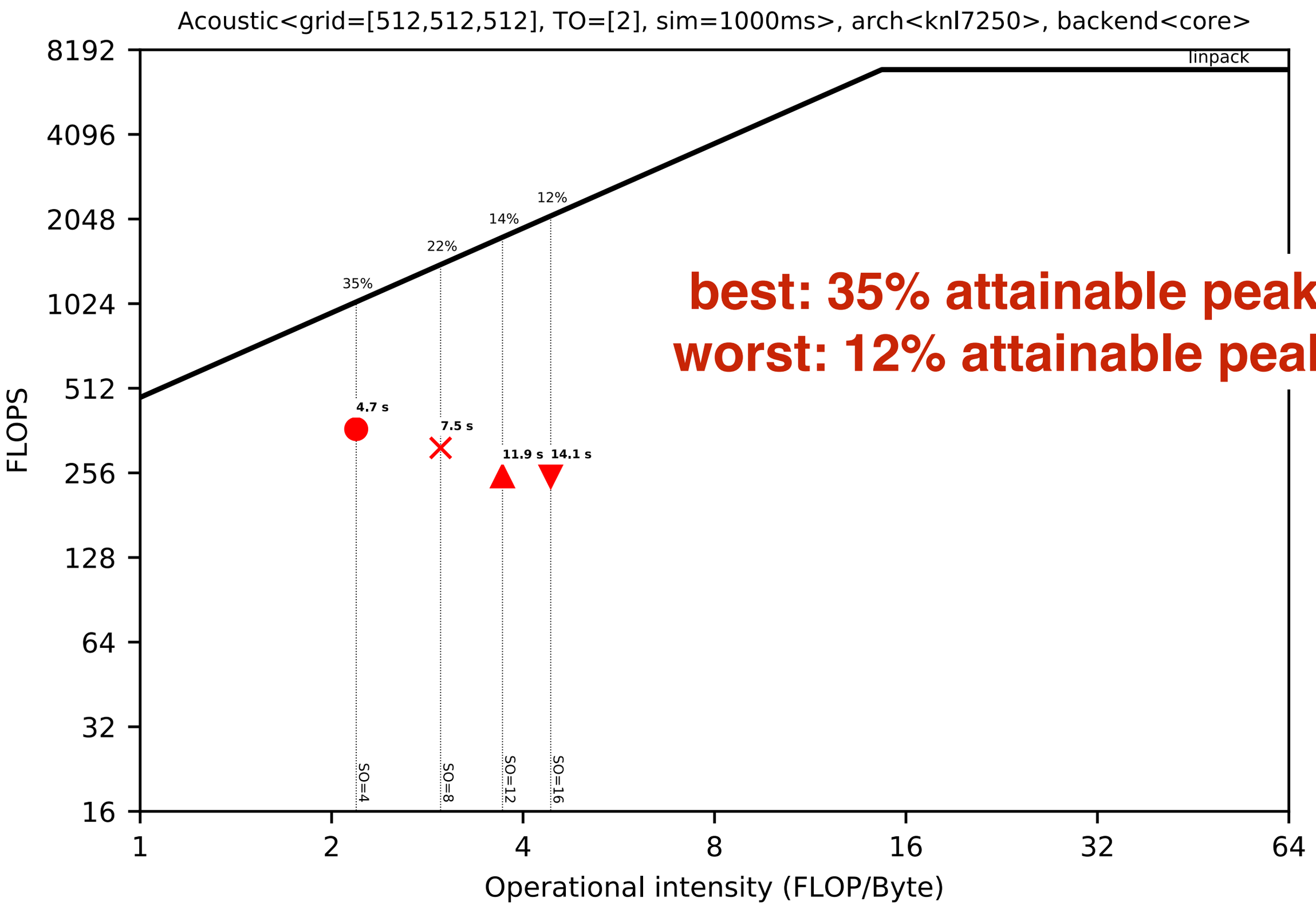
Need to support:

- Control flow (e.g., for data sub-sampling)
- Optimizations (e.g., fusion, code motion, …)
- Human-readable (e.g., generate routines to avoid repetitions)

The key ingredients:

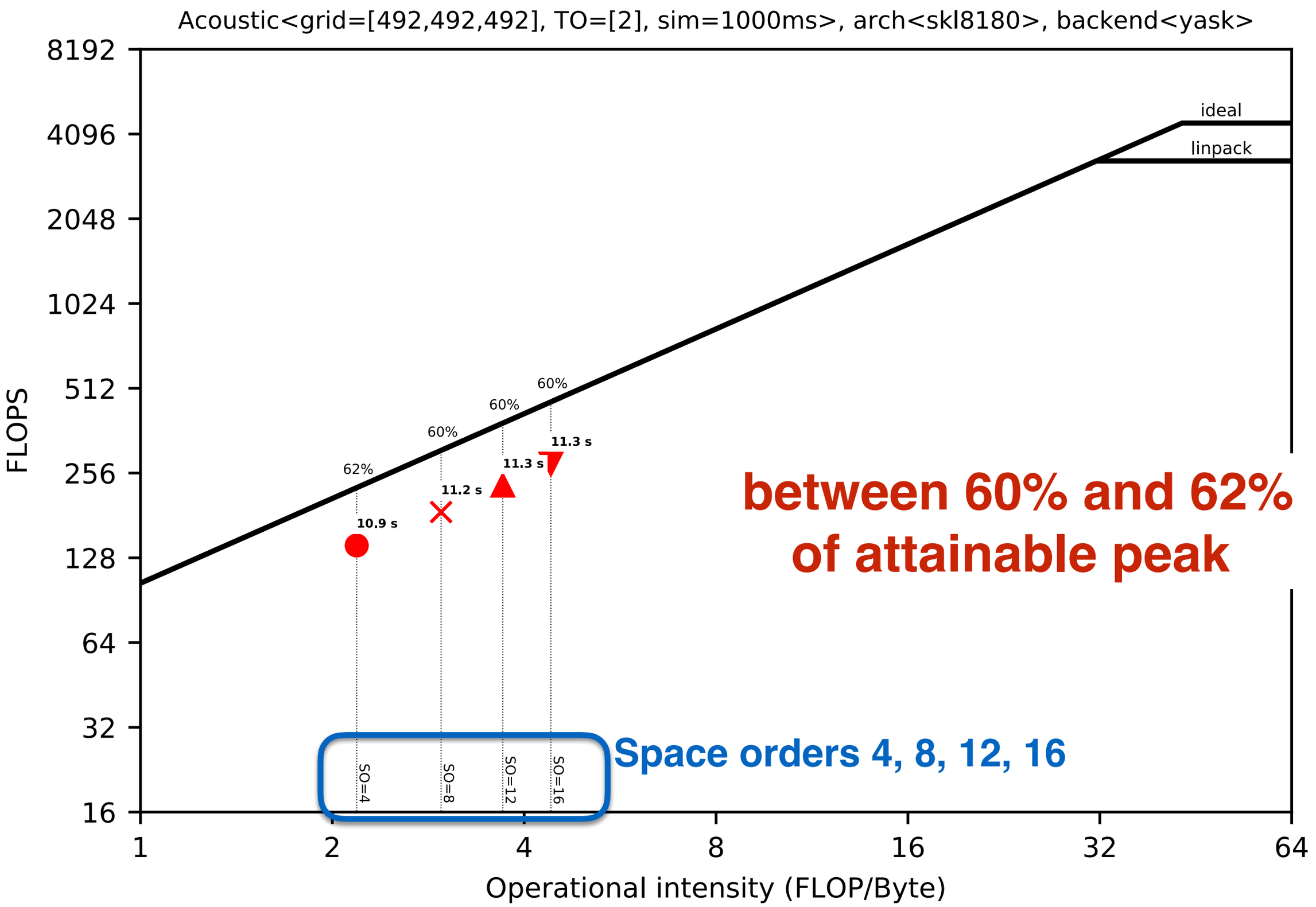
- Data dependence analysis
- Topological sorting

Single-socket — Isotropic acoustic on Xeon Phi 7250

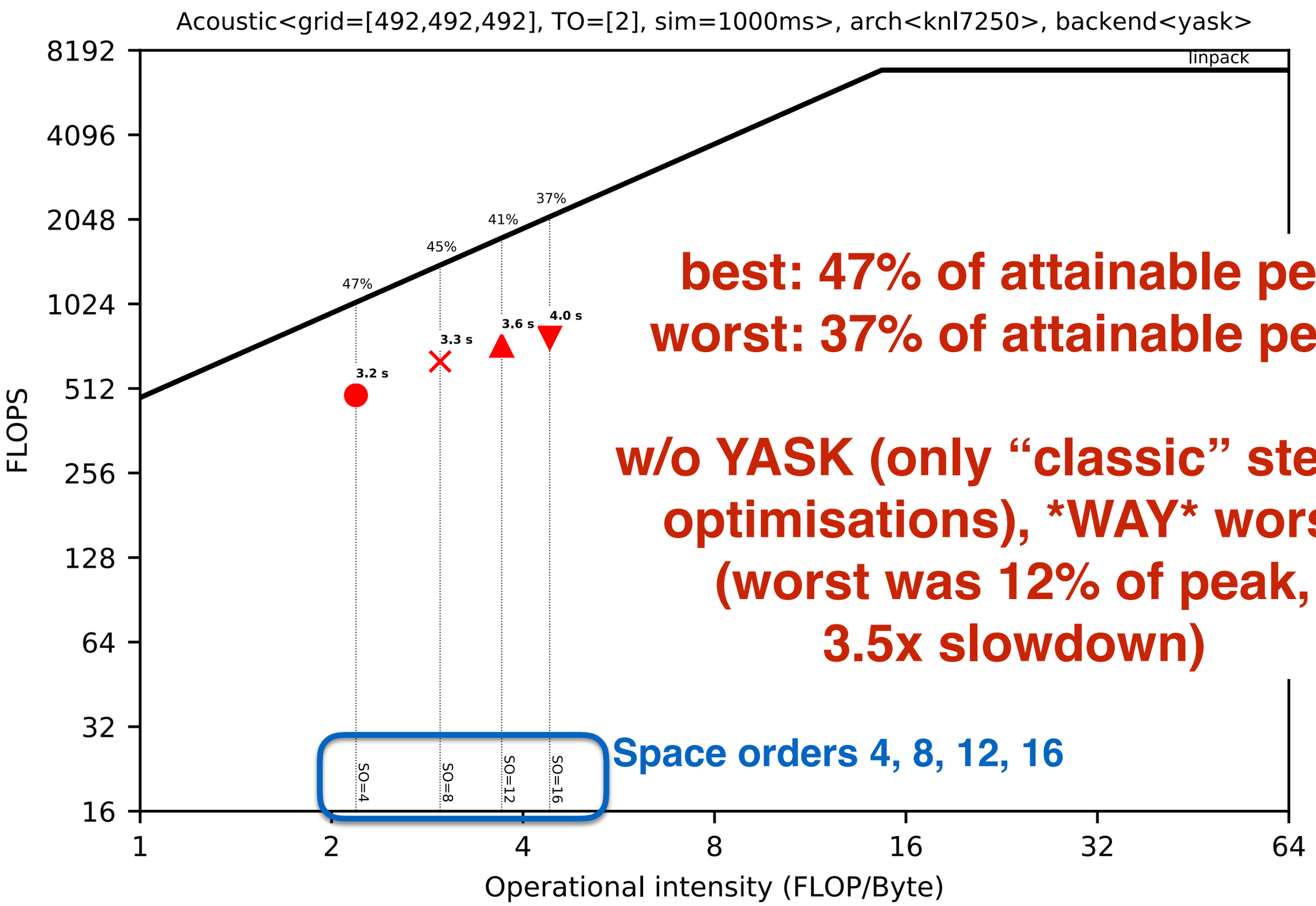


best: 35% attainable peak
worst: 12% attainable peak

Acoustic on Skylake 8180 with YASK



Acoustic on Xeon Phi 7250 with YASK



best: 47% of attainable peak
worst: 37% of attainable peak

w/o YASK (only "classic" stenci optimisations), *WAY* worse (worst was 12% of peak, 3.5x slowdown)

Space orders 4, 8, 12, 16