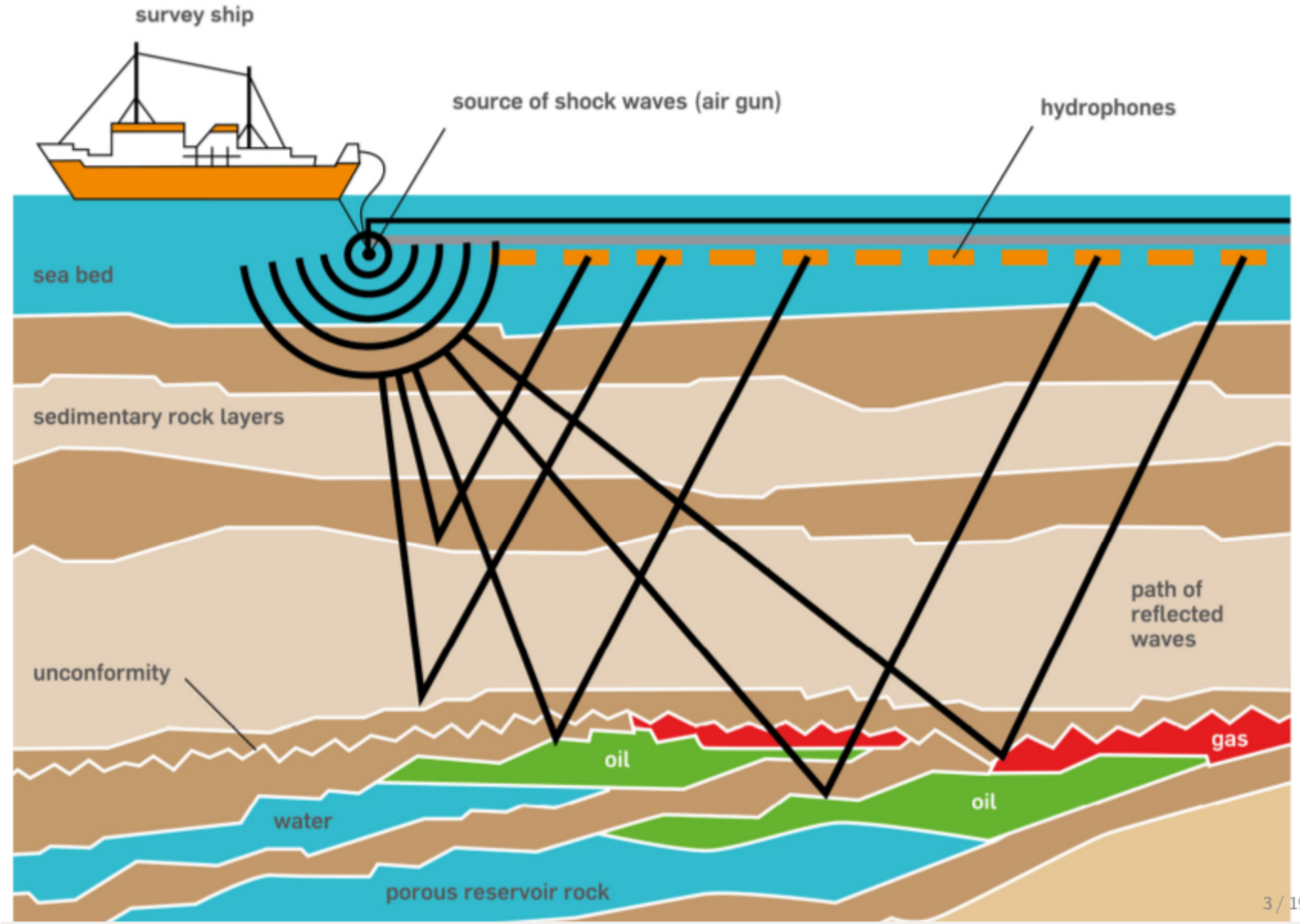
DEVITO
AUTOMATED HIGH-PERFORMANCE FINITE DIFFERENCES FOR GEOPHYSICAL EXPLORATION

F. Luporini[1], C. Yount[4], M. Louboutin[3], N. Kukreja[1], P. Witte[2], M. Lange[5], P. Kelly[1], F. Herrmann[3], G.Gorman[1]

[1]Imperial College London
[2]The University of British Columbia
[3]Georgia Institute of Technology
[4]Intel Corporation
[5]European Centre for Medium-Range Weather Forecasts
(former Imperial College London)

IXPUG 2018

Driving application: inversion algorithms for seismic imaging



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

Issue 1: Computational cost

Realistic full-waveform inversion (FWI) scenario:

- **$O(10^3)$ FLOPs per loop iteration** or **high memory pressure**
- Realistic 3D grids with **$>10^9$ grid points**
- Often more than **3000 time steps**
- **Two** operators: forward + adjoint, to be executed **~15 times**
- Usually **30000 shots**
- ≈ **O(billions) TFLOPs**
- >>> Days, weeks, months on supercomputers

Issue 2: Variations in physics and mathematics

- Overarching strategy for inversion
- Formulations of wave equations
- Space and time discretizations
- Boundary conditions, data acquisition, sources/receivers …

Issue 3: Time flies…

- Proliferation of computer architectures
- Unmaintainable, impenetrable, non-portable legacy code
- Skepticism: C/C++/Fortran **IS** the way

Raising the level of abstraction

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

```
void finite_difference_solver(…)
{
  …
  <impenetrable code with crazy
   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 finite_difference_solver(…)
{
  …
  <impenetrable code with crazy
   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

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

```
u = TimeFunction(…)
m = Function(…)
eqn = Eq(m * u.dt2 + eta * u.dt - u.laplace, 0)
```

Raising the level of abstraction

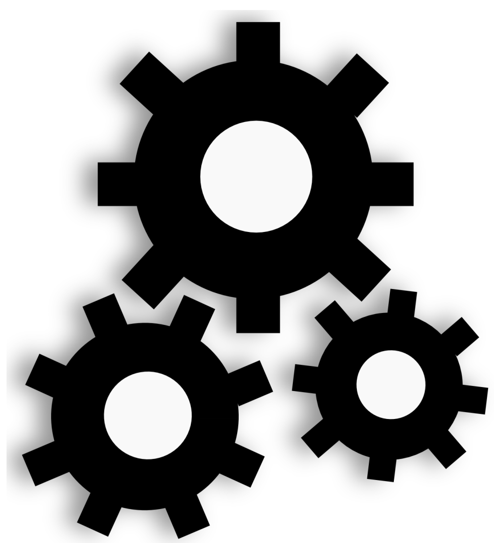
$$m\frac{\partial^2 u}{\partial t^2} + \eta\frac{\partial u}{\partial t} - \Delta u = 0$$

```
u = TimeFunction(…)
m = Function(…)
eqn = Eq(m * u.dt2 + eta * u.dt - u.laplace, 0)
```

```
void finite_difference_solver(…) { … }
```

Devito and its key ingredients

- SymPy to express stencil equations
- NumPy for data management
- >>> can exploit SciPy and Dask
- … simplicity!



- "Real" compiler technology (not a template-based code generator)
- Symbolic processing
- AST processing

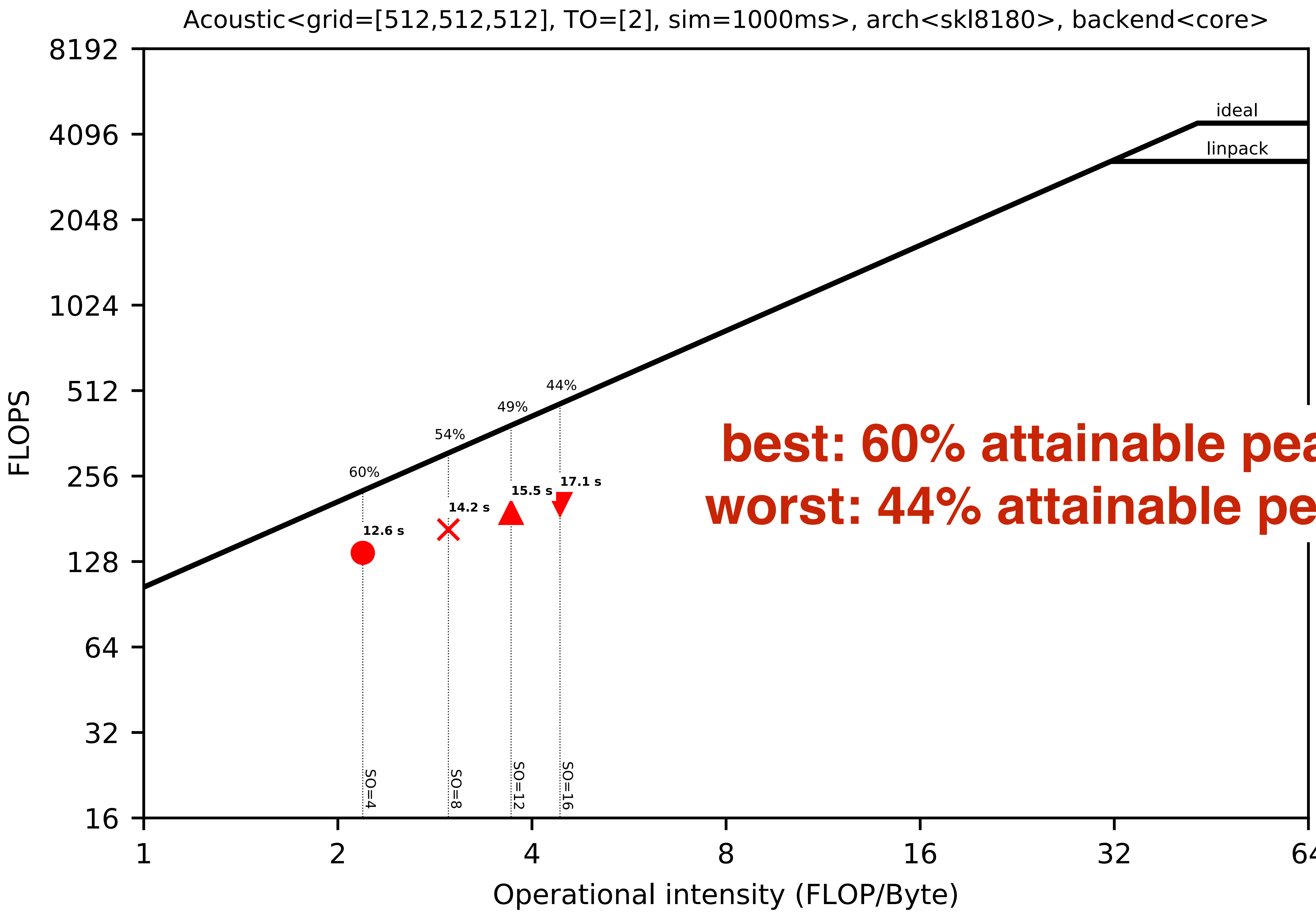
- Amongst the most advanced software to optimize the performance of stencil codes
- Data layout, multi-level blocking, prefetching, vector-folding, …

Loop-level optimizations

- We started off with standard stuff:
  - Loop blocking (no time loop)
  - Empirical auto-tuning for loop blocking
  - OpenMP parallelism
  - SIMD vectorization via OpenMP 4.5 pragmas
- Our initial thought was:

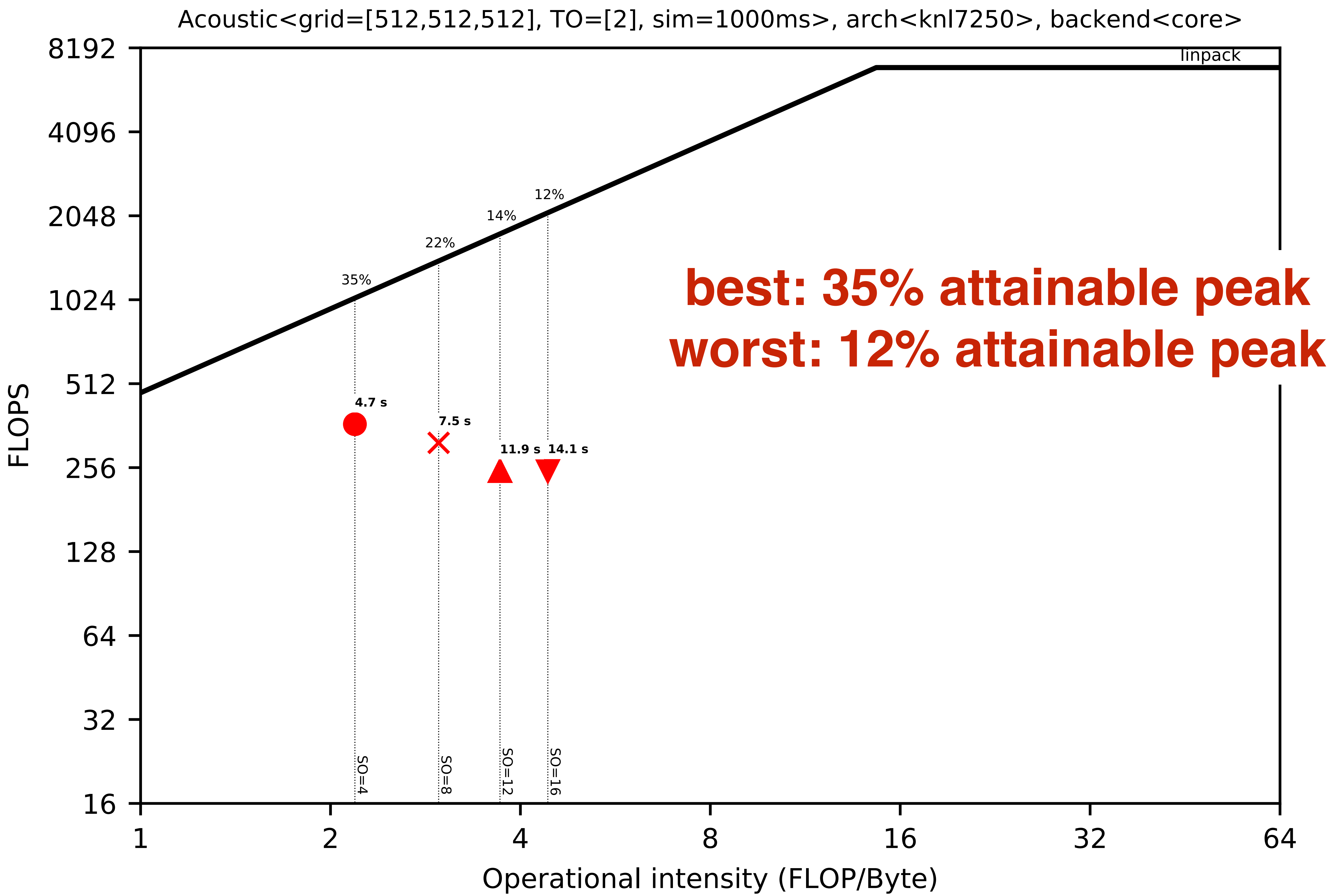
“well, OK, we don’t do anything fancy such as blocking over time, but this plus all the flop-level optimizations should be enough to get some decent performance”

Did it work? Acoustic on Skylake 8180



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

Did it work? Acoustic on Xeon Phi 7250



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

A new backend for Devito: YASK

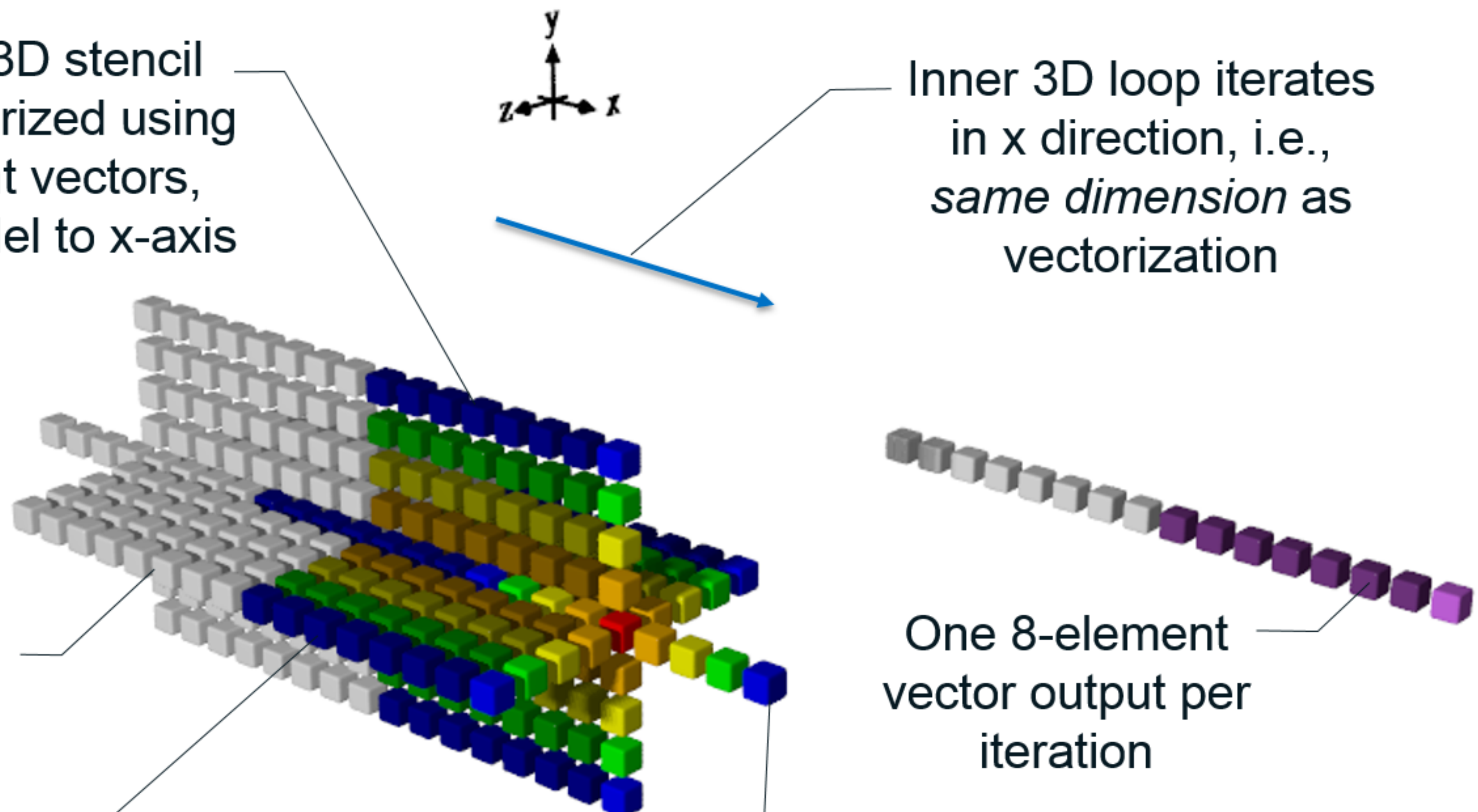
Now Devito dynamically generates and runs code that "offloads" parts of the computations (i.e., some stencil equations) onto dynamically generated YASK libraries

```
void f0(…) { … }
void f1(…) { … }

void entry_point_from_python ( … ) {
  …
  for t = 0 to num_of_timesteps:
    for i = 0 to Ni:  // Devito-optimized loop
      f0( … );

    yask_lib_0 ( … );  // jump to YASK-land

    for j = 0 to Nj:  // Some other Devito-optimized loop
      f1 ( … );
}
```

Towards “vector folding” in YASK

Traditional “1D” vectorization requires lots of bandwidth

25-point 3D stencil input vectorized using 8-element vectors, each parallel to x-axis



Inner 3D loop iterates in x direction, i.e., *same dimension* as vectorization

Previous iteration

Current iteration

One 8-element vector output per iteration

Need to read 17 new cache lines* for each iteration (8 of the 25 vectors overlap in x dimension)

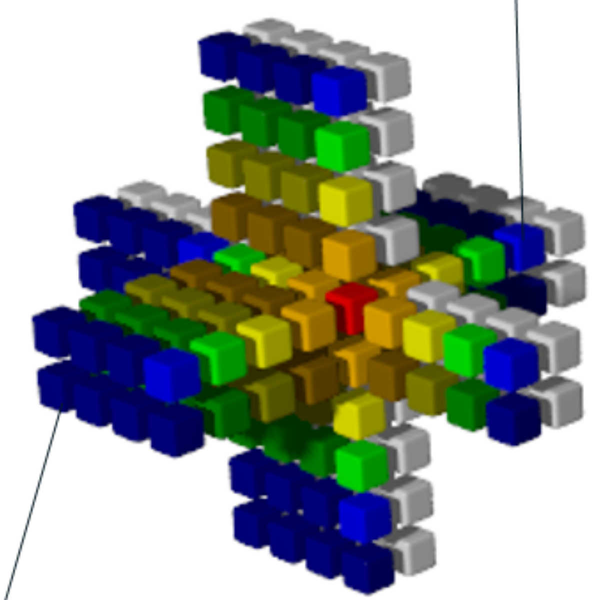
Steady-state memory BW = **17** new cache lines input to calculate each vector of output

*Assuming cache line size = vector size.

Vector folding in YASK

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

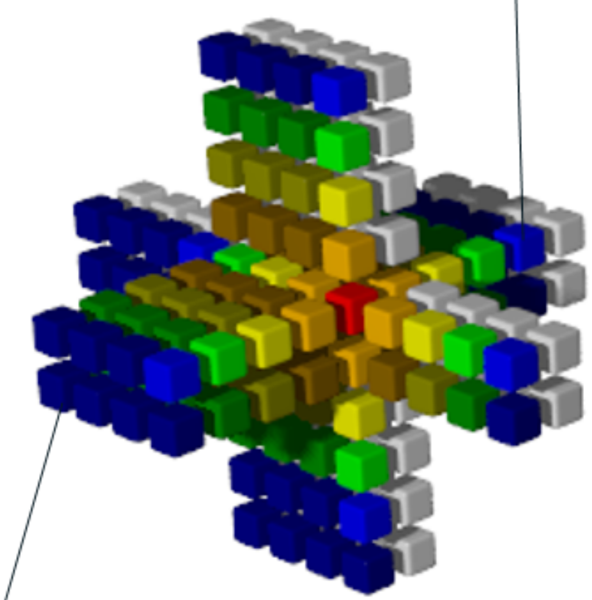
Need to read only 7 new cache lines for each iteration (vectors overlap in x-y dimensions within an iteration and in z dimension between iterations)

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

Vector folding in YASK

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*



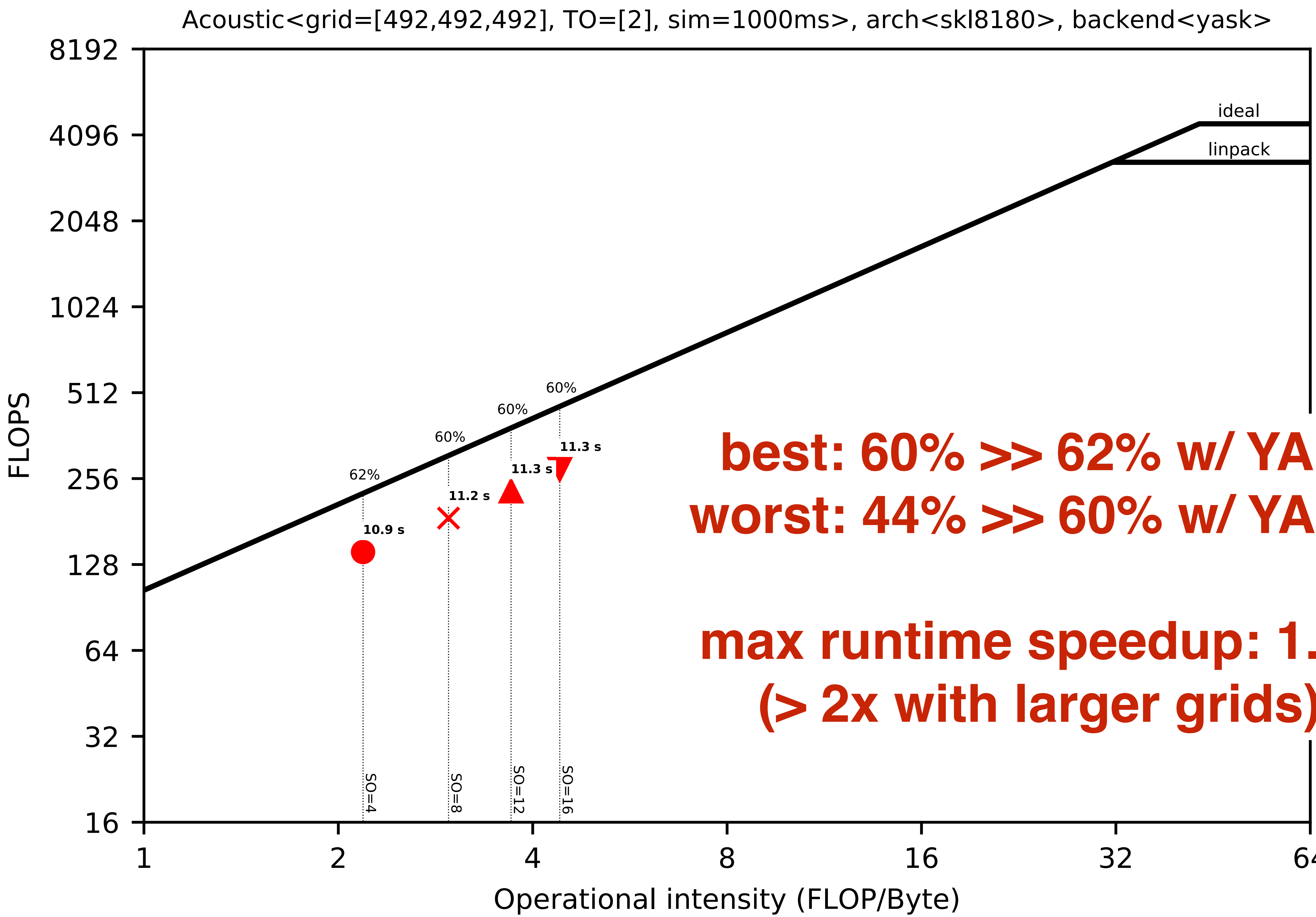
Inner 3D loop iterates in z direction, i.e., *perpendicular* to 2D vector

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:
<u>multi-level tiling</u>
<u>software prefetching</u>
<u>temporal wavefront blocking</u>

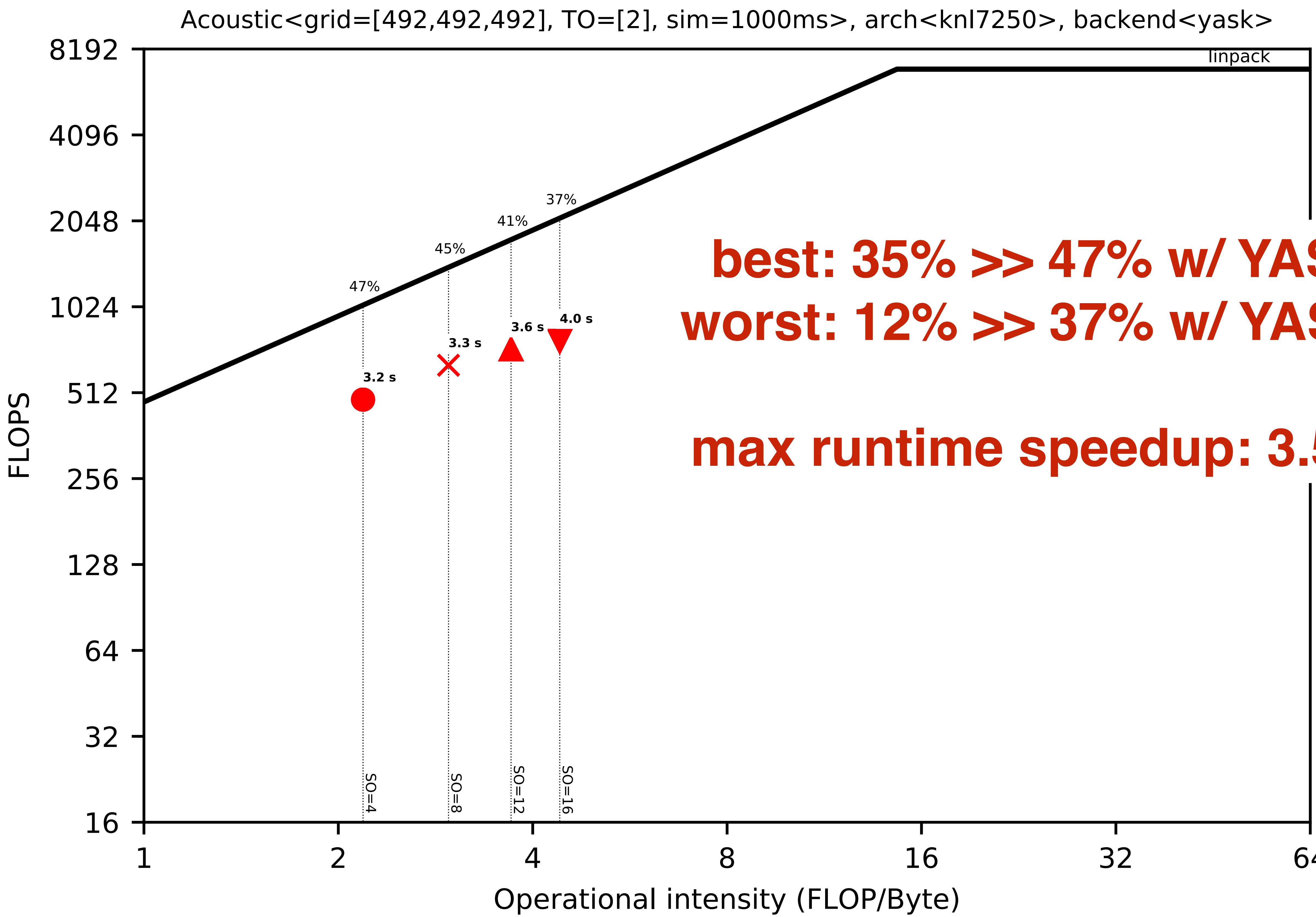
Acoustic on Skylake 8180 with YASK



best: 60% >> 62% w/ YASK
worst: 44% >> 60% w/ YASK

max runtime speedup: 1.5x
(> 2x with larger grids)

Acoustic on Xeon Phi 7250 with YASK



best: 35% >> 47% w/ YASK

worst: 12% >> 37% w/ YASK

max runtime speedup: 3.5x

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks. Intel internal measurements as of Dec 2017 on Intel® Xeon Phi™ processor 7250 with 16 GiB MCDRAM, 96 GiB DDR4 and/or Intel® Xeon® processor 8108 with 128 GiB DDR. Benchmark results were obtained prior to implementation of recent software patches and firmware updates intended to address exploits referred to as "Spectre" and "Meltdown". Implementation of these updates may make these results inapplicable to your device or system.

More aggressive FLOP reduction strategies

- Common sub-expressions elimination, factorization, …

More aggressive FLOP reduction strategies

- Common sub-expressions elimination, factorization, …
- **Cross-iteration redundancies elimination**

```
for i, for j, …
  sin(phi[i,j]) + sin(phi[i-1,j-1]) + sin(phi[i+2,j+2])
```

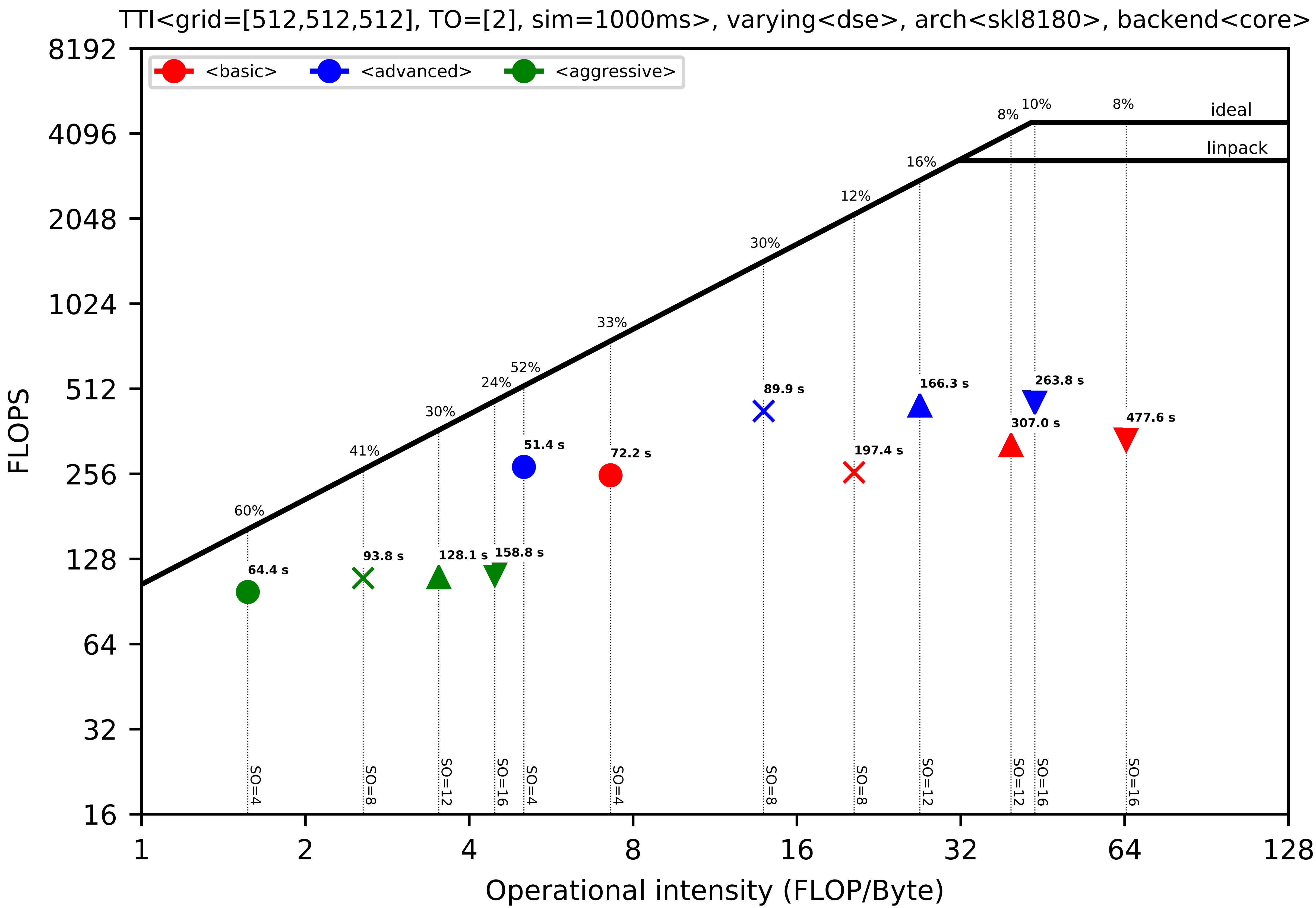
Observations:

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

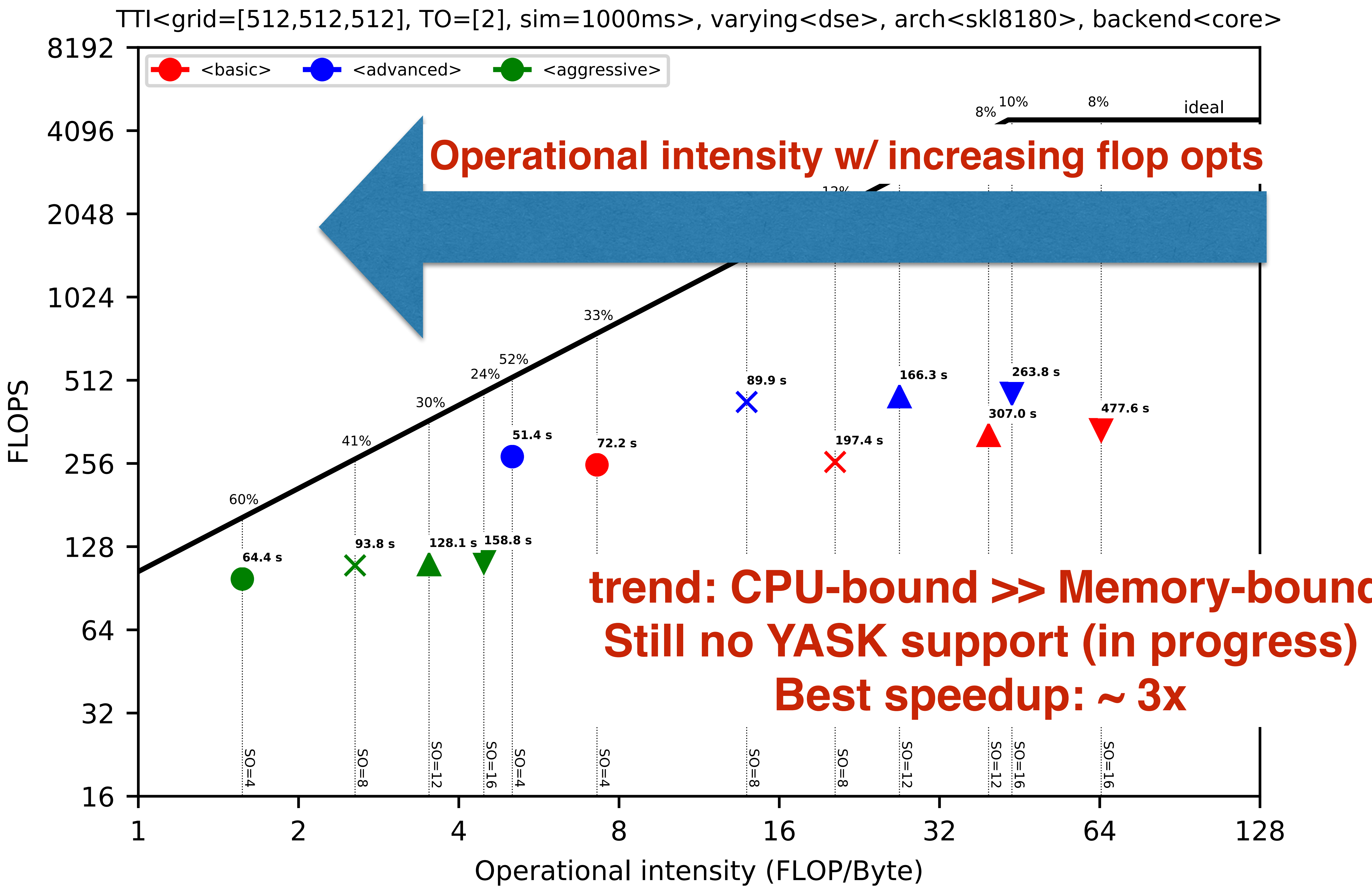
```
for i, for j
  B[i,j] = sin(phi[i,j])
for i, for j, …
  B[i,j] + B[i-1,j-1] + B[i+2,j+2]
```

Trading FLOPs for storage?

More aggressive FLOP reduction strategies



More aggressive FLOP reduction strategies



Legal Disclaimer & Optimization Notice

Conclusions and resources

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

Useful links

- **Website: http://www.devitoproject.org**
- **GitHub: https://github.com/opesci/devito**
- **Slack: https://opesci-slackin.now.sh**

Imperial College London

EPSRC Engineering and Physical Sciences Research Council

intel

Appendix

Experimentation details

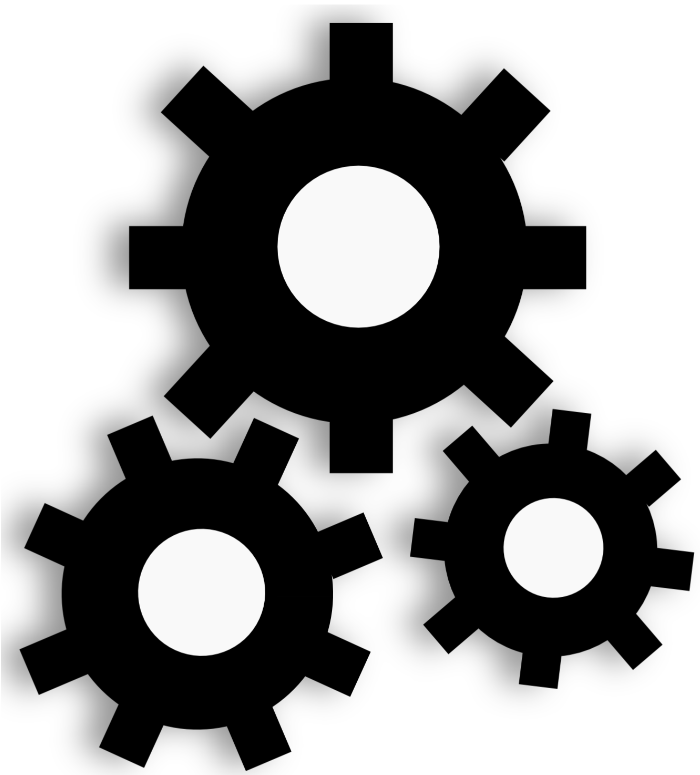
- Architectures
  - Intel® Xeon® Platinum 8180 Processor ("Skylake", 28 cores)
  - Intel(R) XeonPhi(R) 7250 (68 cores)
    - Quadrant mode (still no support for NUMA)
    - Tried 1, 2, 4 threads per core. Shown best.

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

- OpenMP
  - Single socket
  - Thread pinning via Numactl

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

Philosophy: optimizations at the <u>RIGHT</u> level of abstraction



Example: optimizations for FLOPs reduction

`Operator([eqn1, eqn2, …, eqn3])`

- Runtime constant propagation
- Equation clustering, **NOT** loop fusion
- Symbolic transformations to minimize the operation count of the equations

all based on Python and SymPy; no trace of loops yet!