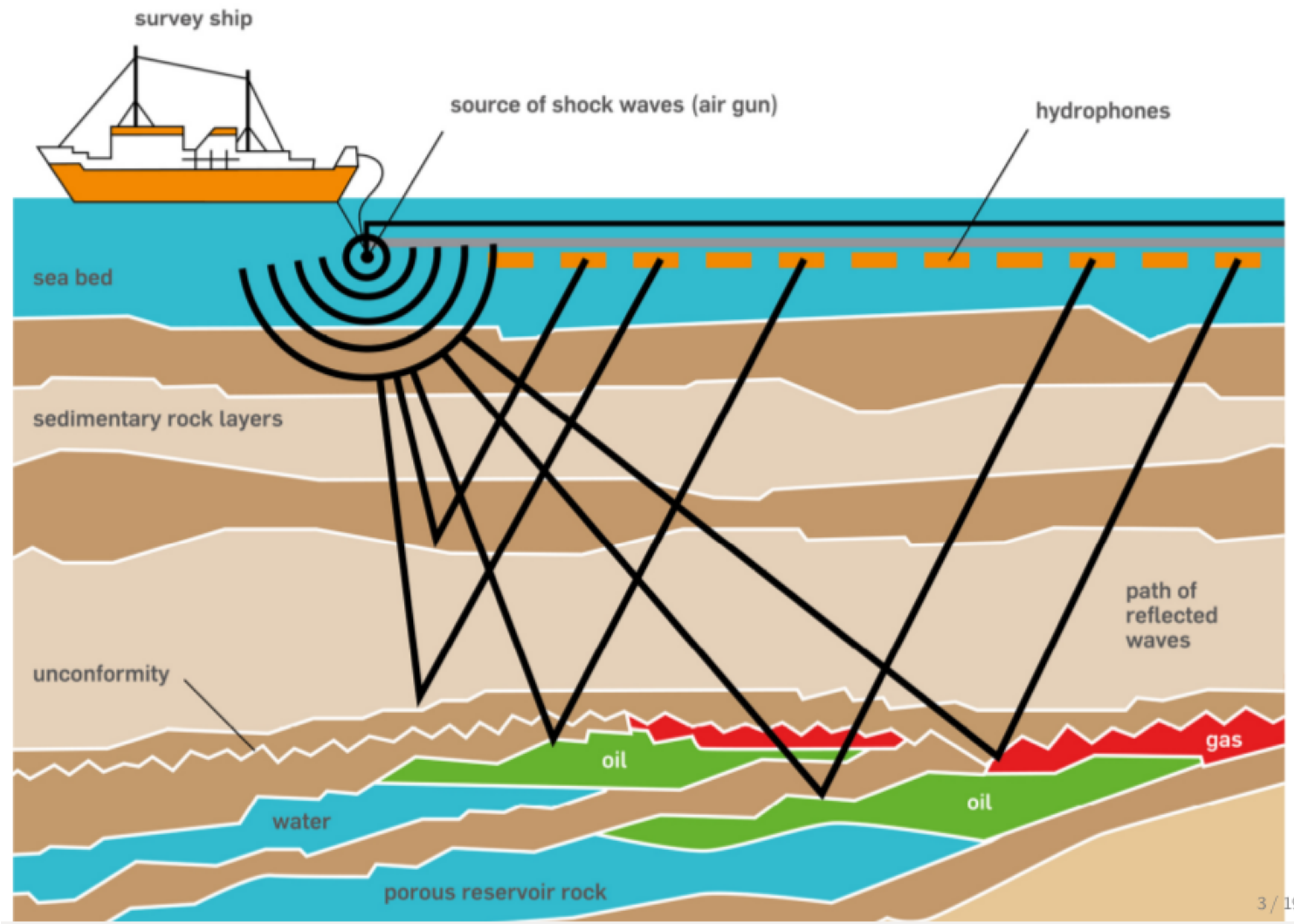
AUTOMATED LOOP GENERATION FOR HIGH-PERFORMANCE FINITE DIFFERENCES (AND BEYOND)

F. Luporini[1], C. Yount[4], M. Louboutin[3], N. Kukreja[1], P. Witte[2], T. Burges[5], M. Lange[6], P. H. J. 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]DUG - DownUnder Geosolutions
[6]European Centre for Medium-Range Weather Forecasts (former Imperial College London)

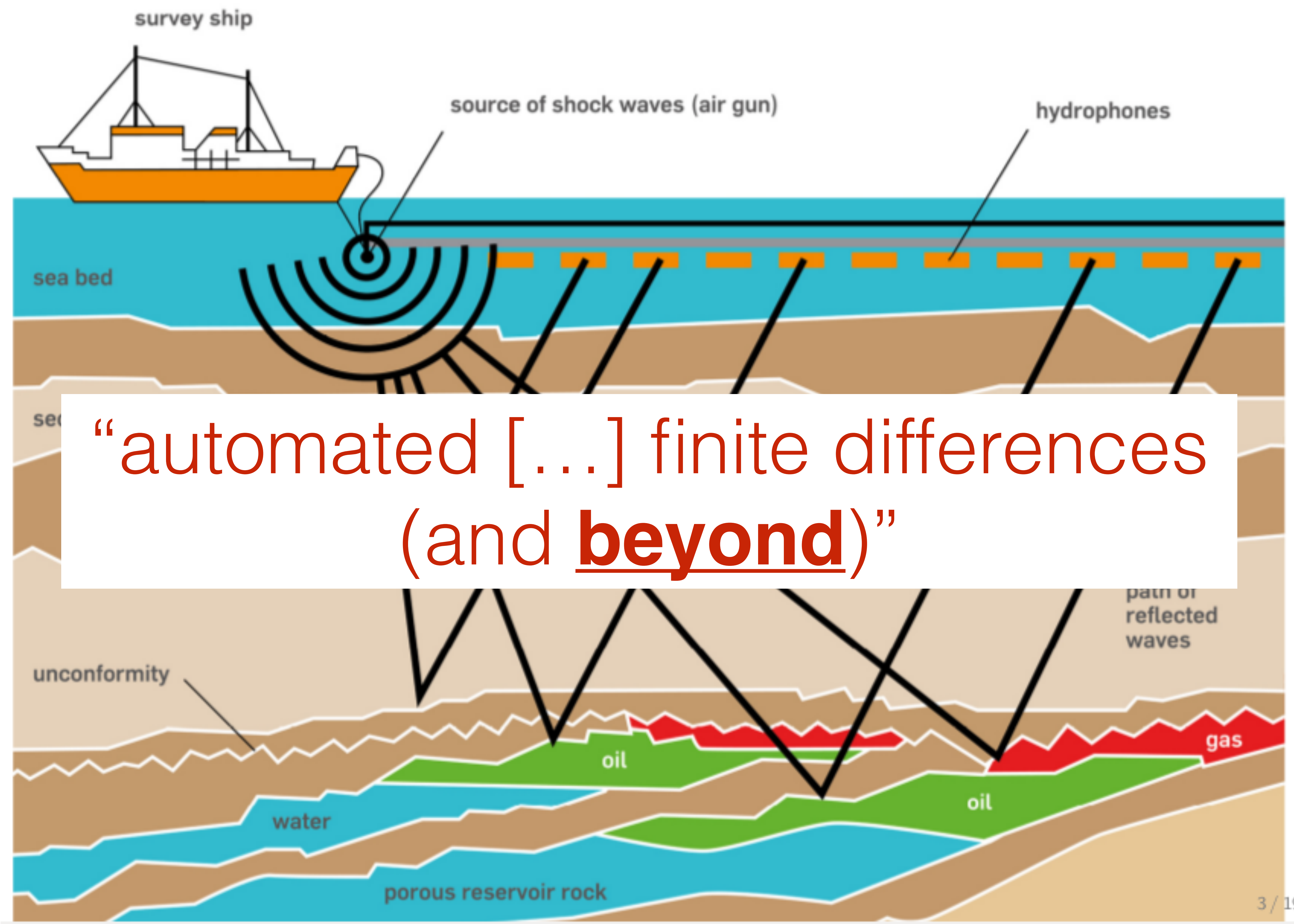
Dagstuhl Seminar, March 2018

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

Driving application: inversion algorithms for seismic imaging



“automated [...] finite differences (and **<u>beyond</u>**)”

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 kernel(…) {
  …
  <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 kernel(…) {
   …
   <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

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

```
eqn = m * u.dt2 + eta * u.dt - u.laplace
solve(eqn, u.forward)
```

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
solve(eqn, u.forward)
```

```
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
solve(eqn, u.forward)
```

```
void kernel(…) { … }
```

1) Flexibility in space/time discretization

```
u = TimeFunction(…, space_order=so)
eqn = m * u.dt2 + eta * u.dt - u.laplace
solve(eqn, u.forward)
```

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]) +
```

2) Backward propagation

```
u = TimeFunction(…, space_order=so)
eqn = m * u.dt2 - eta * u.dt - u.laplace
solve(eqn, u.backward)
```

produces

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

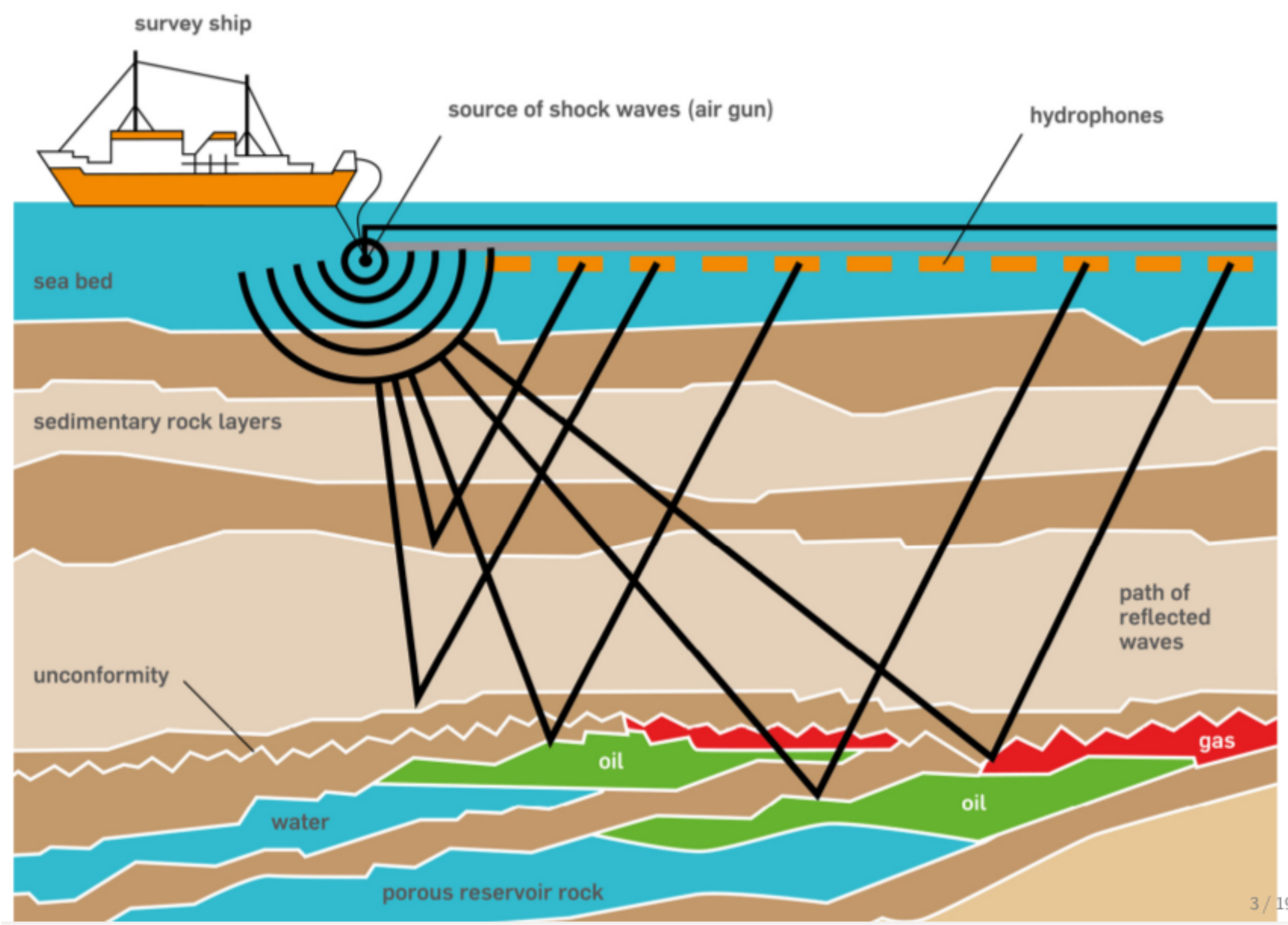
instead of

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

So we must march backwards in time to let the information flow from an iteration to another ("true" flow dependences)

3) beyond finite differences (sparse functions)

```
u = TimeFunction(…, space_order=so)
src = SparseFunction(…)
rec = SparseFunction(…)
eqns = […, src.inject(…), rec.interpolate(…)]
```



Example
Hydrophones only at the top of a 3D grid, but in general unaligned with the computational grid

Eventually, the generated loop nest can be quite complex

```
for (int time = time_m, t0 = …, t1 = …, …) {
  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] = …
      }
    }
  }
  for (int p_src = p_src_m; p_src <= p_src_M; p_src += 1) {
    u[t1][map[…]][map[…]][map[…]] = …
  }
  for (int p_rec = p_rec_m; p_rec <= p_rec_M; p_rec += 1) {
    rec[time][p_rec] = …
  }
}
```

Indirection array

4) beyond finite differences (chained BLAS/contractions)

```
A = Function(…)
B = Function(…)
…
eqns = [Eq(D, A*B + A*C), Eq(F, D*E)]
```

4) beyond finite differences (chained BLAS/contractions)

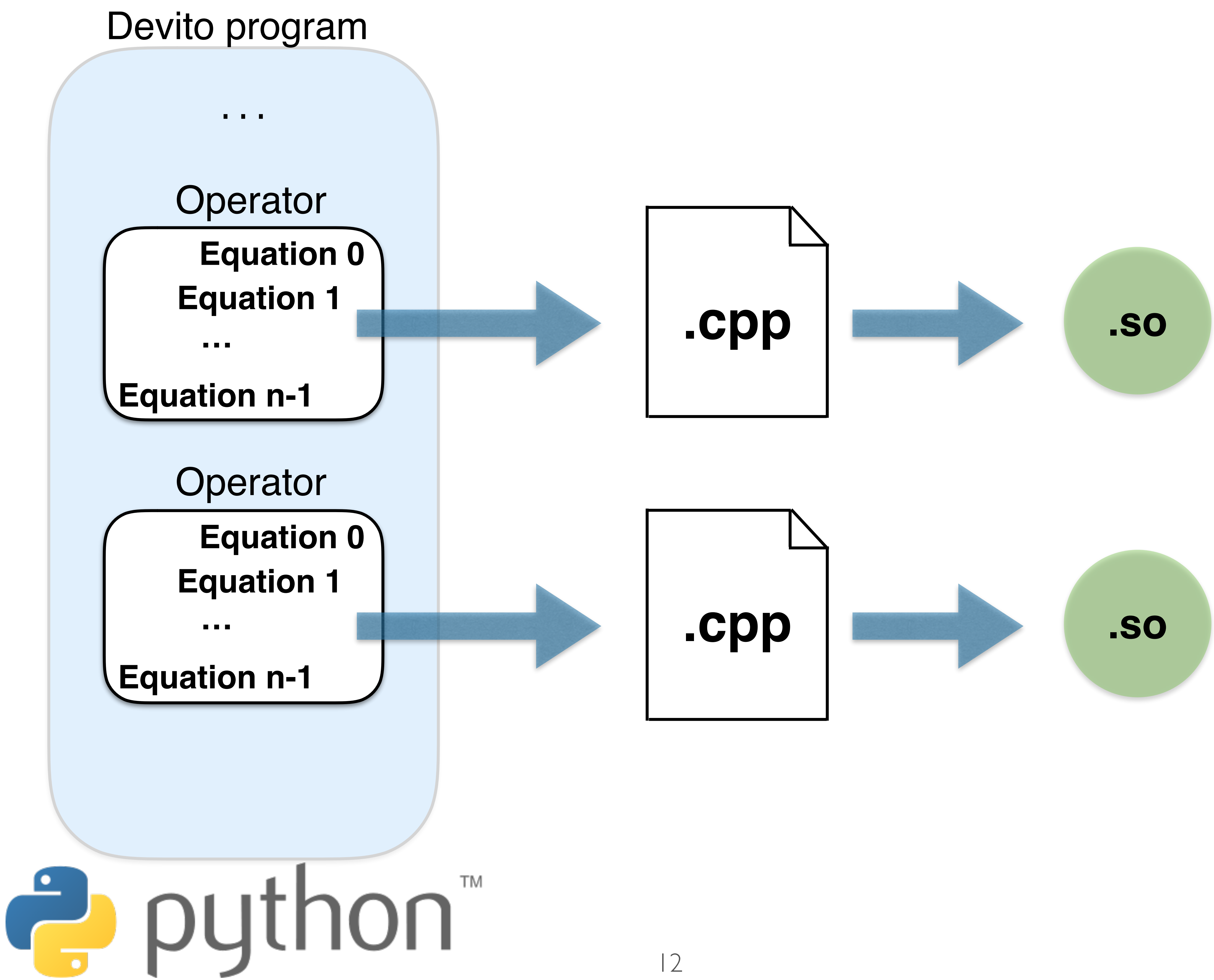
```
A = Function(…)
B = Function(…)
…
eqns = [Eq(D, A*B + A*C), Eq(F, D*E)]
```

Reuse D along i

```
for (int i = i_s; i < i_e; i += 1) {
  for (int k = k_s; k < k_e; k += 1) {
    for (int j = j_s; j < j_e; j += 1) {
      D[i][k] = A[i][j]*B[j][k] + A[i][j]*C[j][k];
    }
  }
  for (int l = l_s; l < l_e; l += 1) {
    for (int k = k_s; k < k_e; k += 1) {
      F[i][l] = D[i][k]*E[k][l];
    }
  }
}
```

Inlined sum

Devito: program model

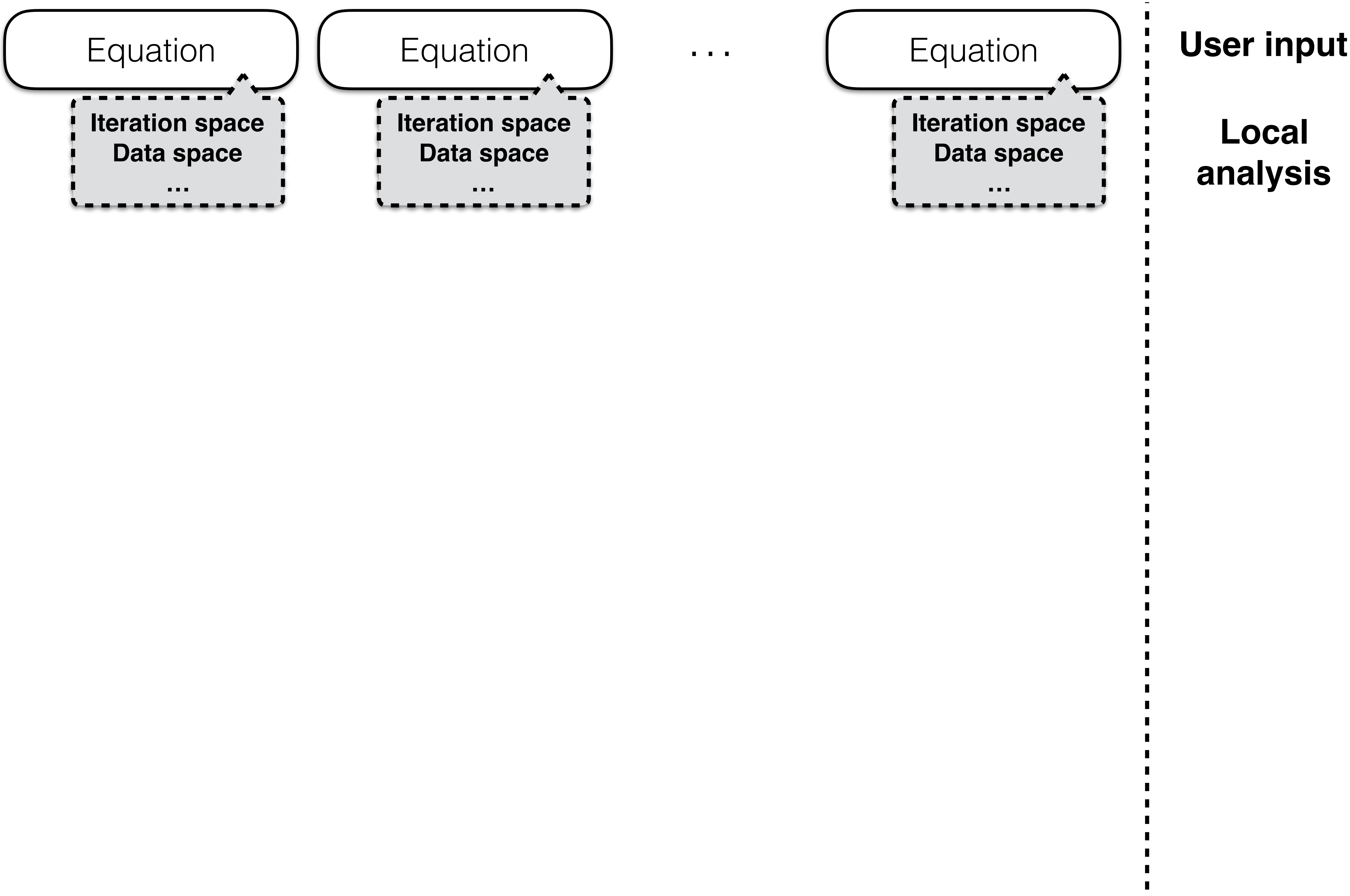


Devito: compilation passes and IRs overview

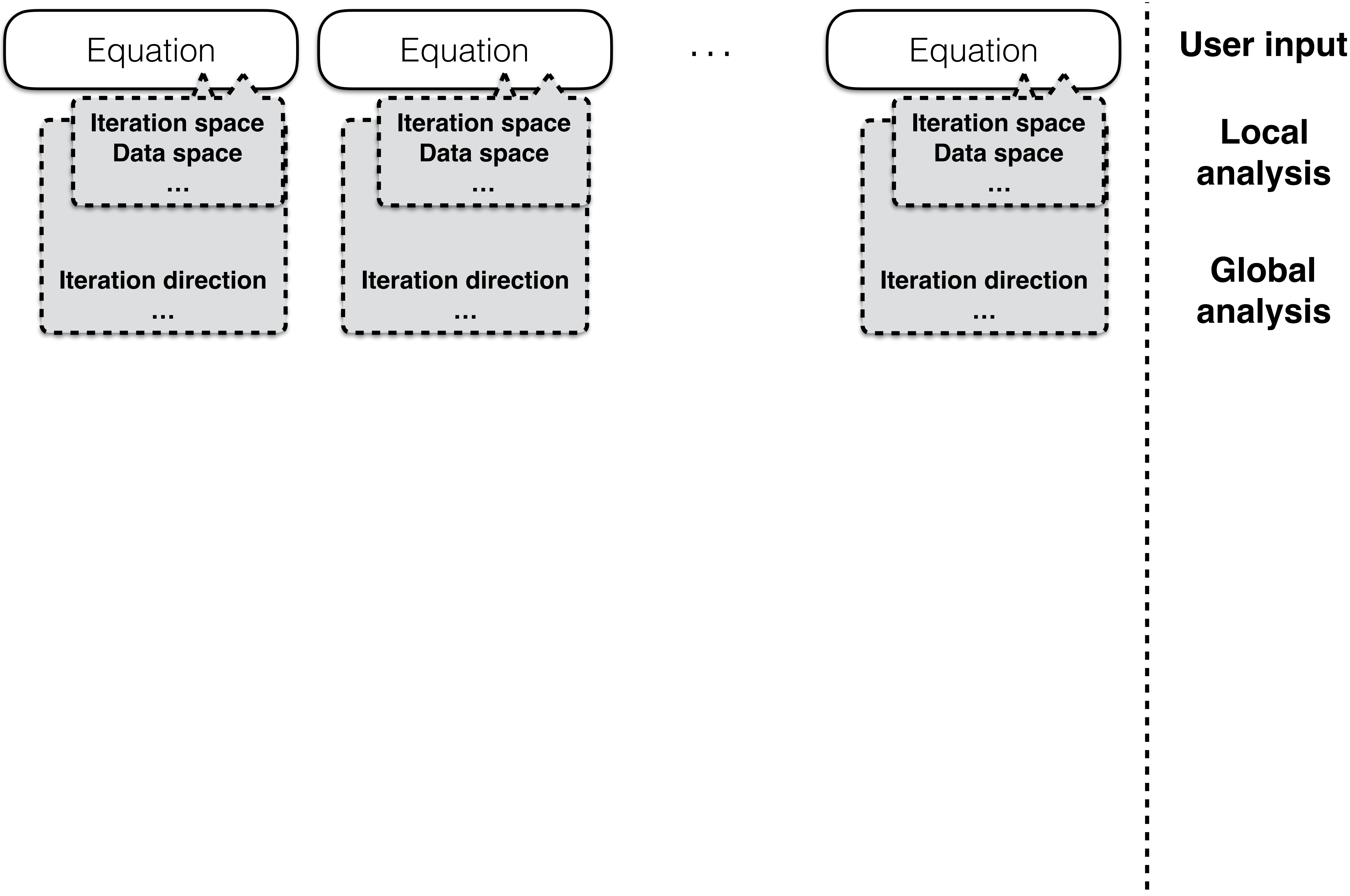
Equation | Equation | . . . | Equation

User input

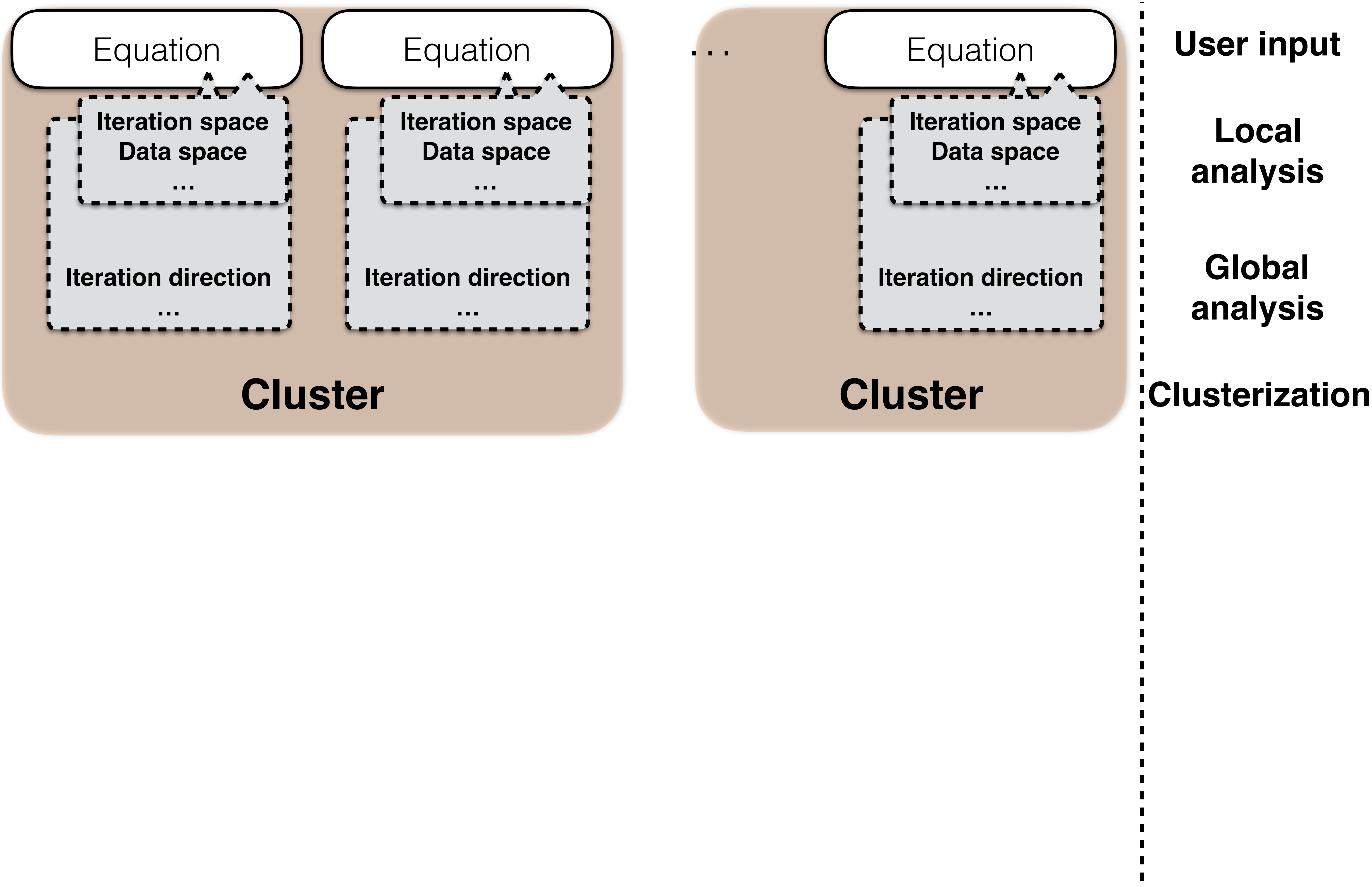
Devito: compilation passes and IRs overview



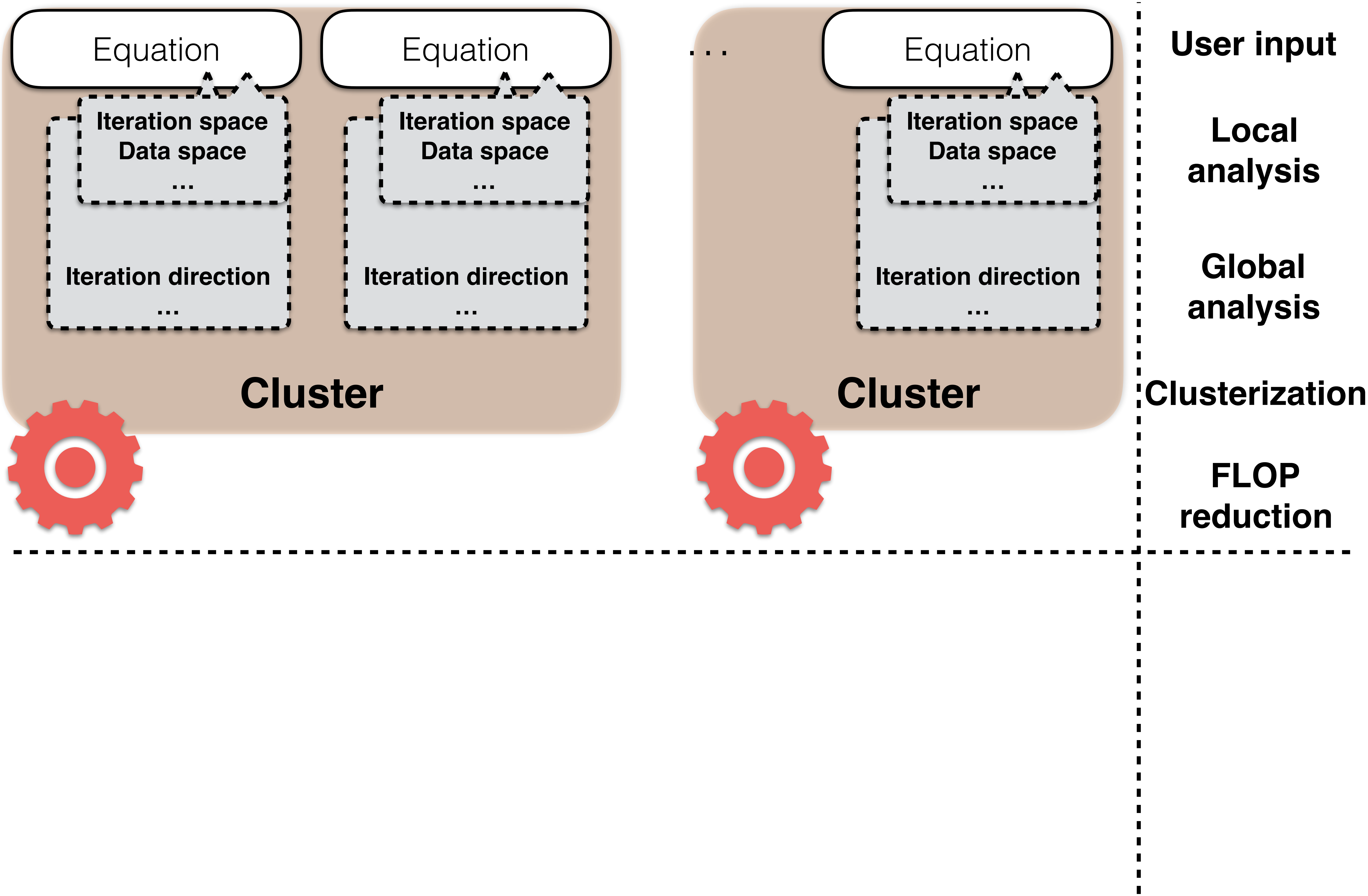
Devito: compilation passes and IRs overview



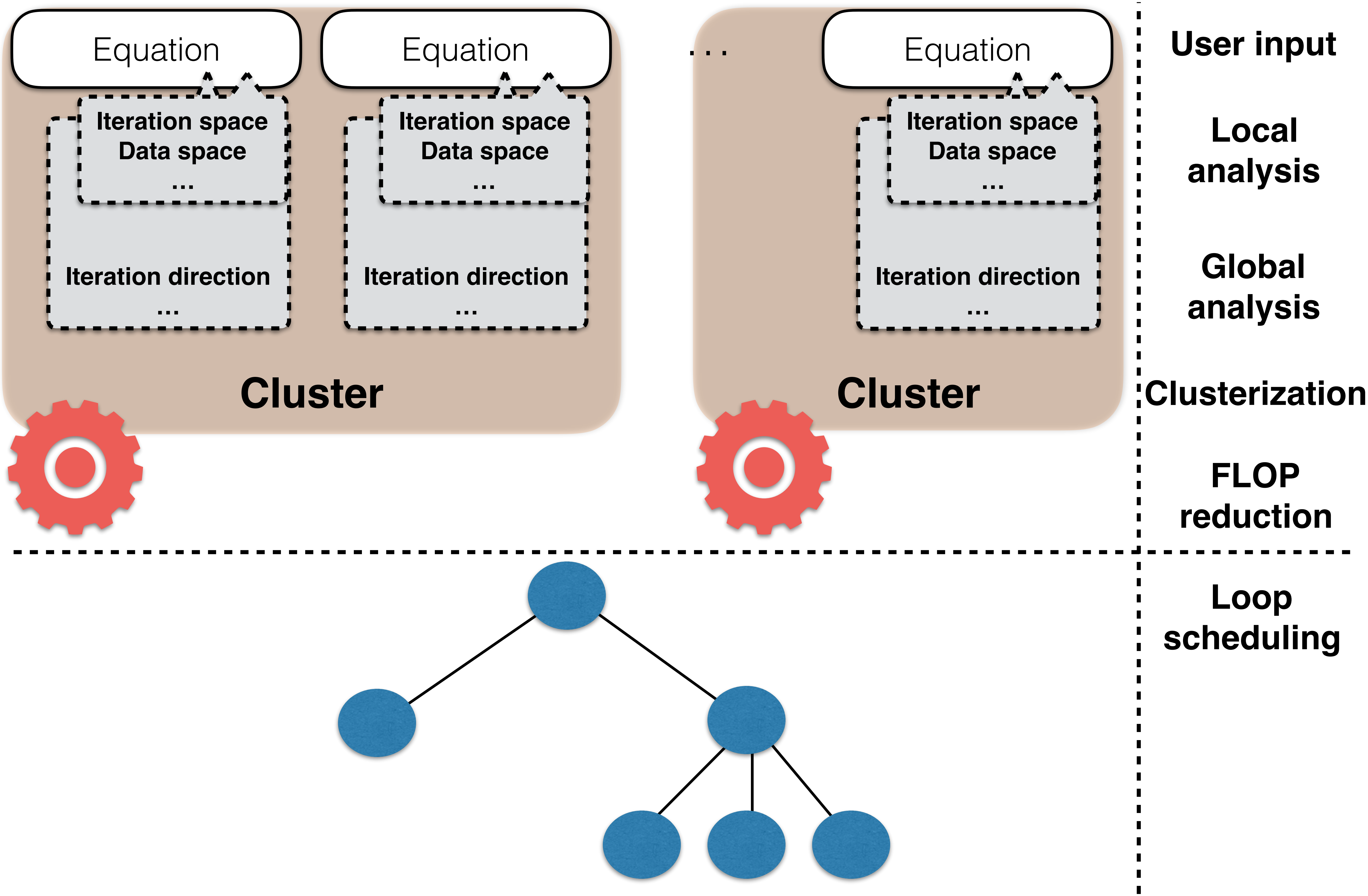
Devito: compilation passes and IRs overview



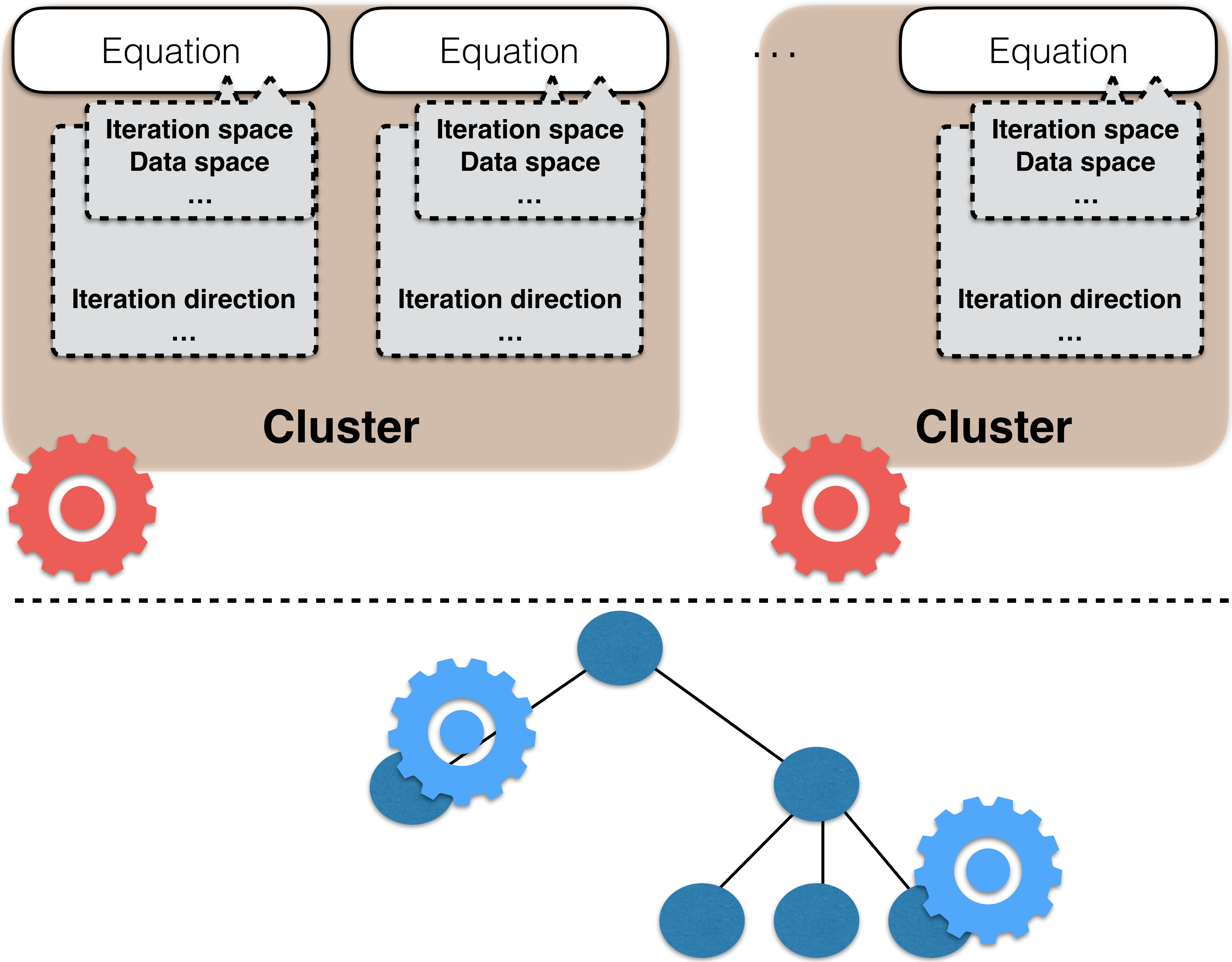
Devito: compilation passes and IRs overview



Devito: compilation passes and IRs overview



Devito: compilation passes and IRs overview



User input

Local analysis

Global analysis

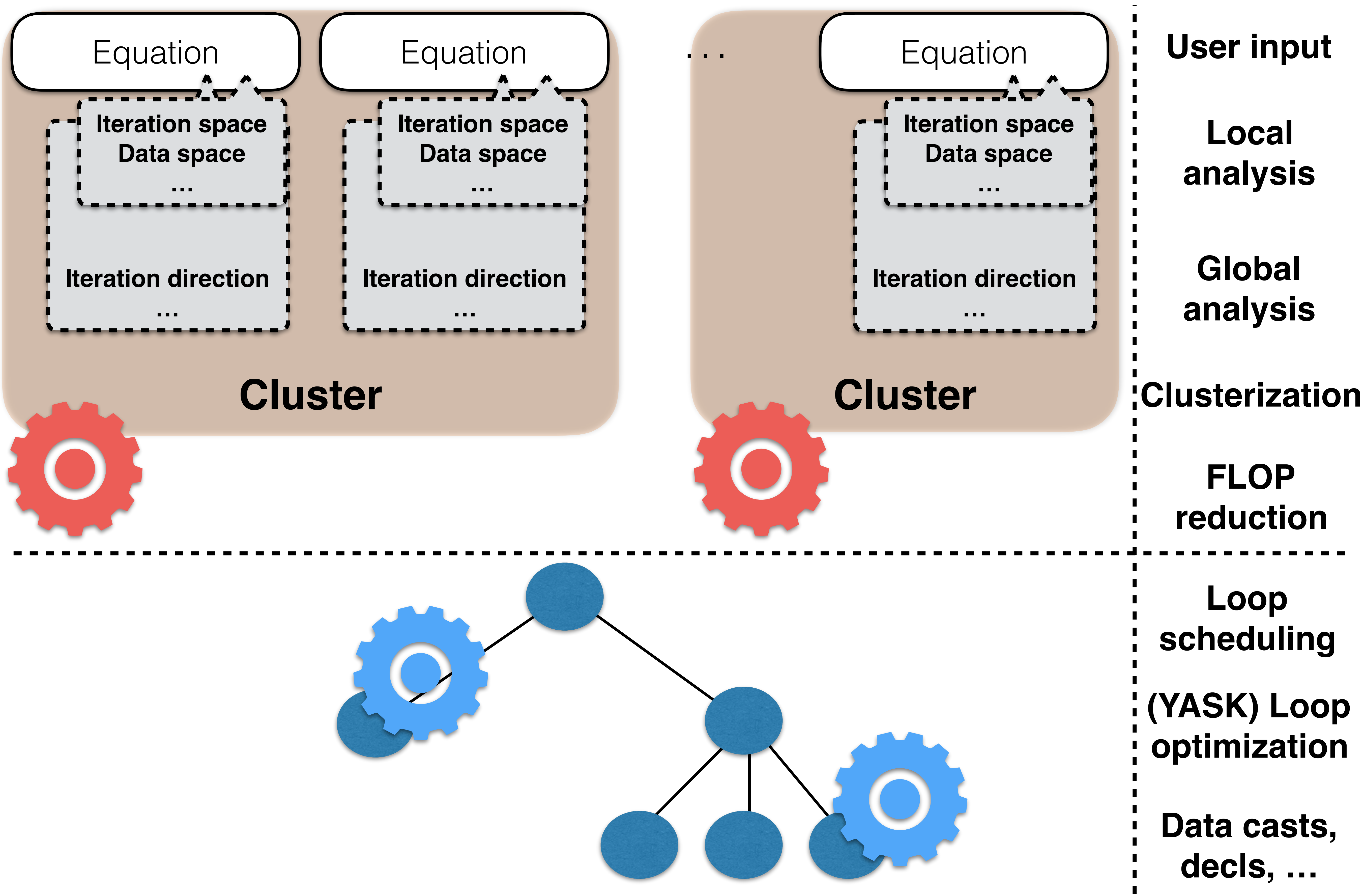
Clusterization

FLOP reduction

Loop scheduling

(YASK) Loop optimization

Devito: compilation passes and IRs overview



FLOPs reduction by symbolic transformations

- Common sub-expressions elimination, factorization, …

FLOPs reduction by symbolic transformations

- 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]
```

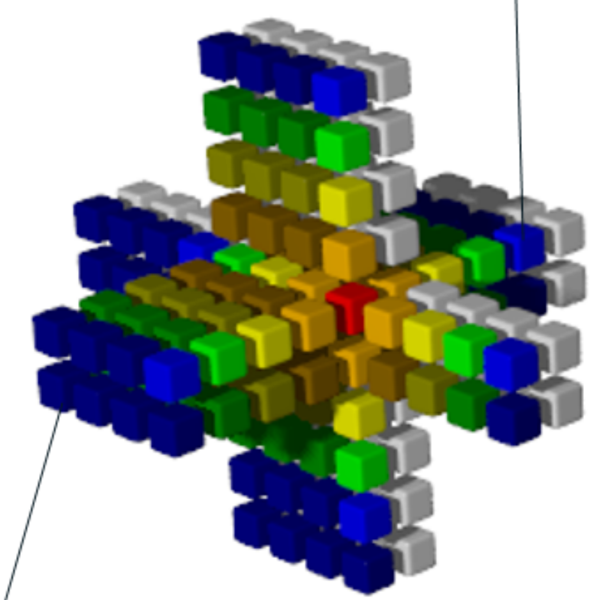
Trading FLOPs for storage?

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



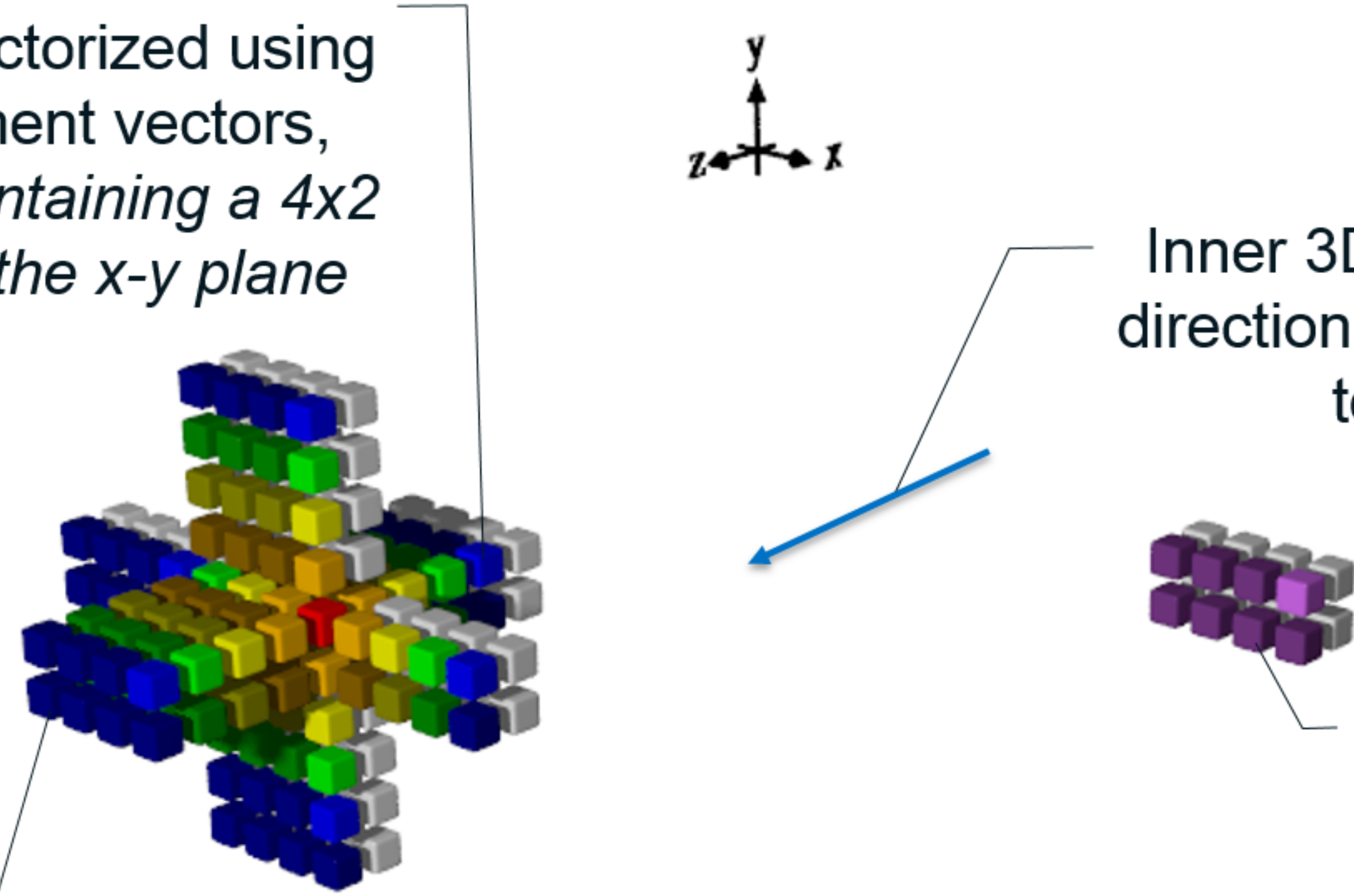
One 8-element (4x2) vector output per iteration

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

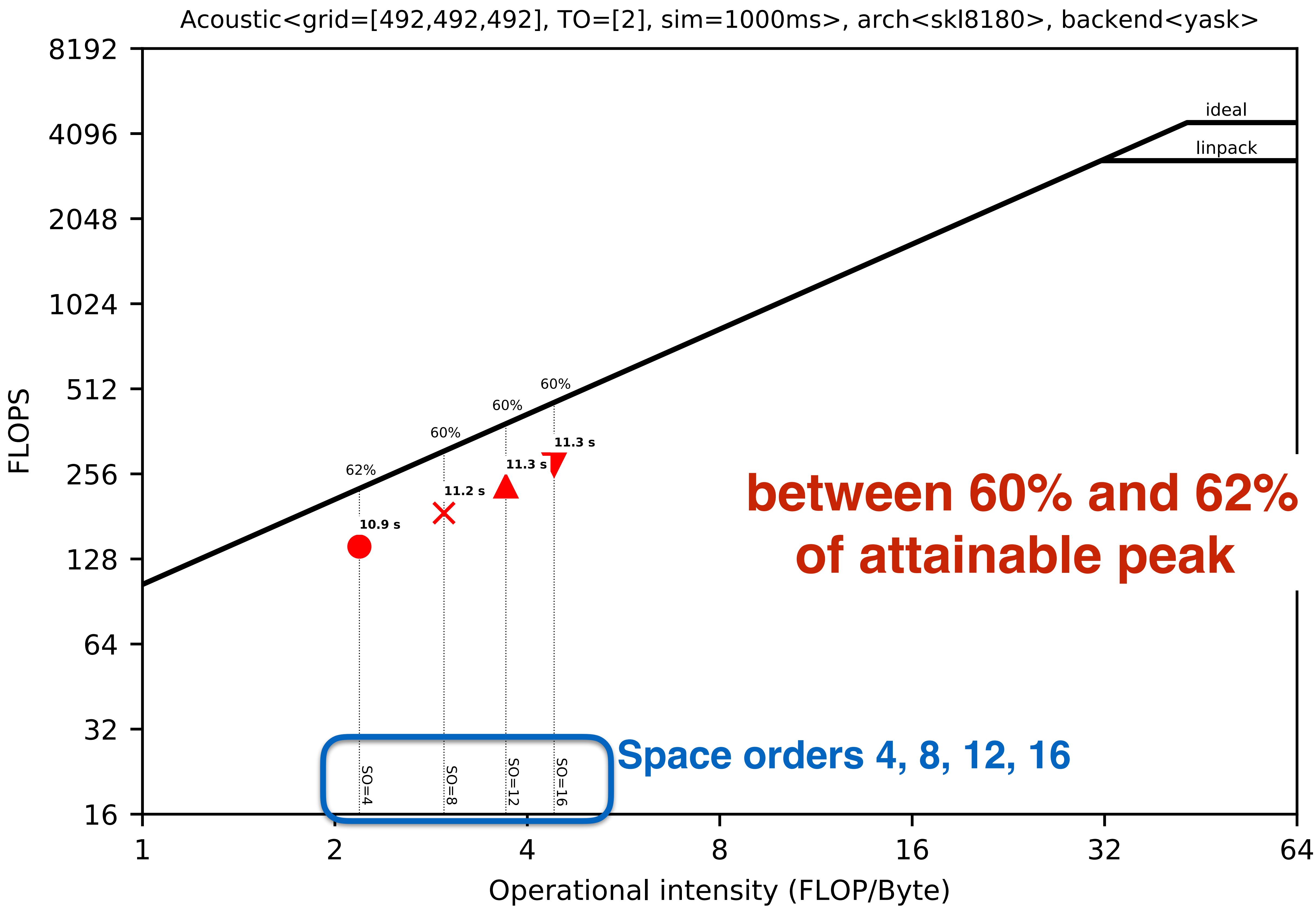
Vector folding via YASK (a Devito backend)

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

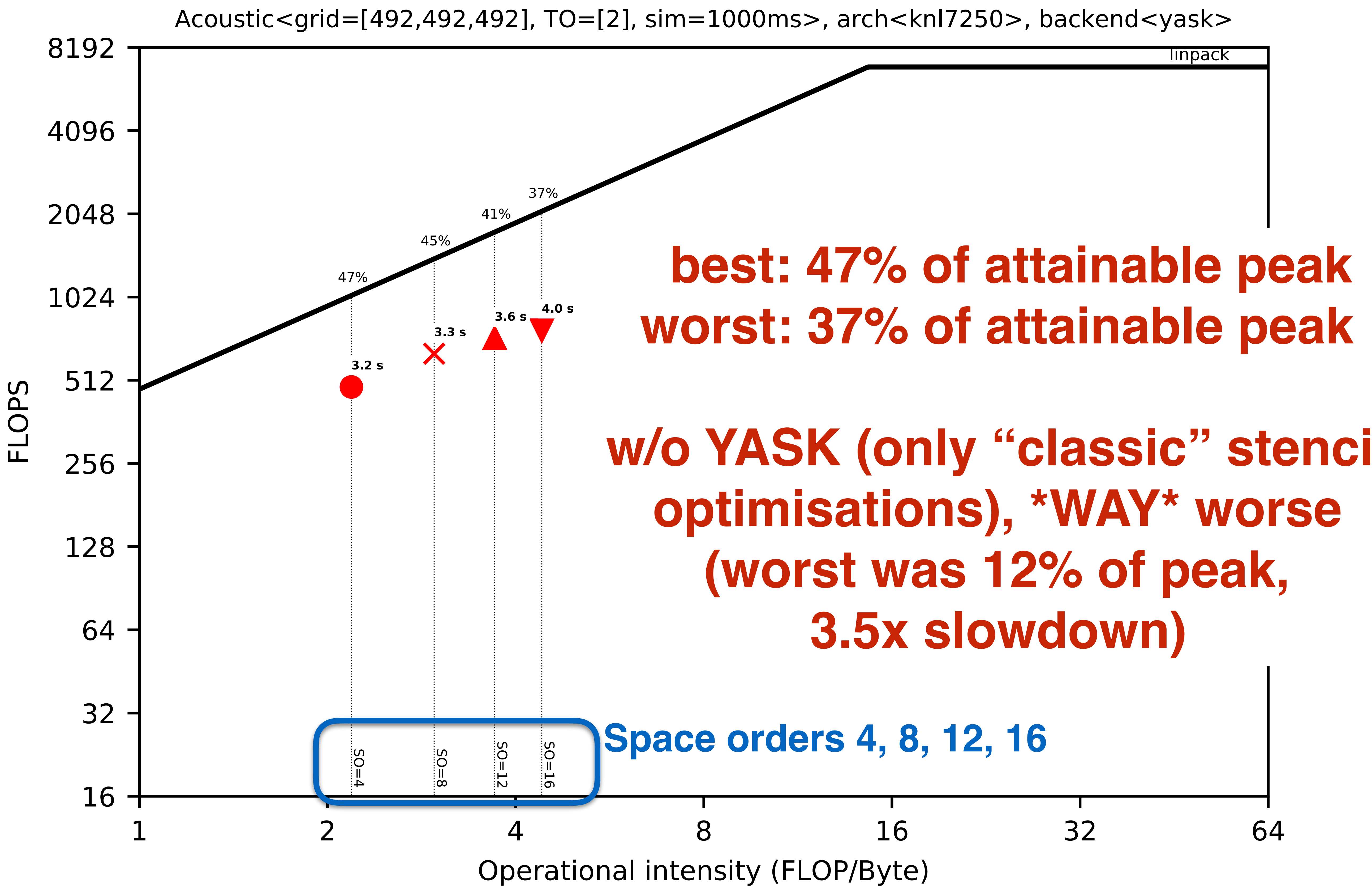


There's actually much more:
<u>multi-level tiling</u>
<u>software prefetching</u>
<u>temporal wavefront blocking</u>

Acoustic on Skylake 8180 with YASK



Acoustic on Xeon Phi 7250 with YASK



Legal Disclaimer & Optimization Notice

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

No computer system can be absolutely secure.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit http://www.intel.com/performance.

Cost reduction scenarios described are intended as examples of how a given Intel-based product, in the specified circumstances and configurations, may affect future costs and provide cost savings. Circumstances will vary. Intel does not guarantee any costs or cost reduction.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Statements in this document that refer to Intel's plans and expectations for the quarter, the year, and the future, are forward-looking statements that involve a number of risks and uncertainties. A detailed discussion of the factors that could affect Intel's results and plans is included in Intel's SEC filings, including the annual report on Form 10-K.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

Optimization notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. (Notice revision #20110804)

Conclusions and resources

- Devito: an efficient and sustainable ~~finite difference DSL~~ system to express and execute "numerical kernels"
- Driven by real-world seismic imaging, inspired by projects such as FEniCS/Firedrake
- 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

POLYHEDRAL??

Appendix

Experimentation details

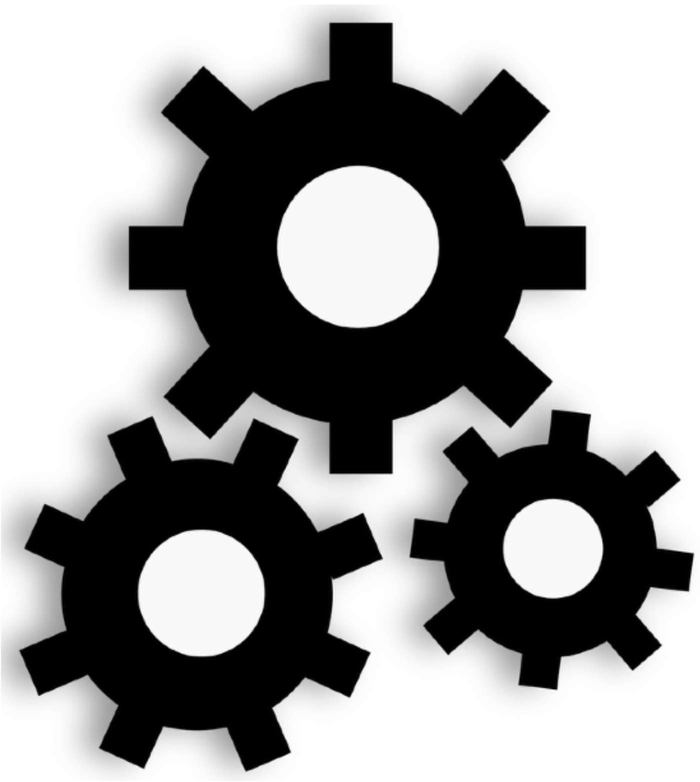
- Architectures
  - Intel® Xeon® Platinum 8180 Processor (“Skylake”, 28 cores)
  - Intel® XeonPhi® 7250 (68 cores)
    - Quadrant mode (still no support for NUMA)
    - Tried 1, 2, 4 threads per core. Shown best.

- 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)

Philosophy: optimizations at the RIGHT level of abstraction



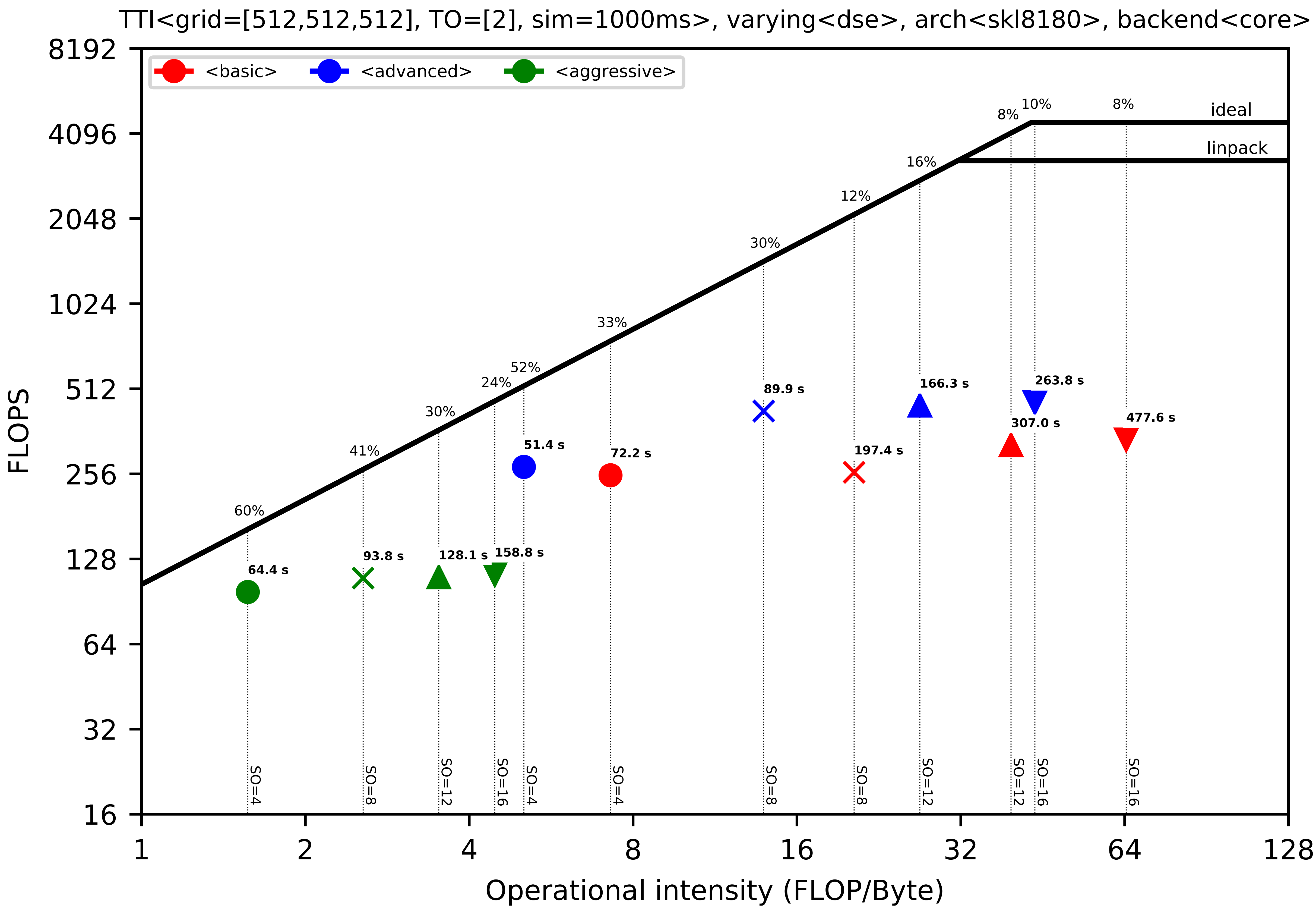
Example: optimizations for FLOPs reduction

```
Operator([eqn1, eqn2, …, eqn3])
```

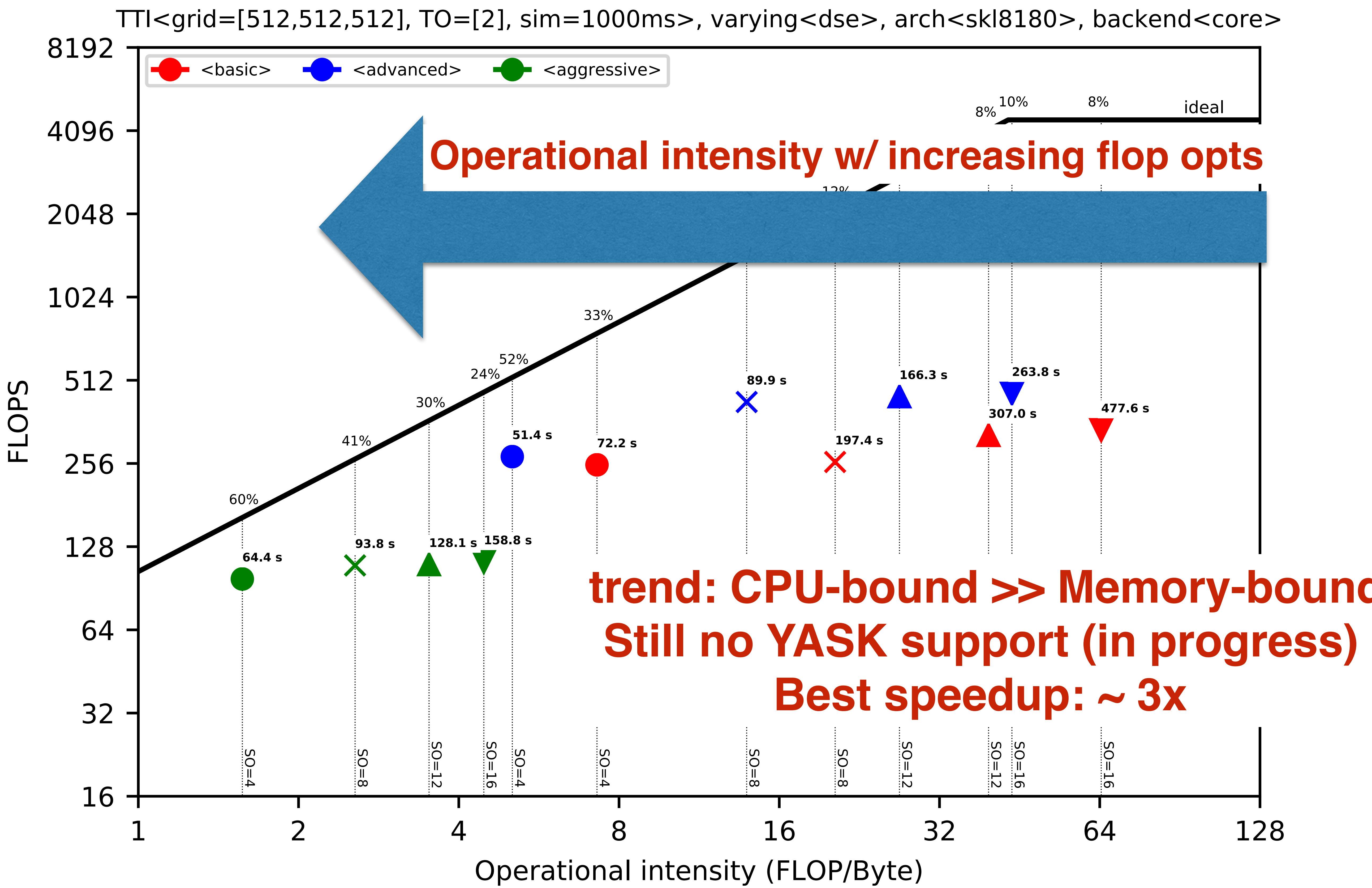
- Runtime constant propagation
- Equation clustering, **NOT** loop fusion
- Symbolic transformations to minimize the operation count of the equations

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

More aggressive FLOP reduction strategies



More aggressive FLOP reduction strategies

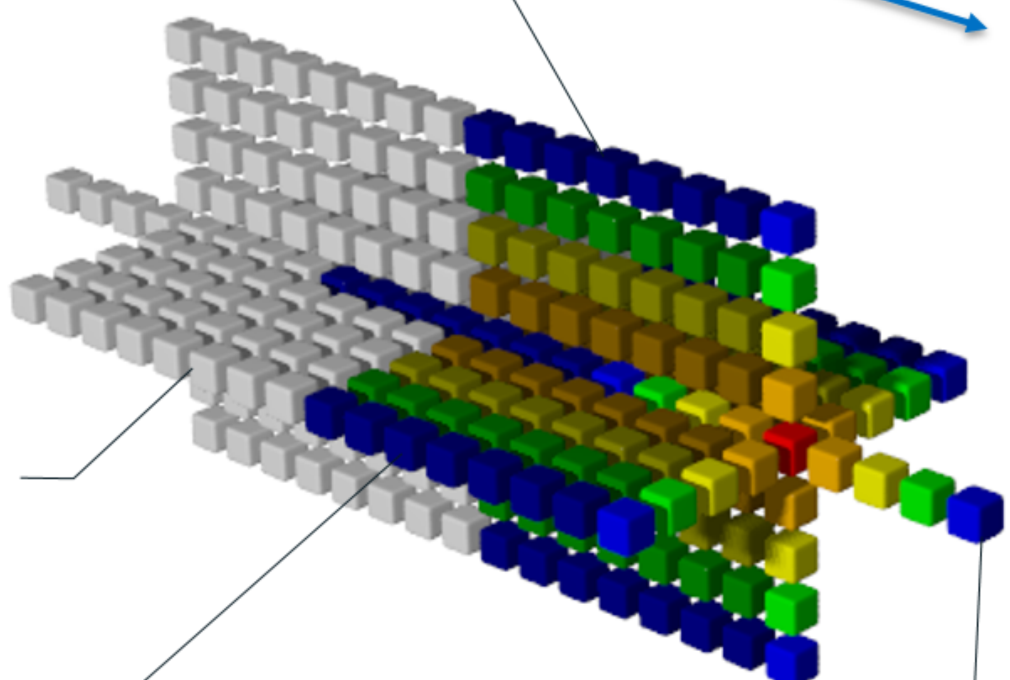


Beyond 1D vectorization …

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.