Automated Temporal Blocking in the Devito DSL and Compiler framework

***George Bisbas*[1,2]**, Fabio Luporini[3], Mathias Louboutin[4], Rhodri Nelson[2], Gerard Gorman[2,3], Paul H.J. Kelly[1]

[1]*Imperial College London, Dept. of Computing*
[2]*Imperial College London, Dept. of Earth Sciences and Engineering*
[3]*Devito Codes, UK*
[4]*Georgia Institute of Technology, Atlanta, USA*

MS243: Stencil Computation for Scientific Applications, SIAM CSE23, Amsterdam

Our motivation:

- **Motivation**: speed up computationally expensive scientific simulations involving the solution of PDEs modelling wave equations through explicit finite-difference methods

- **Cache blocking has been profitable for stencil computations**

- **Temporal cache blocking has been even more profitable!**
  - **Rarely applied in production**
  - **Challenging to apply**
  - **Few libraries, not straightforward**
  - **Why miss out?**

- Through Devito framework we offer the opportunity to go from textbook-like math to HPC temporal blocking code

- **Improved performance** without the fuss!

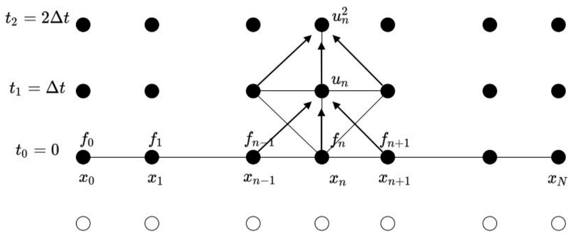
- **Q: Do I need to have CS skills to get perf?**

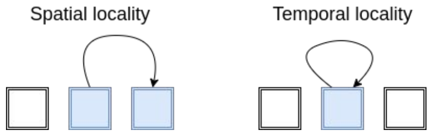
$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \quad 0 < x < L$$

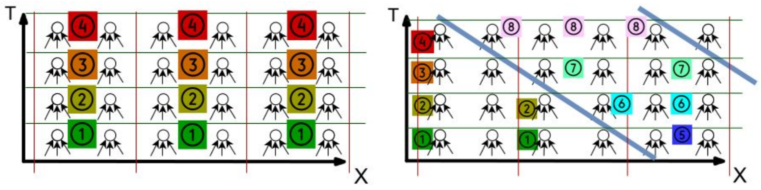
$$\text{Boundary Conditions}: \quad u(0,t) = 0; \quad u(L,t) = 0$$

$$\text{Initial Conditions}: \quad u(x,0) = f(x)$$

$$\frac{\partial u}{\partial t}(x,0) = g(x)$$







Scientific simulations are demanding

🚧 **Very complex to model** (complicated PDEs, BCs, external factors, complex geometries)

✅Software offering **high-level, high-productivity DSLs**
✅Let **domain experts** navigate their design space

🚧 **Resource-demanding** ($O(10^3)$ FLOPs per loop iteration, high memory pressure, 3D grids with > 10^9 grid points, often $O(10^3)$ time steps, inverse problems, ≈O(billions) TFLOPs. Which means days, or weeks, or months on supercomputers!

✅**Offer** automated optimisations and **efficient codegen for HPC** workloads
✅**Higher resolution in space and time** opens up compelling new applications
✅**Unlocks** ever-increasing application **value**

- Complex FD-stencil
- Just a part of these codes!
- No one wants to write it
- No one wants to optimise it
- No one wants to debug

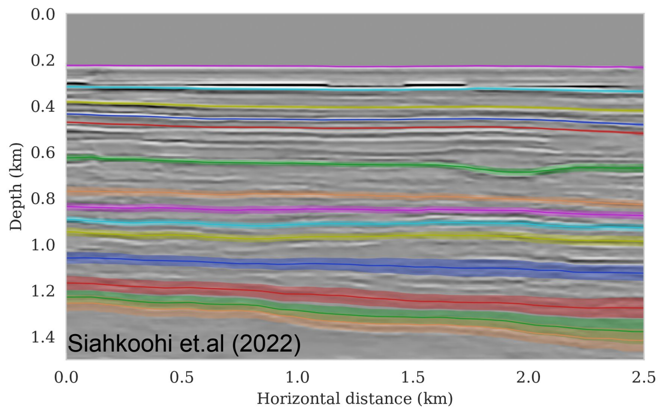
```
void kernel(…) {
  …
  <impenetrable code with
aggressive performance
optimizations, manually
applied, full-time human
resources, less
reproducibility, debugging
nightmares>
  …}
```

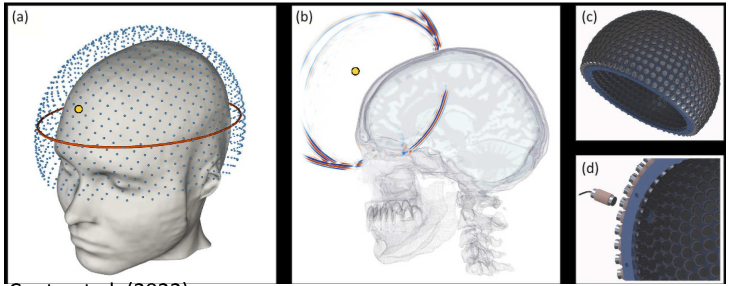
```
for (int time = time_m, t0 = (time)%(3), t1 = (time + 2)%(3), t2 = (time + 1)%(3);
time <= time_M; time += 1, t0 = (time)%(3), t1 = (time + 2)%(3), t2 = (time + 1)%(3))
  {
    /* Begin section0 */
    START_TIMER(section0)
    for (int x0_blk0 = x_m; x0_blk0 <= x_M; x0_blk0 += x0_blk0_size)
    {
      for (int y0_blk0 = y_m; y0_blk0 <= y_M; y0_blk0 += y0_blk0_size)
      {
        for (int x = x0_blk0; x <= MIN(x_M, x0_blk0 + x0_blk0_size - 1); x += 1)
        {
          for (int y = y0_blk0; y <= MIN(y_M, y0_blk0 + y0_blk0_size - 1); y += 1)
          {
            #pragma omp simd aligned(damp,u,vp:32)
            for (int z = z_m; z <= z_M; z += 1)
            {
              float r10 = 1.0F/(vp[x + 12][y + 12][z + 12]*vp[x + 12][y + 12][z +
12]);
              u[t2][x + 12][y + 12][z + 12] = (r10*(-r8*(-2.0F*u[t0][x + 12][y + 12][z
+ 12]) - r8*u[t1][x + 12][y + 12][z + 12]) + r9*damp[x + 12][y + 12][z + 12]*u[t0][x +
12][y + 12][z + 12] + 2.67222496e-7F*(-u[t0][x + 6][y + 12][z + 12] - u[t0][x + 12][y
+ 6][z + 12] - u[t0][x + 12][y + 12][z + 6] - u[t0][x + 12][y + 12][z + 18] - u[t0][x
+ 12][y + 18][z + 12] - u[t0][x + 18][y + 12][z + 12]) + 4.61760473e-6F*(u[t0][x + 7]
[y + 12][z + 12] + u[t0][x + 12][y + 7][z + 12] + u[t0][x + 12][y + 12][z + 7] + u[t0]
[x + 12][y + 12][z + 17] + u[t0][x + 12][y + 17][z + 12] + u[t0][x + 17][y + 12][z +
12]) + 3.96825406e-5F*(-u[t0][x + 8][y + 12][z + 12] - u[t0][x + 12][y + 8][z + 12] -
u[t0][x + 12][y + 12][z + 8] - u[t0][x + 12][y + 12][z + 16] - u[t0][x + 12][y + 16][z
+ 12] - u[t0][x + 16][y + 12][z + 12]) + 2.35155796e-4F*(u[t0][x + 9][y + 12][z + 12]
+ u[t0][x + 12][y + 9][z + 12] + u[t0][x + 12][y + 12][z + 9] + u[t0][x + 12][y + 12]
[z + 15] + u[t0][x + 12][y + 15][z + 12] + u[t0][x + 15][y + 12][z + 12]) +
1.19047622e-3F*(-u[t0][x + 10][y + 12][z + 12] - u[t0][x + 12][y + 10][z + 12] - u[t0]
[x + 12][y + 12][z + 10] - u[t0][x + 12][y + 12][z + 14] - u[t0][x + 12][y + 14][z +
12] - u[t0][x + 14][y + 12][z + 12]) + 7.6190478e-3F*(u[t0][x + 11][y + 12][z + 12] +
u[t0][x + 12][y + 11][z + 12] + u[t0][x + 12][y + 12][z + 11] + u[t0][x + 12][y + 12]
[z + 13] + u[t0][x + 12][y + 13][z + 12] + u[t0][x + 13][y + 12][z + 12]) -
3.97703713e-2F*u[t0][x + 12][y + 12][z + 12])/(r10*r8 + r9*damp[x + 12][y + 12][z +
12]);
            }
          }
        }
      }
    }
    STOP_TIMER(section0,timers)
    /* End section0 */
```

Introducing Devito

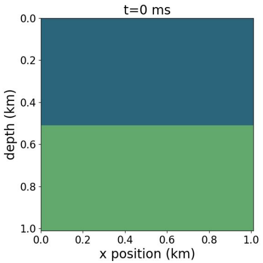
- **Devito is a DSL and compiler framework** for finite difference and stencil computations
- **Solving PDEs** using the **finite-difference method for structured grids** (but not limited to this!)
- Users model in the high-level DSL using symbolic math abstraction, and the compiler auto-generates HPC optimized code
- Inter(-national, -institutional,-disciplinary), lots of users from academia and industry
- Real-world problem simulations! (CFD, seismic/medical imaging, finance, tsunamis)

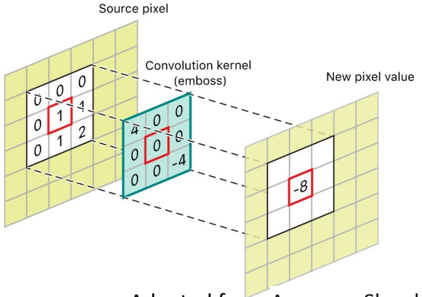
CI-core passing | CI-mpi passing | CI-gpu passing | codecov 88% | chat on slack | benchmarked by asv | pypi package 4.8.0 | launch binder | docker images





Cueto et.al. (2022)



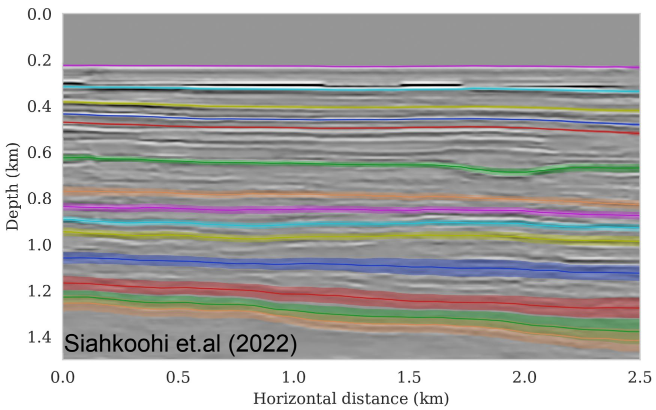


Adapted from Aryaman Sharda

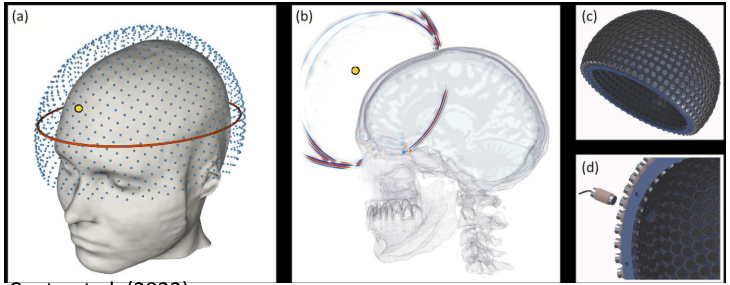
Introducing Devito

- **Open source** - MIT lic. - Try now! https://github.com/devitocodes/devito
- **Compose with** packages from the Python ecosystem (e.g. PyTorch, NumPy, Dask, TensorFlow)
- Best practices in **software engineering**: extensive software testing, code verification, CI/CD, regression tests, documentation, tutorials and PR code review
- Actual compiler technology (not a S2S translator or templates!)

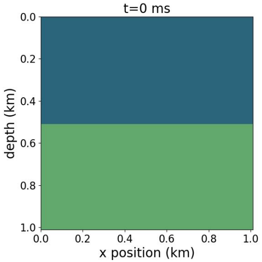
CI-core passing | CI-mpi passing | CI-gpu passing | codecov 88% | chat on slack | benchmarked by asv | pypi package 4.8.0
launch binder | docker images

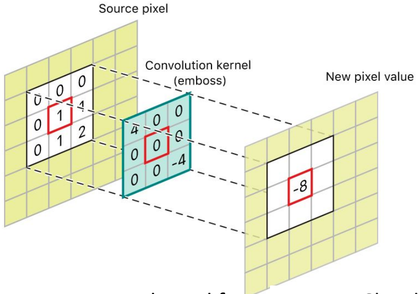


Siahkoohi et.al (2022)



Cueto et.al. (2022)

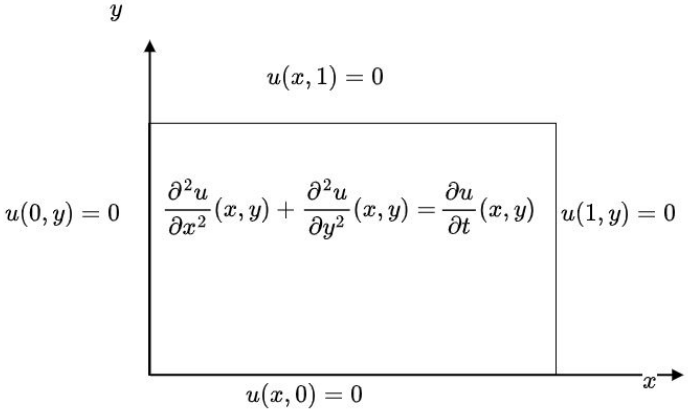




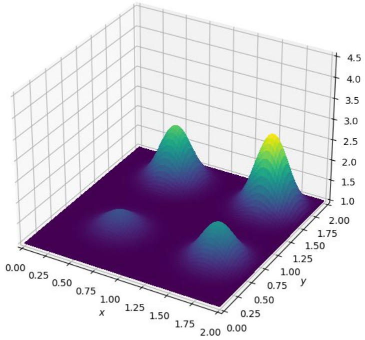
Adapted from Aryaman Sharda

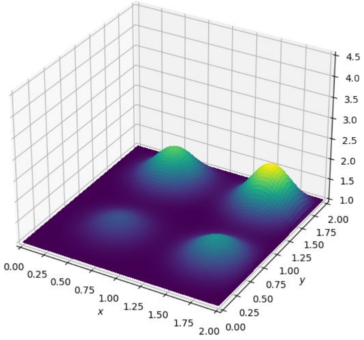
An example from textbook maths to via Devito DSL

2D Heat diffusion modelling



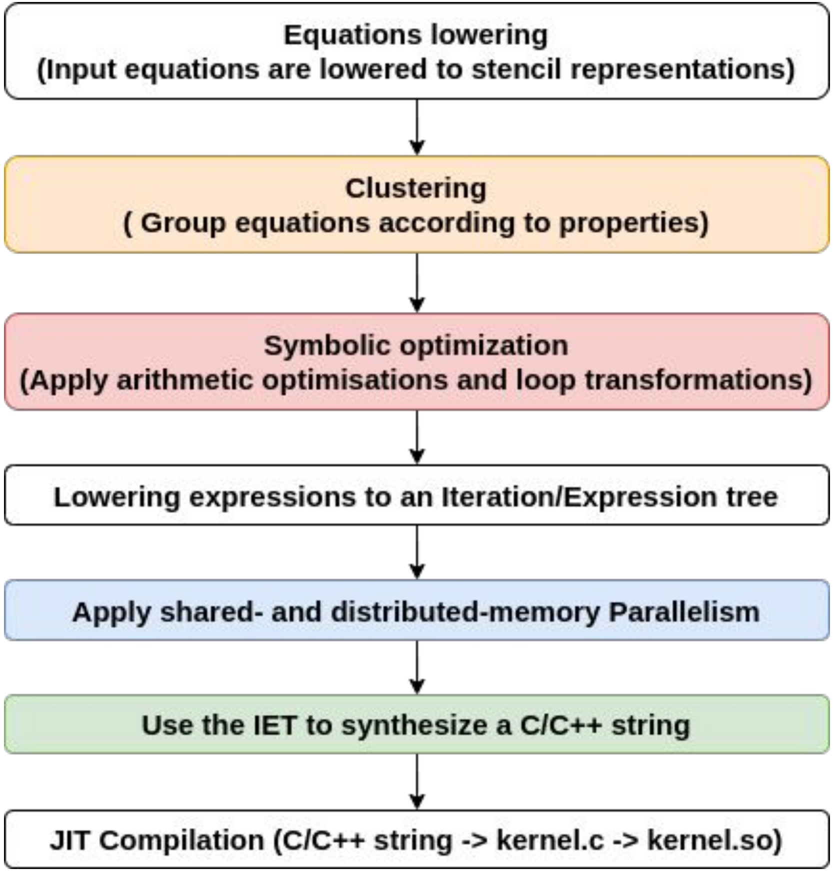
$$\frac{\partial^2 u}{\partial x^2}(x,y) + \frac{\partial^2 u}{\partial y^2}(x,y) = \frac{\partial u}{\partial t}(x,y)$$

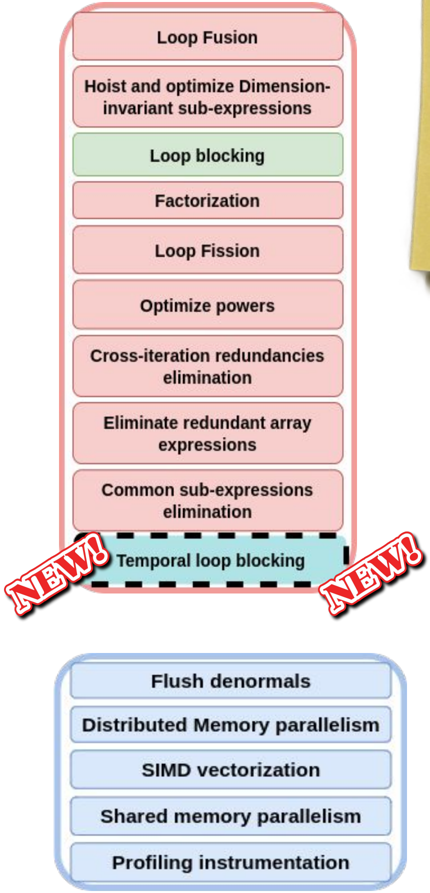




```
from devito import Eq, Grid, TimeFunction, Operator, solve

# Define a structured grid
nx, ny = 10, 10
grid = Grid(shape=(10, 10))

# Define a field on the structured grid
u = TimeFunction(name='u', grid=grid, space_order=2)

# Define a forward time-stepping symbolic equation
eqn = Eq(u.dt, u.laplace)
eqns = [Eq(u.forward, solve(eqn, u.forward))]

# Define boundary conditions
x, y = grid.dimensions
t = grid.stepping_dim

bc_left = Eq(u[t + 1, 0, y], 0.)
bc_right = Eq(u[t + 1, nx-1, y], 0.)
bc_top = Eq(u[t + 1, x, ny-1], 0.)
bc_bottom = Eq(u[t + 1, x, 0], 0.)

eqns += [bc_left, bc_bottom, bc_right, bc_top]
op = Operator(eqns)

# Compute for 3 timesteps
op.apply(time_M=3, dt=0.1)
```

Devito’s compiler optimisations overview





+ advanced combinations of them!
+ heuristics to tune them more!

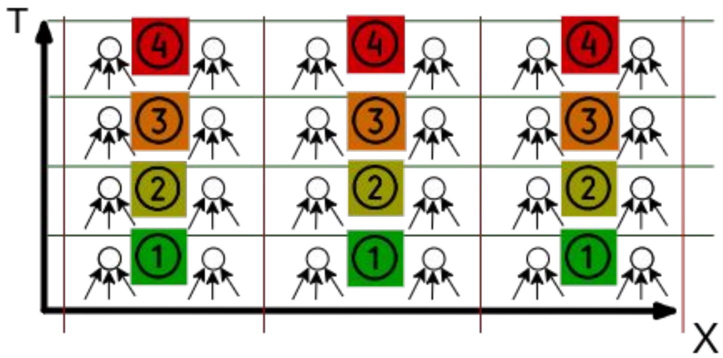
Write once,

Run everywhere!

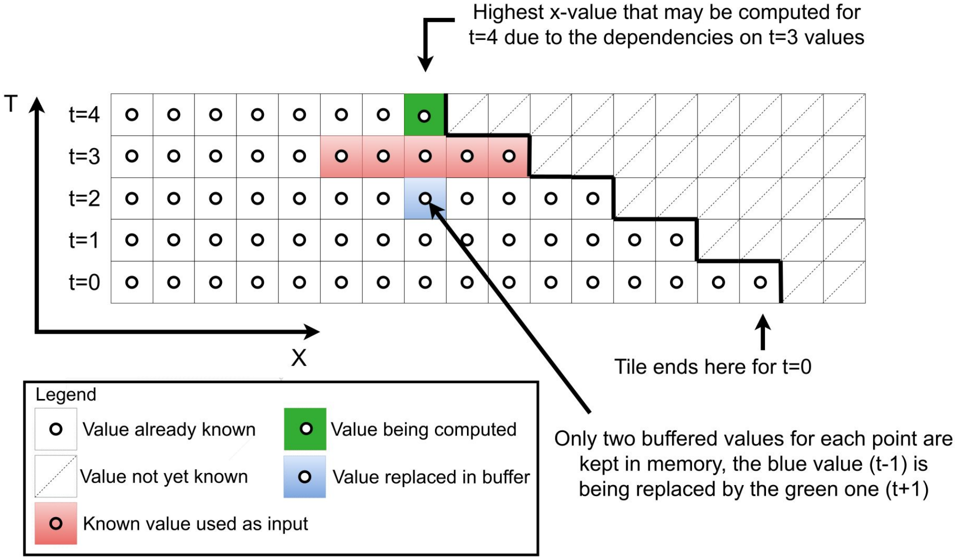
- Serial C/CPP code
- OpenMP parallel code
- MPI (+ OpenMP)
- OpenMP 5 GPU offloading via Clang
- OpenACC GPU offloading

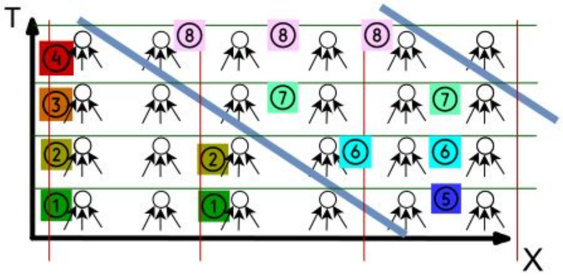
Standard loop blocking (enhancing spatial locality only!)

```
 for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M; time += 1, t0 =
(time)%(2), t1 = (time + 1)%(2))
  {
    for (int x0_blk0 = x_m; x0_blk0 <= x_M; x0_blk0 += x0_blk0_size)
    {
      for (int y0_blk0 = y_m; y0_blk0 <= y_M; y0_blk0 += y0_blk0_size)
      {
        for (int x = x0_blk0; x <= x0_blk0 + x0_blk0_size - 1; x += 1)
        {
          for (int y = y0_blk0; y <= y0_blk0 + y0_blk0_size - 1; y += 1)
          {
            for (int z = z_m; z <= z_M; z += 1)
            {
              float r4 = -2.0F*u[t0][x + 2][y + 2][z + 2];
              u[t1][x + 2][y + 2][z + 2] = dt*(r0*u[t0][x + 2][y + 2][z + 2] + a*(r1*r4 + r1*u[t0][x + 1][y + 2][z
+ 2] + r1*u[t0][x + 3][y + 2][z + 2] + r2*r4 + r2*u[t0][x + 2][y + 1][z + 2] + r2*u[t0][x + 2][y + 3][z + 2]
+ r3*r4 + r3*u[t0][x + 2][y + 2][z + 1] + r3*u[t0][x + 2][y + 2][z + 3]) + 1.0e-1F);
            }
```

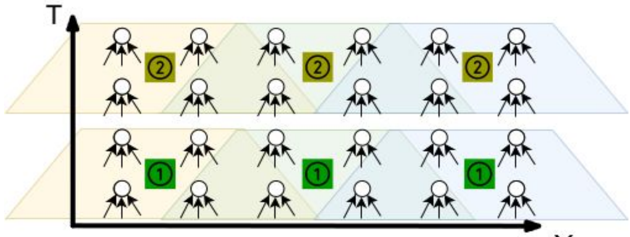


Wavefront temporal blocking

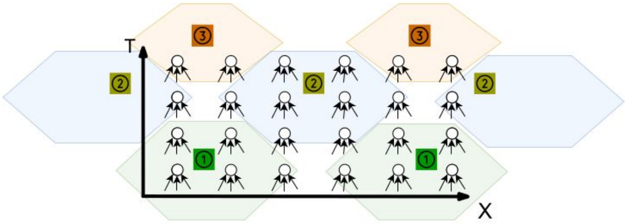




Other temporal blocking variants

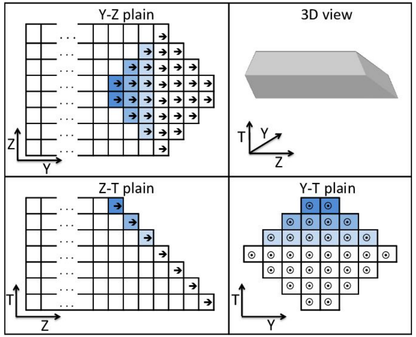


Overlapped temporal blocking

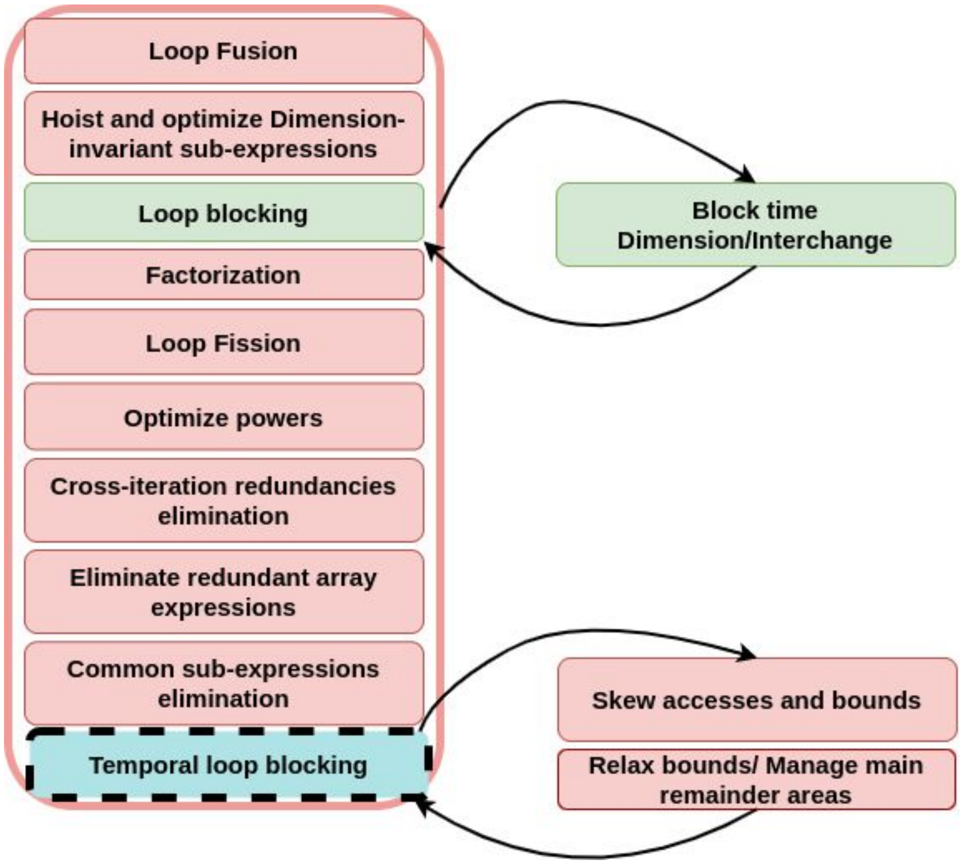


Hexagonal/Diamond temporal blocking

WDT [Malas et.al.]



Synthesizing Temporal blocking in the Devito optimisation pipeline

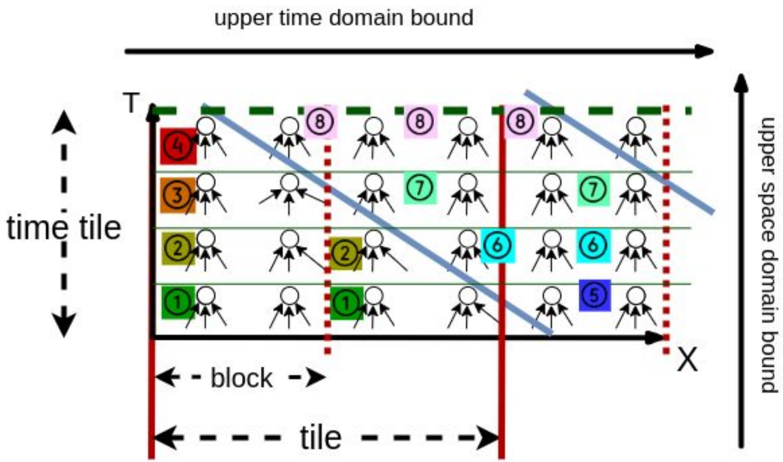


1. Tweak blocking pass to **produce an additional time loop** + space loops, **sort** them accordingly

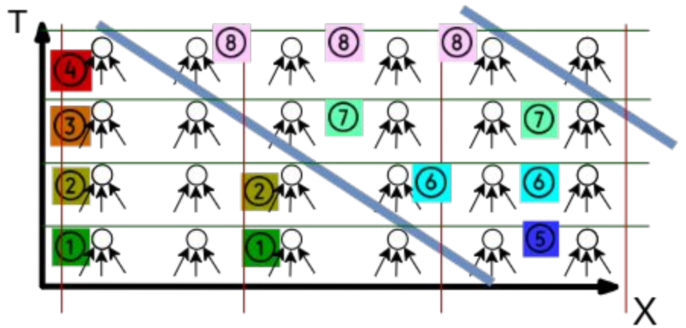
2. **Skew** time accesses and loop bounds

3. Take care of **main/remainder** areas, **time-space diagonals, domain bounds**

✅ Works in tandem with all other Devito opts!



Temporal loop blocking (Wavefront variant)



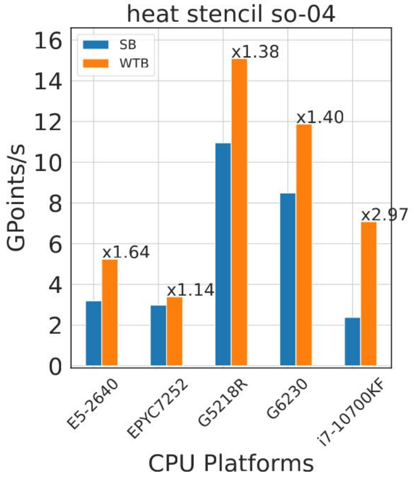
1

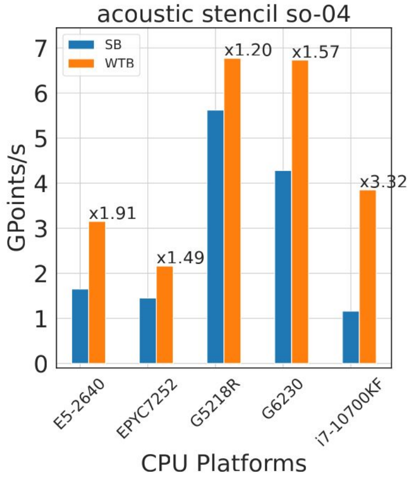
```
for (int time0_blk0 = time_m; time0_blk0 <= time_M; time0_blk0 += time0_blk0_size)
 {
   for (int x0_blk0 = x_m; x0_blk0 <= time_M - time_m + x_M; x0_blk0 += x0_blk0_size)
   {
     for (int y0_blk0 = y_m; y0_blk0 <= time_M - time_m + y_M; y0_blk0 += y0_blk0_size)
     {
       for (int time = time0_blk0, t0 = (time)%(2), t1 = (time + 1)%(2); time <= MIN(time0_blk0 + time0_blk0_size - 1, time_M); time += 1, t0 =
(time)%(2), t1 = (time + 1)%(2))
       {
         for (int x0_blk1 = MAX(x0_blk0, time + x_m); x0_blk1 <= MIN(x0_blk0 + x0_blk0_size - 1, time + x_M); x0_blk1 += x0_blk1_size)
         {
           for (int y0_blk1 = MAX(y0_blk0, time + y_m); y0_blk1 <= MIN(y0_blk0 + y0_blk0_size - 1, time + y_M); y0_blk1 += y0_blk1_size)
           {
             for (int x = x0_blk1; x <= MIN(MIN(x0_blk0 + x0_blk0_size - 1, time + x_M), x0_blk1 + x0_blk1_size - 1); x += 1)
             {
               for (int y = y0_blk1; y <= MIN(MIN(y0_blk0 + y0_blk0_size - 1, time + y_M), y0_blk1 + y0_blk1_size - 1); y += 1)
               {
                 for (int z = z_m; z <= z_M; z += 1)
                   {
                     float r4 = -2.0F*u[t0][-time + x + 2][-time + y + 2][z + 2];
                 u[t1][-time + x + 2][-time + y + 2][z + 2] = dt*(r0*u[t0][-time + x + 2][-time + y + 2][z + 2] + a*(r1*r4 + r1*u[t0][-time + x + 1][-time + y +
2][z + 2] + r1*u[t0][-time + x + 3][-time + y + 2][z + 2] + r2*r4 + r2*u[t0][-time + x + 2][-time + y + 1][z + 2] + r2*u[t0][-time + x + 2][-time + y + 3][z
+ 2] + r3*r4 + r3*u[t0][-time + x + 2][-time + y + 2][z + 1] + r3*u[t0][-time + x + 2][-time + y + 2][z + 3]) + 1.0e-1F);}
```

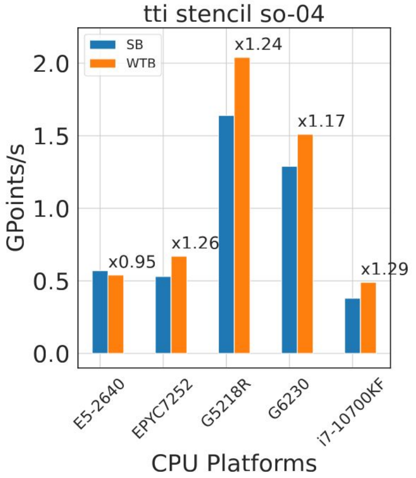
3

2

Experimental evaluation, low discretization orders







CPU characterstics					
	i7-10700KF	Gold 5218R	Gold 6230	EPYC7742	E5-2640
CPU(s)	16	80	40	256	32
Thread(s) per core:	2	2	1	2	2
Core(s) per socket:	8	20	20	64	8
Socket(s):	1	2	2	2	2
NUMA node(s):	1	2	2	8	2
L1d cache:	256KiB	1.3MiB	32KiB	32KiB	512KiB
L1i cache:	256KiB	1.3MiB	32KiB	32KiB	512KiB
L2 cache:	2MiB	40 MiB	1MiB	512KiB	4MiB
L3 cache:	16MiB	55 MiB	27.5MiB	16MiB	40MiB

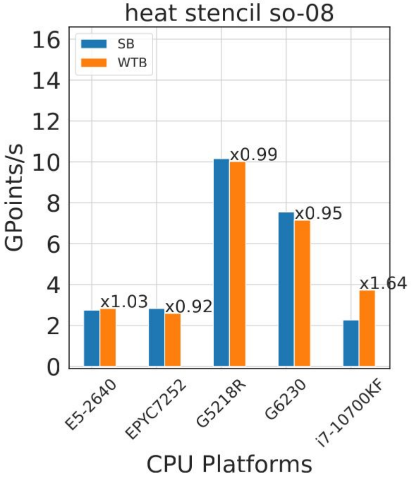
Table 4.1.: Characateristics of CPU platforms used for benchmarking

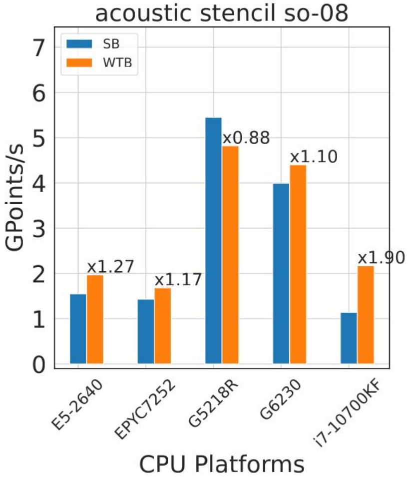
shape 1024 1024 1024, timesteps 512

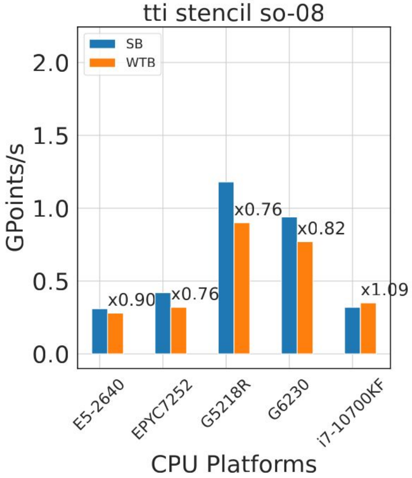
- Kernels are flop-optimized through Devito.
- Gpts/s aka Gcells/s: time to solution metric in stencil computations
- (!) High Gflops/s do not guarantee a faster solution.
- OMP thread pinning, SIMD
- Aggressive auto-tuning

Rooflines available

Experimental evaluation, higher discretization orders







CPU characterstics					
	i7-10700KF	Gold 5218R	Gold 6230	EPYC7742	E5-2640
CPU(s)	16	80	40	256	32
Thread(s) per core:	2	2	1	2	2
Core(s) per socket:	8	20	20	64	8
Socket(s):	1	2	2	2	2
NUMA node(s):	1	2	2	8	2
L1d cache:	256KiB	1.3MiB	32KiB	32KiB	512KiB
L1i cache:	256KiB	1.3MiB	32KiB	32KiB	512KiB
L2 cache:	2MiB	40 MiB	1MiB	512KiB	4MiB
L3 cache:	16MiB	55 MiB	27.5MiB	16MiB	40MiB

Table 4.1.: Characateristics of CPU platforms used for benchmarking

shape 1024 1024 1024, timesteps 512

- Kernels are flop-optimized through Devito.
- Gpts/s aka Gcells/s: time to solution metric in stencil computations
- (!) High Gflops/s do not guarantee a faster solution.
- OMP thread pinning, SIMD
- Aggressive auto-tuning

Rooflines available

Conclusions

- We presented **Devito, a DSL and compiler framework** for explicit finite difference schemes for **solving PDEs** using the **FD method for structured grids (**but not limited to them!)
- The Devito Compiler supports a great variety of optimizations for stencil kernels, **we aim to add another one, to enhance temporal data reuse**
- Promising performance gains of ranging from 3x on low order (4) to 1.6x and 1.9x on higher order (8) problems

Current WIP

- Full integration to DSL (currently in a branch/fork)
- User will get out-of the box time tiled code for all PDEs!

Future plans

- Challenges with interpolations
- Automate more TB schemes
- Add MPI-aware scheme
- Extend TB to GPUs
- Performance for higher-order stencils

- Website
- Slack
- Code

DEVITOPROJECT

Join us, use Devito, work with us!

Imperial College London

EPSRC Engineering and Physical Sciences Research Council

HiPEDS

DEVITO

References

- Bisbas G., Luporini F., Louboutin M., Nelson R., Gorman G. J. and Kelly P. H. J., "Temporal blocking of finite-difference stencil operators with sparse “off-the-grid” sources," *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Portland, OR, USA, 2021, pp. 497-506, doi: 10.1109/IPDPS49936.2021.00058.

- Luporini, F., Lange, M., Louboutin, M., Kukreja, N., Hückelheim, J., Yount, C., Witte, P.A., Kelly, P.H., Gorman, G., & Herrmann, F. (2020). Architecture and Performance of Devito, a System for Automated Stencil Computation. ACM Transactions on Mathematical Software (TOMS), 46, 1 - 28.

- Louboutin, M., M., Lange, F., Luporini, N., Kukreja, P. A., Witte, F. J., Herrmann, P., Velesko, and G. J., Gorman. "Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration".Geoscientific Model Development 12, no.3 (2019): 1165–1187.

- Yount, C., & Duran, A. (2016). Effective Use of Large High-Bandwidth Memory Caches in HPC Stencil Computation via Temporal Wave-Front Tiling. (2016) 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), 65-75.

Appendix

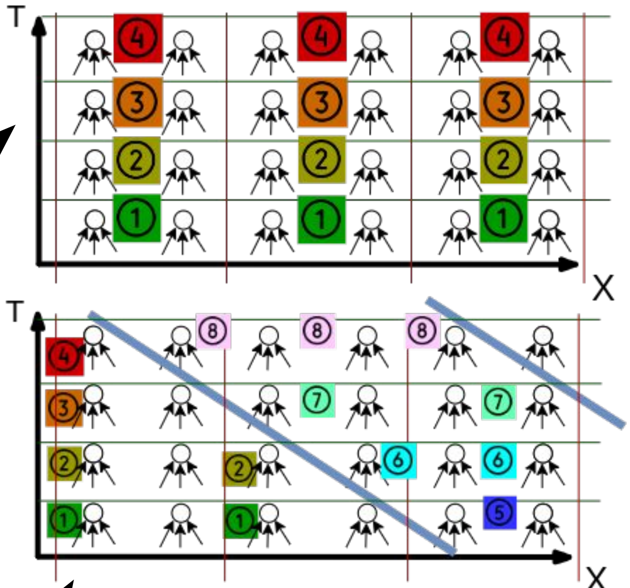
…And works in tandem with all other Devito optimizations!

```
for (int time0_blk0 = time_m; time0_blk0 <= time_M; time0_blk0 += time0_blk0_size)
 {
  for (int x0_blk0 = x_m; x0_blk0 <= time_M - time_m + x_M; x0_blk0 += x0_blk0_size)
  {
   for (int y0_blk0 = y_m; y0_blk0 <= time_M - time_m + y_M; y0_blk0 += y0_blk0_size)
   {
    for (int time = time0_blk0, t0 = (time)%(2), t1 = (time + 1)%(2); time <= MIN(time0_blk0 + time0_blk0_size - 1, time_M); time += 1, t0 = (time)%(2), t1 = (time + 1)%(2))
    {
     #pragma omp parallel num_threads(nthreads)
     {
      #pragma omp for collapse(2) schedule(dynamic,1)
      for (int x0_blk1 = MAX(x0_blk0, time + x_m); x0_blk1 <= MIN(x0_blk0 + x0_blk0_size - 1, time + x_M); x0_blk1 += x0_blk1_size)
      {
       for (int y0_blk1 = MAX(y0_blk0, time + y_m); y0_blk1 <= MIN(y0_blk0 + y0_blk0_size - 1, time + y_M); y0_blk1 += y0_blk1_size)
       {
        for (int x = x0_blk1; x <= MIN(MIN(x0_blk0 + x0_blk0_size - 1, time + x_M), x0_blk1 + x0_blk1_size - 1); x += 1)
        {
         for (int y = y0_blk1; y <= MIN(MIN(y0_blk0 + y0_blk0_size - 1, time + y_M), y0_blk1 + y0_blk1_size - 1); y += 1)
         {
          #pragma omp simd aligned(u:32)
          for (int z = z_m; z <= z_M; z += 1)
           {
           float r4 = -2.0F*u[t0][-time + x + 2][-time + y + 2][z + 2];
          u[t1][-time + x + 2][-time + y + 2][z + 2] = dt*(r0*u[t0][-time + x + 2][-time + y + 2][z + 2] + a*(r1*r4 + r1*u[t0][-time + x + 1][-time + y + 2][z + 2] + r1*u[t0][-time + x + 3][-time + y + 2][z + 2] + r2*r4 + r2*u[t0][-time + x + 2][-time + y + 1][z + 2] + r2*u[t0][-time + x + 2][-time + y + 3][z
```

Cache blocking optimizations in Devito

- Loop Fusion
- Hoist and optimize Dimension-invariant sub-expressions
- Loop blocking
- Factorization
- Loop Fission
- Optimize powers
- Cross-iteration redundancies elimination
- Eliminate redundant array expressions
- Common sub-expressions elimination
- Temporal loop blocking

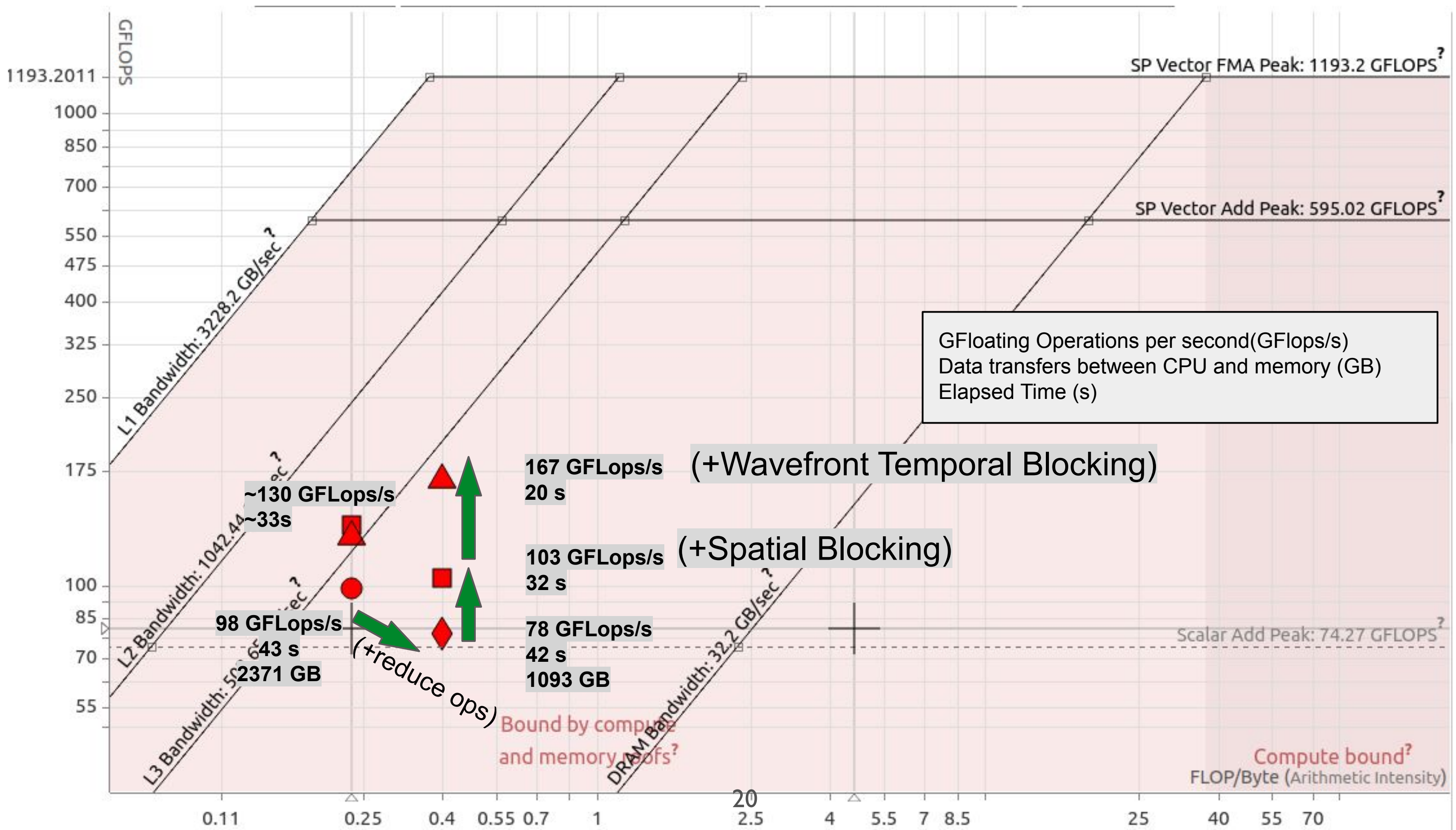
NEW!



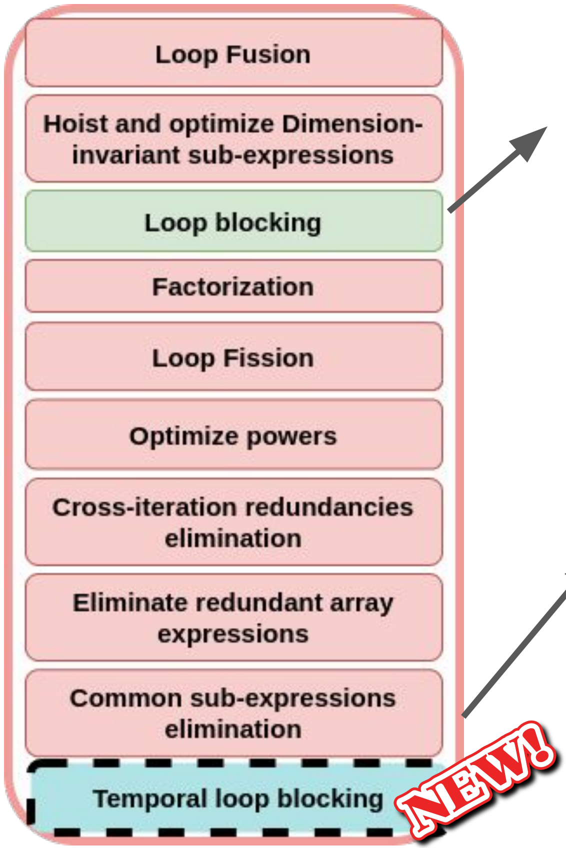
- **Cache blocking** is a performance optimisation to enhance spatial and temporal locality.
- **Spatial** locality: Exploit data closely present in memory
- **Temporal** locality: Exploit same data in a short time-span
- **Can yield large-performance gains but its error-prone** and **tedious** to **apply by hand!**
- **Advanced cache blocking** is **even more challenging** (hierarchical loop blocking, temporal loop blocking).
- Our **motivation** is to **automate** the application of **temporal blocking through compiler passes.**
- **No changes whatsoever in user-code!**

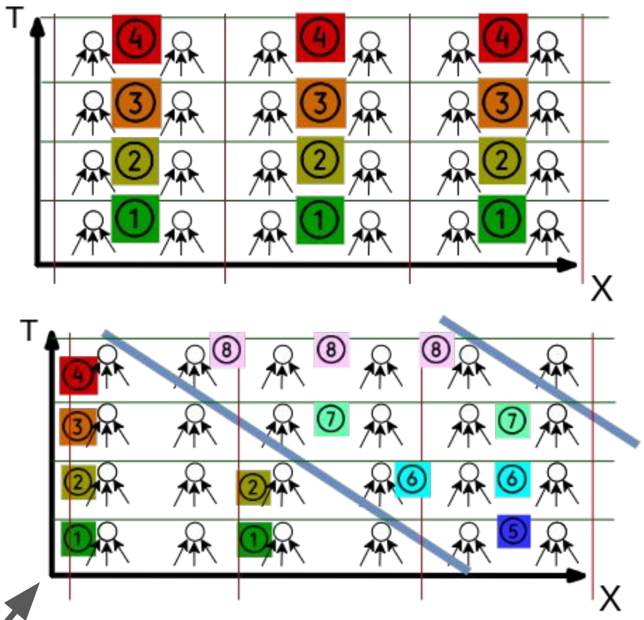
Works in tandem with all other Devito opts!

Cache-aware Roofline model :Laplacian, discretization order 8, 512^3 grid points, 512ms (Intel Skylake)



Cache blocking optimizations in Devito





For a given amount of available data, try to compute whatever can be computed!

Temporal blocking Methodology:

Split wavefront temporal blocking in subpasses

1. A pass where time loop is blocked
2. A pass where accesses are skewed
3. A pass where space-time wave bounds are computed

1. A pass where time loop is blocked + another space loop level added

- Loop Fusion
- Hoist and optimize Dimension-invariant sub-expressions
- Loop blocking
- Factorization
- Loop Fission
- Optimize powers
- Cross-iteration redundancies elimination
- Eliminate redundant array expressions
- Common sub-expressions elimination
- Temporal loop blocking

```
for (int time0_blk0 = time_m; time0_blk0 <= time_M; time0_blk0 += time0_blk0_size)
  {
    for (int x0_blk0 = x_m; x0_blk0 <= x_M; x0_blk0 += x0_blk0_size)
    {
      for (int y0_blk0 = y_m; y0_blk0 <= y_M; y0_blk0 += y0_blk0_size)
      {
        for (int time = time0_blk0, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time0_blk0 + time0_blk0_size
- 1; time += 1, t0 = (time)%(2), t1 = (time + 1)%(2))
        {
          for (int x0_blk1 = x0_blk0; x0_blk1 <= x0_blk0 + x0_blk0_size - 1; x0_blk1 += x0_blk1_size)
          {
            for (int y0_blk1 = y0_blk0; y0_blk1 <= y0_blk0 + y0_blk0_size - 1; y0_blk1 += y0_blk1_size)
            {
              for (int x = x0_blk1; x <= x0_blk1 + x0_blk1_size - 1; x += 1)
              {
                for (int y = y0_blk1; y <= y0_blk1 + y0_blk1_size - 1; y += 1)
                {
                  for (int z = z_m; z <= z_M; z += 1)
                  {
              float r4 = -2.0F*u[t0][x + 2][y + 2][z + 2];
              u[t1][x + 2][y + 2][z + 2] = dt*(r0*u[t0][x + 2][y + 2][z + 2] + a*(r1*r4 + r1*u[t0][x + 1][y + 2][z + 2] +
r1*u[t0][x + 3][y + 2][z + 2] + r2*r4 + r2*u[t0][x + 2][y + 1][z + 2] + r2*u[t0][x + 2][y + 3][z + 2] + r3*r4 +
r3*u[t0][x + 2][y + 2][z + 1] + r3*u[t0][x + 2][y + 2][z + 3]) + 1.0e-1F);
                  }
```

2. A pass where accesses are skewed

- Loop Fusion
- Hoist and optimize Dimension-invariant sub-expressions
- Loop blocking
- Factorization
- Loop Fission
- Optimize powers
- Cross-iteration redundancies elimination
- Eliminate redundant array expressions
- Common sub-expressions elimination
- Temporal loop blocking

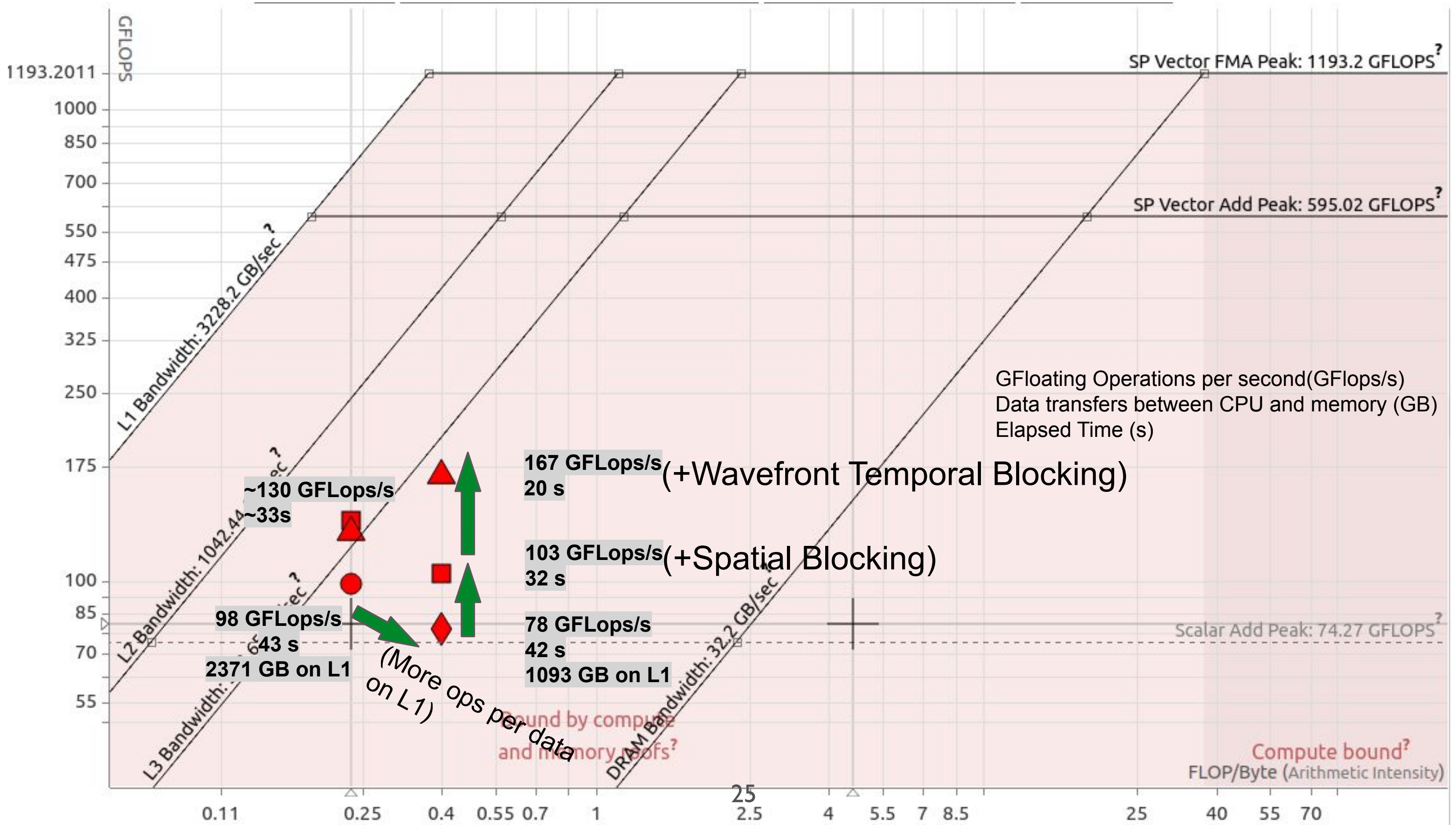
```
for (int time0_blk0 = time_m; time0_blk0 <= time_M; time0_blk0 += time0_blk0_size)
 {
  for (int x0_blk0 = x_m; x0_blk0 <= x_M; x0_blk0 += x0_blk0_size)
  {
   for (int y0_blk0 = y_m; y0_blk0 <= y_M; y0_blk0 += y0_blk0_size)
   {
    for (int time = time0_blk0, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time0_blk0 + time0_blk0_size
- 1; time += 1, t0 = (time)%(2), t1 = (time + 1)%(2))
    {
     for (int x0_blk1 = x0_blk0; x0_blk1 <= x0_blk0 + x0_blk0_size - 1; x0_blk1 += x0_blk1_size)
     {
      for (int y0_blk1 = y0_blk0; y0_blk1 <= y0_blk0 + y0_blk0_size - 1; y0_blk1 += y0_blk1_size)
      {
       for (int x = x0_blk1; x <= x0_blk1 + x0_blk1_size - 1; x += 1)
       {
        for (int y = y0_blk1; y <= y0_blk1 + y0_blk1_size - 1; y += 1)
        {
         for (int z = z_m; z <= z_M; z += 1)
         {
           {
            float r4 = -2.0F*u[t0][-time + x + 2][-time + y + 2][z + 2];
          u[t1][-time + x + 2][-time + y + 2][z + 2] = dt*(r0*u[t0][-time + x + 2][-time + y + 2][z + 2] +
a*(r1*r4 + r1*u[t0][-time + x + 1][-time + y + 2][z + 2] + r1*u[t0][-time + x + 3][-time + y + 2][z + 2] + r2*r4 +
r2*u[t0][-time + x + 2][-time + y + 1][z + 2] + r2*u[t0][-time + x + 2][-time + y + 3][z + 2] + r3*r4 +
r3*u[t0][-time + x + 2][-time + y + 2][z + 1] + r3*u[t0][-time + x + 2][-time + y + 2][z + 3]) + 1.0e-1F);}
```

3. A pass to adjust loop bounds to satisfy space-time diagonals and arbitrary block shapes

```
for (int time0_blk0 = time_m; time0_blk0 <= time_M; time0_blk0 += time0_blk0_size)
 {
   for (int x0_blk0 = x_m; x0_blk0 <= time_M - time_m + x_M; x0_blk0 += x0_blk0_size)
   {
     for (int y0_blk0 = y_m; y0_blk0 <= time_M - time_m + y_M; y0_blk0 += y0_blk0_size)
     {
       for (int time = time0_blk0, t0 = (time)%(2), t1 = (time + 1)%(2); time <= MIN(time0_blk0 + time0_blk0_size - 1, time_M); time += 1, t0 =
(time)%(2), t1 = (time + 1)%(2))
       {
         for (int x0_blk1 = MAX(x0_blk0, time + x_m); x0_blk1 <= MIN(x0_blk0 + x0_blk0_size - 1, time + x_M); x0_blk1 += x0_blk1_size)
         {
           for (int y0_blk1 = MAX(y0_blk0, time + y_m); y0_blk1 <= MIN(y0_blk0 + y0_blk0_size - 1, time + y_M); y0_blk1 += y0_blk1_size)
           {
             for (int x = x0_blk1; x <= MIN(MIN(x0_blk0 + x0_blk0_size - 1, time + x_M), x0_blk1 + x0_blk1_size - 1); x += 1)
             {
               for (int y = y0_blk1; y <= MIN(MIN(y0_blk0 + y0_blk0_size - 1, time + y_M), y0_blk1 + y0_blk1_size - 1); y += 1)
               {
                 for (int z = z_m; z <= z_M; z += 1)
                      {
                      float r4 = -2.0F*u[t0][-time + x + 2][-time + y + 2][z + 2];
                   u[t1][-time + x + 2][-time + y + 2][z + 2] = dt*(r0*u[t0][-time + x + 2][-time + y + 2][z + 2] + a*(r1*r4 + r1*u[t0][-time + x + 1][-time + y +
2][z + 2] + r1*u[t0][-time + x + 3][-time + y + 2][z + 2] + r2*r4 + r2*u[t0][-time + x + 2][-time + y + 1][z + 2] + r2*u[t0][-time + x + 2][-time + y + 3][z
+ 2] + r3*r4 + r3*u[t0][-time + x + 2][-time + y + 2][z + 1] + r3*u[t0][-time + x + 2][-time + y + 2][z + 3]) + 1.0e-1F);}
```

Works in tandem with all other Devito opts!

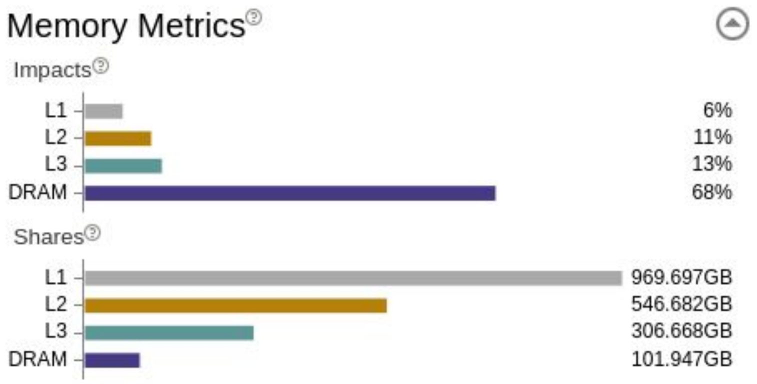
Cache-aware Roofline model :Laplacian, discretization order 8, 512^3 grid points, 512ms (Intel Skylake)



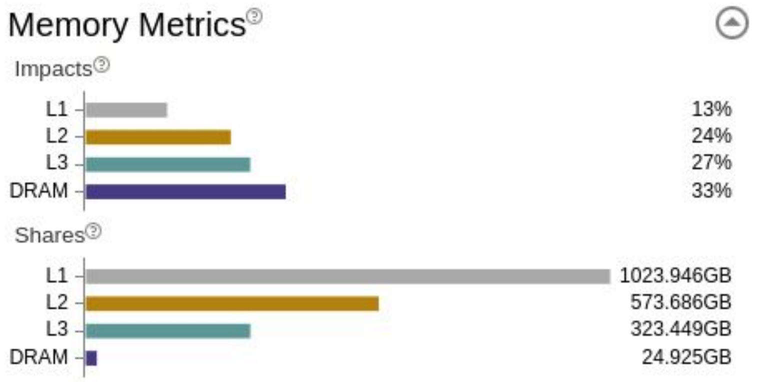
Cache shares
Laplacian, discretization order 8, 512^3 grid points, 512ms (Intel Skylake)

Highly optimized, spatially blocked, vectorized kernel + wavefront temporal blocking

Opt + SB (103Gflops/s, 32 secs, 1922GB)

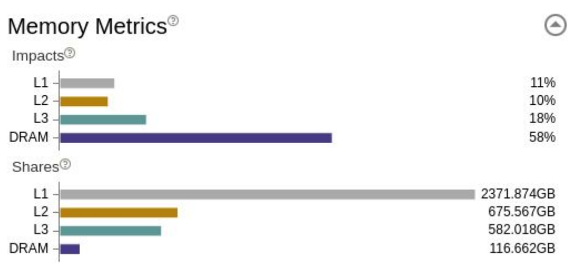


Opt + TB (167Gflops/s, 20 secs, 1943GB)

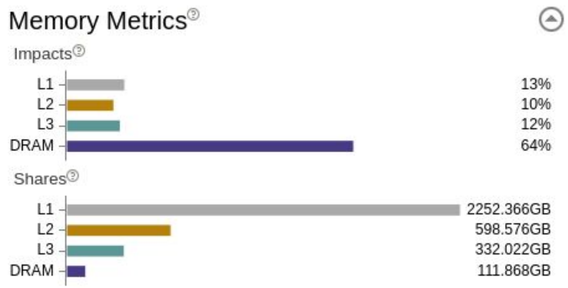


Roofline model
Laplacian, discretization order 8, 512^3 grid points, 512ms (Intel Skylake)

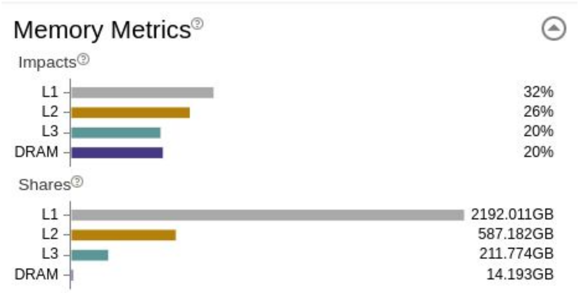
No-opt (98Gflops/s, 43 secs, 3744GB)



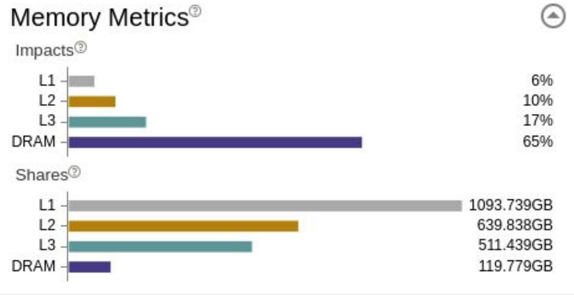
No-opt + SB(~130Gflops/s, 33 secs, 3293GB)



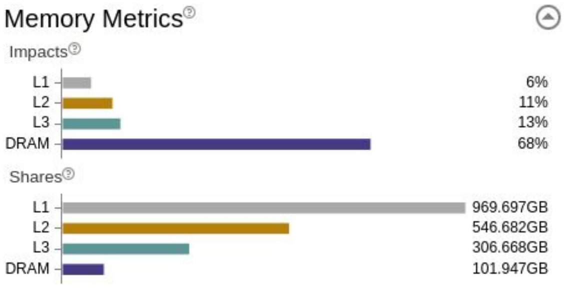
No-opt + TB (~130Gflops/s, 33 secs, 3004GB)



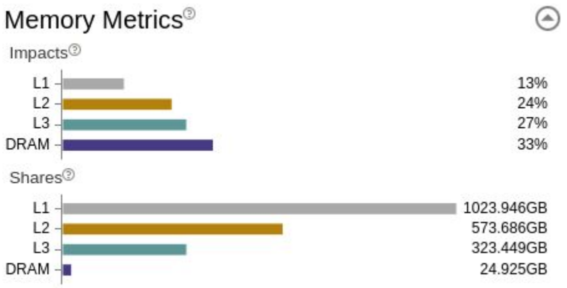
Opt (78Gflops/s, 42 secs, 2362GB)



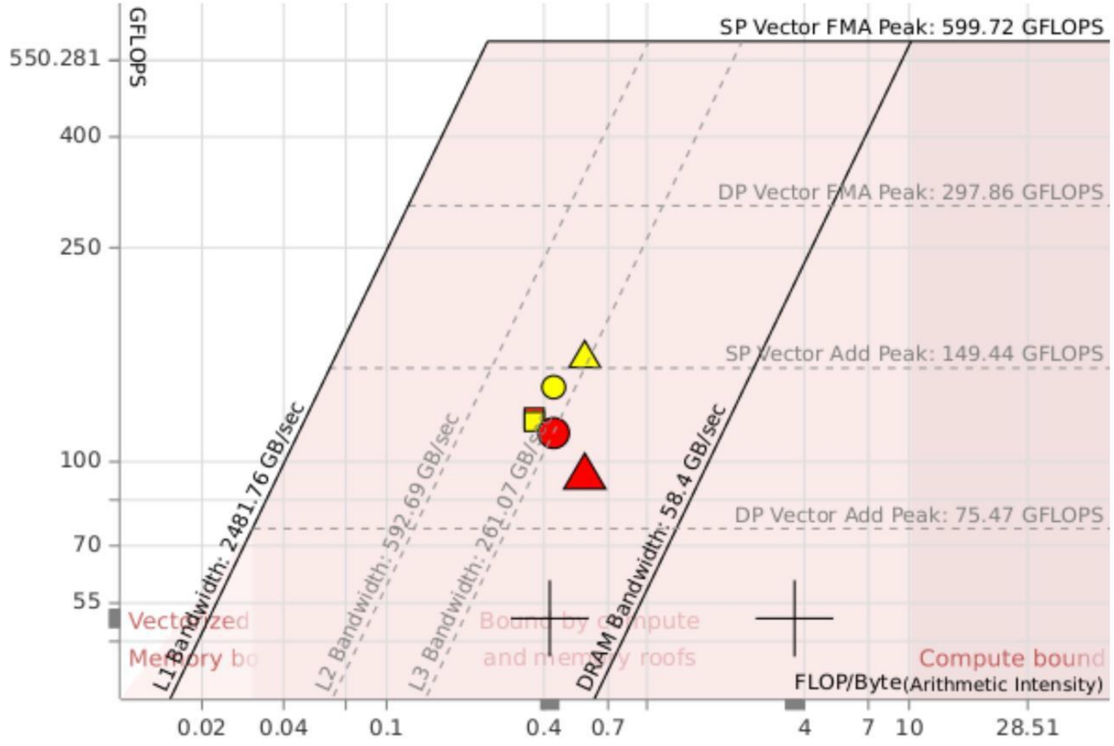
Opt + SB (103Gflops/s, 32 secs, 1922GB)



Opt + TB (167Gflops/s, 20 secs, 1943GB)



Roofline model (Broadwell, isotropic acoustic, 512^3 grid points, 512ms)



Space order:

- △ 4
- ○ 8
- □ 12

Temporal Blocking

Spatial Blocking

From high to low…

```python
# High-level DSL syntax
from devito import Eq, Grid,
TimeFunction, Operator

grid = Grid(shape=(4, 4))
u = TimeFunction(name='u', grid=grid,
space_order=2)
u.data[:] = 1

eq = Eq(u.forward, u.laplace + 1)
op = Operator(eq)
op.apply(time_M=3)
```

Groups of expressions, Cluster-level

```
(Cluster([Eq(u[t1, x + 2, y + 2],
u[t0, x + 1, y + 2]/h_x**2 -
2.0*u[t0, x + 2, y + 2]/h_x**2 +
u[t0, x + 3, y + 2]/h_x**2 +
u[t0, x + 2, y + 1]/h_y**2 -
2.0*u[t0, x + 2, y + 2]/h_y**2 +
u[t0, x + 2, y + 3]/h_y**2 +
1)]),)
```

Groups of expressions, Cluster-level (Optimized)

```
[Cluster([Eq(r0, 1/(h_x*h_x))
          Eq(r1, 1/(h_y*h_y))]),
Cluster([Eq(r2, -2.0*u[t0, x + 2,
y + 2])
          Eq(u[t1, x + 2, y + 2],
r0*r2 + r0*u[t0, x + 1, y + 2] +
r0*u[t0, x + 3, y + 2] + r1*r2 +
r1*u[t0, x + 2, y + 1] + r1*u[t0,
x + 2, y + 3] + 1)])]
```

```
<Callable Kernel>
  <CallableBody <allocs=0, casts=0, maps=0> <unmaps=0, frees=0>>
    <List (4, 0, 0)>
      <C.Comment /* Flush denormal numbers to zero in hardware */>
      <C.Statement
_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_O
N);>
      <C.Statement
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);>
      <C.Line >
    <List (0, 2, 0)>

      <ExpressionBundle (2)>

        <Expression r0 = 1/(h_x*h_x)>
        <Expression r1 = 1/(h_y*h_y)>

      <[affine,sequential] Iteration time::time::(time_m, time_M, 1)>
        <Section (section0)>

          <OverlappableHaloSpot(u)>
            <OmpRegion (1, 1, 0)>
              <C.Pragma #pragma omp parallel num_threads(nthreads)>
              <ParallelTree (0, 1, 0)>

                <[affine,collapsed[1],parallel] Iteration x::x::(x_m, x_M, 1)>
                  <[affine,parallel,vector-dim] Iteration y::y::(y_m, y_M, 1)>
                    <ExpressionBundle (2)>

                      <Expression r2 = -2.0*u[t0, x + 2, y + 2]>
                      <Expression u[t1, x + 2, y + 2] = r0*r2 + r0*u[t0, x + 1, y + 2]
+ r0*u[t0, x + 3, y + 2] + r1*r2 + r1*u[t0, x + 2, y + 1] + r1*u[t0, x + 2, y +
3] + 1>
```

Mapping from IET level to c-code

```
<Callable Kernel>
  <CallableBody <allocs=0, casts=0, maps=0> <unmaps=0, frees=0>>
    <List (0, 2, 0)>

      <ExpressionBundle (2)>

        <Expression r0 = 1/(h_x*h_x)>
        <Expression r1 = 1/(h_y*h_y)>

      <[affine,sequential] Iteration time::time::(time_m, time_M, 1)>
        <Section (section0)>

          <HaloSpot(u)>
            <[affine,parallel] Iteration x::x::(x_m, x_M, 1)>
              <[affine,parallel] Iteration y::y::(y_m, y_M, 1)>
                <ExpressionBundle (2)>

                  <Expression r2 = -2.0*u[t0, x + 2, y + 2]>
                  <Expression u[t1, x + 2, y + 2] = r0*r2 + r0*u[t0, x + 1, y + 2] + r0*u[t0, x + 3, y + 2] + r1*r2 + r1*u[t0, x + 2, y + 1] + r1*u[t0, x + 2, y + 3] + 1>
```

```
int Kernel(const float h_x, const float h_y, struct dataobj *restrict u_ve
const int time_M, const int time_m, const int x_M, const int x_m, const
int y_M, const int y_m)
{
  r0 = 1.0F/(h_x*h_x);
  r1 = 1.0F/(h_y*h_y);

  for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M; time += 1, t0 = (time)%(2), t1 = (time + 1)%(2))
  {
    /* Begin section0 */
    for (int x = x_m; x <= x_M; x += 1)
    {
      for (int y = y_m; y <= y_M; y += 1)
      {
        r2 = -2.0F*u[t0][x + 2][y + 2];
        u[t1][x + 2][y + 2] = r0*r2 + r0*u[t0][x + 1][y + 2] + r0*u[t0][x + 3][y + 2] + r1*r2 + r1*u[t0][x + 2][y + 1] + r1*u[t0][x + 2][y + 3] + 1;
      }
    }
    /* End section0 */
  }
}
```

Mapping from IET level to c-code - Add denormals

```
<Callable Kernel>
  <CallableBody <allocs=0, casts=0, maps=0> <unmaps=0, frees=0>>
    <List (4, 0, 0)>
      <C.Comment /* Flush denormal numbers to zero in hardware */>
      <C.Statement
_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);>
      <C.Statement _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);>
      <C.Line >

    <List (0, 2, 0)>

      <ExpressionBundle (2)>

        <Expression r0 = 1/(h_x*h_x)>
        <Expression r1 = 1/(h_y*h_y)>

      <[affine,sequential] Iteration time::time::(time_m, time_M, 1)>
        <Section (section0)>

          <HaloSpot(u)>
            <[affine,parallel] Iteration x::x::(x_m, x_M, 1)>
              <[affine,parallel] Iteration y::y::(y_m, y_M, 1)>
                <ExpressionBundle (2)>

                  <Expression r2 = -2.0*u[t0, x + 2, y + 2]>
                  <Expression u[t1, x + 2, y + 2] = r0*r2 + r0*u[t0, x + 1, y + 2] + r0*u[t0, x
+ 3, y + 2] + r1*r2 + r1*u[t0, x + 2, y + 1] + r1*u[t0, x + 2, y + 3] + 1>
```

```
int Kernel(const float h_x, const float h_y, struct dataobj *restrict u_vec, const
int time_M, const int time_m, const int x_M, const int x_m, const int y_M,
const int y_m)
{
  /* Flush denormal numbers to zero in hardware */
  _MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);

  r0 = 1.0F/(h_x*h_x);
  r1 = 1.0F/(h_y*h_y);

  for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M;
time += 1, t0 = (time)%(2), t1 = (time + 1)%(2))
  {
    /* Begin section0 */
    for (int x = x_m; x <= x_M; x += 1)
    {
      for (int y = y_m; y <= y_M; y += 1)
      {
        r2 = -2.0F*u[t0][x + 2][y + 2];
        u[t1][x + 2][y + 2] = r0*r2 + r0*u[t0][x + 1][y + 2] + r0*u[t0][x + 3][y + 2] +
r1*r2 + r1*u[t0][x + 2][y + 1] + r1*u[t0][x + 2][y + 3] + 1;
      }
    }
    /* End section0 */
  }
}
```

Mapping from IET level to c-code - Add parallelism

```
<Callable Kernel>
  <CallableBody <allocs=0, casts=0, maps=0> <unmaps=0, frees=0>>
    <List (4, 0, 0)>
      <C.Comment /* Flush denormal numbers to zero in hardware */>
      <C.Statement
_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);>
      <C.Statement _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);>
      <C.Line >
    <List (0, 2, 0)>

      <ExpressionBundle (2)>

        <Expression r0 = 1/(h_x*h_x)>
        <Expression r1 = 1/(h_y*h_y)>

      <[affine,sequential] Iteration time::time::(time_m, time_M, 1)>
        <Section (section0)>

          <OverlappableHaloSpot(u)>
            <OmpRegion (1, 1, 0)>
              <C.Pragma #pragma omp parallel num_threads(nthreads)>
              <ParallelTree (0, 1, 0)>

                <[affine,collapsed[1],parallel] Iteration x::x::(x_m, x_M, 1)>
                  <[affine,parallel,vector-dim] Iteration y::y::(y_m, y_M, 1)>
                    <ExpressionBundle (2)>

                      <Expression r2 = -2.0*u[t0, x + 2, y + 2]>
                      <Expression u[t1, x + 2, y + 2] = r0*r2 + r0*u[t0, x + 1, y + 2] +
r0*u[t0, x + 3, y + 2] + r1*r2 + r1*u[t0, x + 2, y + 1] + r1*u[t0, x + 2, y + 3] + 1>
```

```
int Kernel(...)
{
  /* Flush denormal numbers to zero in hardware */
  _MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
  _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);

  r0 = 1.0F/(h_x*h_x);
  r1 = 1.0F/(h_y*h_y);

  for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M; time +=
1, t0 = (time)%(2), t1 = (time + 1)%(2))
  {
    /* Begin section0 */
    #pragma omp parallel num_threads(nthreads)
    {
      #pragma omp for collapse(1) schedule(dynamic,1)
      for (int x = x_m; x <= x_M; x += 1)
      {
        #pragma omp simd aligned(u:32)
        for (int y = y_m; y <= y_M; y += 1)
        {
          r2 = -2.0F*u[t0][x + 2][y + 2];
          u[t1][x + 2][y + 2] = r0*r2 + r0*u[t0][x + 1][y + 2] + r0*u[t0][x + 3][y + 2] + r1*r2 +
r1*u[t0][x + 2][y + 1] + r1*u[t0][x + 2][y + 3] + 1;
        }
      }
    }
    /* End section0 */
  }
}
```

Mapping from IET level to c-code - Add parallelism

```
<Callable Kernel>
  <CallableBody <allocs=0, casts=0, maps=0> <unmaps=0, frees=0>>
    <List (4, 0, 0)>
      <C.Comment /* Flush denormal numbers to zero in hardware */>
      <C.Statement
_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);>
      <C.Statement _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);>
      <C.Line >
    <List (0, 2, 0)>

      <ExpressionBundle (2)>

        <Expression r0 = 1/(h_x*h_x)>
        <Expression r1 = 1/(h_y*h_y)>

      <[affine,sequential] Iteration time::time::(time_m, time_M, 1)>
        <Section (section0)>

          <OverlappableHaloSpot(u)>
            <OmpRegion (1, 1, 0)>
              <C.Pragma #pragma omp parallel num_threads(nthreads)>
              <ParallelTree (0, 1, 0)>

                <[affine,collapsed[1],parallel] Iteration x::x::(x_m, x_M, 1)>
                  <[affine,parallel,vector-dim] Iteration y::y::(y_m, y_M, 1)>
                    <ExpressionBundle (2)>

                      <Expression r2 = -2.0*u[t0, x + 2, y + 2]>
                      <Expression u[t1, x + 2, y + 2] = r0*r2 + r0*u[t0, x + 1, y + 2] +
r0*u[t0, x + 3, y + 2] + r1*r2 + r1*u[t0, x + 2, y + 1] + r1*u[t0, x + 2, y + 3] + 1>
```

```
int Kernel(...)
{
  /* Flush denormal numbers to zero in hardware */
  _MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
  _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);

  r0 = 1.0F/(h_x*h_x);
  r1 = 1.0F/(h_y*h_y);

  for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M; time +=
1, t0 = (time)%(2), t1 = (time + 1)%(2))
  {
    /* Begin section0 */
    #pragma omp parallel num_threads(nthreads)
    {
      #pragma omp for collapse(1) schedule(dynamic,1)
      for (int x = x_m; x <= x_M; x += 1)
      {
        #pragma omp simd aligned(u:32)
        for (int y = y_m; y <= y_M; y += 1)
        {
          r2 = -2.0F*u[t0][x + 2][y + 2];
          u[t1][x + 2][y + 2] = r0*r2 + r0*u[t0][x + 1][y + 2] + r0*u[t0][x + 3][y + 2] + r1*r2 +
r1*u[t0][x + 2][y + 1] + r1*u[t0][x + 2][y + 3] + 1;
        }
      }
    }
    /* End section0 */
  }
}
```

Pipeline for each target: CPU/OpenMP

```c
int Kernel(const float h_x, const float h_y, struct dataobj *restrict u_vec, const int time_M, const int time_m, const int x_M, const int x_m, const int y_M, const int y_m, const int nthreads, struct profiler * timers)
{
  float (*restrict u)[u_vec->size[1]][u_vec->size[2]] __attribute__ ((aligned (64))) = (float (*)[u_vec->size[1]][u_vec->size[2]]) u_vec->data;

  /* Flush denormal numbers to zero in hardware */
  _MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
  _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);

  float r0 = 1.0F/(h_x*h_x);
  float r1 = 1.0F/(h_y*h_y);

  for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M; time += 1, t0 = (time)%(2), t1 = (time + 1)%(2))
  {
    /* Begin section0 */
    START_TIMER(section0)
    #pragma omp parallel num_threads(nthreads)
    {
      #pragma omp for collapse(1) schedule(dynamic,1)
      for (int x = x_m; x <= x_M; x += 1)
      {
        #pragma omp simd aligned(u:32)
        for (int y = y_m; y <= y_M; y += 1)
        {
          float r2 = -2.0F*u[t0][x + 2][y + 2];
          u[t1][x + 2][y + 2] = r0*r2 + r0*u[t0][x + 1][y + 2] + r0*u[t0][x + 3][y + 2] + r1*r2 + r1*u[t0][x + 2][y + 1] + r1*u[t0][x + 2][y + 3] + 1;
        }
      }
    }
    STOP_TIMER(section0,timers)
    /* End section0 */
  }

  return 0;
}
```

Pipeline for each target: GPU/OpenACC

```
int Kernel(const float h_x, const float h_y, struct dataobj *restrict u_vec, const int time_M, const int time_m, const int x_M, const int x_m, const int y_M, const int
y_m, const int deviceid, const int devicerm, struct profiler * timers)
{
  /* Begin of OpenACC setup */
  acc_init(acc_device_nvidia);
  if (deviceid != -1)
  {
    acc_set_device_num(deviceid,acc_device_nvidia);
  }
  /* End of OpenACC setup */

  float (*restrict u)[u_vec->size[1]][u_vec->size[2]] __attribute__ ((aligned (64))) = (float (*)[u_vec->size[1]][u_vec->size[2]]) u_vec->data;

  #pragma acc enter data copyin(u[0:u_vec->size[0]][0:u_vec->size[1]][0:u_vec->size[2]])

  float r0 = 1.0F/(h_x*h_x);
  float r1 = 1.0F/(h_y*h_y);

  for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M; time += 1, t0 = (time)%(2), t1 = (time + 1)%(2))
  {
    /* Begin section0 */
    START_TIMER(section0)
    #pragma acc parallel loop collapse(2) present(u)
    for (int x = x_m; x <= x_M; x += 1)
    {
      for (int y = y_m; y <= y_M; y += 1)
      {
        float r2 = -2.0F*u[t0][x + 2][y + 2];
        u[t1][x + 2][y + 2] = r0*r2 + r0*u[t0][x + 1][y + 2] + r0*u[t0][x + 3][y + 2] + r1*r2 + r1*u[t0][x + 2][y + 1] + r1*u[t0][x + 2][y + 3] + 1;
      }
    }
    STOP_TIMER(section0,timers)
    /* End section0 */
  }

  #pragma acc exit data copyout(u[0:u_vec->size[0]][0:u_vec->size[1]][0:u_vec->size[2]])
  #pragma acc exit data delete(u[0:u_vec->size[0]][0:u_vec->size[1]][0:u_vec->size[2]])    if(devicerm)

  return 0;
}
```

Talk outline

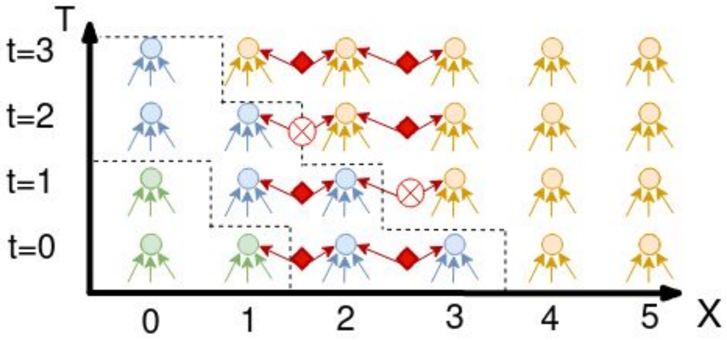
- Motivation: speed up computationally expensive scientific simulations involving the solution of PDEs modelling wave equations through explicit FD methods (seismic and medical imaging)

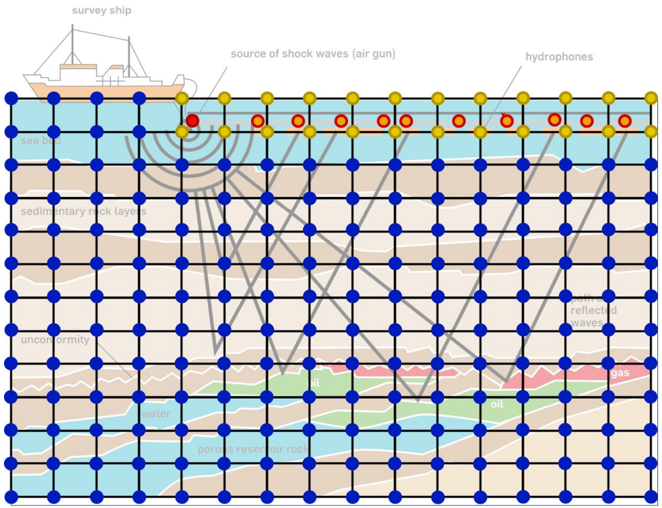
- Accelerating through cache optimizations, more **specifically through temporal blocking**

- Enabling **temporal blocking** on practical wave-propagation simulations is complicated as they consist of **sparse "off-the-grid"** operators (**Not** a typical stencil benchmark!) => **Applicability issues**

- We present **an approach to overcome limitations** and enable TB

- Experimental results show **improved** performance





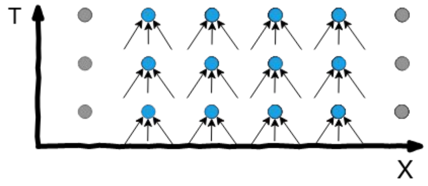
Modelling practical applications

- Stencils everywhere, not only though.
  What else?

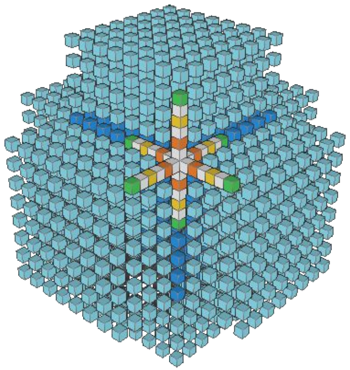
- Remarkable amount of work in the past on optimizing stencils...
  (Parallelism, cache optimizations, accelerators)

- Sources injecting and receivers interpolating at sparse off-the-grid coordinates.
  Non-conventional update patterns.

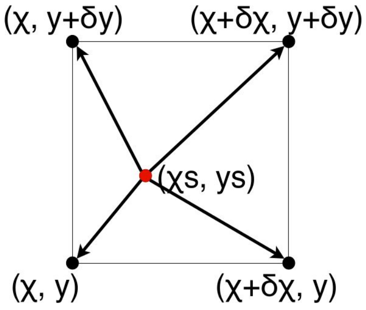
- Usually their coordinates are not aligned with the computational grid. How do we iterate over them?

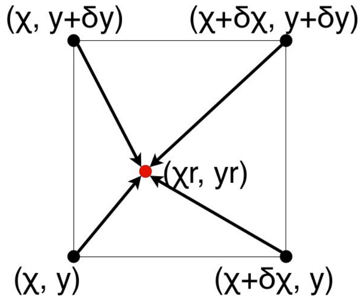


A 1d 3pt stencil update



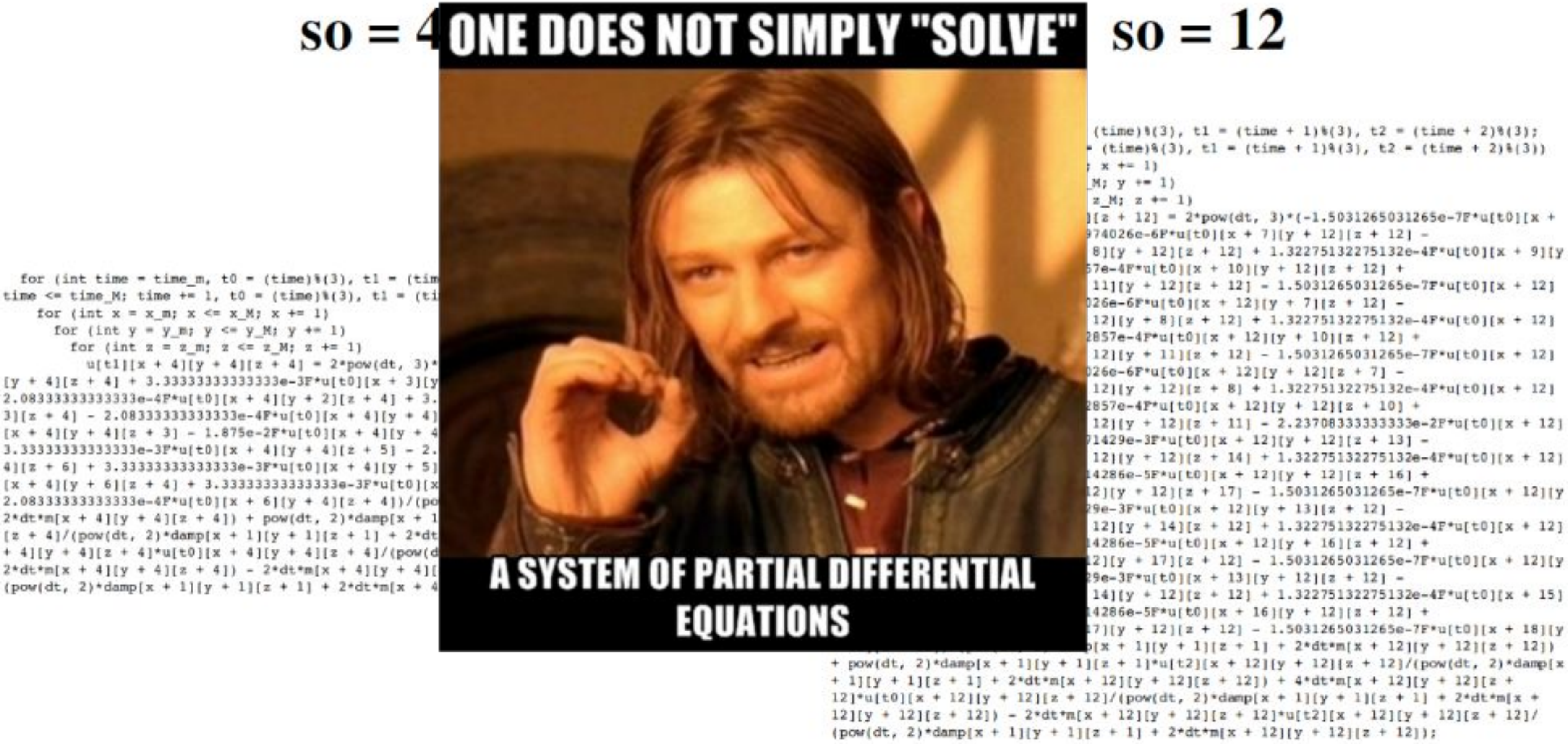
A 3d-19pt stencil update



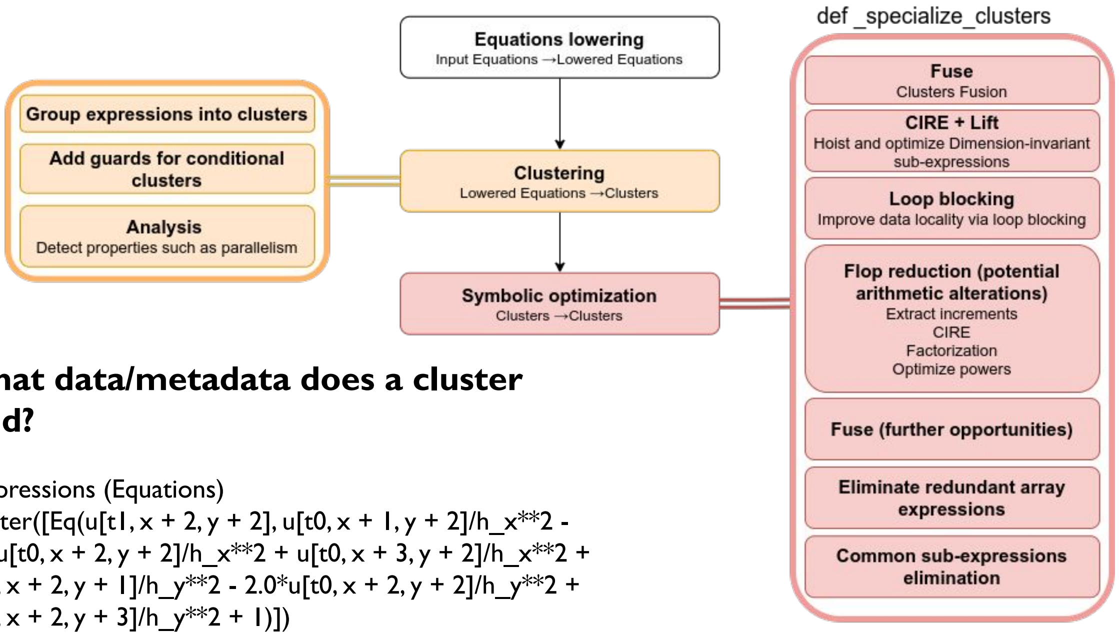


Off-the-grid operators (Source injection/Receiver interpolation)

If you do not use Devito (or other high-level tool)...good luck!



The cluster level



What data/metadata does a cluster hold?

- Expressions (Equations)

Cluster([Eq(u[t1, x + 2, y + 2], u[t0, x + 1, y + 2]/h_x**2 - 2.0*u[t0, x + 2, y + 2]/h_x**2 + u[t0, x + 3, y + 2]/h_x**2 + u[t0, x + 2, y + 1]/h_y**2 - 2.0*u[t0, x + 2, y + 2]/h_y**2 + u[t0, x + 2, y + 3]/h_y**2 + 1)])

- IterationSpace

IterationSpace[time[0,0]<008>++, x[0,0]<008>++, y[0,0]<008>++]

- Detect computational properties

<frozendict {time: {affine, sequential}, x: {affine, tilable, skewable, parallel}, y: {affine, tilable, skewable, parallel}}>

What data/metadata does a cluster hold?

- Relations

clusters[0].ispace.relations
{(time, t), (time, x, y), (t, x, y), ()}

- Sub-Iterators

<frozendict {time: (t0, t1), x: (), y: ()}>

- Directions

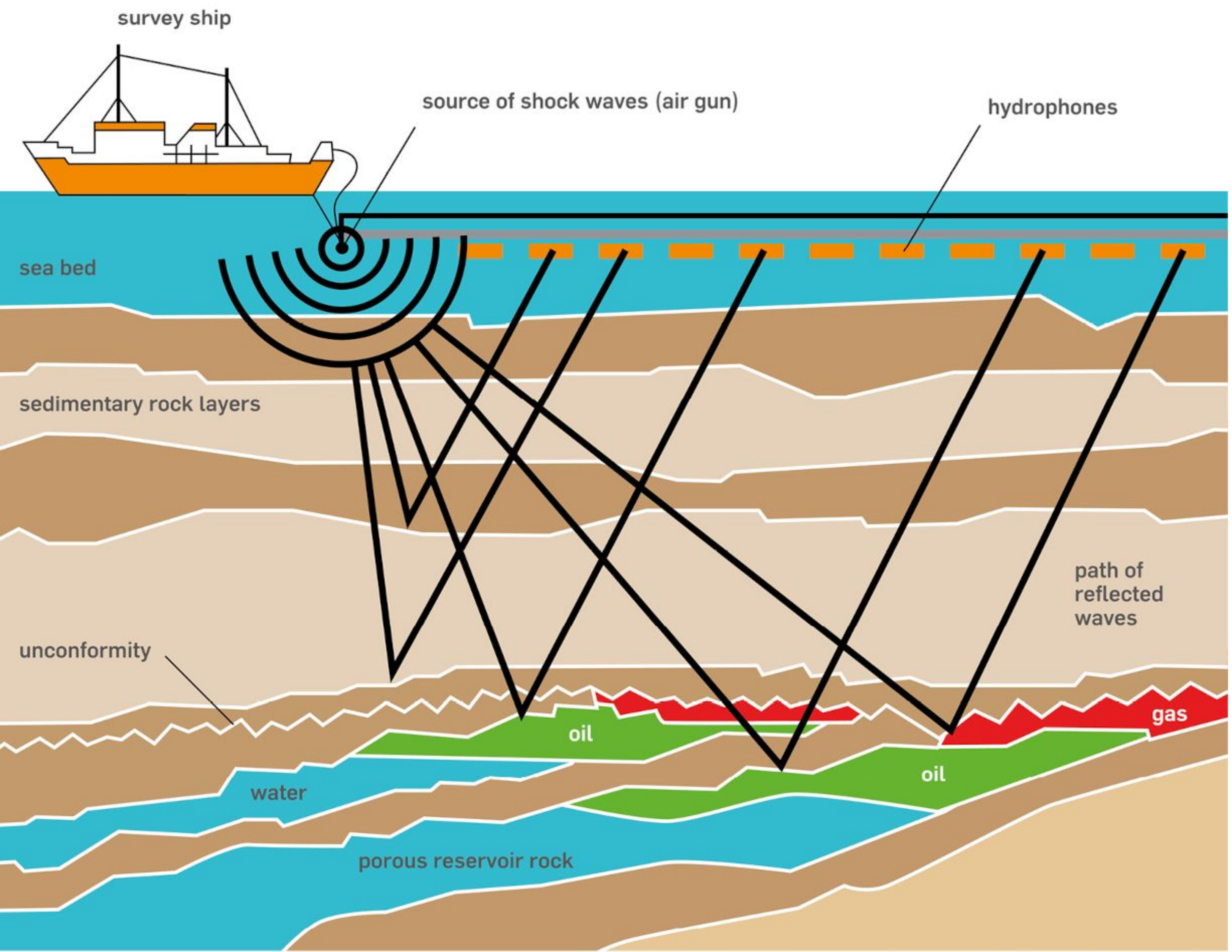
<frozendict {time: ++, x: ++, y: ++}>

- Dimensions

{t, y, time, x}

Dummy cluster pass:
https://gist.github.com/georgebisbas/8115b94b86a3ffbc25e179f1e22c49e7

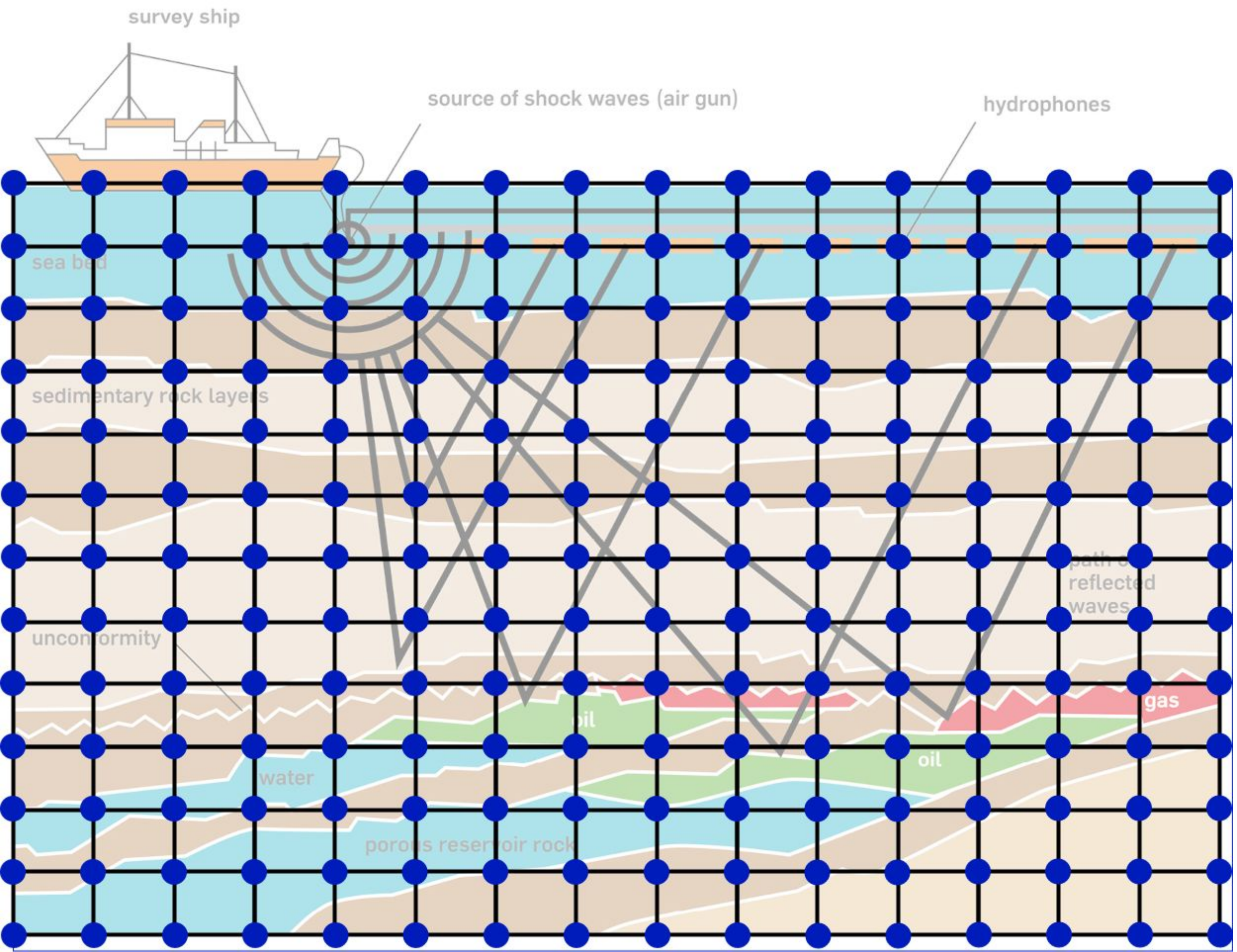
Sparse off-the-grid operators



- How a seismic survey looks like

Source: KrisEnergy 2021

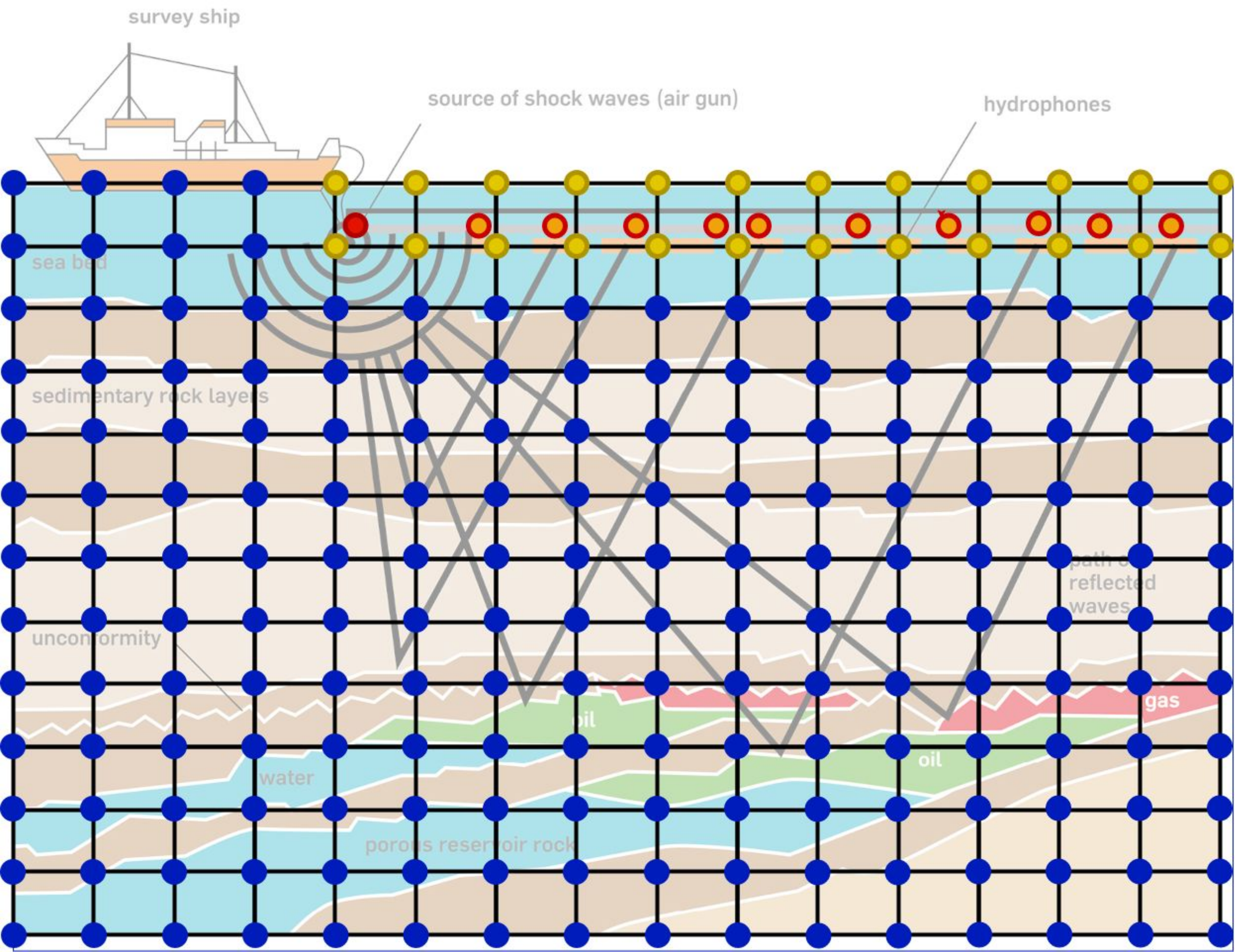
Sparse off-the-grid operators



- How a seismic survey looks like
- Discretizing the computational domain (the FD-grid). Solution computed on the points

Source: KrisEnergy 2021

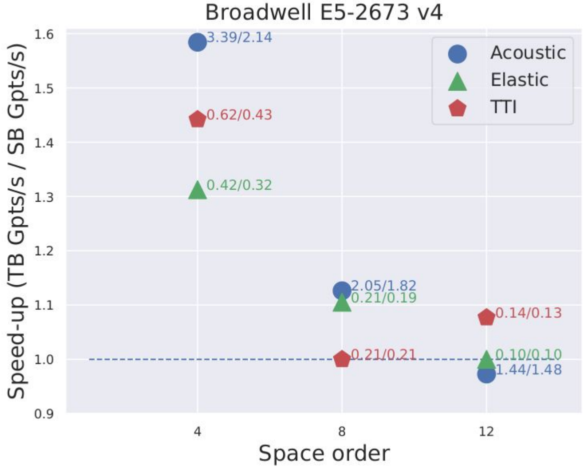
Sparse off-the-grid operators



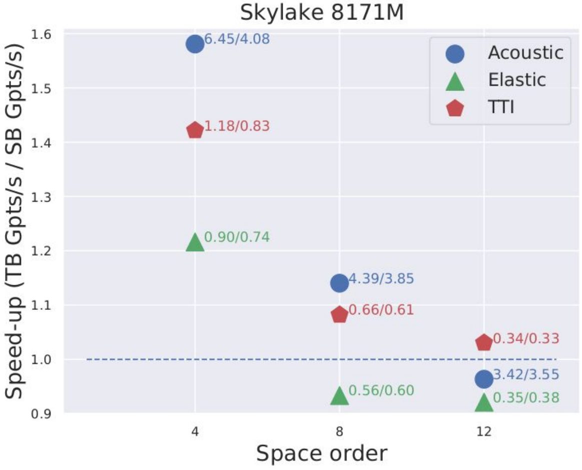
- How a seismic survey looks like
- Discretizing the computational domain (the FD-grid). Solution computed on the points
- Not-aligned "off-the-grid" operators exist (source injection/receiver interpolation)

Source: KrisEnergy 2021

Experimental evaluation



(a) Throughput speed-up of kernels for Broadwell.



(b) Throughput speed-up of kernels for Skylake.

- Kernels are flop-optimized through Devito.
- Gpts/s aka Gcells/s: time to solution metric in stencil computations
- (!) High Gflops/s do not guarantee a faster solution.

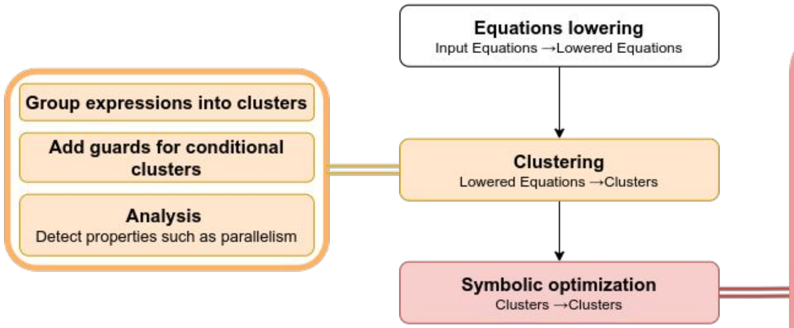
Azure model Architecture	**E16s v3 Broadwell**	**E32s v3 Skylake**
vCPUs	16	32
GiB memory	128	256
Model name	E5-2673 v4	8171M
CPUs	16	32
Thread(s) per core	2	2
Core(s) per socket	8	16
Socket(s)	1	1
NUMA node(s)	1	1
Model	79	85
CPU MHz	2300	2100
L1d cache	32K	32K
L1i cache	32K	32K
L2 cache	256K	1024K
L3 cache	51200K	36608K

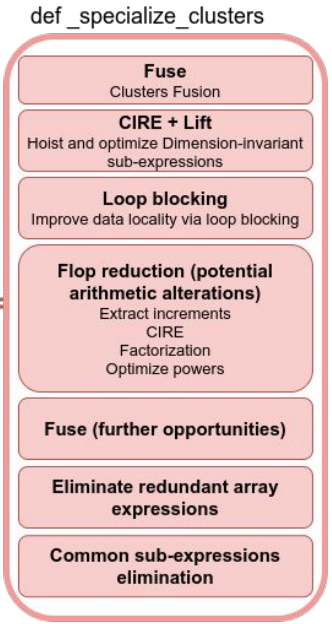
TABLE I: VM specification

- Benchmark on Azure VMs
- GCC, ICC
- Thread pinning
- OpenMP, SIMD
- Aggressive auto-tuning

Open-source, on top of latest Devito!

The cluster level






```
 float r0 = 1.0F/(h_x*h_x);
 float r1 = 1.0F/(h_y*h_y);

for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M; time
+= 1, t0 = (time)%(2), t1 = (time + 1)%(2))
  {
    /* Begin section0 */
    START_TIMER(section0)
    for (int x = x_m; x <= x_M; x += 1)
    {
      #pragma omp simd aligned(u:32)
      for (int y = y_m; y <= y_M; y += 1)
      {
        float r2 = -2.0F*u[t0][x + 2][y + 2];
        u[t1][x + 2][y + 2] = r0*r2 + r0*u[t0][x + 1][y + 2] + r0*u[t0][x + 3][y + 2] +
r1*r2 + r1*u[t0][x + 2][y + 1] + r1*u[t0][x + 2][y + 3] + 1;
      }
    }
    STOP_TIMER(section0,timers)
    /* End section0 */
  }

  return 0;
}
```

Optimized! 😀😀😀

What data/metadata does a cluster hold?

- **Expressions (Equations)**

(Pdb) clusters[0]
Cluster([Eq(r0, 1/(h_x*h_x))
        Eq(r1, 1/(h_y*h_y))])
(Pdb) clusters[1]
Cluster([Eq(r2, -2.0*u[t0, x + 2, y + 2])
        Eq(u[t1, x + 2, y + 2], r0*r2 + r0*u[t0, x + 1, y + 2] + r0*u[t0, x + 3, y + 2] + r1*r2 + r1*u[t0, x + 2, y + 1] + r1*u[t0, x + 2, y + 3] + 1)])

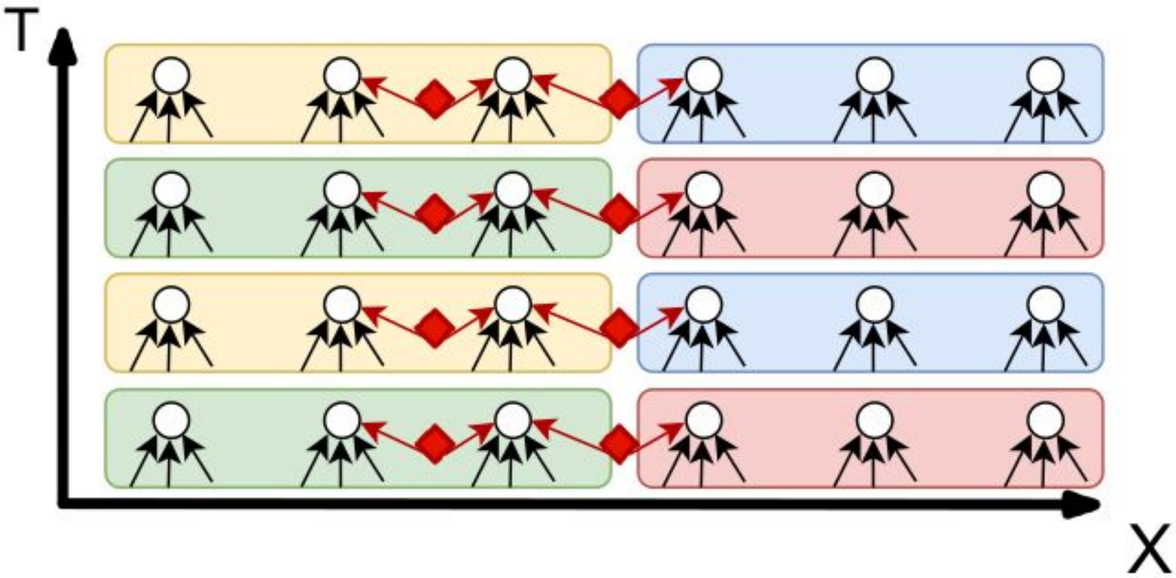
- **IterationSpace**

(Pdb) clusters[0].ispace
IterationSpace[]
(Pdb) clusters[1].ispace
IterationSpace[time[0,0]<960>++, x[0,0]<960>++, y[0,0]<960>++]

Applying loop-blocking

Loop blocking (aka space blocking, loop tiling):

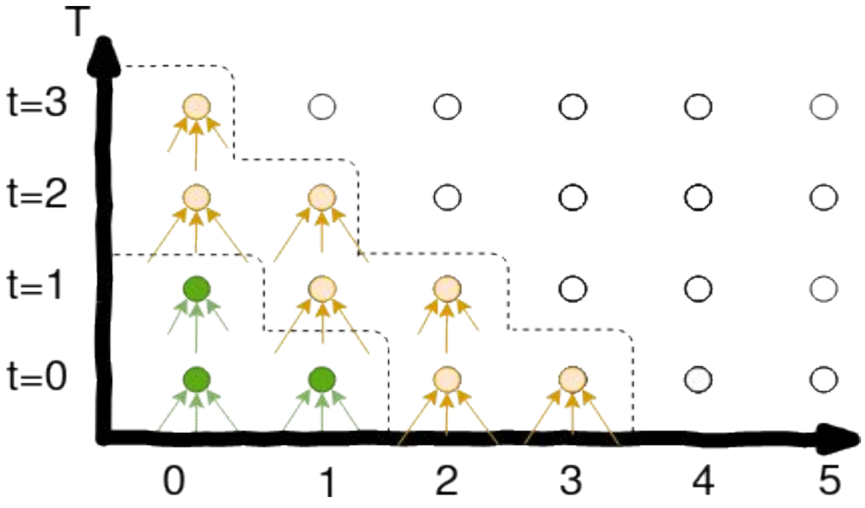
- Decompose grids into blocks/tiles. Iteration space partitioned to smaller chunks/blocks
- Improved data locality ⇛ Increased performance (Rich literature)
- Sparse off-the-grid operators are iterated as without blocking

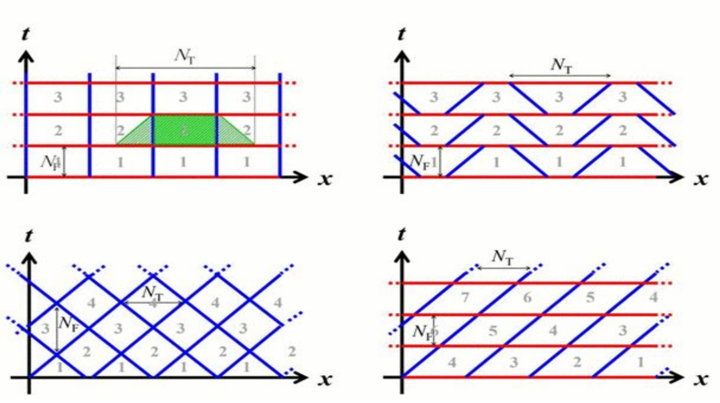


Applying temporal-blocking

Temporal blocking (Time-Tiling):

- Space blocking but data reuse is extended to time-dimension.
- Update grid points in future where (space) and when (time) possible
- Rich literature, several variants of temporal blocking, shapes, schemes
  - -**Wave-front / Skewed** (Approach followed in the paper)
  - -Diamonds, Trapezoids, Overlapped, Hybrid models

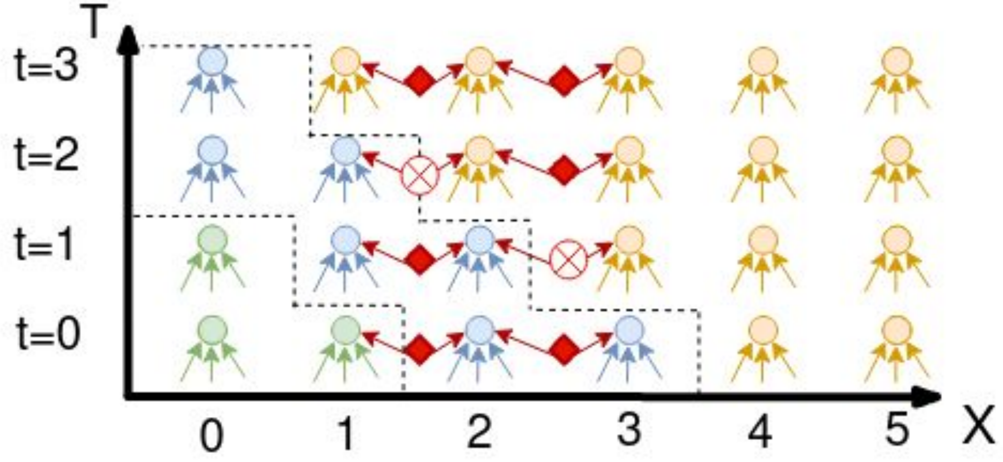




Tanaka et.al. (2018)

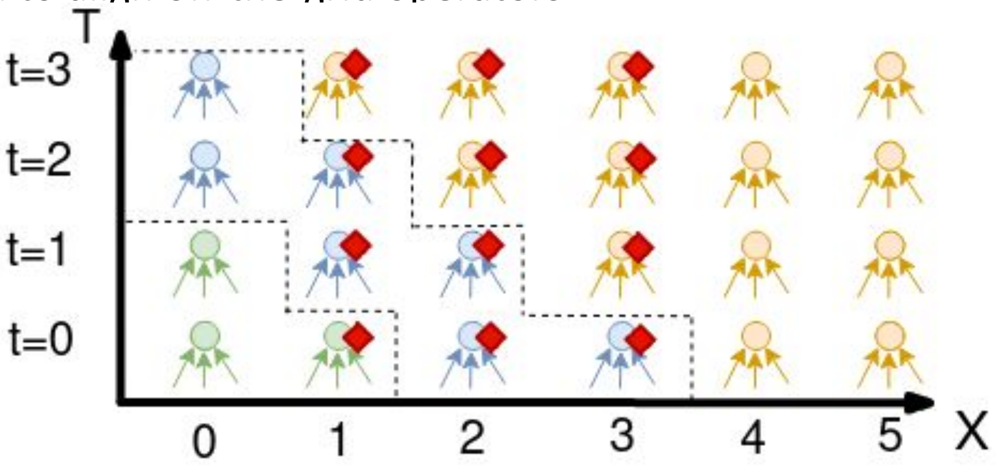
Off-the-grid operators: the issue

- Data dependences violations happen while a temporal update
- Source injection is in a different iteration space
- When a sparse operator exists in the boundary between space-time blocks, the order of updates is not preserved
- **Solution**: Need to align off-the-grid operators



Off-the-grid operators: the issue

- Data dependences violations happen while a temporal update
- Source injection is in a different iteration space
- When a sparse operator exists in the boundary between space-time blocks, the order of updates is not preserved
- **Solution**: Need to align off-the-grid operators



Methodology

- A negligible-cost scheme to precompute the source injection contribution.
- Align source injection data dependences to the grid
- This scheme is applicable to other fields as well (e.g. medical imaging)

Iterate over sources and store indices of affected points

- Inject to a **zero-valued initialized grid** for one (or a few more) timesteps
- **Hypothesis:** non-zero source-injection values at the first time-steps
- **Independent** of the injection and interpolation type (e.g. non-linear injection)

Listing 2: Source injection over an empty grid. No PDE stencil update is happening.

```
for t = 1 to 2 do
    foreach s in sources do
        for i = 1 to np do
            xs, ys, zs = map(s, i);
            u[t, xs, ys, zs] + = f(src(t, s))
```

- Then, we **store the non-zero** grid point coordinates

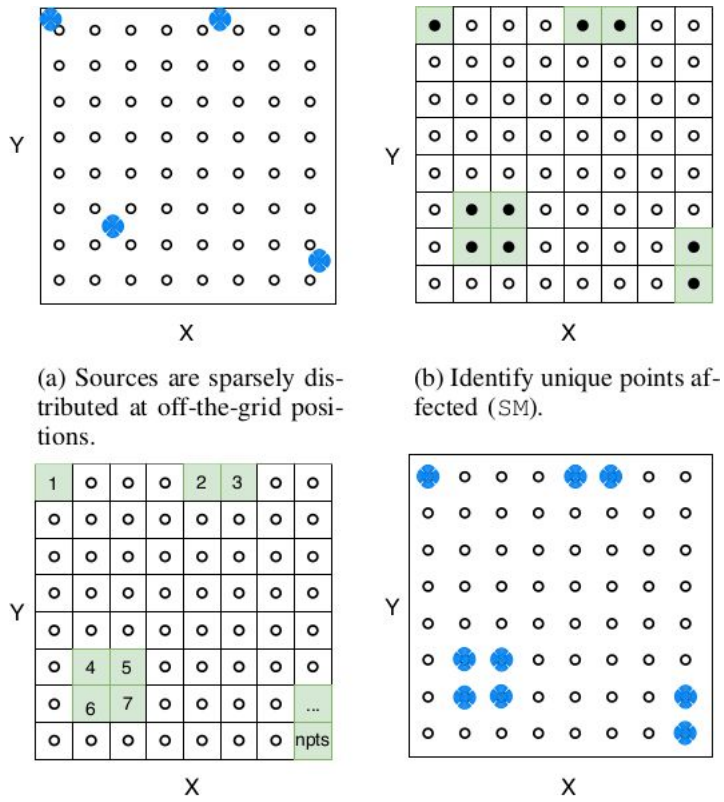
Generate sparse binary mask, unique IDs and decompose wavefields

- Perform source injection to decompose the off-the-grid wavefields to on-the-grid per point wavefields.
- Inject sources to sources

	Off-the-grid	Aligned
len(sources)	n_src	n_aff_pts
len(sources.coords)	(n_src, 3)	(n_aff_pts, 3)
len(sources.data)	(n_src, nt)	(n_aff_pts, nt)

Listing 3: Decomposing the source injection wavefields.

```
for t = 1 to nt do
    foreach s in sources do
        for i = 1 to np do
            xs, ys, zs = map(s, i);
            src_dcmp[t, SID[xs, ys, zs]] + = f(src(t, s));
```



(a) Sources are sparsely distributed at off-the-grid positions.

(b) Identify unique points affected (SM).

(c) Assign a unique ID to every affected point (SID).

(d) Sources are aligned with grid positions.

Fuse iteration spaces

- Indirection mapping has changed. We still use indirections but now they are on the point.
- By using the aligned structure, we fuse the source injection loop inside the kernel update iteration space.
- The source mask SM is used to add (if 1) or not (if 0) the impact and SID is used to indirect to the impact values using the traversed grid coordinates.

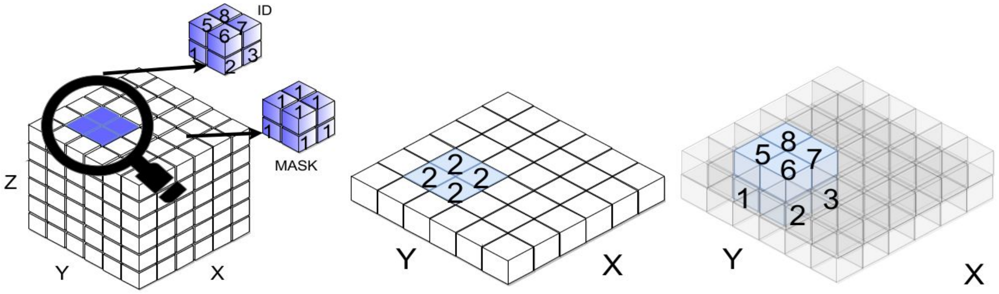
Listing 4: Stencil kernel update with fused source injection.

```
for t = 1 to nt do
    for x = 1 to nx do
        for y = 1 to ny do
            for z = 1 to nz do
                A(t, x, y, z, s);
            for z2 = 1 to nz do
                u[t, x, y, z2] + = SM[x, y, z2] * src_dcmp[t, SID[x, y, z2]];
```

SIMD? (AVX512)

Reducing the iteration space size

- Lots of redundant ops due to sparsity
- A schedule to perform only necessary operations
- Aggregate NZ along the z-axis keeping count of them in a reduced-size structure named ***nnz_mask***
- Reduce the size of SID by cutting off zero z-slices



(a) SID and SM are very sparse in the general case. (b) nnz_mask Aggregating non-zero values along z-axis. (c) Sp_SID, a reduced size SID.

Listing 5: Stencil kernel update with reduced size iteration space for source injection.

```
for t = 1 to nt do
  | for x = 1 to nx do
  |   | for y = 1 to ny do
  |   |   | for z = 1 to nz do
  |   |   |   | A(t, x, y, z, s);
  |   |   | for z2 = 1 to nnz_mask[x][y] do
  |   |   |   | I(t, x, y, z, s) ≡{ zind = Sp_SID[x, y, z2];
  |   |   |   | u[t, x, y, z2] += src_dcmp[t, SID[x, y, zind]]; }
```

Listing 1: A typical time-stepping loop nest structure for a stencil update with source injection. This stencil has one temporal and three spatial dimensions.

```
for t = 1 to nt do
  for x = 1 to nx do
    for y = 1 to ny do
      for z = 1 to nz do
        A(t, x, y, z) ≡ u[t, x, y, z] = u[t-1, x, y, z] + Σ_{r=1}^{r=so/2} w_r (
          u[t-1, x - r, y, z] + u[t-1, x + r, y, z] + u[t-1, x, y - r, z] +
          u[t-1, x, y + r, z] + u[t-1, x, y, z - r] + u[t-1, x, y, z + r] );
  foreach s in sources do // For every source
    for i = 1 to np do // Get the points affected
      xs, ys, zs = map(s, i) // through indirection
      u[t, xs, ys, zs] + = f(src(t, s)) // add their impact
       on the field
```

Non-aligned

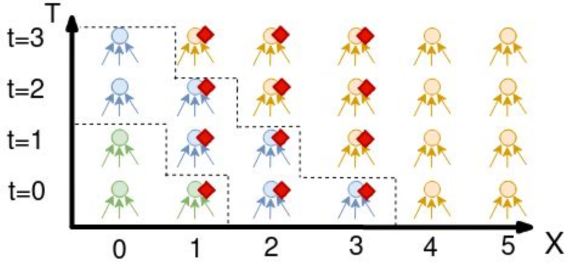
Listing 5: Stencil kernel update with fused - reduced size iteration space - source injection.

```
for t = 1 to nt do
  for x = 1 to nx do
    for y = 1 to ny do
      for z = 1 to nz do
        A(t, x, y, z, s);
      for z2 = 1 to nnz_mask[x][y] do
        I(t, x, y, z, s) ≡{ zind = Sp_SM[x, y, z2];
        u[t, x, y, z2] +=
        SM[x, y, zind] * src_dcmp[t, SID[x, y, zind]]; }
```

Aligned

- Aligned to grid
- Same OPS
- Parallelism
- SIMD
- Apply TB

Everything so far, automated in Devito DSL



Applying wave-front temporal blocking

- TB with manually editing the Devito generated code
- Skewing factor depends on data dependency distances (higher for higher SO, multigrid)

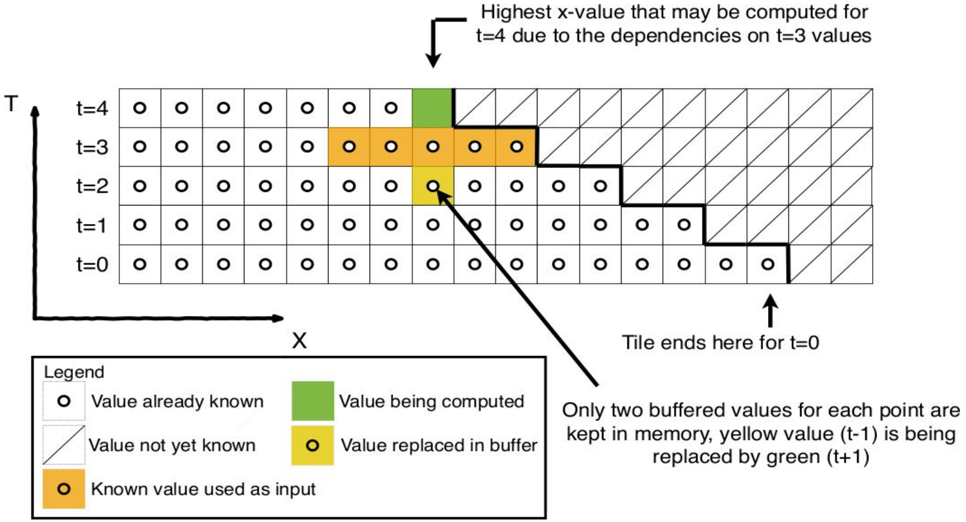
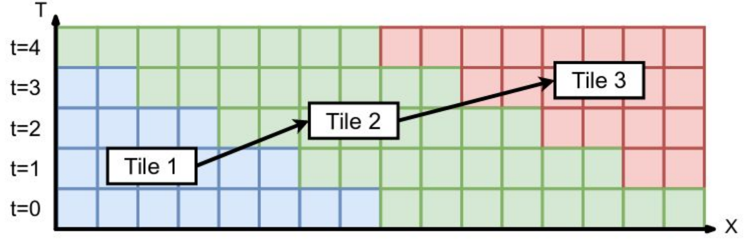
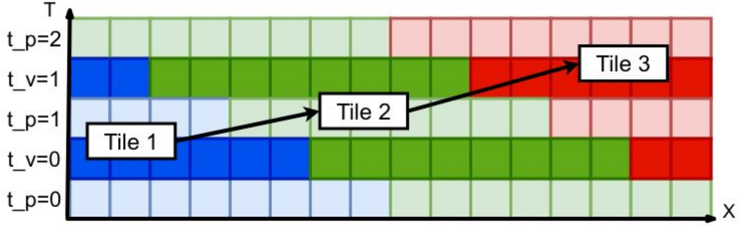


Figure from YASK, Yount et. al (2016)



(a) The figure shows multiple wave-front tiles evaluated sequentially, partially adapted from [15].



(b) The figure shows multiple wave-front tiles evaluated sequentially in multigrid stencil codes.

Listing 6: The figure shows the loop structure after applying our proposed scheme.

```
for t_tile in time_tiles do
  for xtile in xtiles do
    for ytile in ytiles do
      for t in t_tile do
        OpenMP parallelism
        for xblk in xtile do
          for yblk in ytile do
            for x in xblk do
              for y in yblk do
                SIMD vectorization
                for z = 1 to nz do
                  A(t, x - time, y - time, z, s);
                for z2 = 1 to nnz_mask[x][y] do
                  I(t, x - time, y - time, z2, s);
```

Iterate space-time tiles

Time-stepping in the tile and loop-blocking within the tile. Collapse outer loops that are loop-blocked

No loop blocking on z-dim, full stride for max-vectorization performance

Experimental evaluation: the models

- **Isotropic Acoustic**

  Generally known, single scalar PDE, laplacian like, low cost

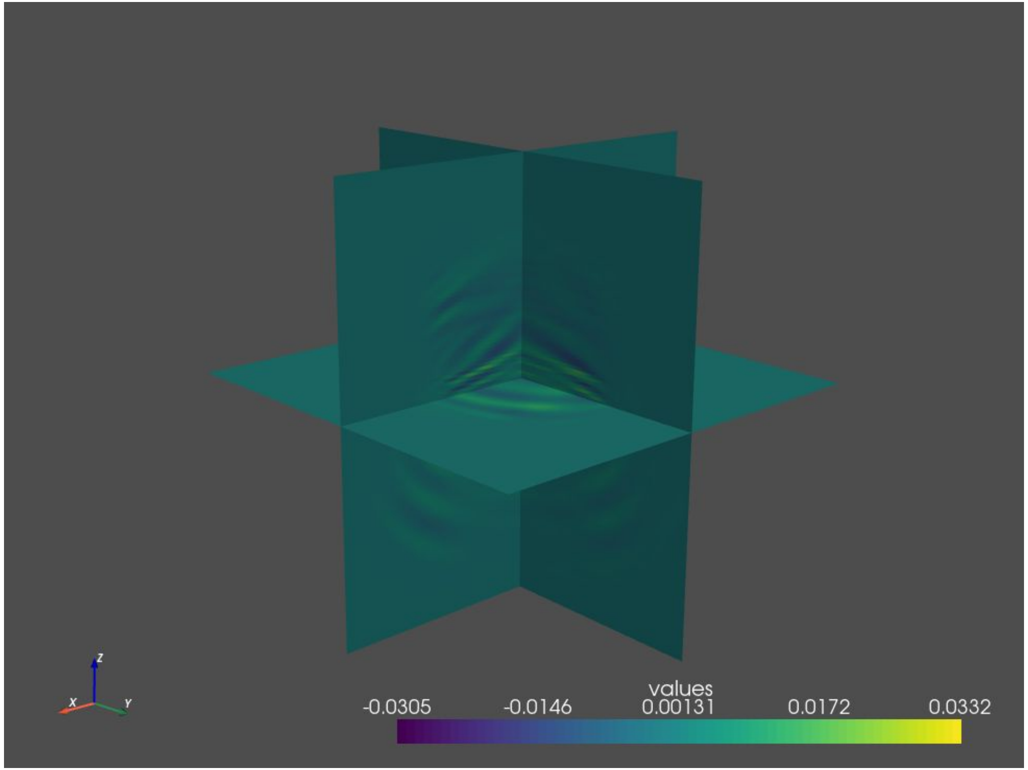
- **Isotropic Elastic**

  Coupled system of a vectorial and tensorial PDE, explosive source, increased data movement, first order in time, cross-loop data dependencies

- **Anisotropic Acoustic (aka TTI)**

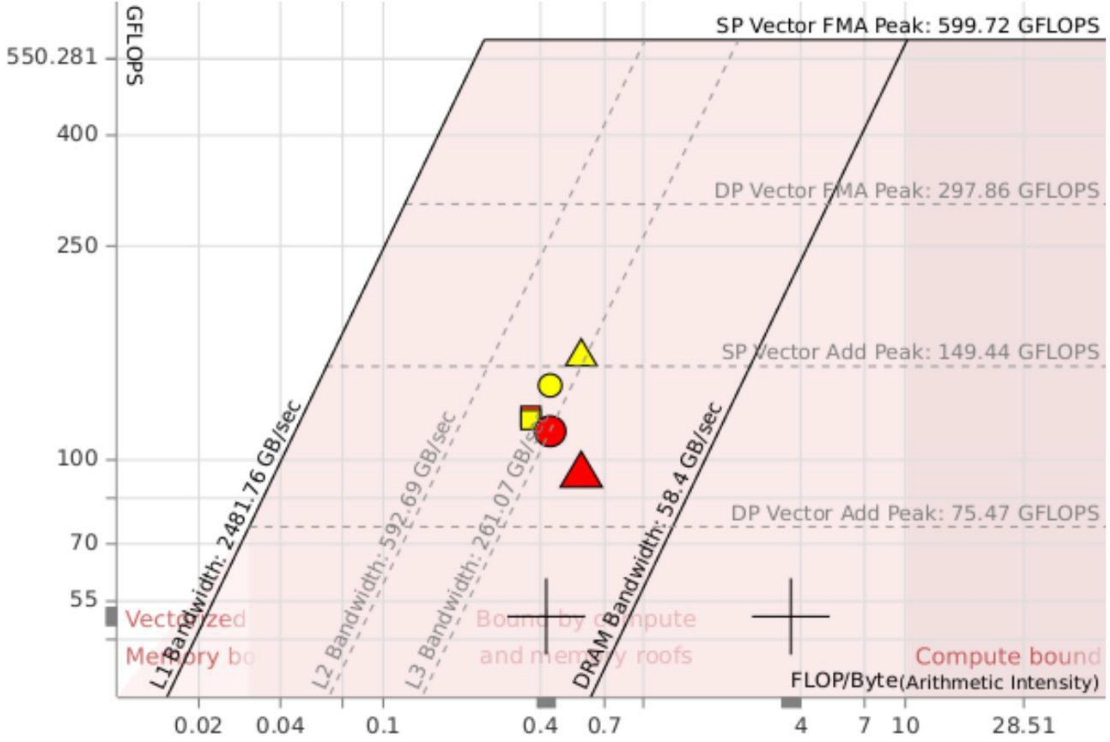
  Industrial applications, rotated laplacian, coupled system of two scalar PDEs

Industrial-level, 512^3 grid points, 512ms simulation time, damping fields ABCs



Velocity field, TTI wave propagation after 512ms

Cache-aware roofline model



Space order:

- △ 4
- ○ 8
- □ 12

Temporal Blocking

Spatial Blocking

Broadwell, isotropic acoustic, 512^3 grid points, 512ms

Acknowledgements

Thanks to collaborators and contributors:

- Navjot Kukreja (Imperial College)
- John Washbourne (Chevron)
- Edward Caunt (Imperial College)

http://www.devitoproject.org

https://github.com/devitocodes/devito

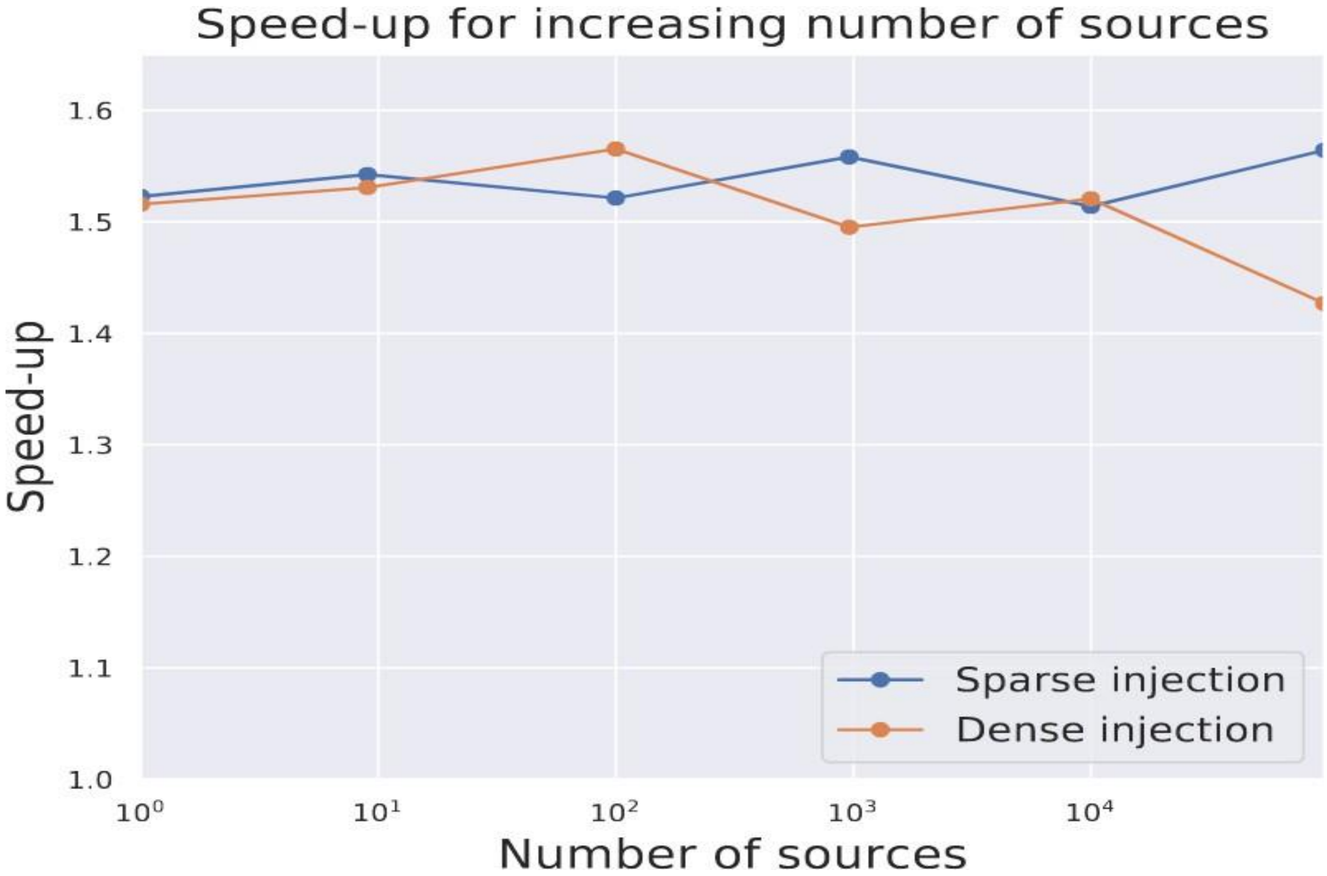
https://opesci-slackin.now.sh

Imperial College London

EPSRC Engineering and Physical Sciences Research Council

HiPEDS

Corner cases, increasing number of sources



The generated C code - stencil update


```
#pragma omp for collapse(1) schedule(dynamic,1)
for (int x0_blk0 = x_m; x0_blk0 <= x_M; x0_blk0 += x0_blk0_size)
{
  for (int y0_blk0 = y_m; y0_blk0 <= y_M; y0_blk0 += y0_blk0_size)
  {
    for (int x = x0_blk0; x <= x0_blk0 + x0_blk0_size - 1; x += 1)
    {
      for (int y = y0_blk0; y <= y0_blk0 + y0_blk0_size - 1; y += 1)
      {
        #pragma omp simd aligned(damp,uref,vp:32)
        for (int z = z_m; z <= z_M; z += 1)
        {
          float r14 = -2.84722222F*uref[t1][x + 8][y + 8][z + 8];
          float r13 = 1.0/dt;
          float r12 = 1.0/(dt*dt);
          float r11 = 1.0/(vp[x + 8][y + 8][z + 8]*vp[x + 8][y + 8][z + 8]);
          uref[t0][x + 8][y + 8][z + 8] = (r11*(-r12*(-2.0F*uref[t1][x + 8][y + 8][z + 8] +
uref[t2][x + 8][y + 8][z + 8])) + r13*(damp[x + 1][y + 1][z + 1]*uref[t1][x + 8][y + 8][z + 8])
(r14 - 1.78571429e-3F*(uref[t1][x + 8][y + 8][z + 4] + uref[t1][x + 8][y + 8][z + 12]) +
2.53968254e-2F*(uref[t1][x + 8][y + 8][z + 5] + uref[t1][x + 8][y + 8][z + 11]) -
2.0e-1F*(uref[t1][x + 8][y + 8][z + 6] + uref[t1][x + 8][y + 8][z + 10]) + 1.6F*(uref[t1][x + 8]
[y + 8][z + 7] + uref[t1][x + 8][y + 8][z + 9]))/((h_z*h_z)) + (r14 - 1.78571429e-3F*(uref[t1][x
+ 8][y + 4][z + 8] + uref[t1][x + 8][y + 12][z + 8]) + 2.53968254e-2F*(uref[t1][x + 8][y + 5][z
8] + uref[t1][x + 8][y + 11][z + 8]) - 2.0e-1F*(uref[t1][x + 8][y + 6][z + 8] + uref[t1][x + 8][
+ 10][z + 8]) + 1.6F*(uref[t1][x + 8][y + 7][z + 8] + uref[t1][x + 8][y + 9][z + 8]))/((h_y*h_y)
+ (r14 - 1.78571429e-3F*(uref[t1][x + 4][y + 8][z + 8] + uref[t1][x + 12][y + 8][z + 8]) +
2.53968254e-2F*(uref[t1][x + 5][y + 8][z + 8] + uref[t1][x + 11][y + 8][z + 8]) -
2.0e-1F*(uref[t1][x + 6][y + 8][z + 8] + uref[t1][x + 10][y + 8][z + 8]) + 1.6F*(uref[t1][x + 7]
[y + 8][z + 8] + uref[t1][x + 9][y + 8][z + 8]))/((h_x*h_x)))/(r11*r12 + r13*damp[x + 1][y + 1][
+ 1]);
        }
      }
    }
  }
}
}
}
```

The generated C code - source injection


```c
    /* Begin section1 */
    #pragma omp parallel num_threads(nthreads_nonaffine)
    {
      int chunk_size = (int)(fmax(1, (1.0F/3.0F)*(p_src_M - p_src_m + 1)/nthreads_nonaffine));
      #pragma omp for collapse(1) schedule(dynamic,chunk_size)
      for (int p_src = p_src_m; p_src <= p_src_M; p_src += 1)
      {
        int ii_src_0 = (int)(floor((-o_x + src_coords[p_src][0])/h_x));
        int ii_src_1 = (int)(floor((-o_y + src_coords[p_src][1])/h_y));
        int ii_src_2 = (int)(floor((-o_z + src_coords[p_src][2])/h_z));
        int ii_src_3 = (int)(floor((-o_z + src_coords[p_src][2])/h_z)) + 1;
        int ii_src_4 = (int)(floor((-o_y + src_coords[p_src][1])/h_y)) + 1;
        int ii_src_5 = (int)(floor((-o_x + src_coords[p_src][0])/h_x)) + 1;
        float px = (float)(-h_x*(int)(floor((-o_x + src_coords[p_src][0])/h_x)) - o_x + src_coords[p_src][0]);
        float py = (float)(-h_y*(int)(floor((-o_y + src_coords[p_src][1])/h_y)) - o_y + src_coords[p_src][1]);
        float pz = (float)(-h_z*(int)(floor((-o_z + src_coords[p_src][2])/h_z)) - o_z + src_coords[p_src][2]);
        if (ii_src_0 >= x_m - 1 && ii_src_1 >= y_m - 1 && ii_src_2 >= z_m - 1 && ii_src_0 <= x_M + 1 && ii_src_1
<= y_M + 1 && ii_src_2 <= z_M + 1)
        {
          float r0 = 4.49016082216644F*(vp[ii_src_0 + 8][ii_src_1 + 8][ii_src_2 + 8]*vp[ii_src_0 + 8][ii_src_1 + 8]
[ii_src_2 + 8])*(-px*py*pz/(h_x*h_y*h_z) + px*py/(h_x*h_y) + px*pz/(h_x*h_z) - px/h_x + py*pz/(h_y*h_z) - py/h_y -
pz/h_z + 1)*src[time][p_src];
          #pragma omp atomic update
          uref[t0][ii_src_0 + 8][ii_src_1 + 8][ii_src_2 + 8] += r0;
        }
        if (ii_src_0 >= x_m - 1 && ii_src_1 >= y_m - 1 && ii_src_3 >= z_m - 1 && ii_src_0 <= x_M + 1 && ii_src_1
<= y_M + 1 && ii_src_3 <= z_M + 1)
        {
          float r1 = 4.49016082216644F*(vp[ii_src_0 + 8][ii_src_1 + 8][ii_src_3 + 8]*vp[ii_src_0 + 8][ii_src_1 + 8]
[ii_src_3 + 8])*(px*py*pz/(h_x*h_y*h_z) - px*pz/(h_x*h_z) - py*pz/(h_y*h_z) + pz/h_z)*src[time][p_src];
          #pragma omp atomic update
          uref[t0][ii_src_0 + 8][ii_src_1 + 8][ii_src_3 + 8] += r1;
        }
        if (ii_src_0 >= x_m - 1 && ii_src_2 >= z_m - 1 && ii_src_4 >= y_m - 1 && ii_src_0 <= x_M + 1 && ii_src_2
<= z_M + 1 && ii_src_4 <= y_M + 1)
        {
          float r2 = 4.49016082216644F*(vp[ii_src_0 + 8][ii_src_4 + 8][ii_src_2 + 8]*vp[ii_src_0 + 8][ii_src_4 + 8]
[ii_src_2 + 8])*(px*py*pz/(h_x*h_y*h_z) - px*py/(h_x*h_y) - py*pz/(h_y*h_z) + py/h_y)*src[time][p_src];
```

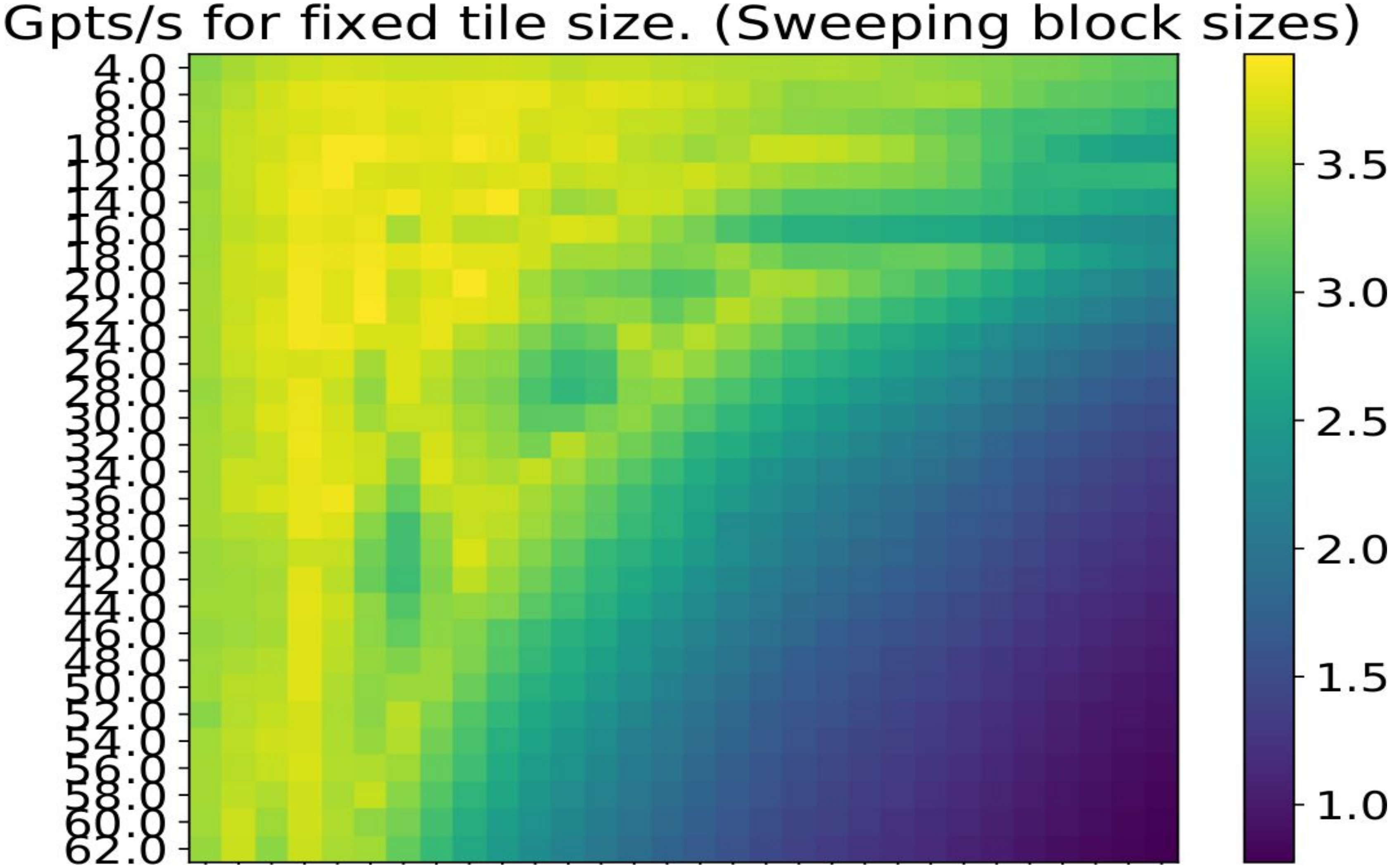
Algorithm 3: Source injection pseudocode.

```
for t = 1 to nt do
    foreach s in sources do
        # Find on the grid coordinates
        src_x_min = floor(src_coords[s][0], ox)
        src_x_max = ceil(src_coords[s][0], ox)
        src_y_min = floor(src_coords[s][1], oy)
        src_y_max = ceil(src_coords[s][1], oy)
        src_z_min = floor(src_coords[s][2], oz)
        src_z_max = ceil(src_coords[s][2], oz)
        # Compute weights
        px = f(src_coords[s][0], ox)
        py = f(src_coords[s][1], oy)
        pz = f(src_coords[s][2], oz)
        # Unrolled for 8 points
        if src_x_min, src_y_min, src_z_min in grid then
            r0 = v(src_x_min, src_y_min, src_z_min, src[t][s]);
            u[t, src_x_min, src_y_min, src_z_min] + = r0)
        :
        if src_x_max, src_y_max, src_z_max in grid then
            r7 = v(src_x_max, src_y_max, src_z_max src[t][s]);
            u[t, src_x_max, src_y_max, src_z_max] + = r7)
```

Floor, ceil of of the-grid

Weights of impact

Unrolled loop for each affected point, compute injection part and add to field



Gpts/s for fixed tile size. (Sweeping block sizes)
4.0
6.0
8.0
10.0
12.0
14.0
16.0
18.0
20.0
22.0
24.0
26.0
28.0
30.0
32.0
34.0
36.0
38.0
40.0
42.0
44.0
46.0
48.0
50.0
52.0
54.0
56.0
58.0
60.0
62.0
3.5
3.0
2.5
2.0
1.5
1.0

Algorithm 3: Source injection pseudocode.

```
for t = 1 to nt do
    foreach s in sources do
        # Find on the grid coordinates
        src_x_min = floor(src_coords[s][0], ox)
        src_x_max = ceil(src_coords[s][0], ox)
        ⋮
        # Compute weights
        px = f(src_coords[s][0], ox)
        ⋮
        # Unrolled for 8 points (2^3, 3D case)
        if src_x_min, ... in grid then
            r0 = v(src_x_min, ...src[t][s]);
            u[t, src_x_min, ...] + = r0)
        ⋮
        if src_x_max, ... in grid then
            r7 = v(src_x_max, ...src[t][s]);
            u[t, src_x_max, ...] + = r7)
```

Cache aware roofline model

From here: https://crd.lbl.gov/departments/computer-science/par/research/roofline/introduction/

Effects of Cache Behavior on Arithmetic Intensity

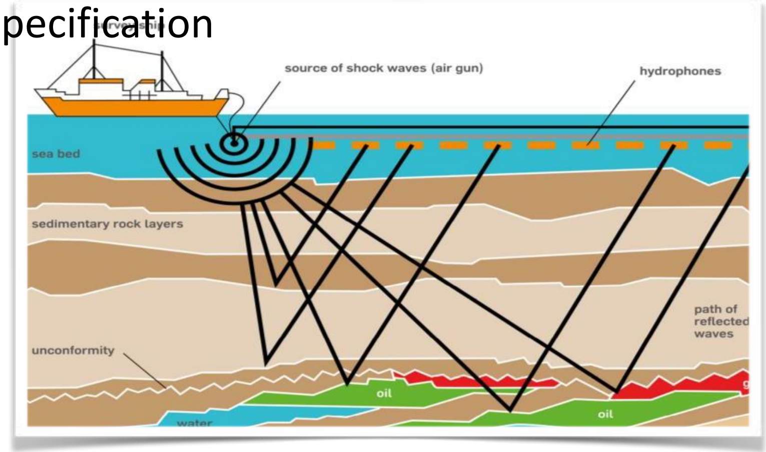
The Roofline model requires an estimate of total data movement. On cache-based architectures, the 3C's cache model highlights the fact that there can be more than simply compulsory data movement. Cache capacity and conflict misses can increase data movement and reduce arithmetic intensity. Similarly, superfluous cache write-allocations can result in a doubling of data movement. The vector initialization operation x[i]=0.0 demands one write allocate and one write back per cache line touched. The write allocate is superfluous as all elements of that cache line are to be overwritten. Unfortunately, the presence of hardware stream prefetchers can make it very difficult to quantify how much beyond compulsory data movement actually occurred.

A bit of background

- **PDEs** are everywhere:
  computational fluid dynamics, image processing, weather forecasting, seismic and medical imaging.

- Numerical analysis => **finite-difference (FD)** methods to solve DEs by approximating derivatives with finite differences.

- **Devito:** Fast Stencil Computation from Symbolic Specification



- **Goal**:
  To improve performance of stencils stemming from practical applications using temporal blocking

CPC Compilers for Parallel Computing

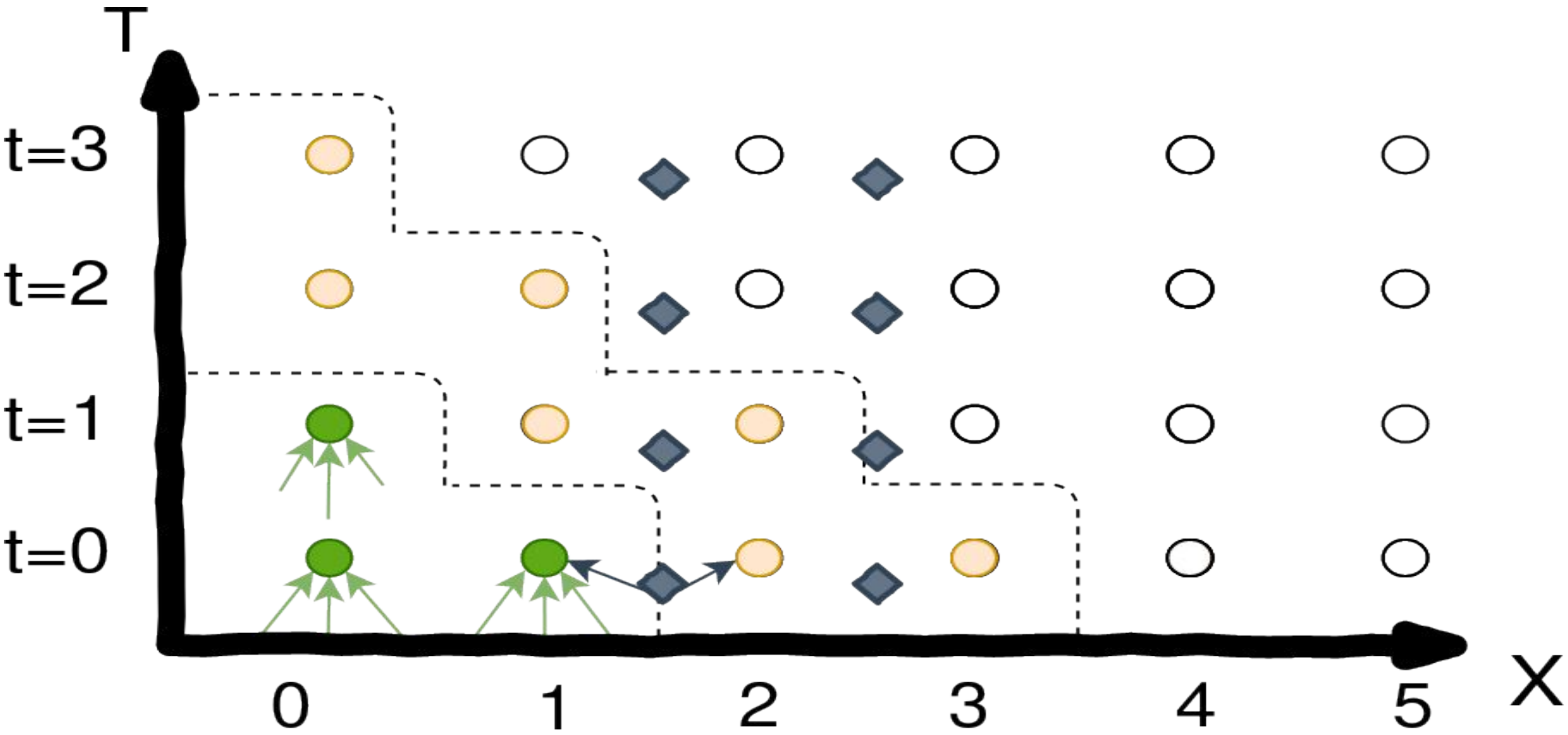
Algorithm 3: Source injection pseudocode.

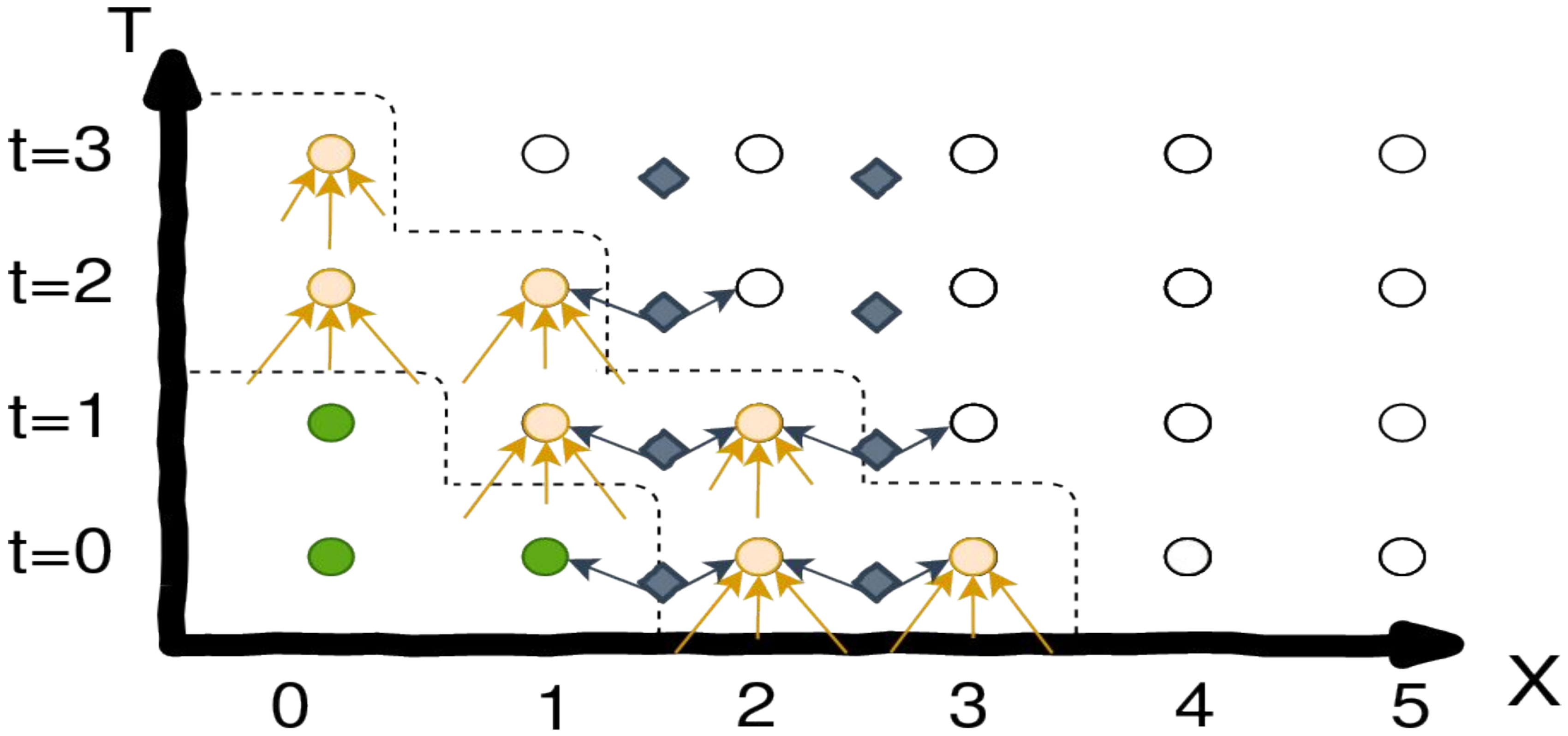
```
for t = 1 to nt do
    foreach s in sources do
        # Find on the grid coordinates
        src_x_min = floor(src_coords[s][0], ox)
        src_x_max = ceil(src_coords[s][0], ox)
        ⋮
        # Compute weights
        px = f(src_coords[s][0], ox)
        ⋮
        # Unrolled for 8 points (2^3, 3D case)
        if src_x_min, ... in grid then
            r0 = v(src_x_min, ...src[t][s]);
            u[t, src_x_min, ...] + = r0)
        ⋮
        if src_x_max, ... in grid then
            r7 = v(src_x_max, ...src[t][s]);
            u[t, src_x_max, ...] + = r7)
```

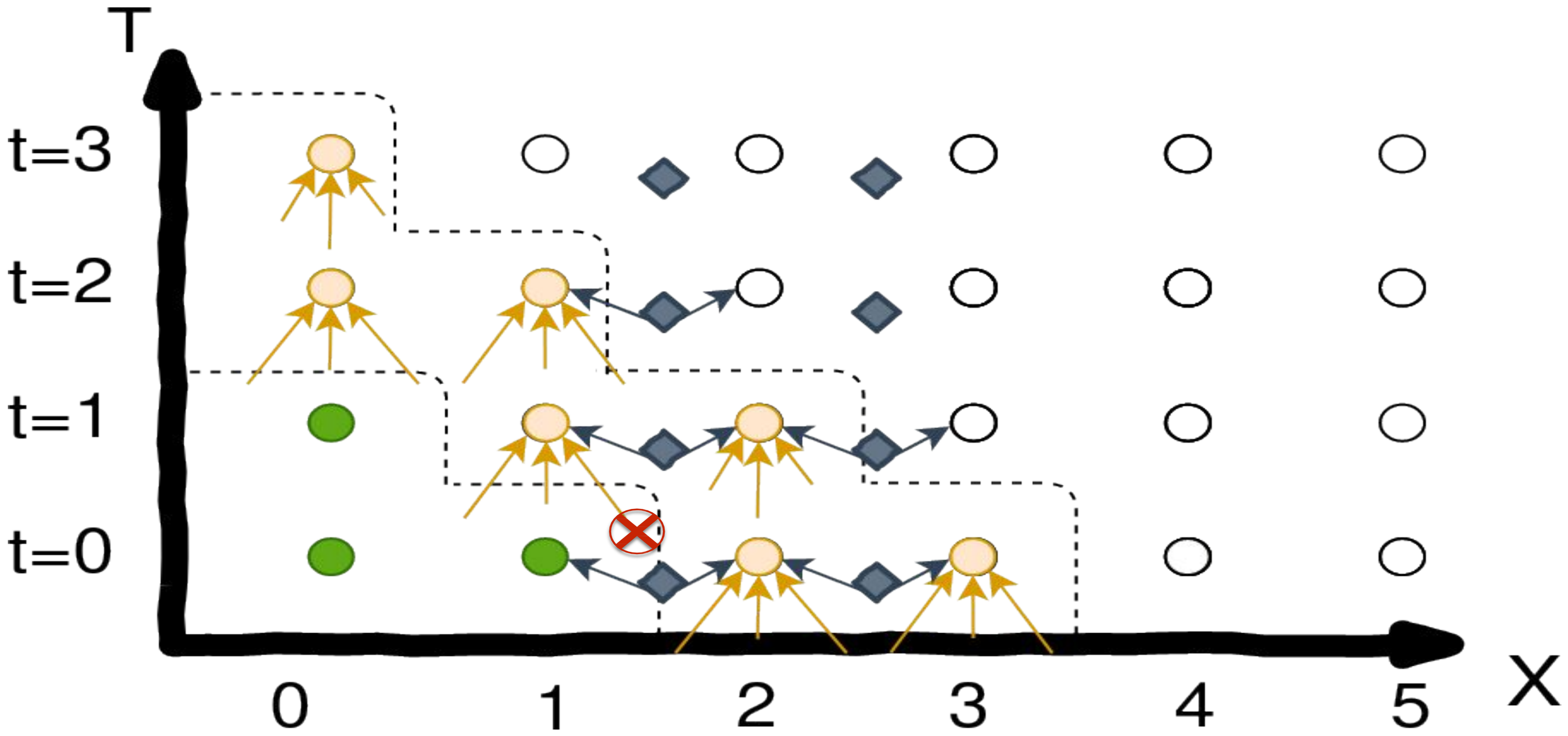
Discover affected points

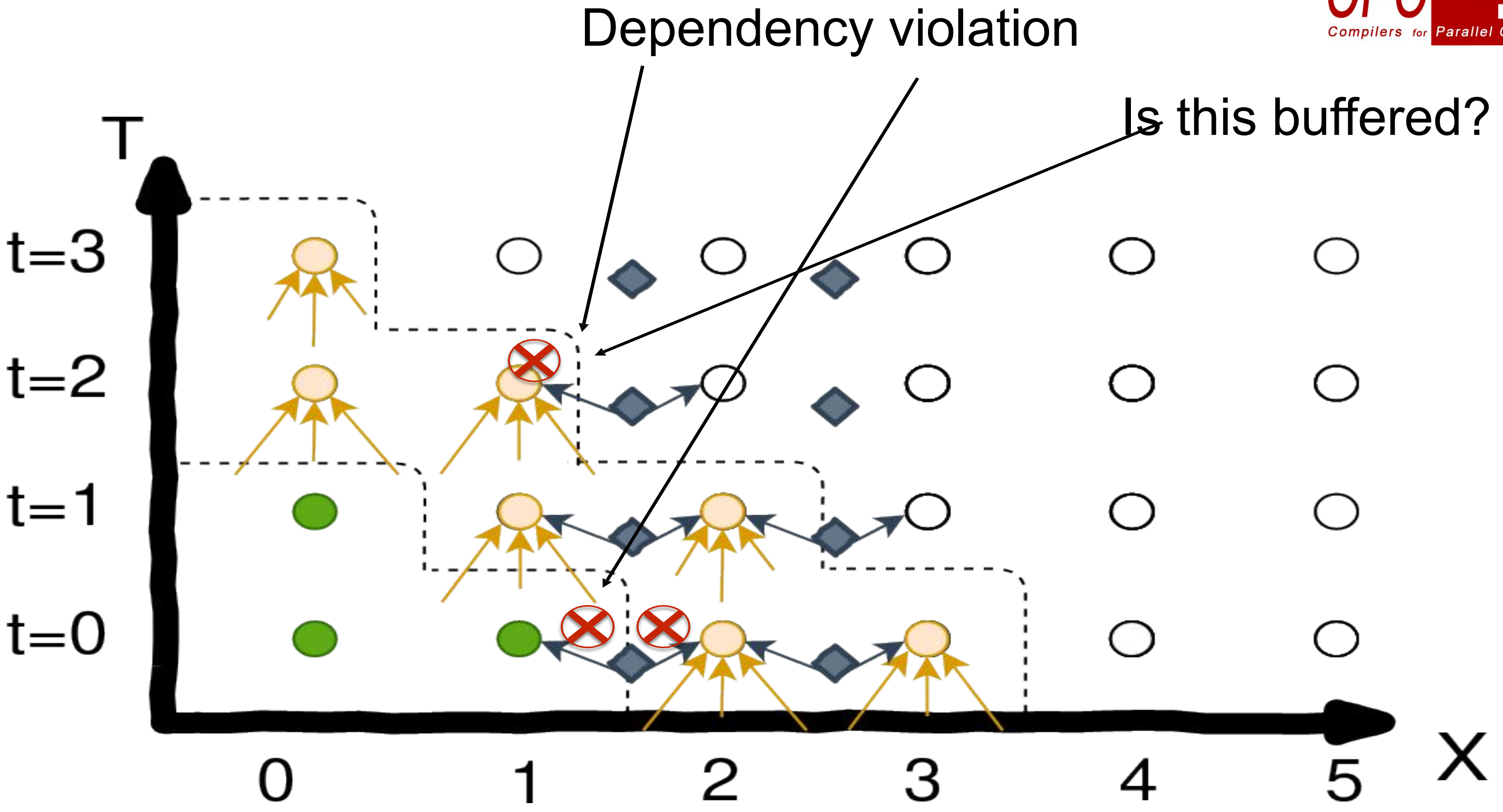
Weights of impact

Unrolled loop for each affected point, compute injection part and add to field

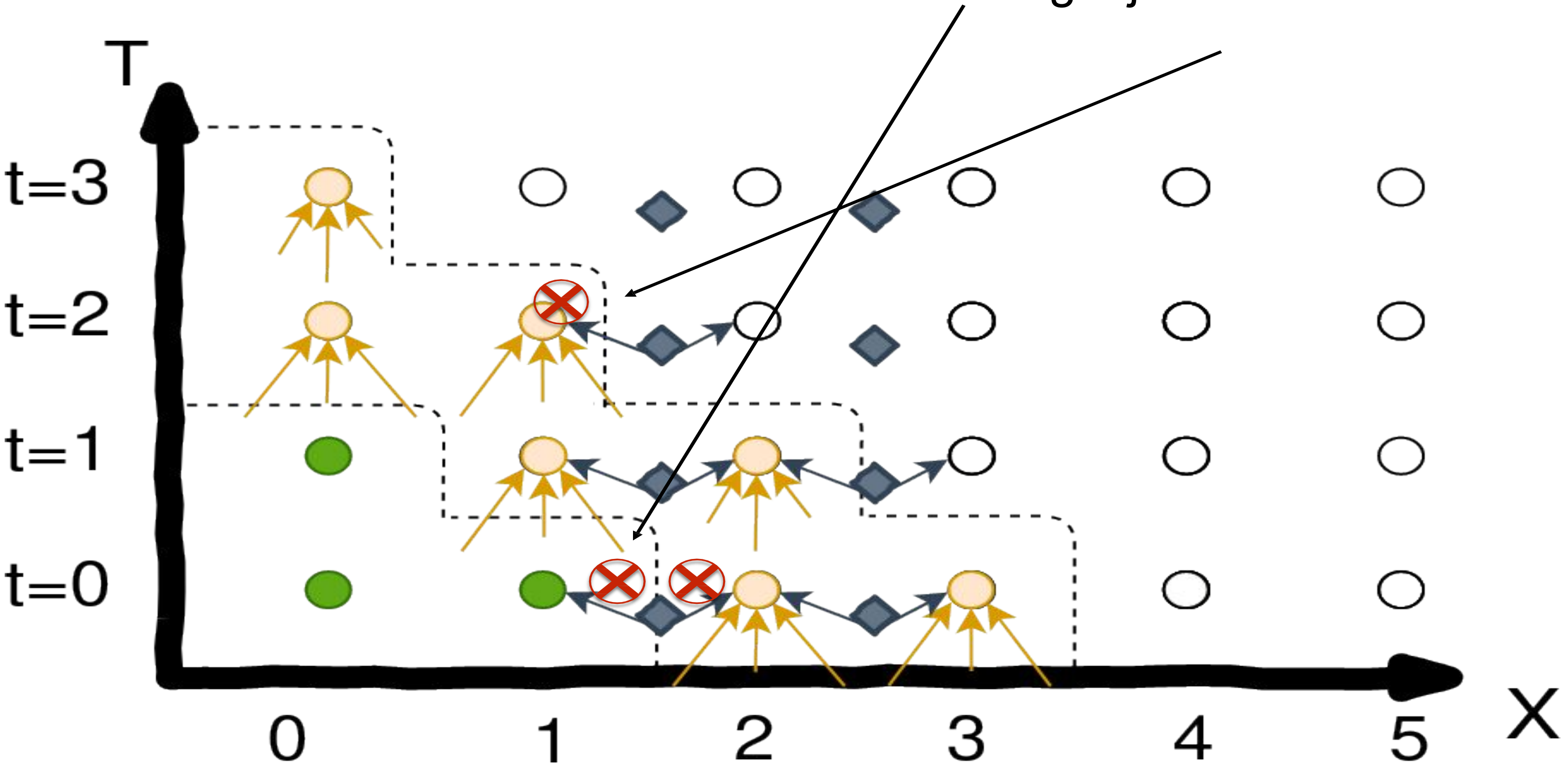

T
t=3
t=2
t=1
t=0
0
1
2
3
4
5
X

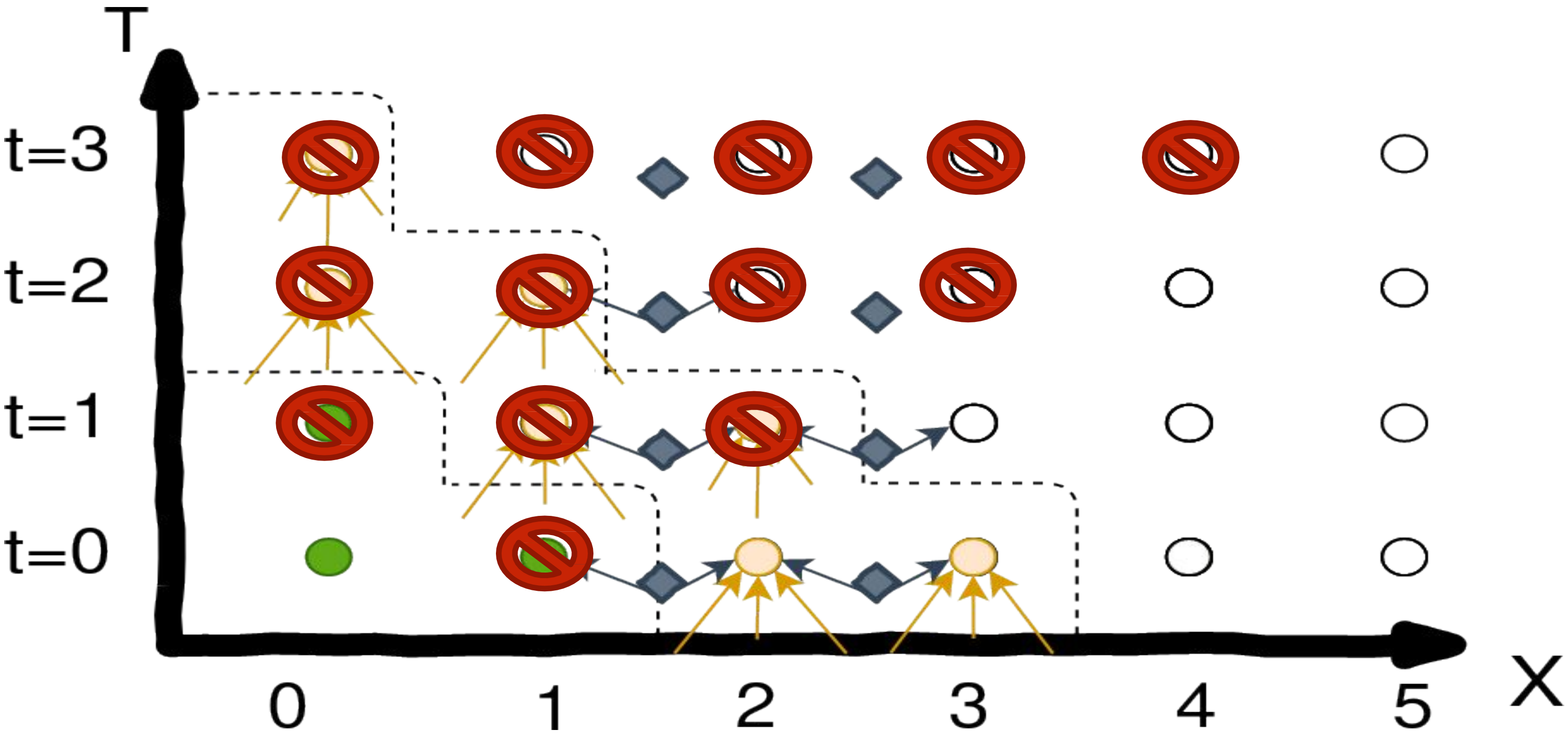

T
t=3
t=2
t=1
t=0
0
1
2
3
4
5
X


T
t=3
t=2
t=1
t=0
0
1
2
3
4
5
X


Dependency violation
Is this buffered?
T
t=3
t=2
t=1
t=0
0
1
2
3
4
5
X

Missing injection?





T
t=3
t=2
t=1
t=0
0
1
2
3
4
5
X

CPC Compilers for Parallel Computing

Algorithm 1: A typical time-stepping loop nest structure for a stencil update with source injection. This stencil has one temporal and three spatial dimensions.

for t = 1 ***to*** nt **do**
  for x = 1 ***to*** nx **do**
    for y = 1 ***to*** ny **do**
      for z = 1 ***to*** nz **do**
        $A(t, x, y, z) \equiv$ u[t, x, y, z] = u[t-1, x, y, z] + $\sum_{r=1}^{r=so/2} w_r \Big[$ u[t-1, x - r, y, z] + u[t-1, x + r, y, z] + u[t-1, x, y - r, z] + u[t-1, x, y + r, z] + u[t-1, x, y, z - r] + u[t-1, x, y, z + r] $\Big]$;
  foreach s ***in*** sources **do**
    for i = 1 ***to*** np **do**
      xs, ys, zs = map(s, i);
      u[t, xs, ys, zs] + = $f(src(t, s))$

Non-aligned

Algorithm 6: Stencil kernel update with fused - reduced size iteration space - source injection.

for t = 1 ***to*** nt **do**
  for x = 1 ***to*** nx **do**
    for y = 1 ***to*** ny **do**
      for z = 1 ***to*** nz **do**
        $A(t, x, y, z, s)$;
      for $z2$ = 1 ***to*** nnz_mask[x][y] **do**
        zind = Sp_SM[x, y, z];
        u[t, x, y, z2] +=
        SM[x, y, zind] * src_dcmp[t, SID[x, y, zind]];

Aligned

Listing 1: A typical time-stepping loop nest structure for a stencil update with source injection. This stencil has one temporal and three spatial dimensions.

```
for t = 1 to nt do
    for x = 1 to nx do
        for y = 1 to ny do
            for z = 1 to nz do
                A(t, x, y, z) ≡ u[t, x, y, z] = u[t-1, x, y, z] + Σ_{r=1}^{r=so/2} w_r (
                  u[t-1, x - r, y, z] + u[t-1, x + r, y, z] + u[t-1, x, y - r, z] +
                  u[t-1, x, y + r, z] + u[t-1, x, y, z - r] + u[t-1, x, y, z + r] );
    foreach s in sources do // For every source
        for i = 1 to np do // Get the points affected
            xs, ys, zs = map(s, i) // through indirection
            u[t, xs, ys, zs] + = f(src(t, s)) // add their impact
             on the field
```

Non-aligned

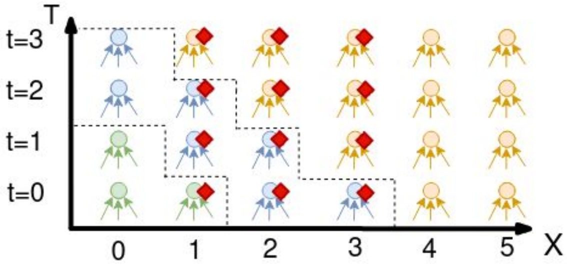
Listing 5: Stencil kernel update with fused - reduced size iteration space - source injection.

```
for t = 1 to nt do
    for x = 1 to nx do
        for y = 1 to ny do
            for z = 1 to nz do
                A(t, x, y, z, s);
            for z2 = 1 to nnz_mask[x][y] do
                I(t, x, y, z, s) ≡{ zind = Sp_SM[x, y, z2];
                u[t, x, y, z2] +=
                SM[x, y, zind] * src_dcmp[t, SID[x, y, zind]]; }
```

Aligned

Compilers for Parallel Computing

- ✓ Aligned to grid
- ✓ Same OPS
- ✓ Parallelism
- ✓ SIMD (?)
- ⏭ Apply TB



The transformation in Devito-DSL

```
u = TimeFunction(name="u", grid=model.grid, space_order=so, time_order=2)
src_term = src.inject(field=u.forward, expr=src * dt**2 / model.m)
pde = model.m * u.dt2 - u.laplace + model.damp * u.dt
stencil = Eq(u.forward, solve(pde, u.forward))
op = Operator([stencil, src_term])
```

The transformation in Devito-DSL

```
# f : perform source injection on an empty grid
f = TimeFunction(name="f", grid=model.grid, space_order=so, time_order=2)  src_f
= src.inject(field=f.forward, expr=src * dt**2 / model.m)
op_f = Operator([src_f])
op_f_sum = op_f.apply(time=3)

nzinds = np.nonzero(f.data[0]) # nzinds is a tuple
```

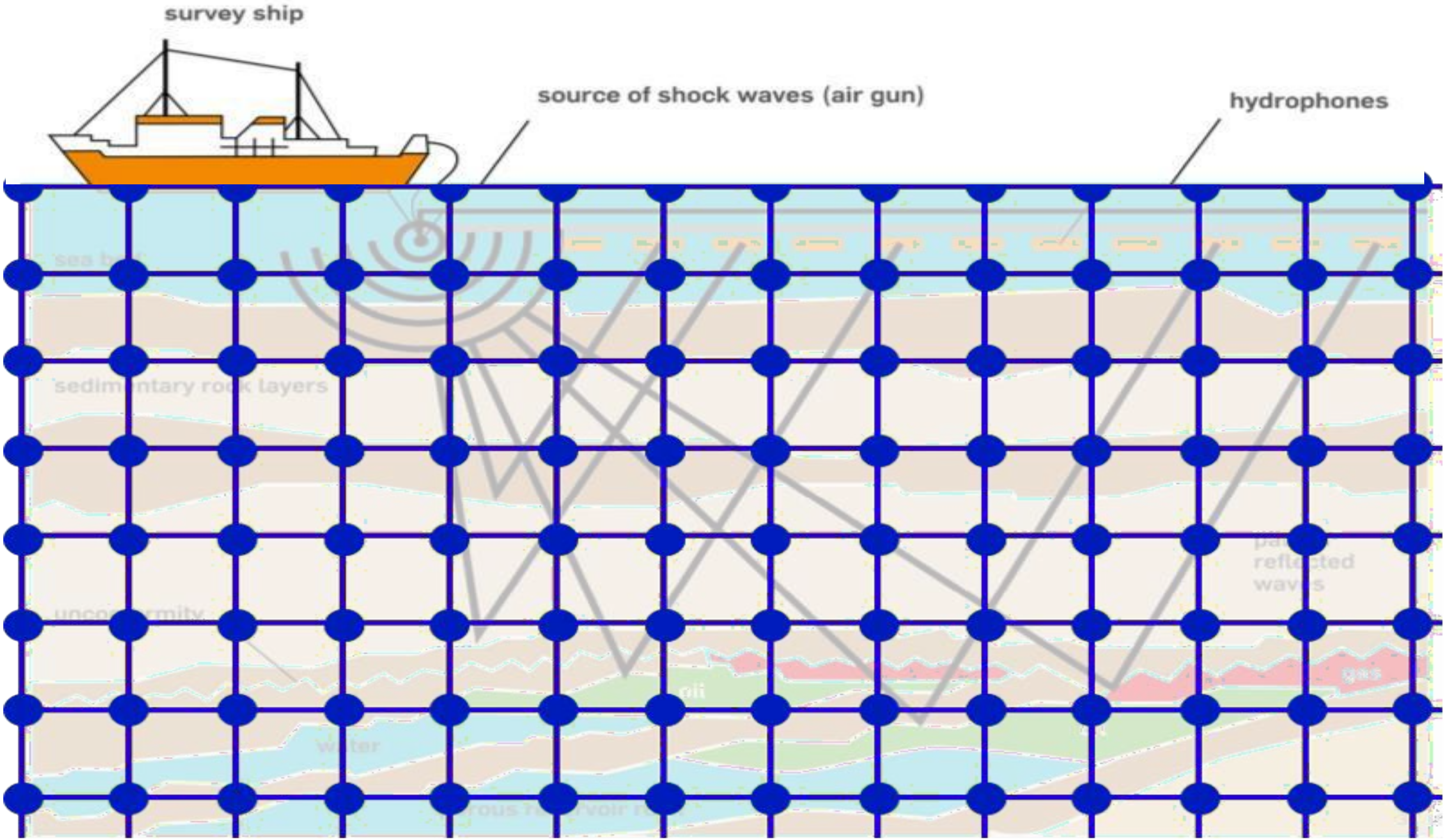
```
eq0 = Eq(sp_zi.symbolic_max, nnz_sp_source_mask[x, y] - 1, implicit_dims=(time, x, y))  eq1 =
Eq(zind, sp_source_mask[x, y, sp_zi], implicit_dims=(time, x, y, sp_zi))

mask_expr = source_mask[x, y, zind] * save_src[time, source_id[x, y, zind]]
eq2 = Inc(usol.forward[t+1, x, y, zind], mask_expr, implicit_dims=(time, x, y, sp_zi))  pde_2 =

model.m * usol.dt2 - usol.laplace + model.damp * usol.dt

stencil_2 = Eq(usol.forward, solve(pde_2, usol.forward))
```

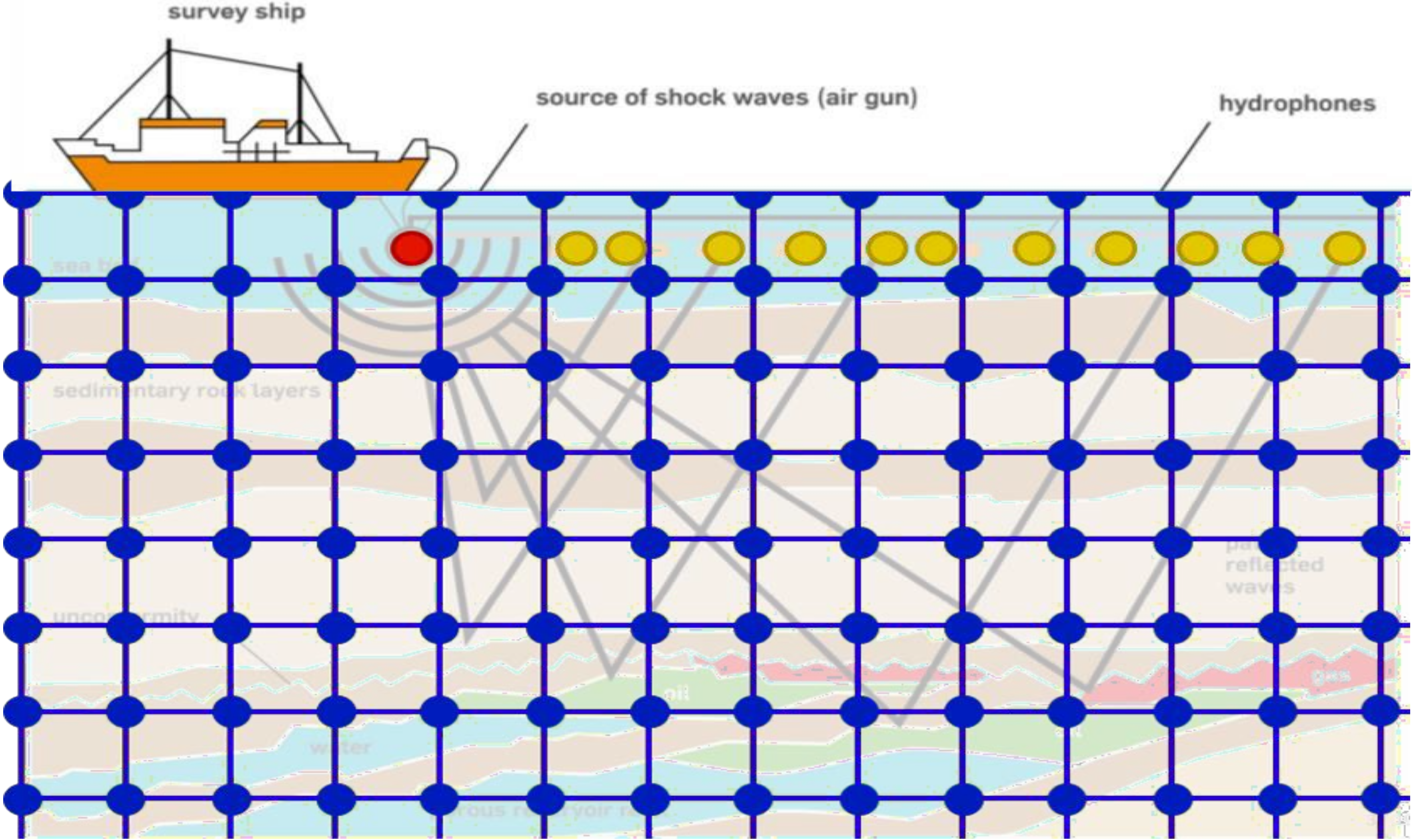
Fuse iteration spaces

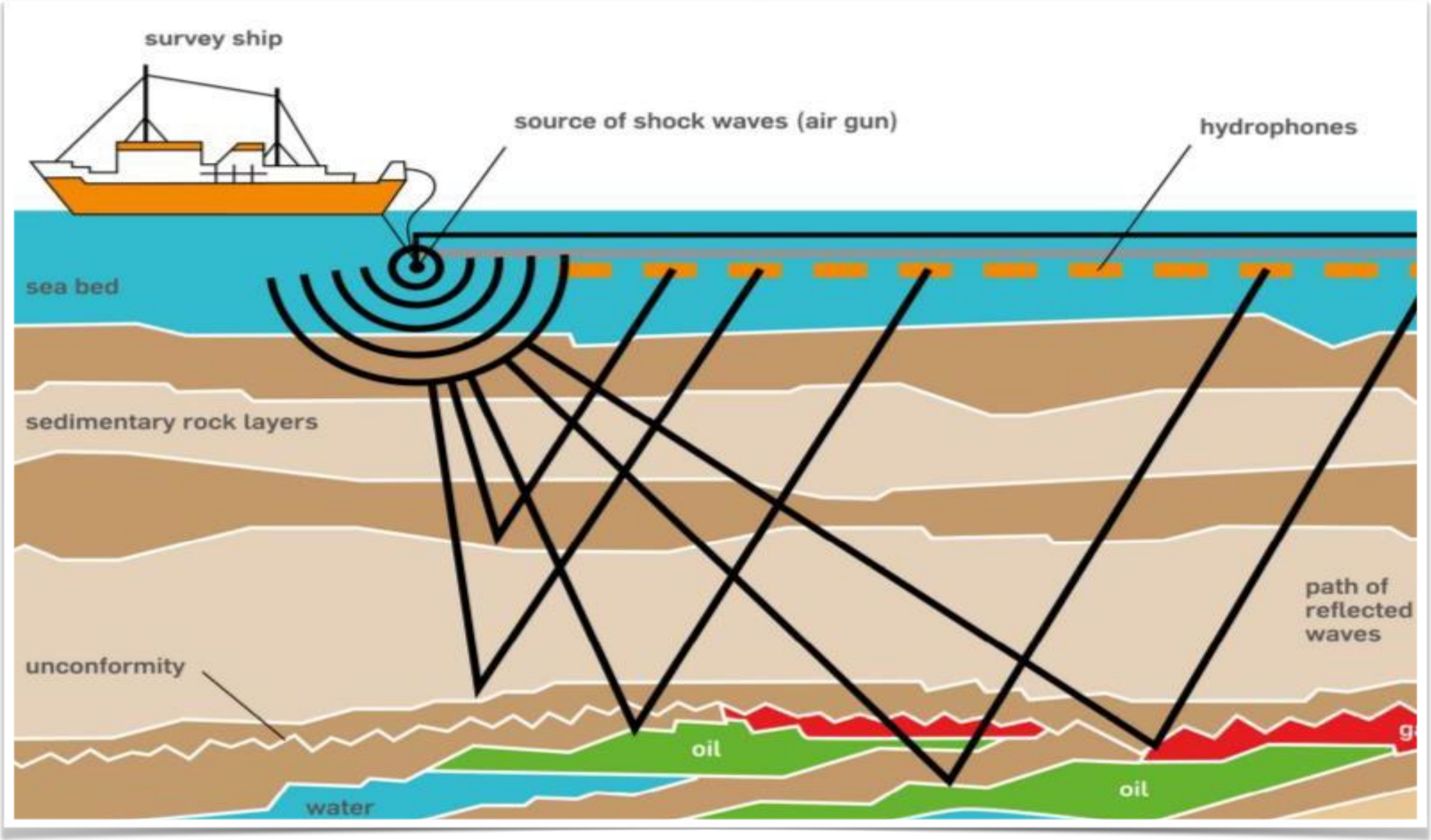
- Indirection mapping has changed. We still use indirections but now they are on the point.
- By using the aligned structure, we fuse the source injection loop inside the kernel update iteration space.
- The source mask SM is used to add (if 1) or not (if 0) the impact and SID is used to indirect to the impact values using the traversed grid coordinates

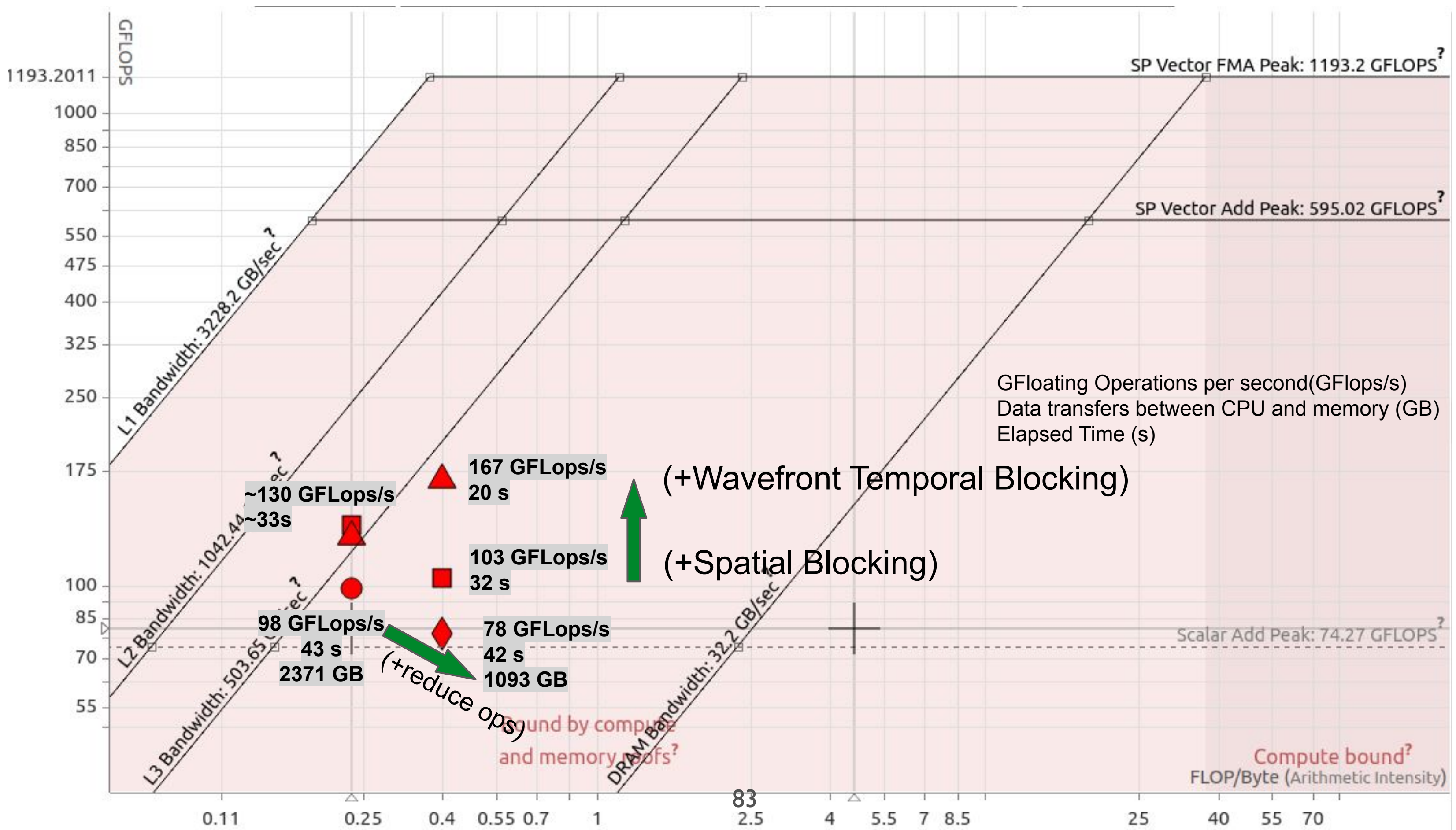
Listing 4: Stencil kernel update with fused source injection.

```
for t = 1 to nt do
  for x = 1 to nx do
    for y = 1 to ny do
      for z = 1 to nz do
        A(t, x, y, z, s);
      for z2 = 1 to nz do
        u[t, x, y, z2] + = SM[x, y, z2] * src_dcmp[t, SID[x, y, z2]];
```


survey ship
source of shock waves (air gun)
hydrophones


survey ship
source of shock waves (air gun)
hydrophones


survey ship
source of shock waves (air gun)
hydrophones
sea bed
sedimentary rock layers
path of reflected waves
unconformity
oil
oil
water


GFLOPS
1193.2011
1000
850
700
550
475
400
325
250
175
100
85
70
55
SP Vector FMA Peak: 1193.2 GFLOPS
SP Vector Add Peak: 595.02 GFLOPS
L1 Bandwidth: 3228.2 GB/sec
L2 Bandwidth: 1042.44 GB/sec
L3 Bandwidth: 503.65 GB/sec
DRAM Bandwidth: 32.2 GB/sec
Scalar Add Peak: 74.27 GFLOPS
GFloating Operations per second(GFlops/s)
Data transfers between CPU and memory (GB)
Elapsed Time (s)
167 GFLops/s
20 s
~130 GFLops/s
~33s
103 GFLops/s
32 s
98 GFLops/s
43 s
2371 GB
78 GFLops/s
42 s
1093 GB
(+Wavefront Temporal Blocking)
(+Spatial Blocking)
(+reduce ops)
Bound by compute and memory roofs
Compute bound
FLOP/Byte (Arithmetic Intensity)
0.11
0.25
0.4
0.55
0.7
1
2.5
4
5.5
7
8.5
25
40
55
70