TEMPORAL BLOCKING OF FINITE-DIFFERENCE STENCIL OPERATORS WITH SPARSE "OFF-THE-GRID" SOURCES IN DEVITO

George Bisbas[1], Fabio Luporini[2], Mathias Louboutin[3], Rhodri Nelson[1], Gerard Gorman[1], Paul Kelly[1]

[1]Imperial College London
[2]Devito Codes
[3]Georgia Tech

From Data Analysis to High-Performance Computing
joint online conference on
Domain-Specific Languages in High-Performance Computing
and
Intelligent Sensor Data Analysis for Smart Systems

What our work is about

- Temporal Blocking on practical simulations on top of **Devito-DSL**

- Practical simulations are complicated

- They consist of **sparse "off-the-grid"** operators
  (Not the typical stencil benchmark!)

- **Temporal blocking** is challenging to apply

- We present an approach to overcome limitations and improve performance

Motivation

- Domain-specific languages in high-performance computing
- Current status: Using a DSL to generate high performance code

High level - DSL specification

Optimization passes

HPC generated code

Motivation

- Domain-specific languages in high-performance computing
- Current status: Using a DSL to generate high performance code
- Goal : Using a DSL to generate **HIGHER** performance code

High level - DSL specification

Optimization passes

HPC generated code

Raise a bit more the performance bar

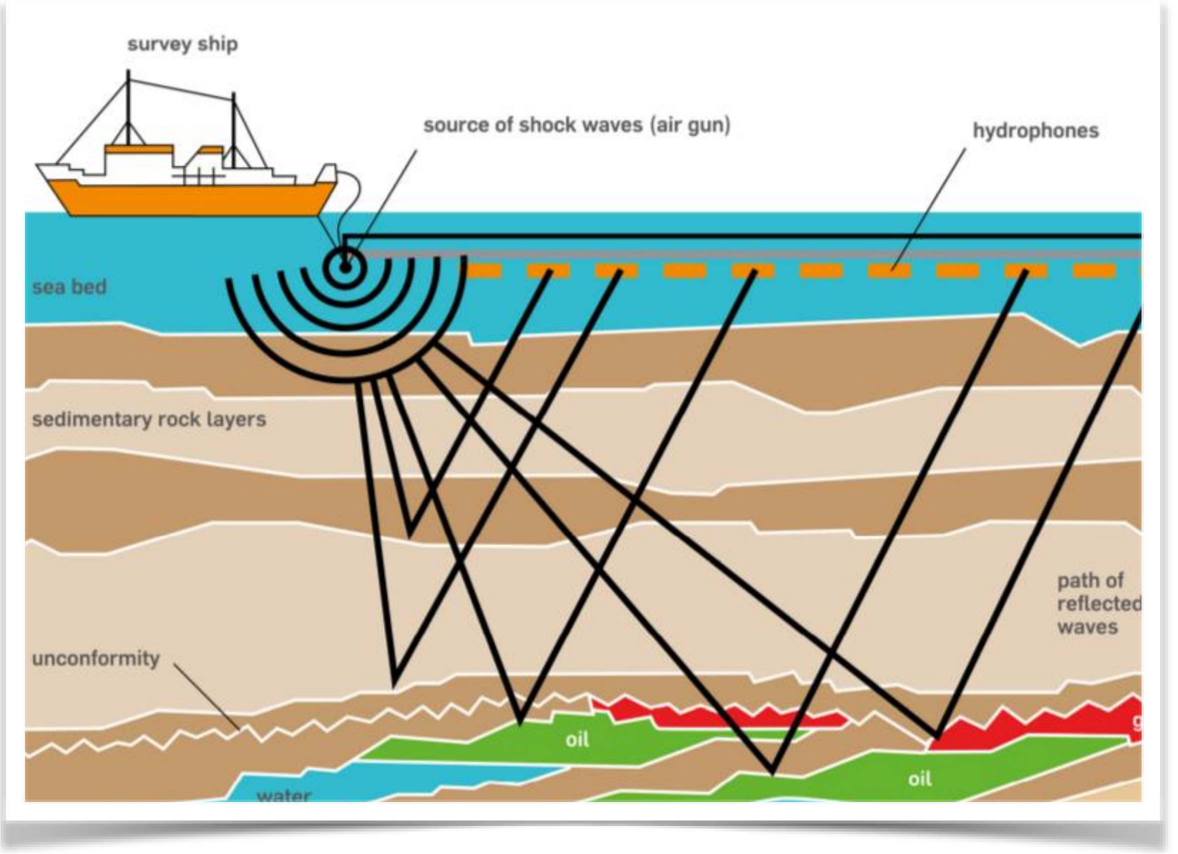
A bit of background

- **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

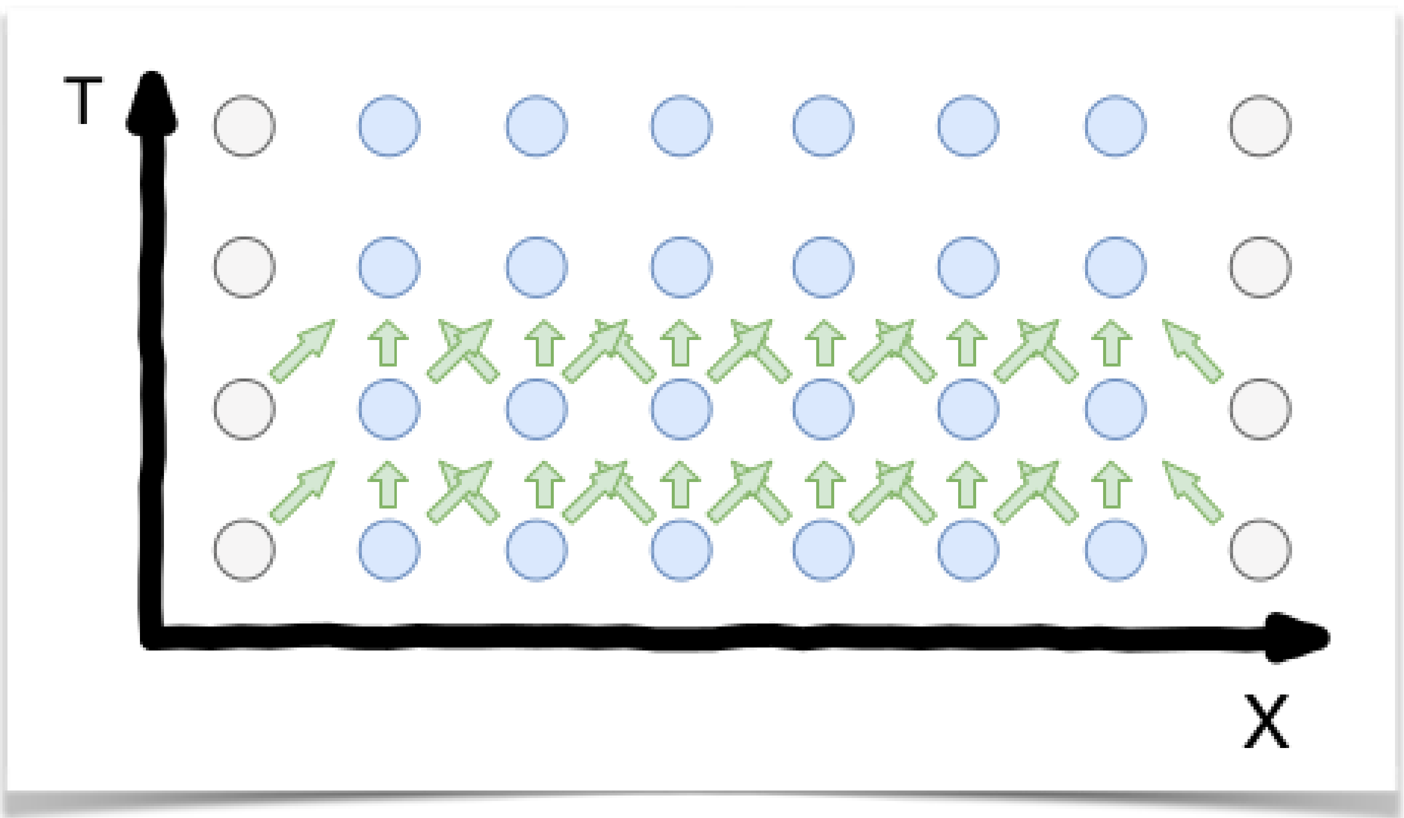
- **Goal**:
  To improve performance of stencils stemming from practical applications using temporal blocking



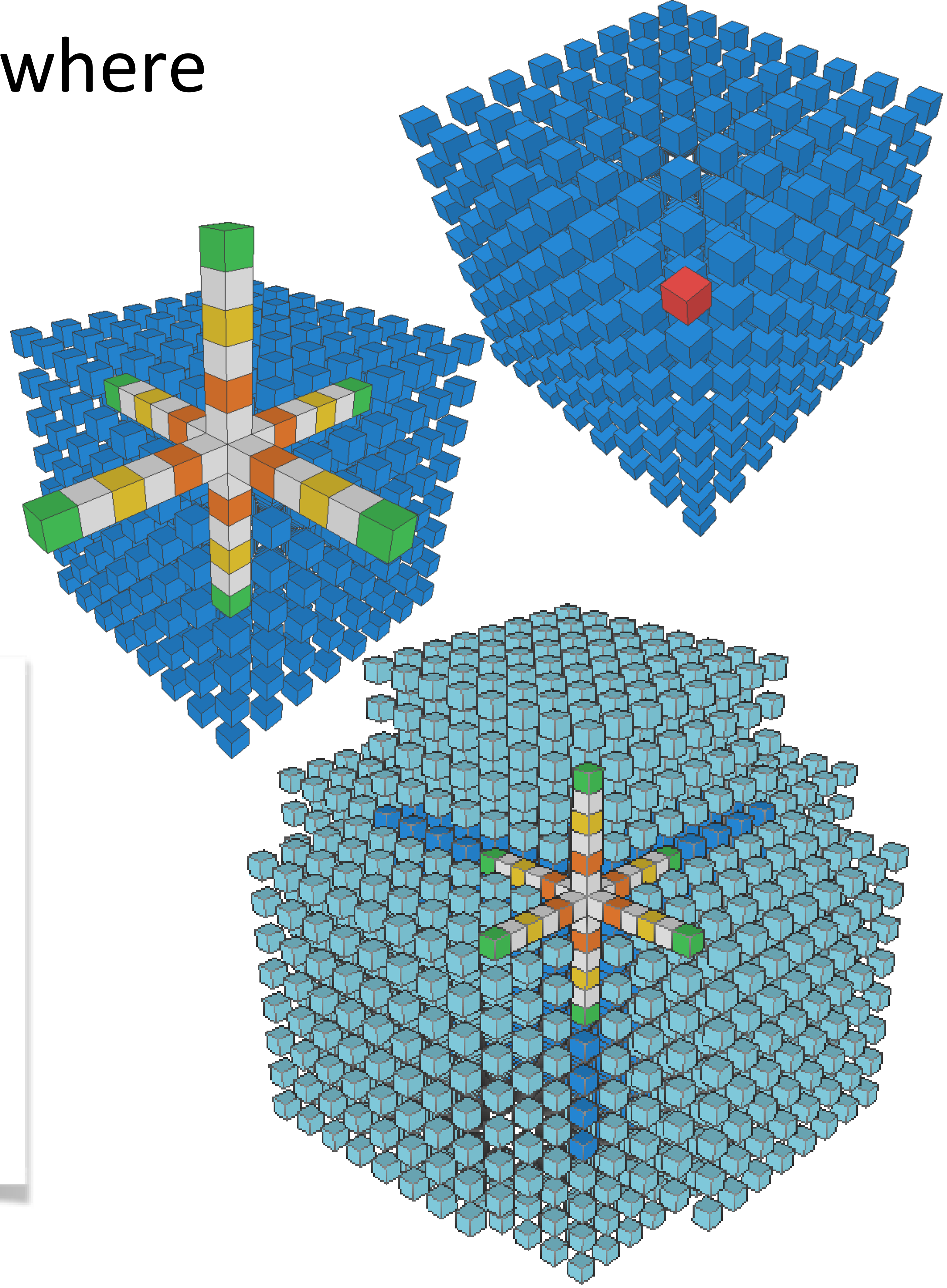
http://www.open.edu/openlearn/science--maths--technology/science/environmental--science/earths--physical--resources--petroleum/content--section--3.2.1

Stencils are everywhere

- Computing stencils on the FD grid
- Stencils used for benchmarking, vast literature on optimizing stencils...
- Parallelism (OpenMP, SIMD, MPI)
- From simplistic (1d-3pt), to wide and complex...



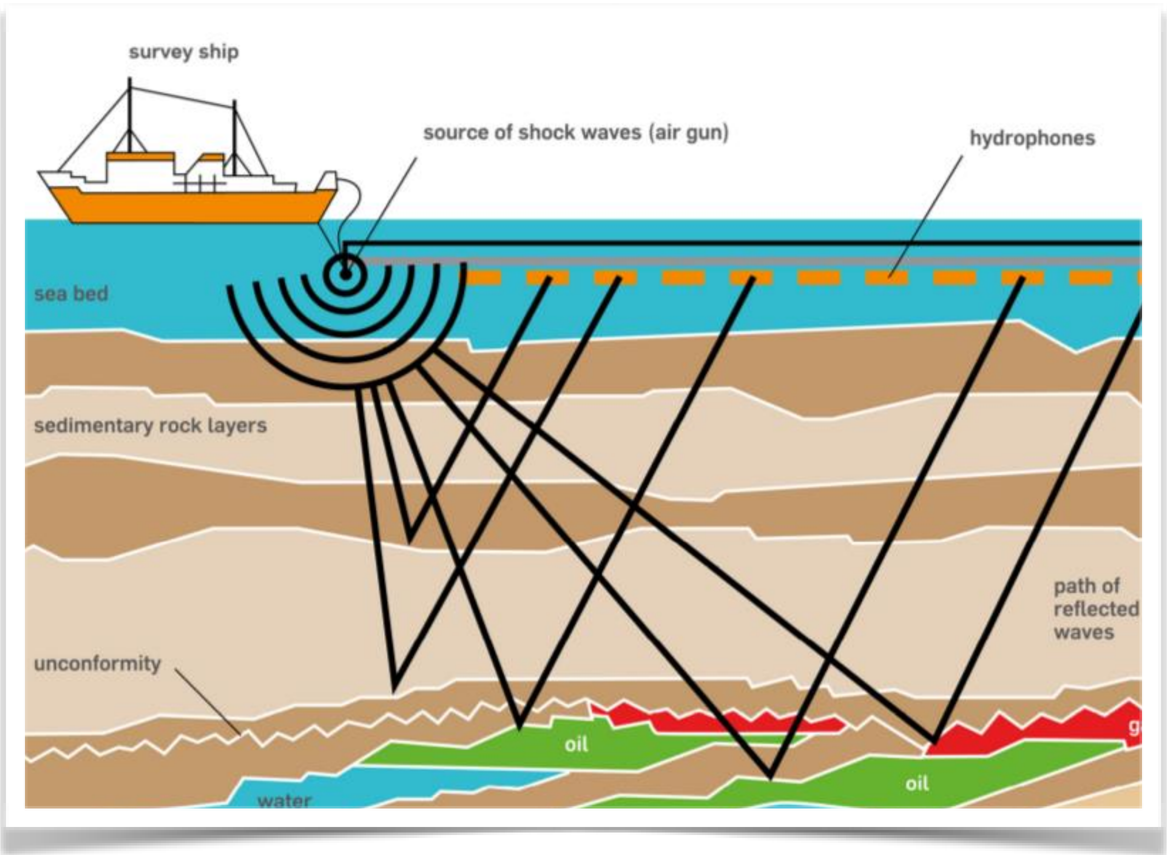
A 1d 3pt stencil update

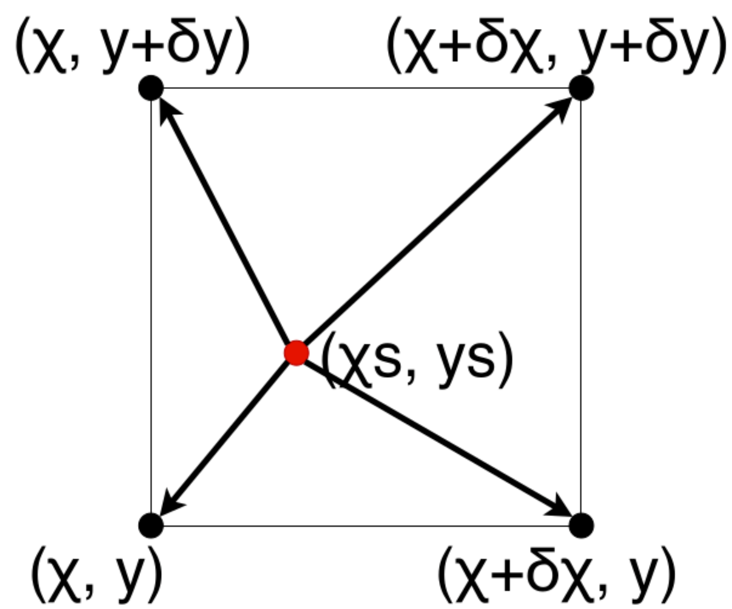


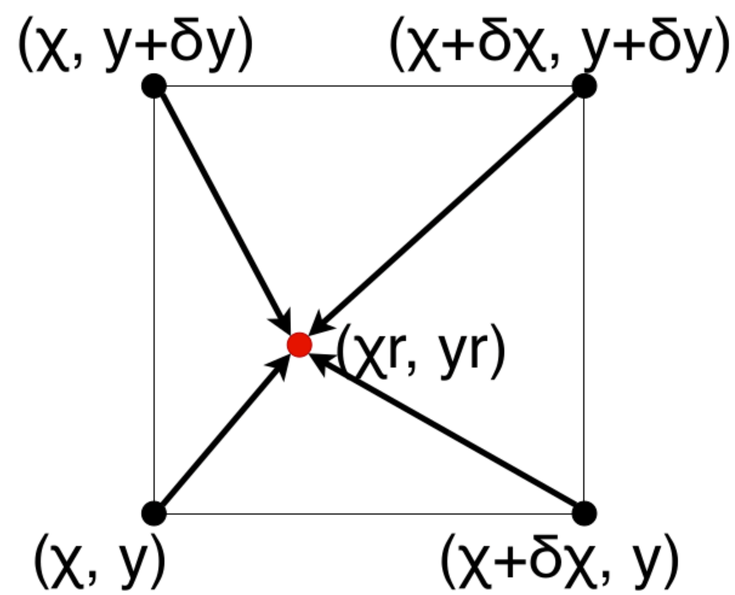
A 3d-19pt stencil update

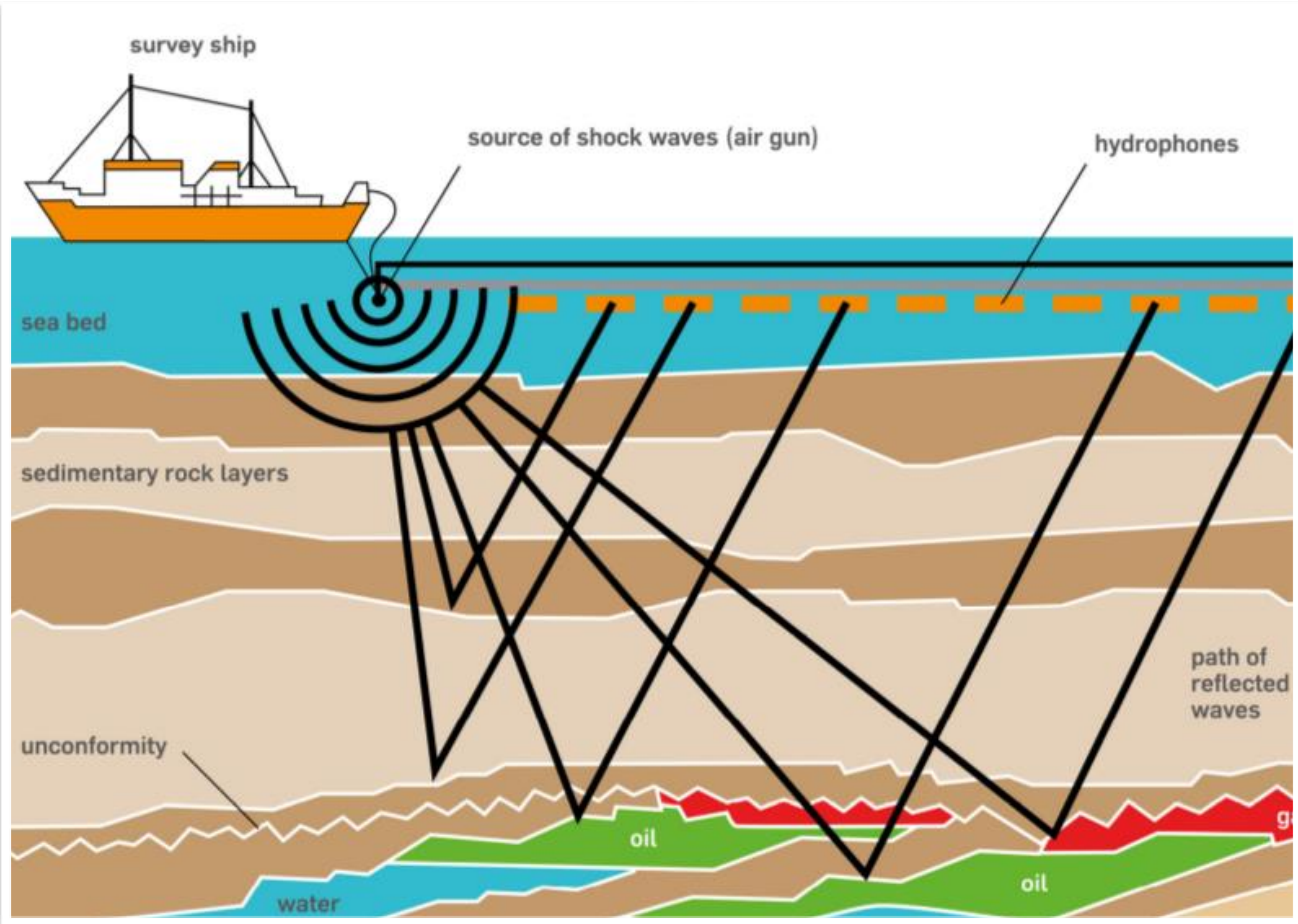
Modelling practical applications

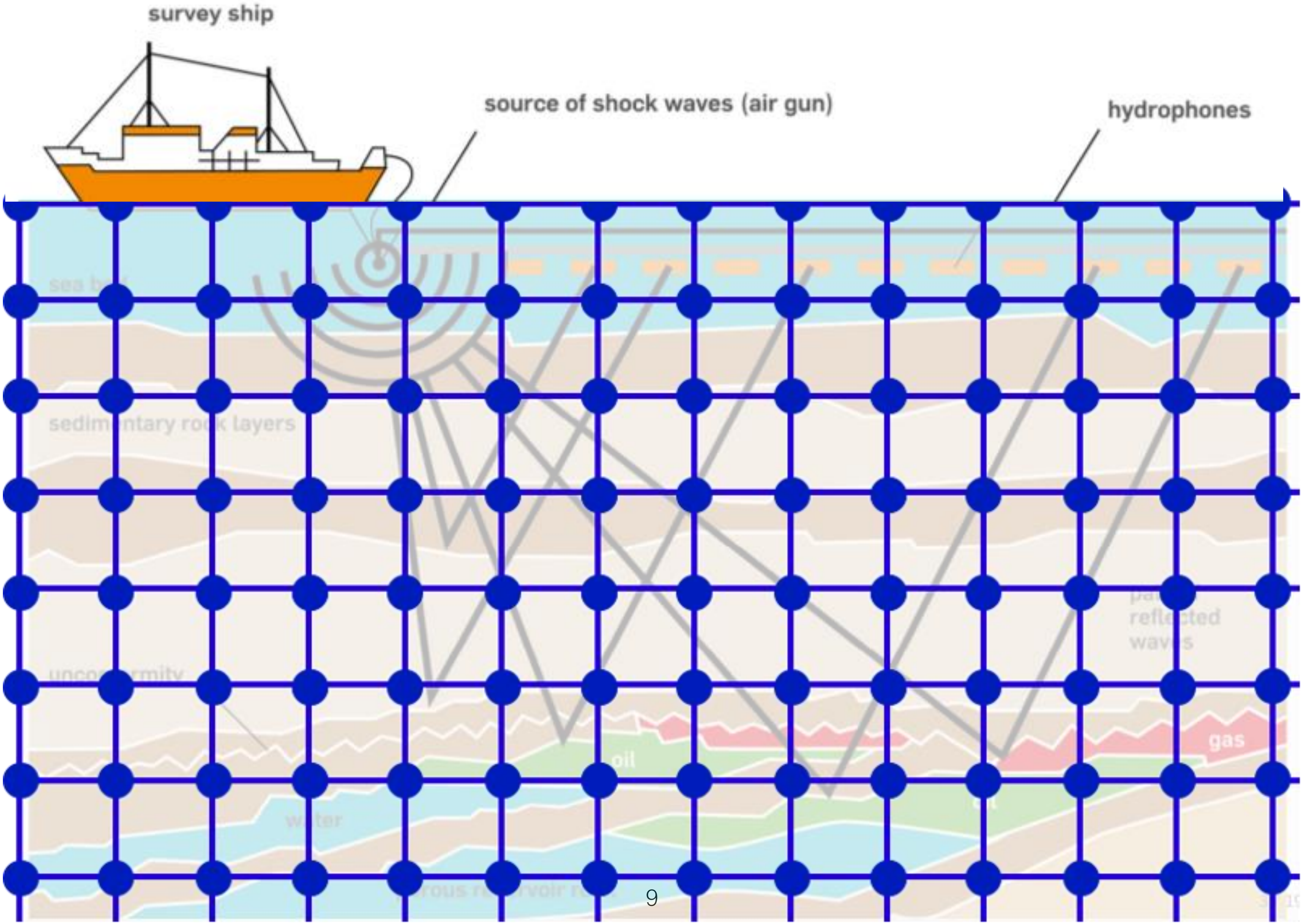
- Not only stencils in the game. What else?
- Sources injecting and receivers interpolating at sparse off-the-grid coordinates. **Non-conventional update patterns.**
- Usually their coordinates are not aligned with the computational grid. How do we iterate over them?



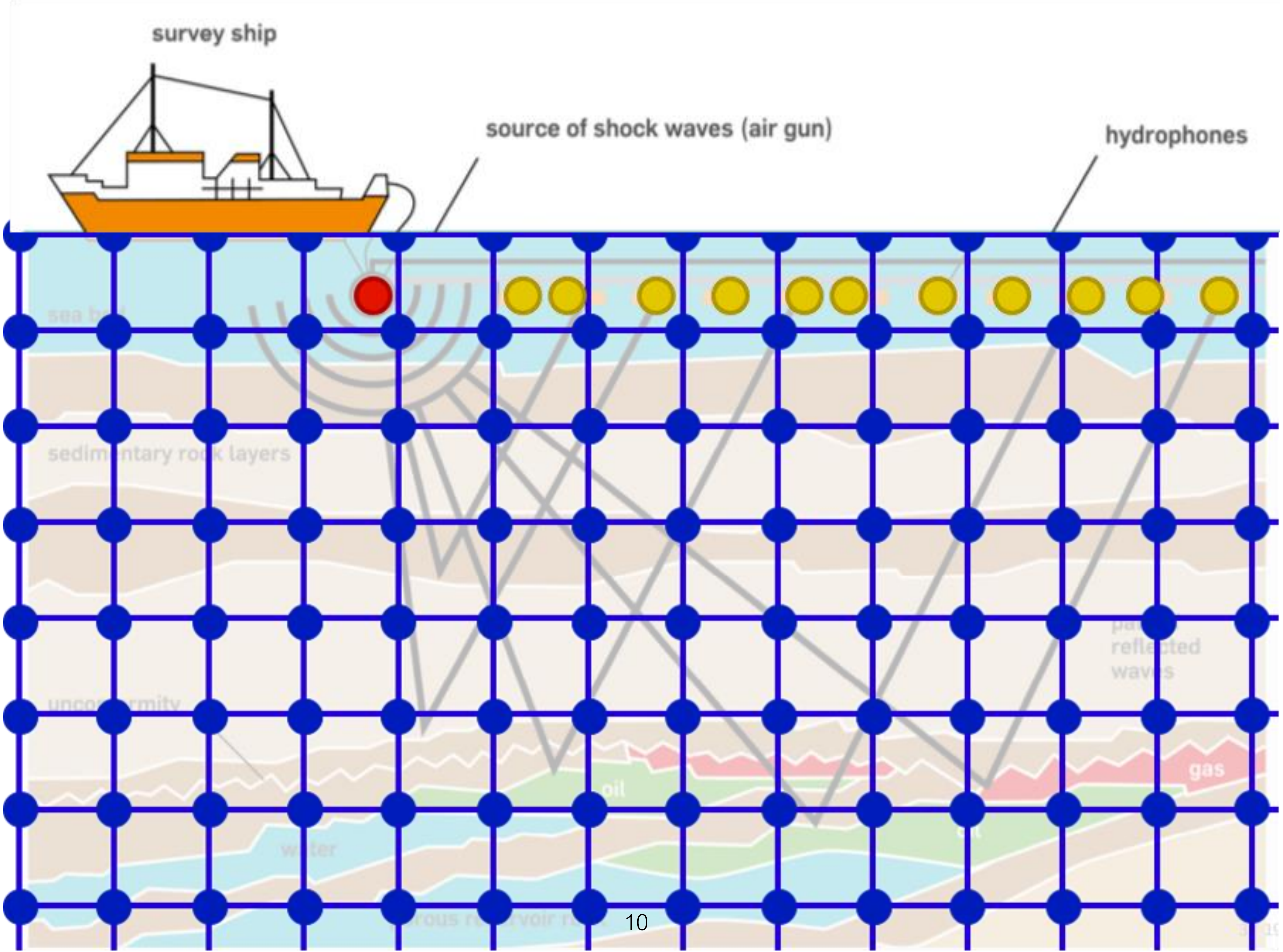


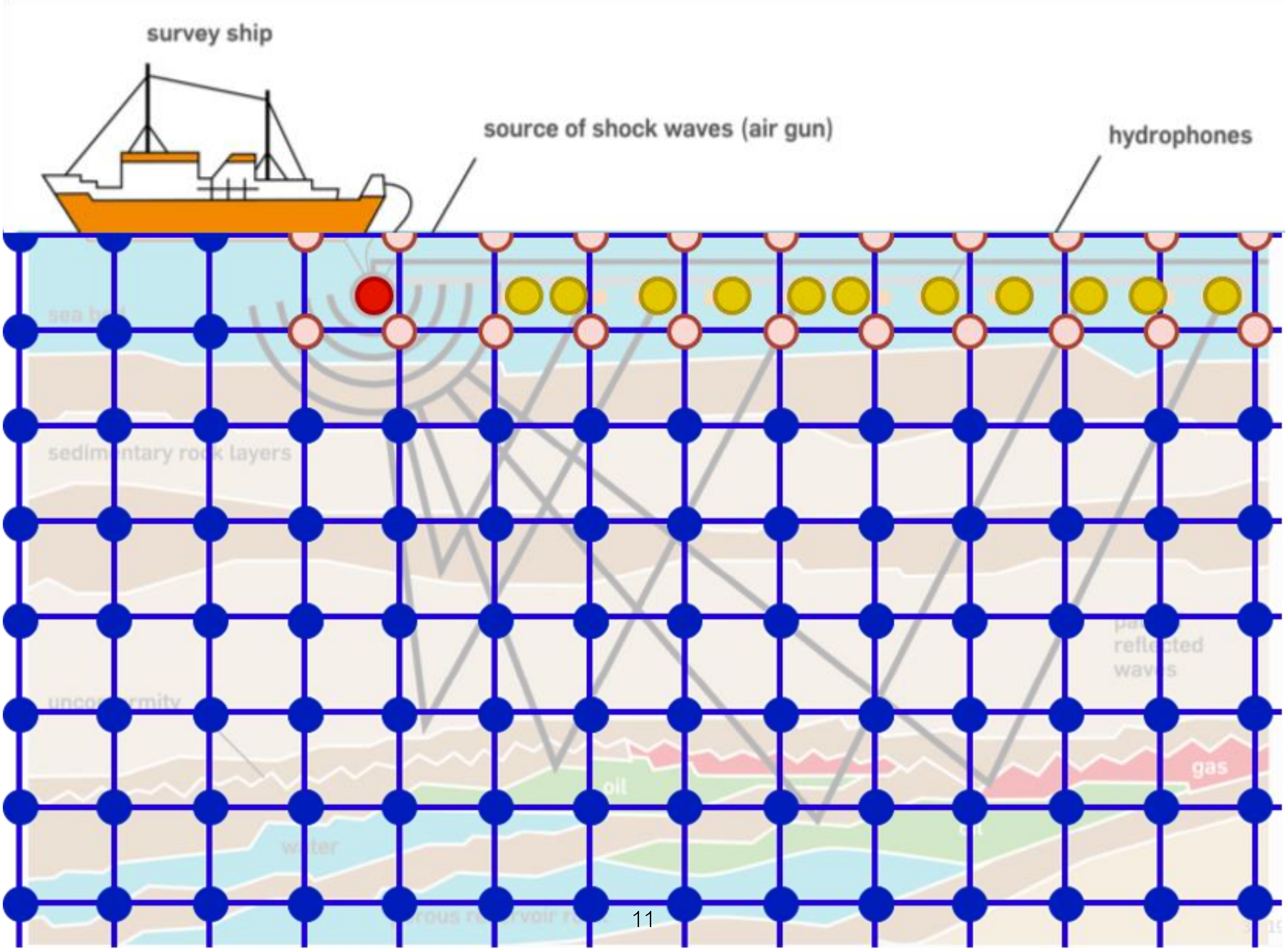



survey ship
source of shock waves (air gun)
hydrophones
sea bed
sedimentary rock layers
path of
reflected
waves
unconformity
oil
oil
water



survey ship
source of shock waves (air gun)
hydrophones
gas
oil


survey ship
source of shock waves (air gun)
hydrophones


survey ship
source of shock waves (air gun)
hydrophones
oil
gas

A typical time-stepping loop with source injection

- Iterate over sources, each has 3-d coordinates
- Indirect accesses to scatter injection to neighbouring points
- Aligned in time, not in space

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] + $\sum_{r=1}^{r=so/2} w_r$ [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 i = 1 ***to*** np **do**
      xs, ys, zs = map(s, i);
      u[t, xs, ys, zs] + = $f(src(t, s))$

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)
```

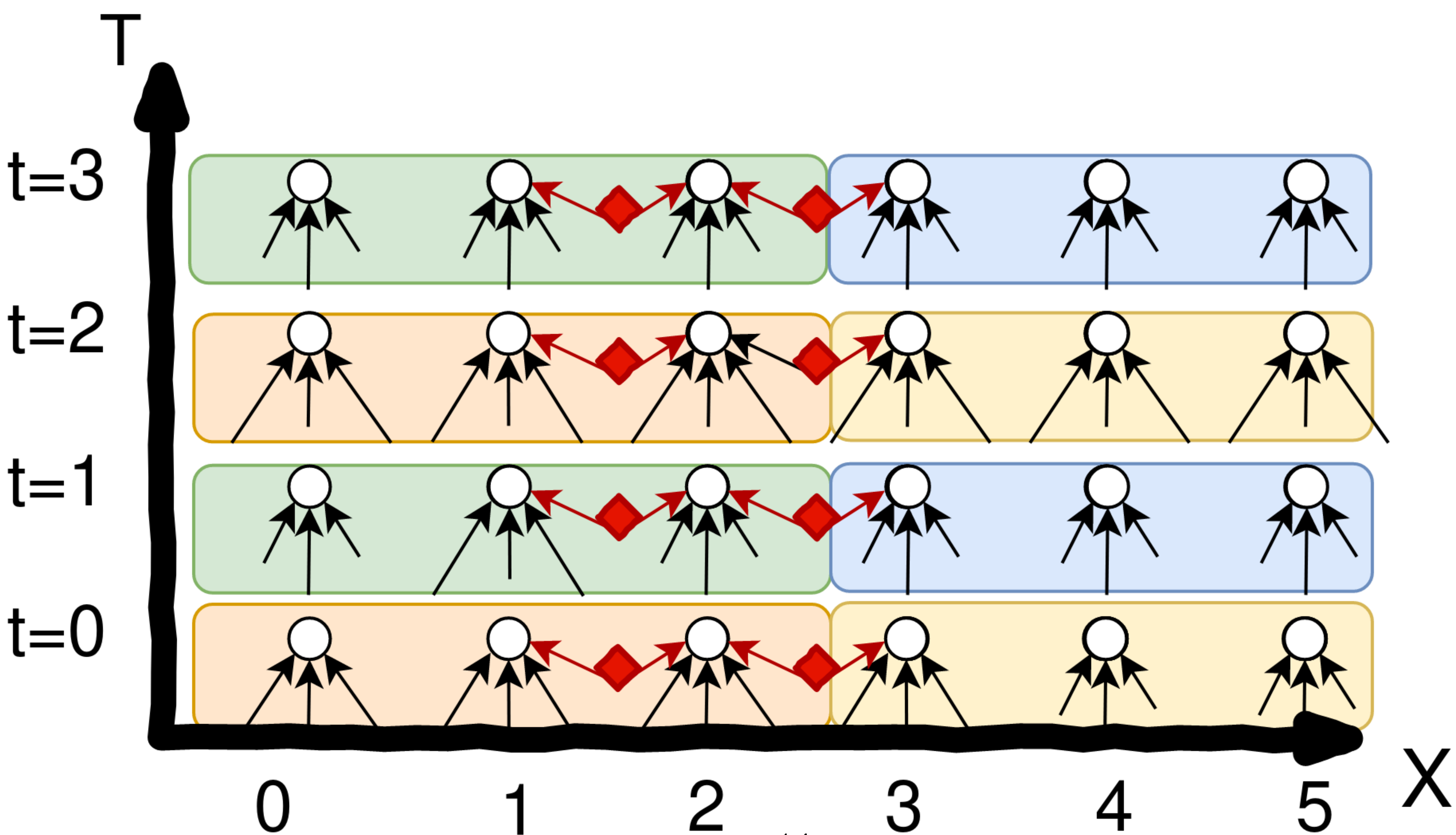
Discover affected points

Weights of impact

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

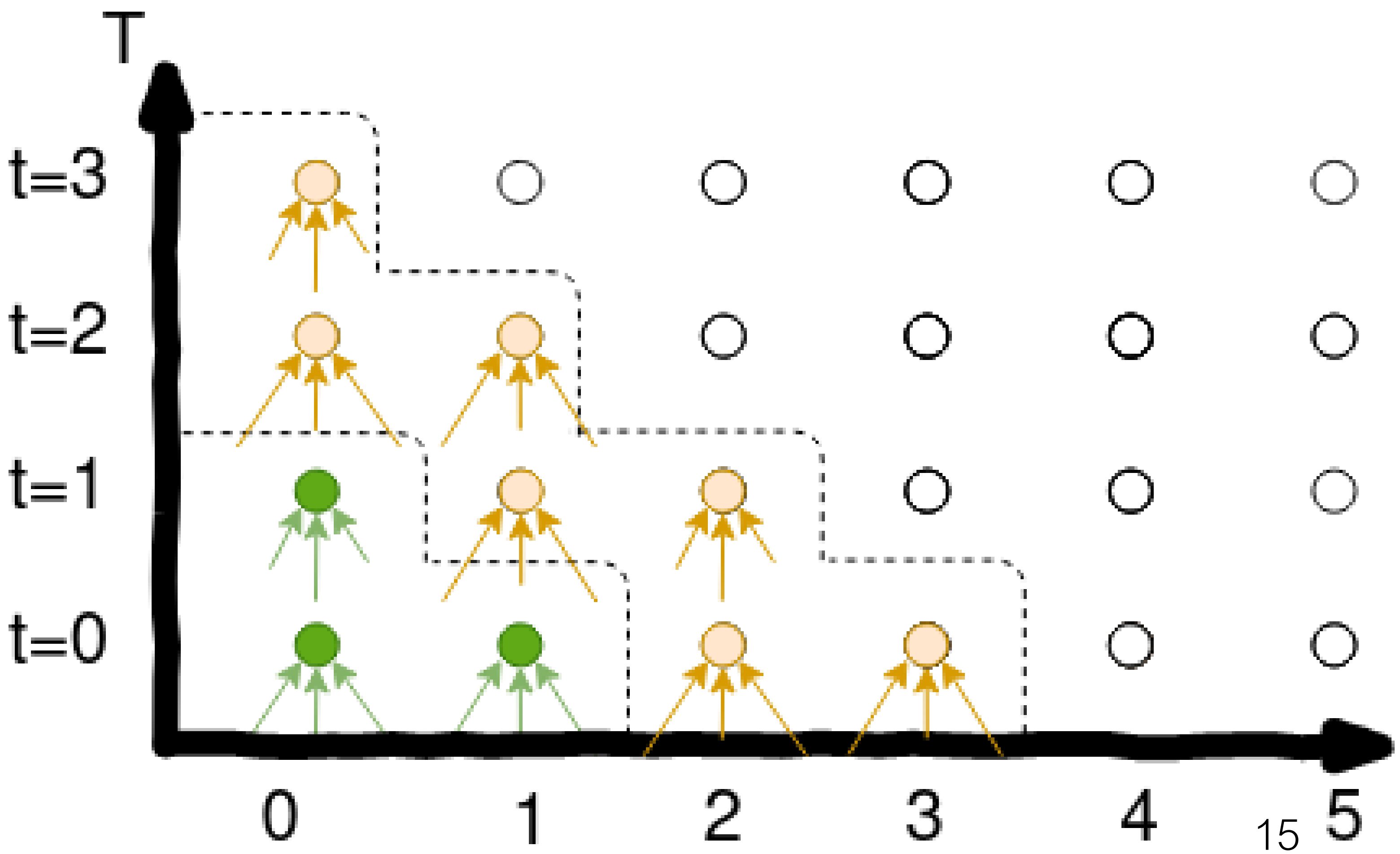
Applying space-blocking

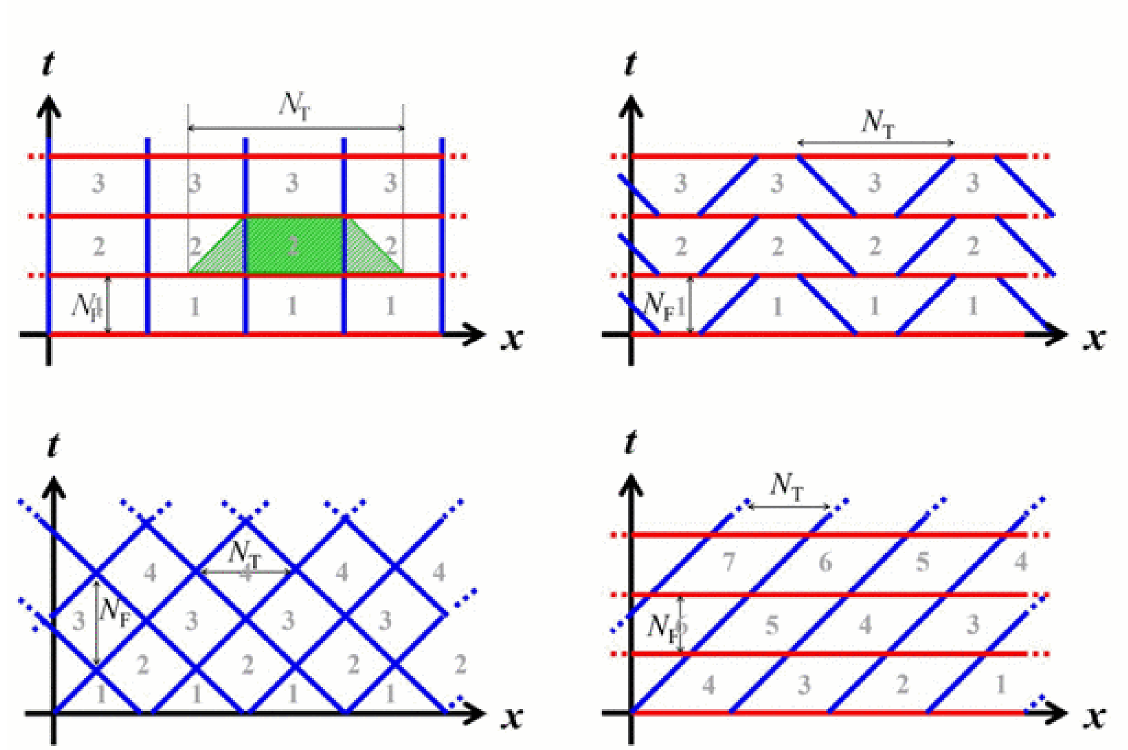
- Spatial blocking:
  - Decompose grids into block tiles/ Partitioning iteration space to smaller chunks/blocks
  - Improve data locality => Increase performance (Rich literature)
  - Sparse off-the-grid operators are iterated as without blocking



Applying temporal-blocking

- Temporal blocking:
  - Space blocking but extend reuse to time-dimension.
  - Update grid points in future where/when (space+time) possible
  - Rich literature, several variants of temporal blocking, shapes, schemes
    - Wave-front / Skewing (Our POC approach)
    - Diamonds, Trapezoids, Overlapped, Hybrid models

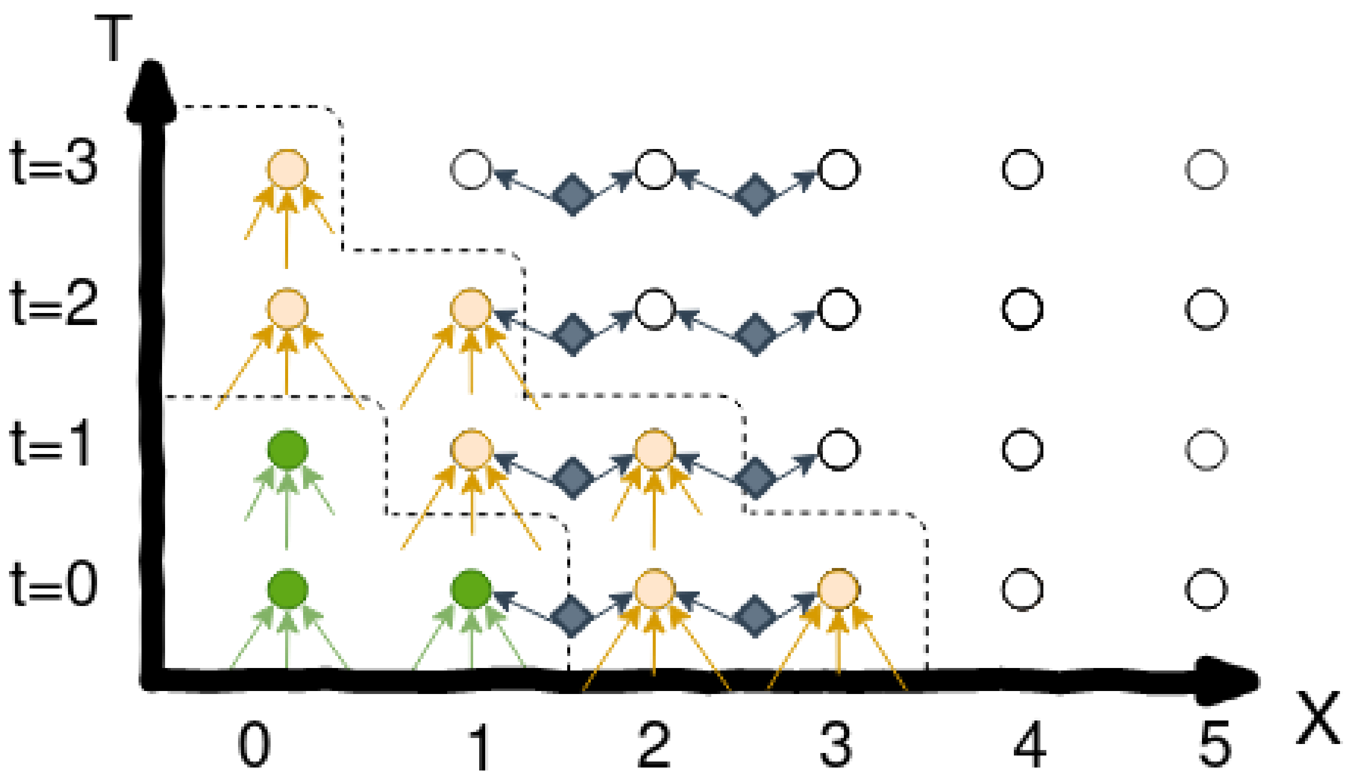


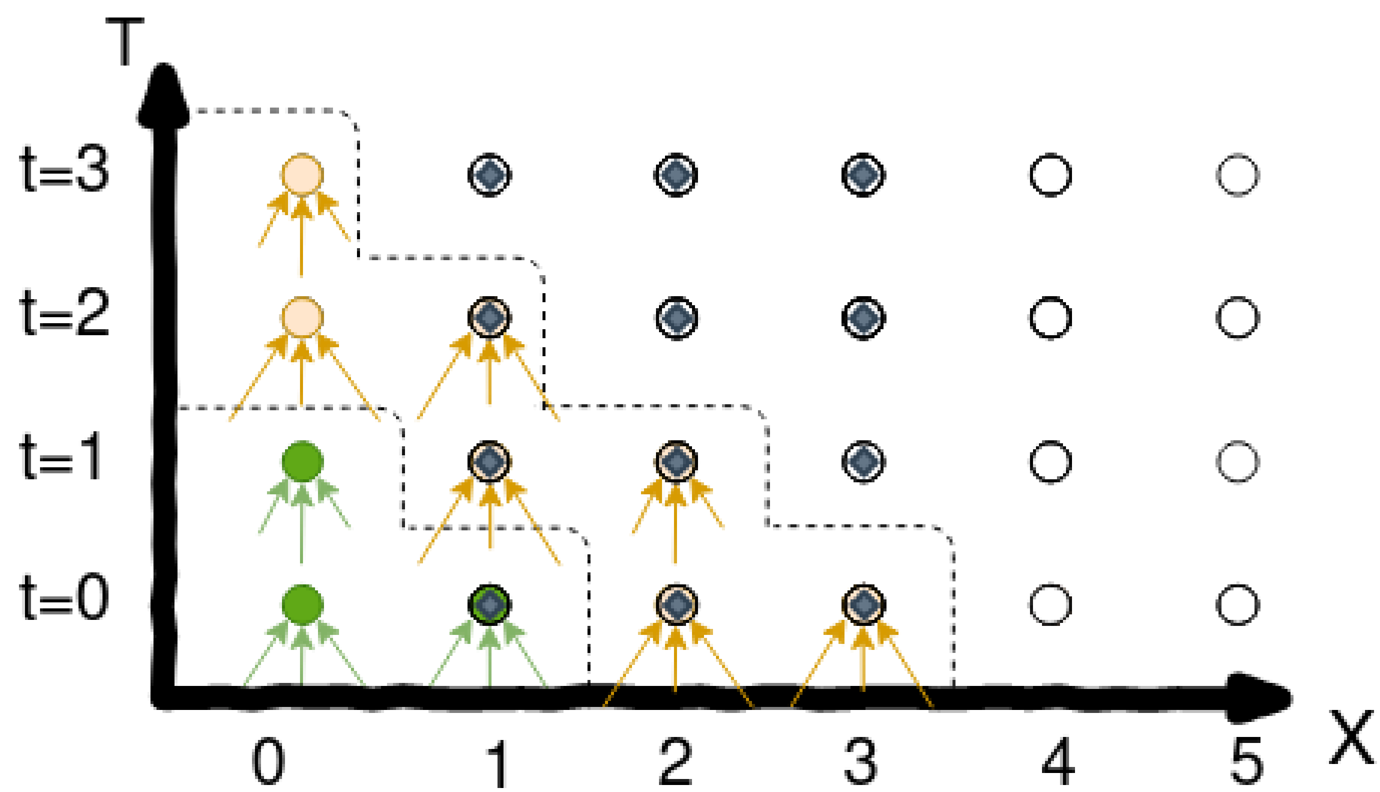


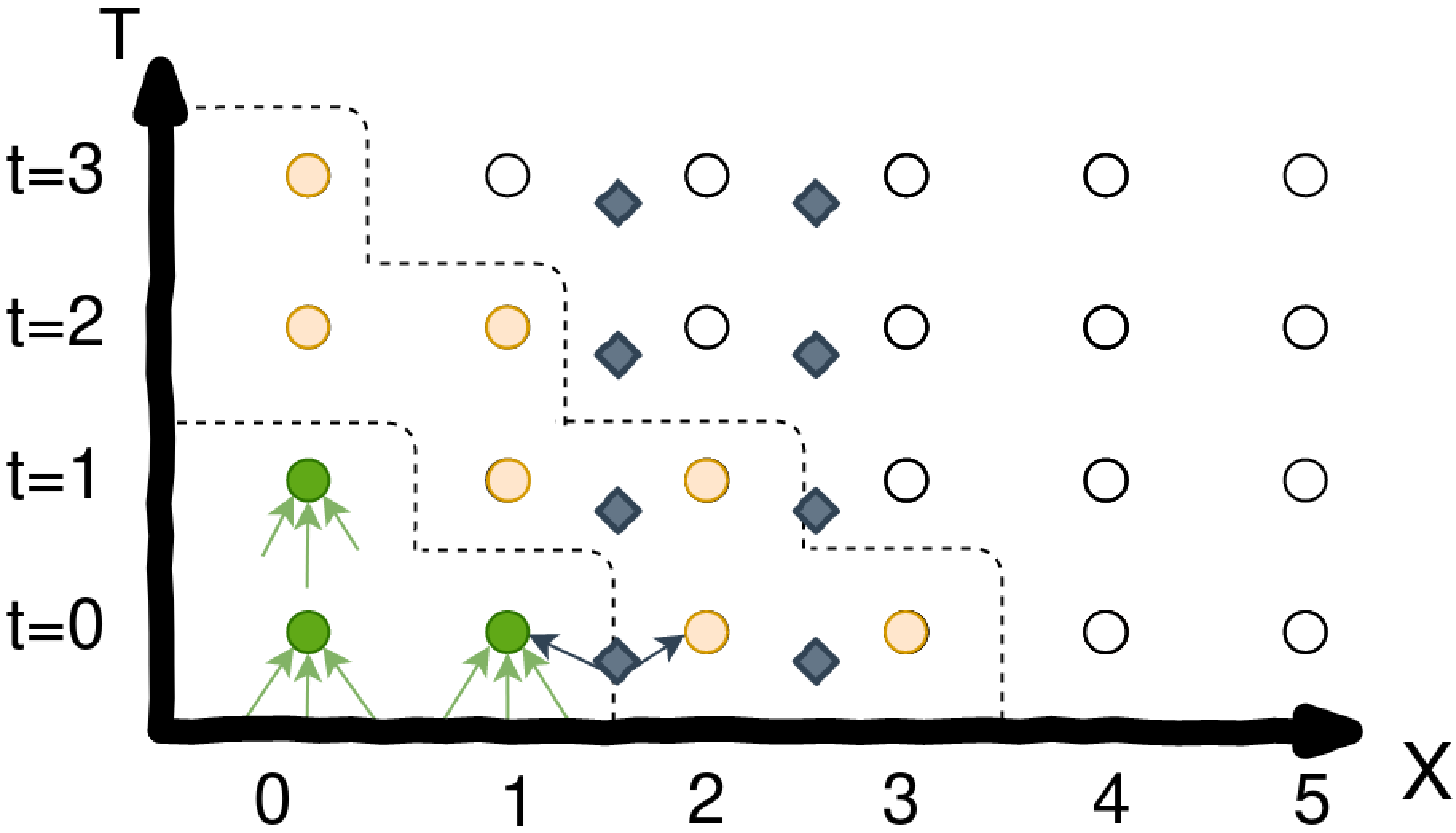
Tanaka et.al. (2018)

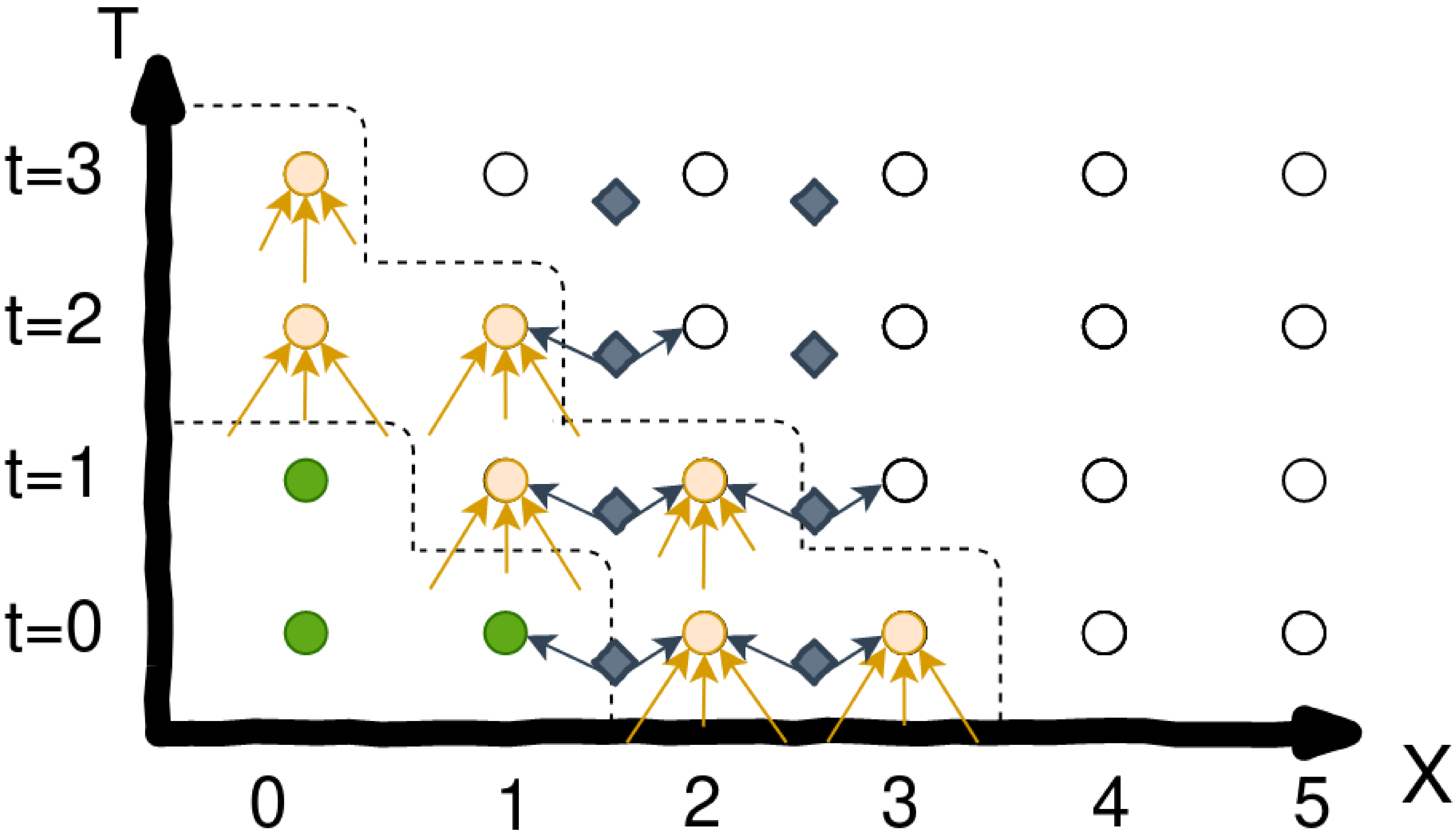
Off-the-grid operators: the issue

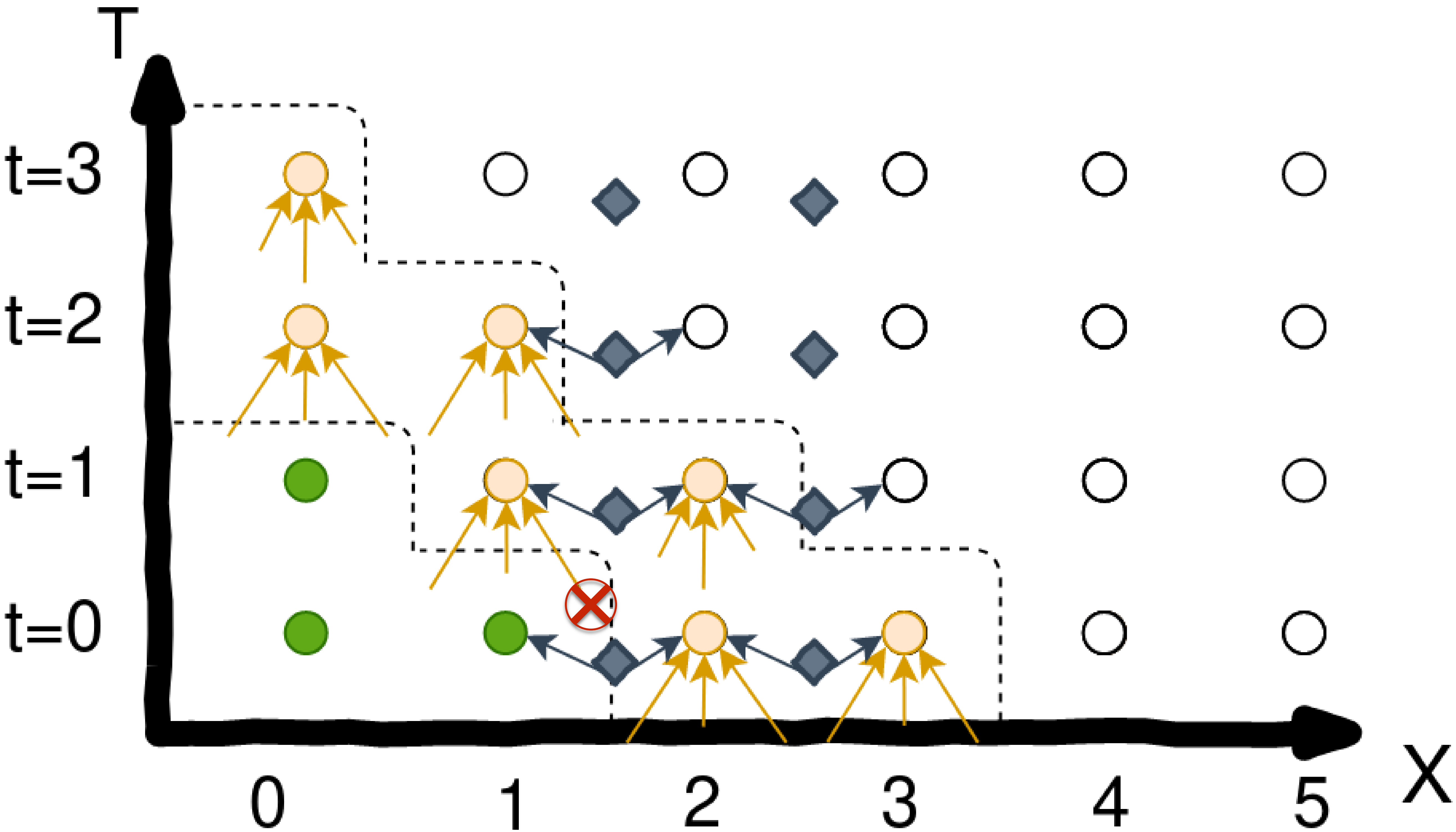
- Data dependences violations happen while a temporal update
- When a sparse operator exists in the boundary between space-time blocks, order of updates is not preserved
- Solution: Need to align off-the-grid operators

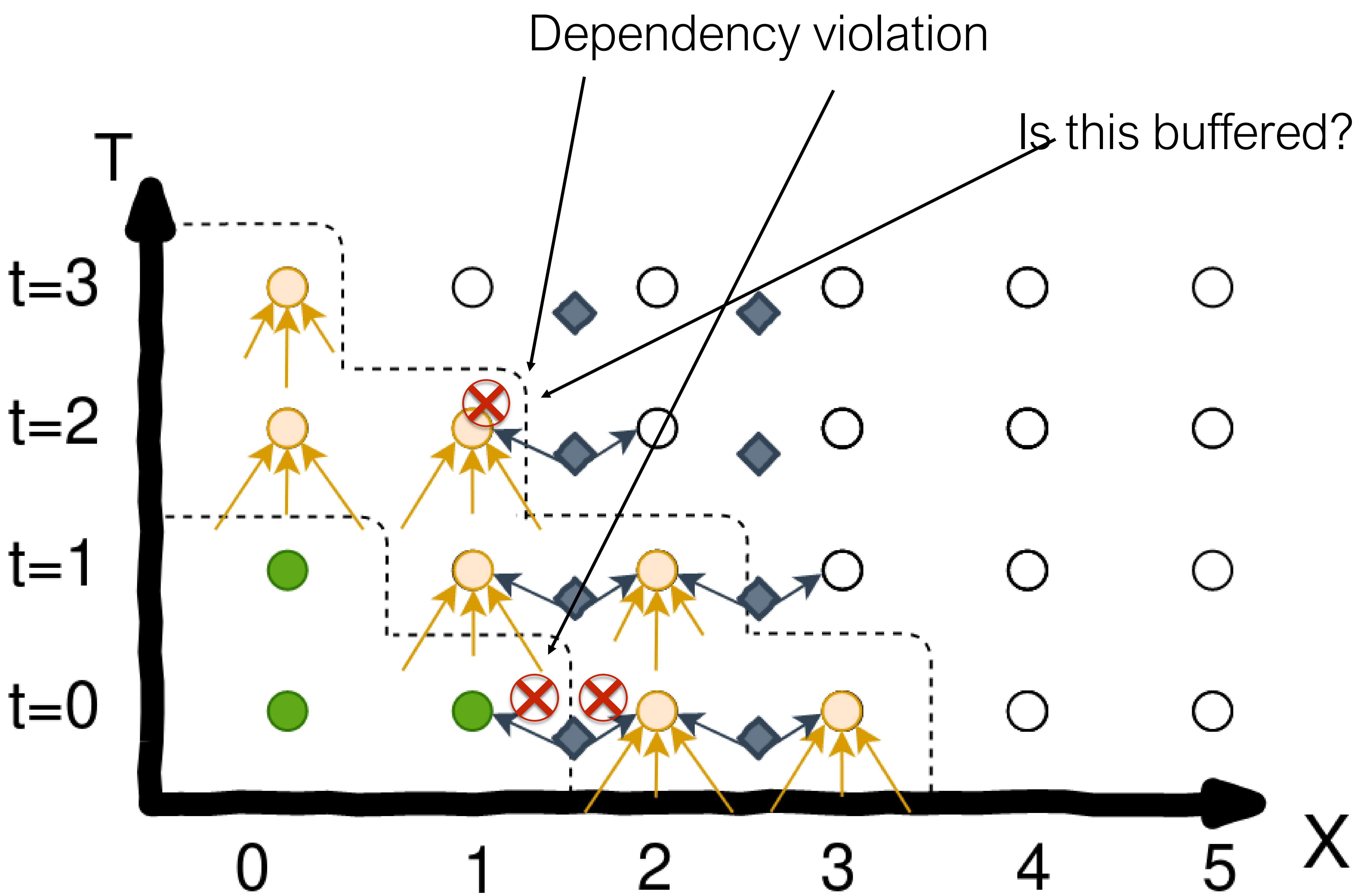


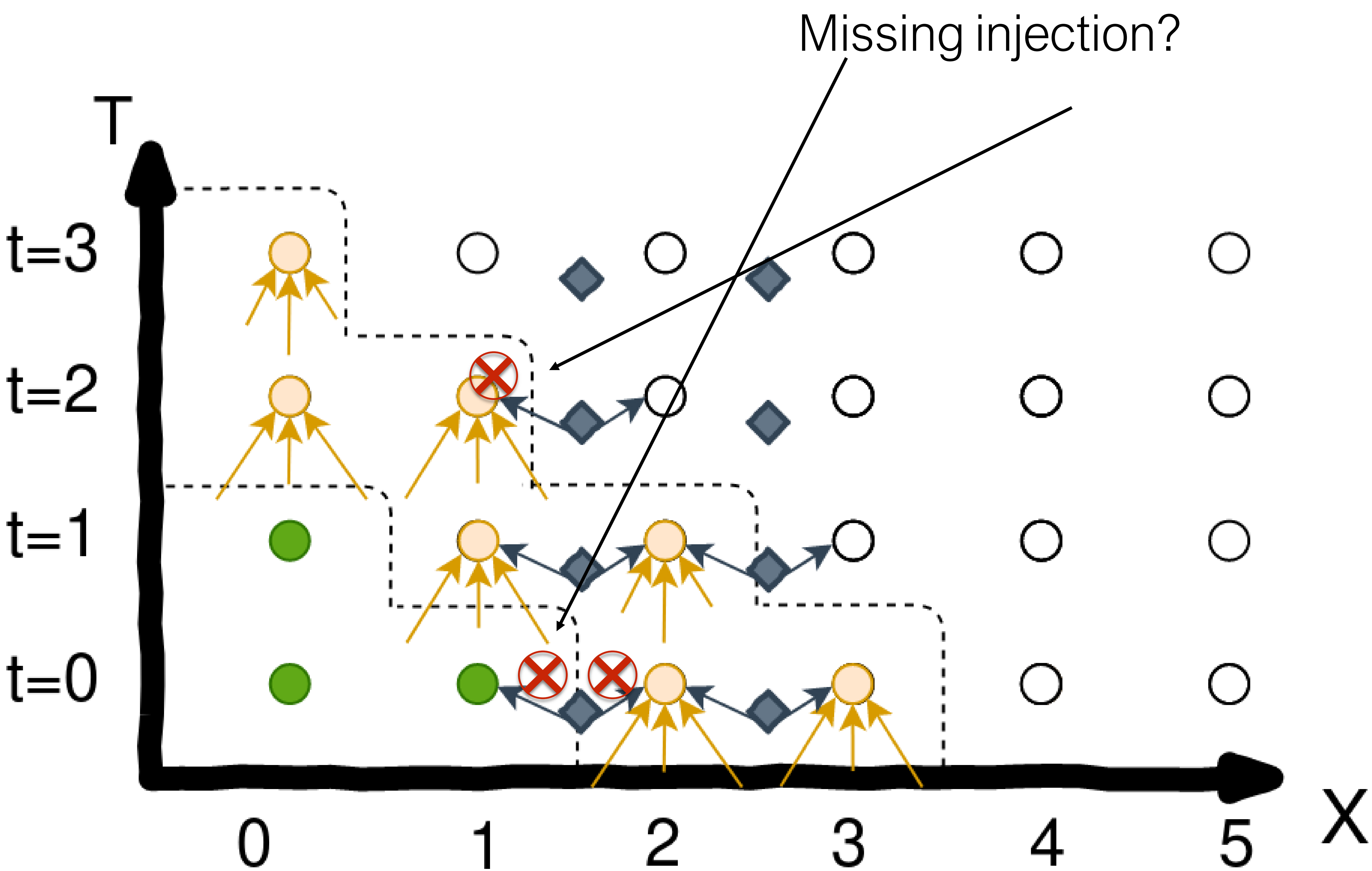


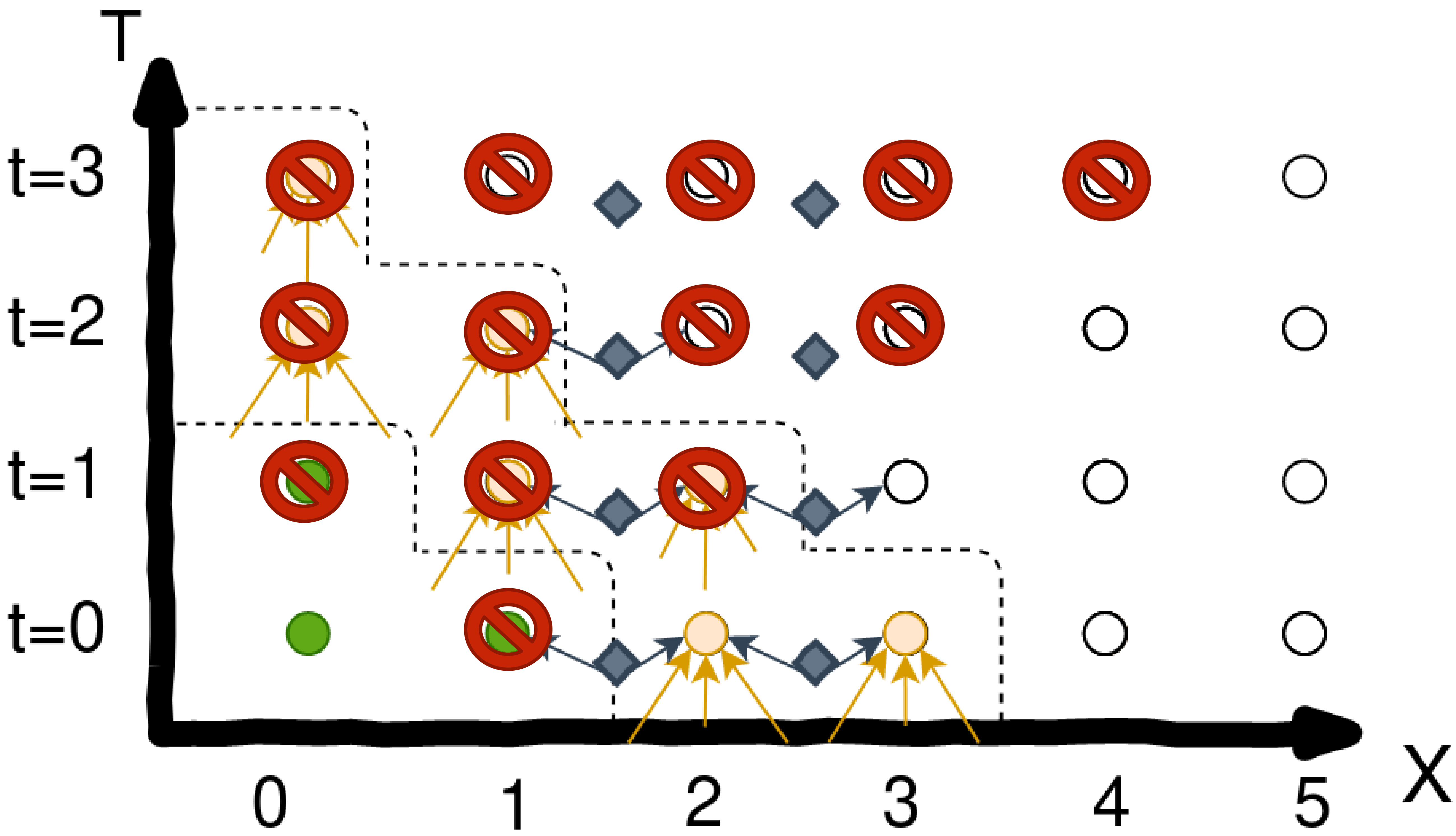

T
t=3
t=2
t=1
t=0
0
1
2
3
4
5
X


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


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

Methodology

- A scheme to precompute the source injection contribution.
- Align to the grid source injection data dependences
- Negligible cost
- All built using Devito's DS Language
- Applicable to other fields as well

Iterate over sources and store indices of affected points

- Inject to a zero-initialized grid for one (or a few more)

- Hypothesis: non-zero values at the first time-steps

- Automatically generate code with Devito. Independent of the injection and interpolation type (e.g. non-linear injection)

Algorithm 2: Source injection is taking place 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

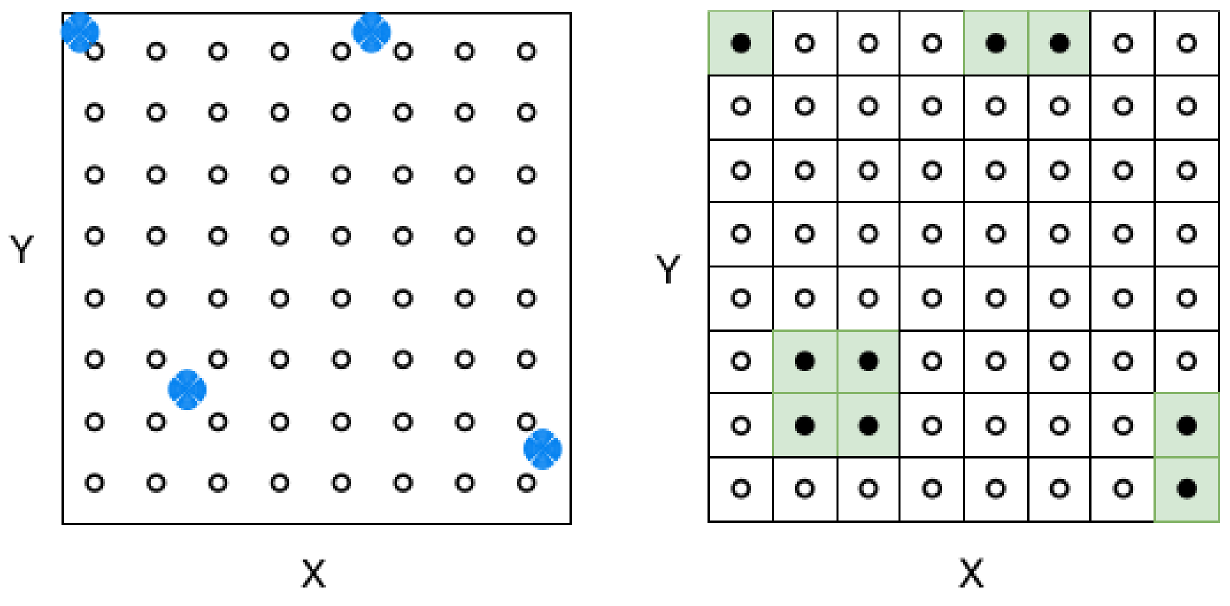
Generate sparse binary mask, unique IDs and decompose wavefields

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

	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)

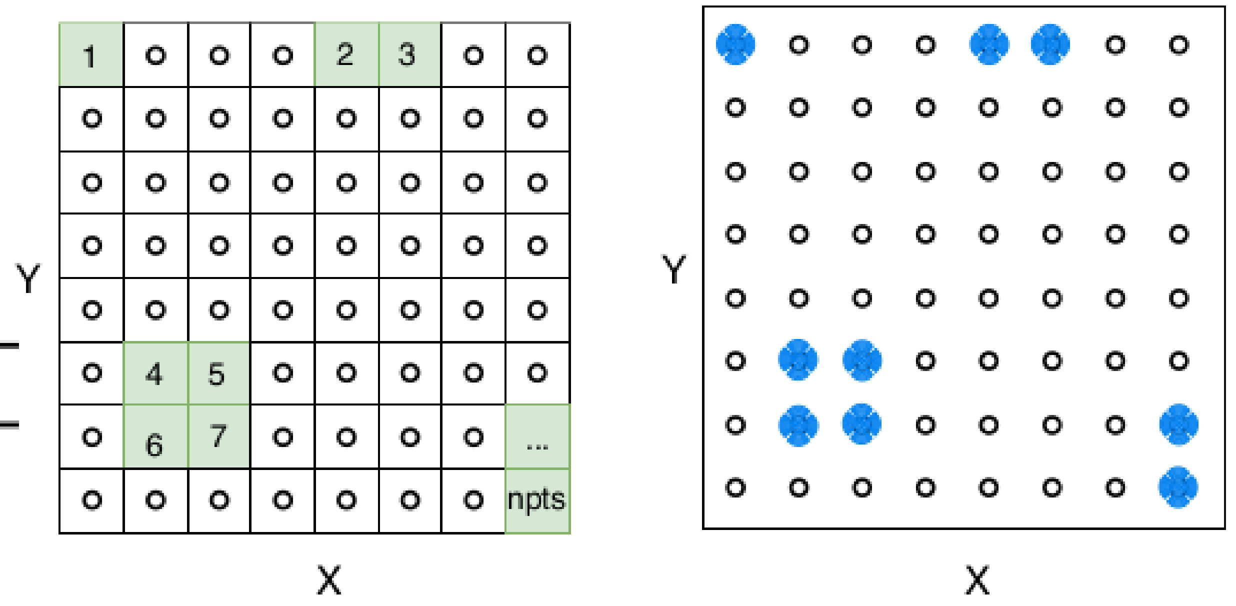
Algorithm 3: Decomposing the source injection wavefields.

```
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.

Algorithm 5: 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 z = 1 to nz do
        u[t, x, y, z] + = SM[x, y, z] * src_dcmp[t, SID[x, y, z]];
```

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.

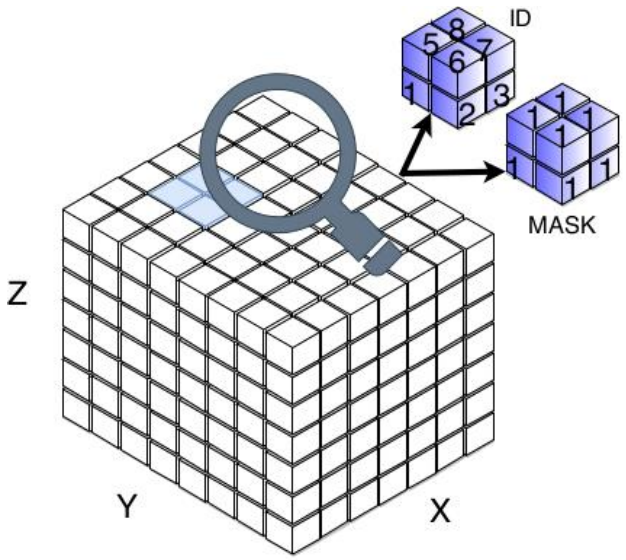
Algorithm 5: Stencil kernel update with fused source injection.

```
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 z = 1 to nz do
                u[t, x, y, z] + = SM[x, y, z] * src_dcmp[t, SID[x, y, z]];
```

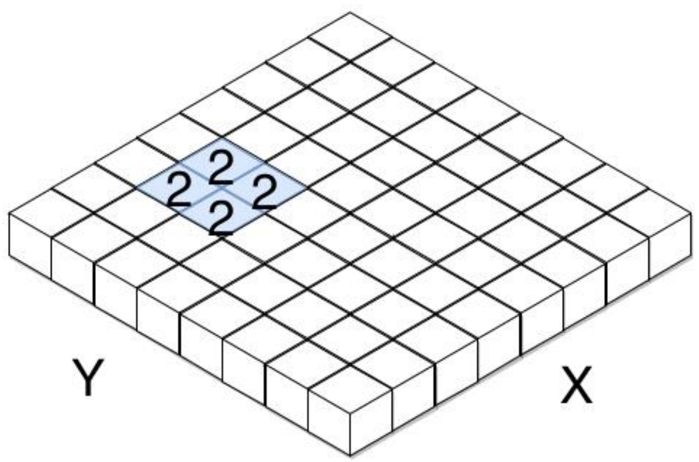
SIMD? (AVX512)

Reducing the iteration space size

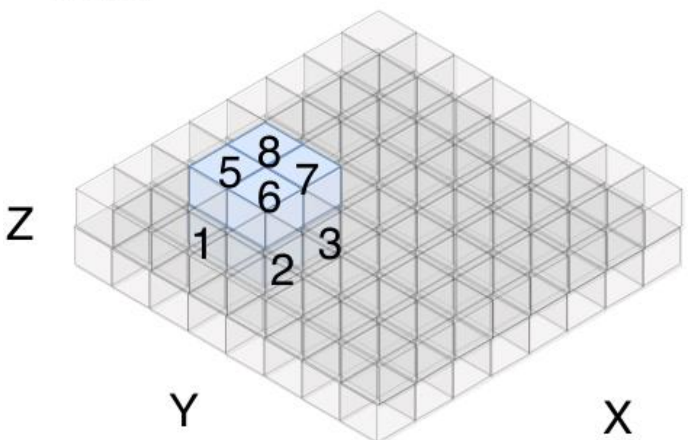
- Perform only necessary operations
- Aggregate NZ along the z- axis keeping count of them in a structure named *nnz_mask*.
- Reduce the size of SM and SID by cutting off zero z-slices



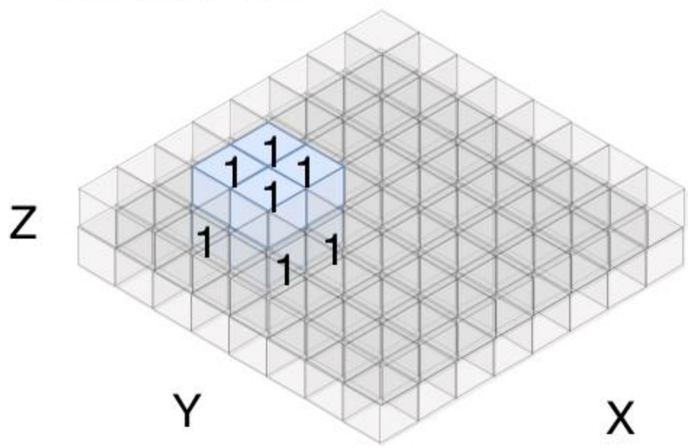
(a) 3D masks and arrays are very sparse in the general case.



(b) Aggregating non-zero elements along z-axis in nnz_mask.



(c) Sp_SID, a reduced size SID



(d) Sp_SM, a reduced size SM

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]];
```

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) ≡ 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 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

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] + $\sum_{r=1}^{r=so/2} w_r$ [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 i = 1 ***to*** np **do**
      xs, ys, zs = map(s, i);
      u[t, xs, ys, zs] + = $f(src(t, s))$

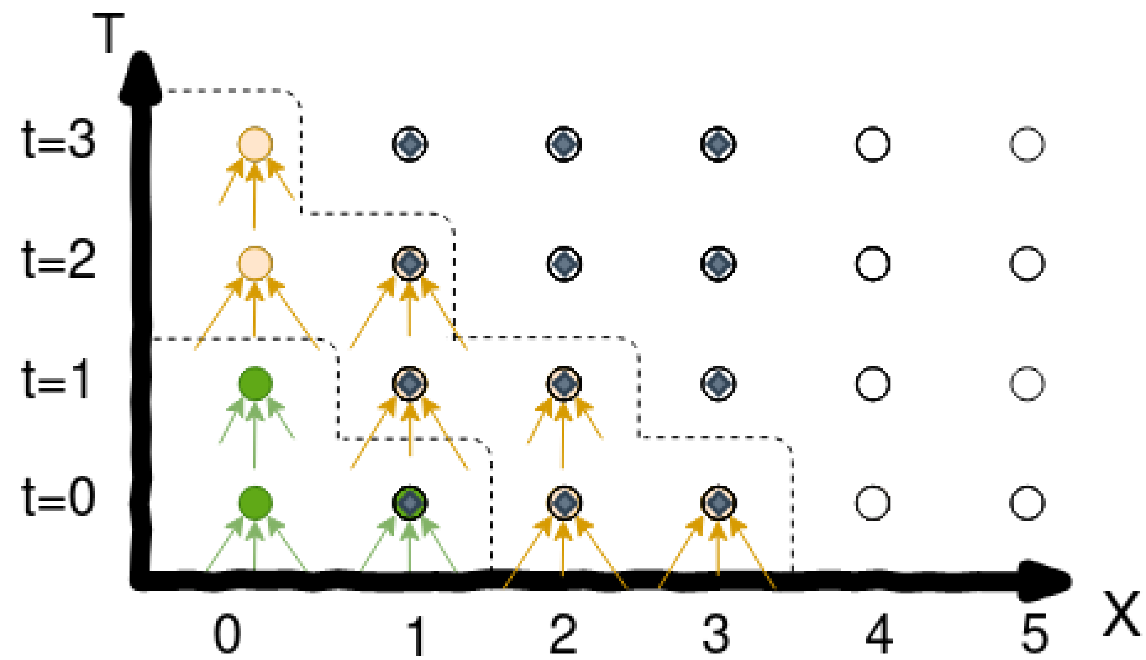
Non-aligned

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

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

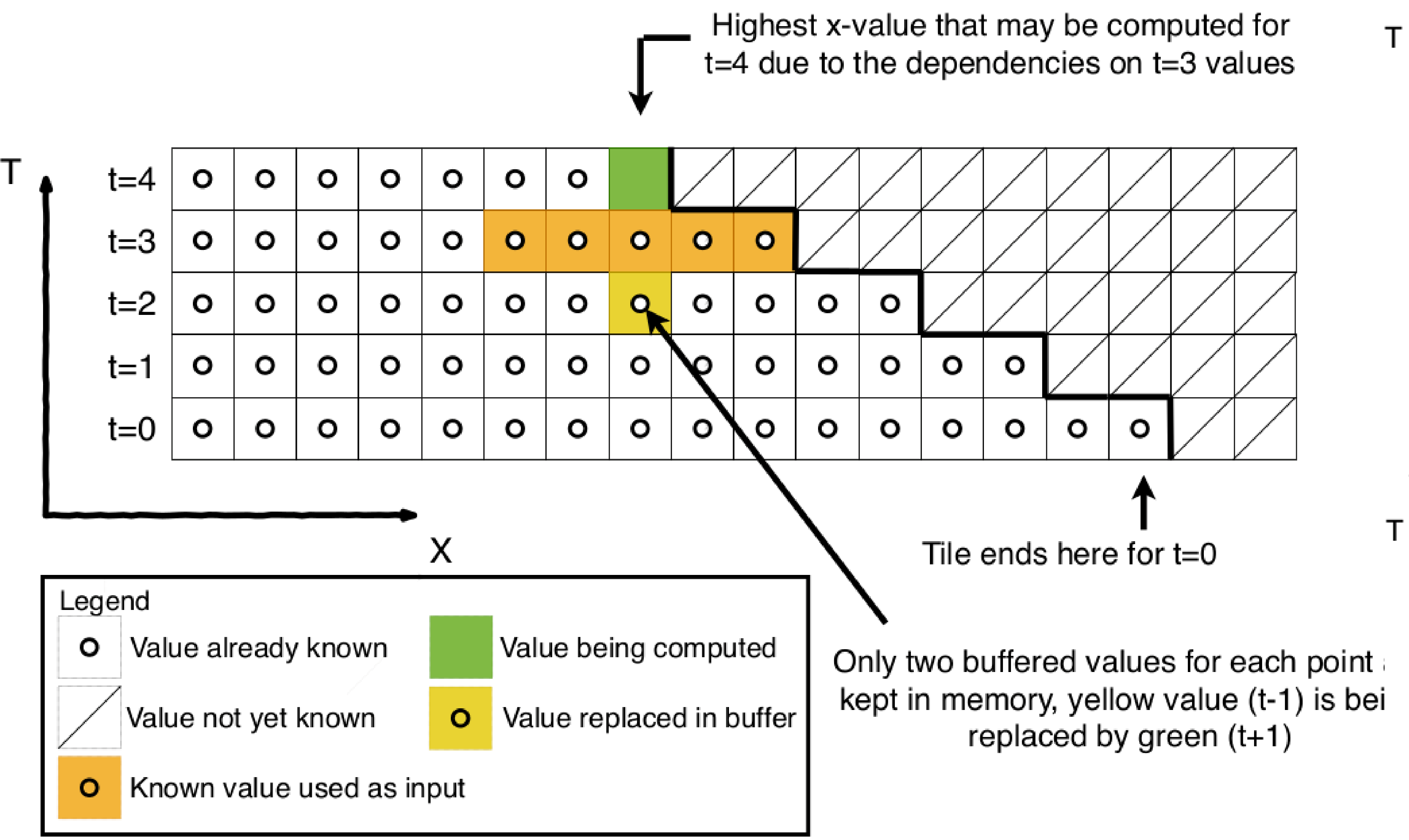
for t = 1 ***to*** nt **do**
  for x = 1 ***to*** nx **do**
    for y = 1 ***to*** ny **do**
      for z = 1 ***to*** nz **do**
        $A(t, x, y, z, s)$;
      for $z2$ = 1 ***to*** nnz_mask[x][y] **do**
        zind = Sp_SM[x, y, z];
        u[t, x, y, z2] +=
        SM[x, y, zind] * src_dcmp[t, SID[x,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

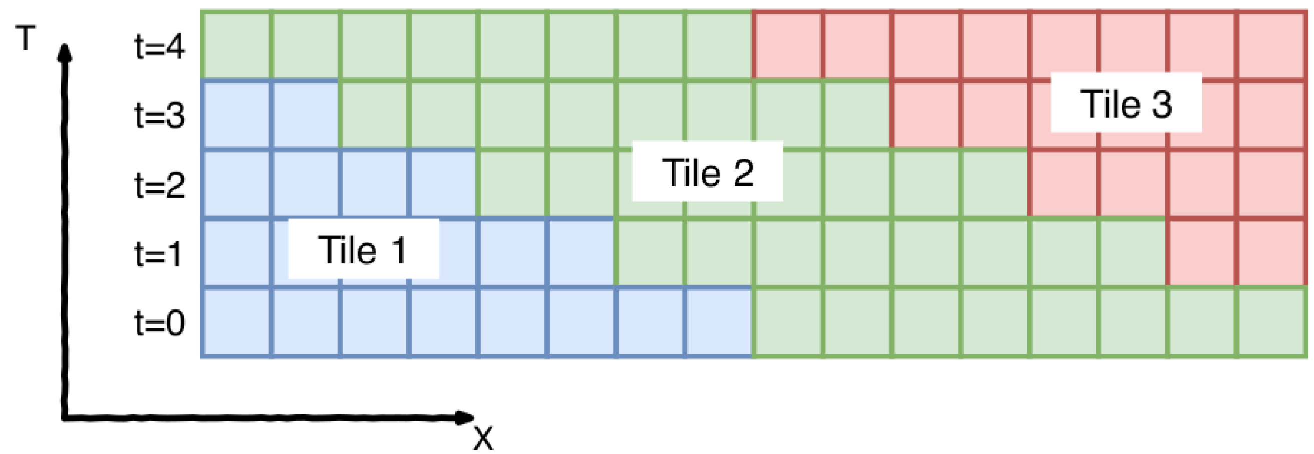
Aligned



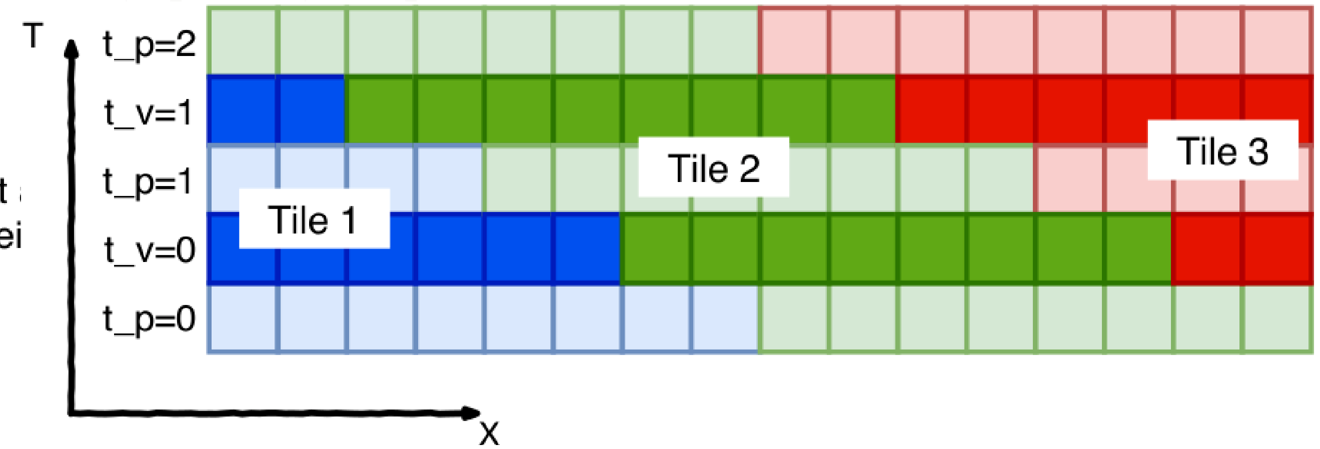
Applying wave-front temporal blocking

- Aligning, automated in DSL; TB with manual loop transformation
- All sources aligned to the grid now. Coordinates aligned with points
- Skewing factor depends on data dependency distances





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



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

Algorithm 7: 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 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);
```

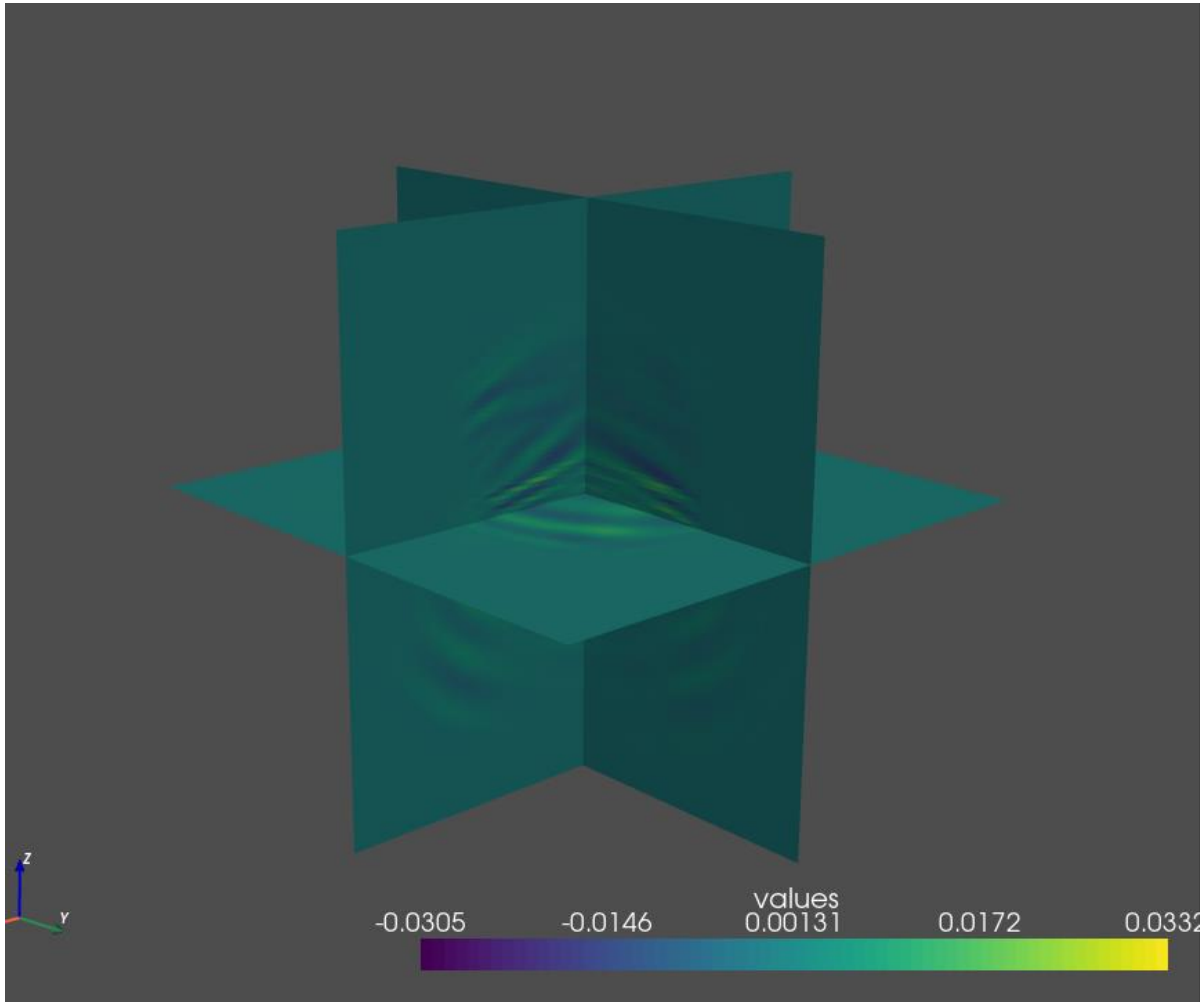
Experimental evaluation: the models

- **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 dependences

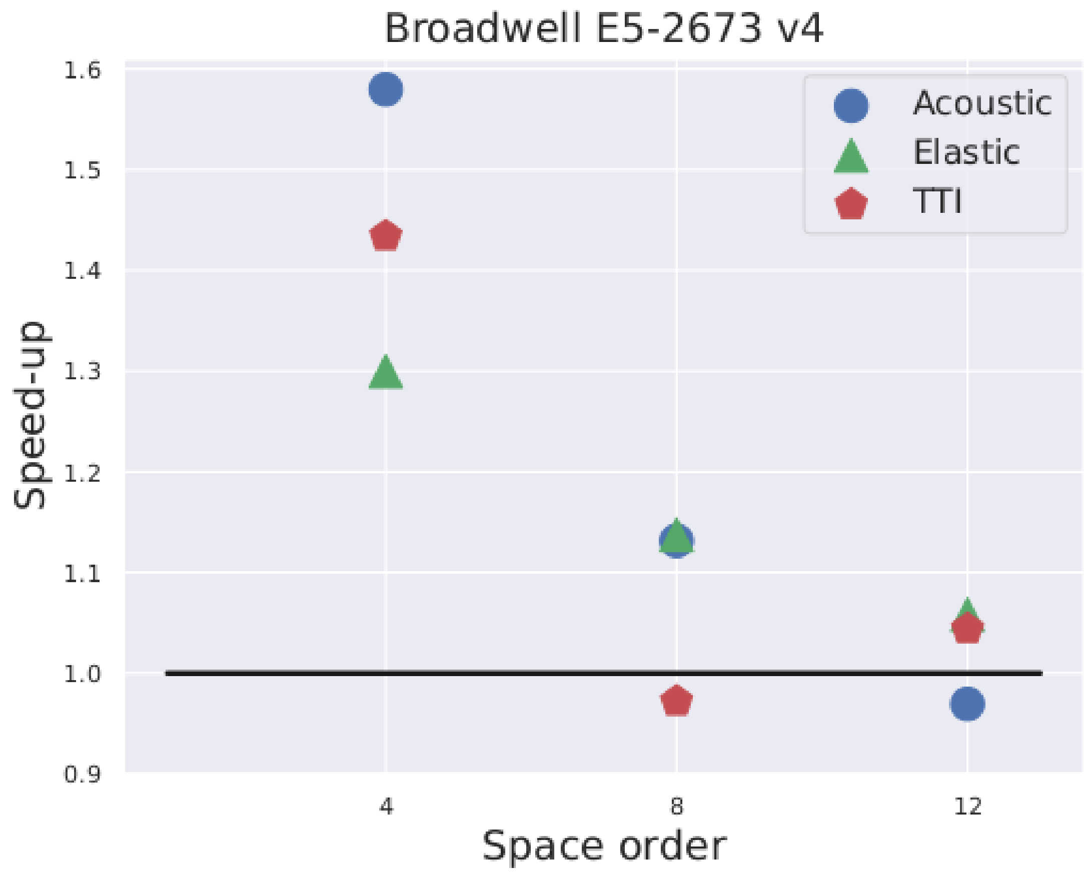
- **Anisotropic Acoustic**
  Industrial applications, rotated laplacian, coupled system of two scalar PDEs

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

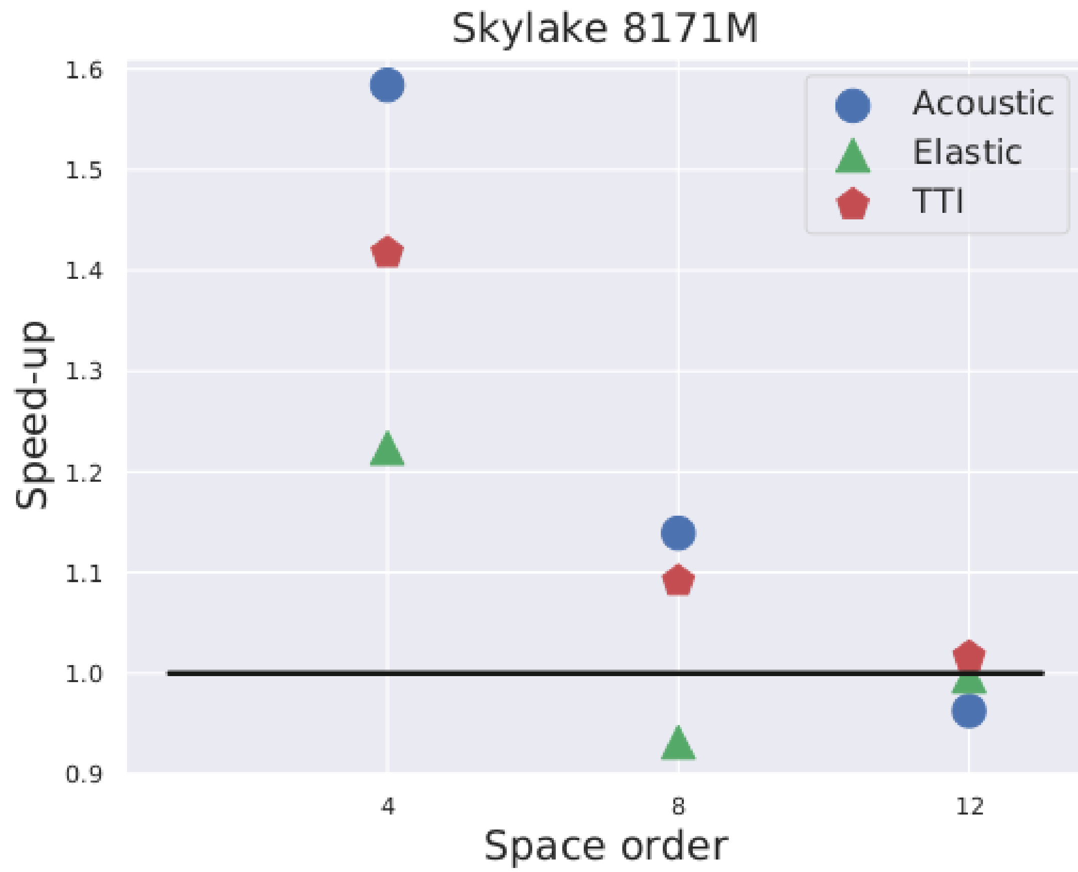


Velocity field, TTI wave propagation after 512ms

Experimental evaluation: the results



(a) Speed-up of kernels for Broadwell.



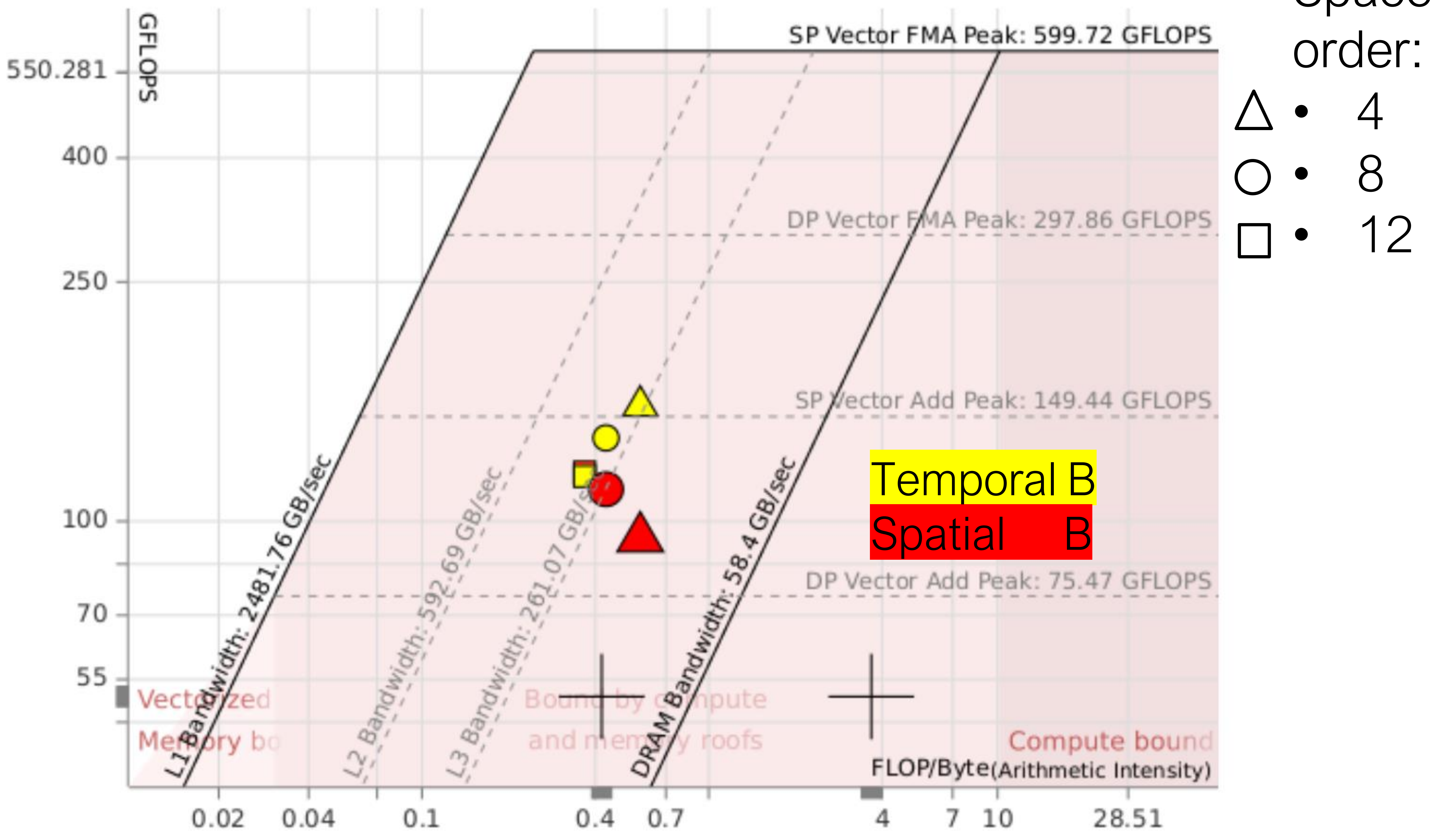
(b) Speed-up of kernels for Skylake.

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

Cache aware roofline model



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

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))
```

Conclusions

- We presented an approach to apply temporal blocking on stencil kernels with sparse off-the-grid operators.
- The additional cost is negligible compared to the achieved gains.
- Solution built on top of Devito-DSL
- Performance gains of up to 1.6x on low order (4) and 1.2x on medium order (8).

Work presented is inherited from: Bisbas, G., Luporini, F., Louboutin, M., Nelson, R., Gorman, G., & Kelly, P.H. (2020). Temporal blocking of finite-difference stencil operators with sparse "off-the-grid" sources. Available online: https://arxiv.org/abs/2010.10248

Future plans →

- Integration/ Automation
- GPUs
- High-order stencils

- Open source, on top of Devito v4.2.3 - https://github.com/georgebisbas/devito

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

Imperial College London

EPSRC Engineering and Physical Sciences Research Council

HiPEDS

Acknowledgements

Thanks to collaborators and contributors:

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

Thank you for your attention! Questions?

Imperial College London

EPSRC
Engineering and Physical Sciences
Research Council

HiPEDS

References

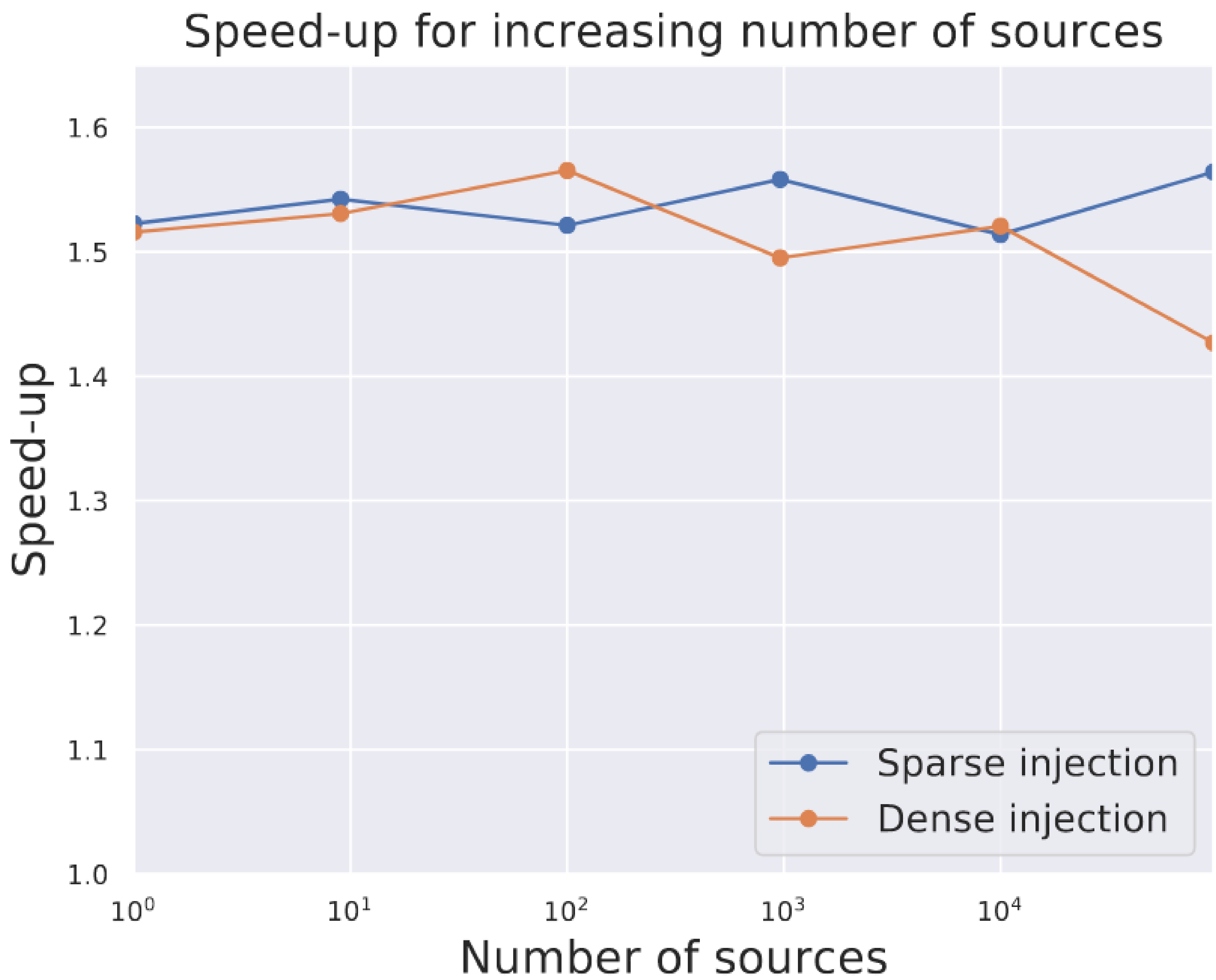
- Bisbas, G., Luporini, F., Louboutin, M., Nelson, R., Gorman, G., & Kelly, P.H. (2020). Temporal blocking of finite-difference stencil operators with sparse "off-the-grid" sources.

- 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.

Corner cases, increasing number of sources



The generated C code - stencil update

```c
    #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

```
/* 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];
```

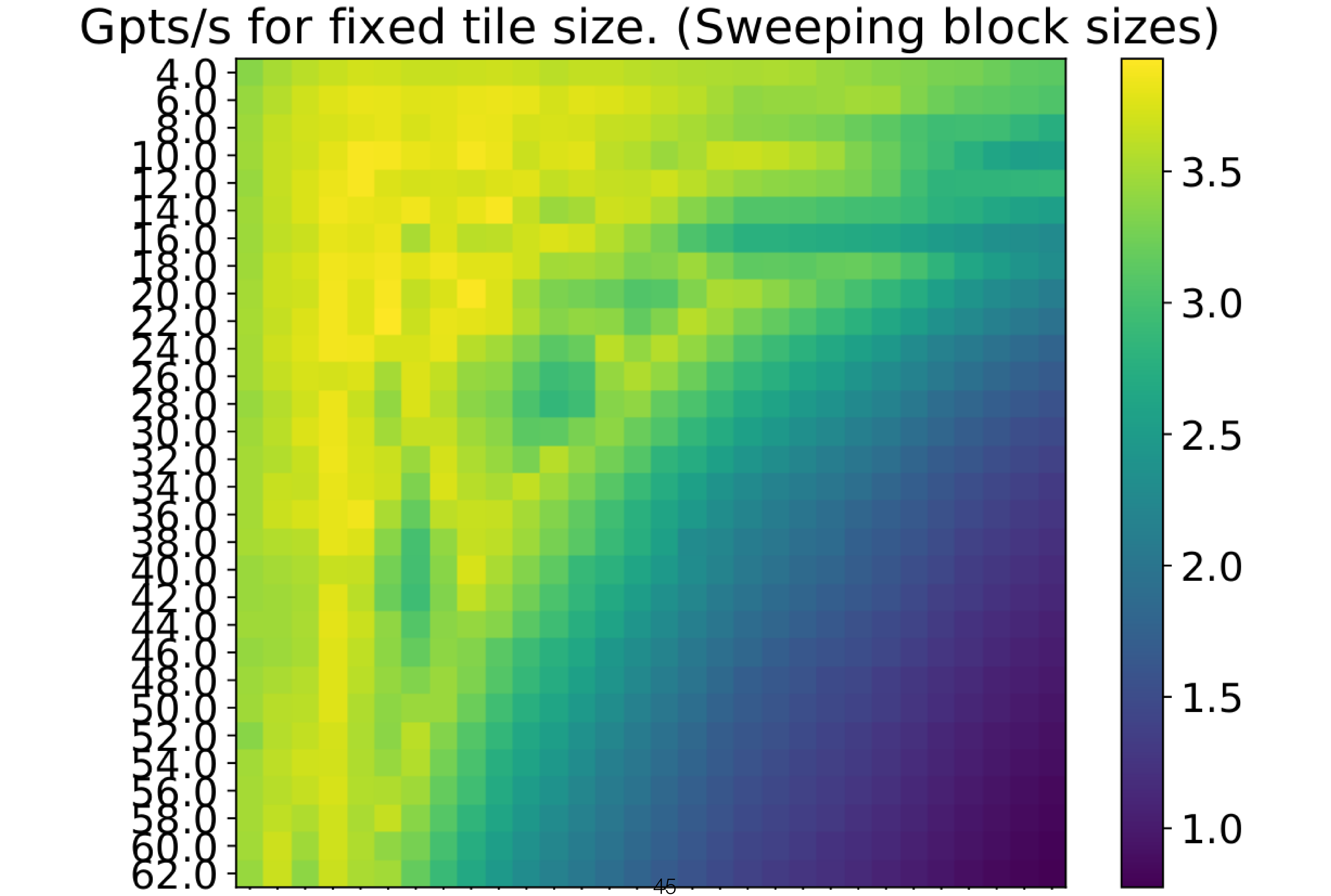
Floor, ceil of off-the-grid

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)
```

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.