TEMPORAL BLOCKING FOR WAVE-PROPAGATION KERNELS WITH SPARSE OFF-THE-GRID SOURCES

George Bisbas\textsuperscript{1}, Fabio Luporini\textsuperscript{2}, Mathias Louboutin\textsuperscript{3}, Rhodri Nelson\textsuperscript{1}, Gerard Gorman\textsuperscript{1}, Paul Kelly\textsuperscript{1}

\textsuperscript{1}Imperial College London
\textsuperscript{2}Devito Codes
\textsuperscript{3}Georgia Tech

Oil and Gas
High-Performance Computing Conference 2021
Rice University, Ken Kennedy Institute, Houston
Motivation

- Accelerating wave-propagation kernels of practical interest through cache optimizations, more specifically through temporal blocking

- Enabling **Temporal Blocking** on practical wave-propagation simulations is complicated

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

- Applicability issues due to seismic imaging kernels consisting of sparse off-the-grid operators

- We present an approach to overcome limitations and enable TB

- Experimental results show improved performance
Modelling practical applications

- Stencils everywhere, but not only. What else?

- Vast literature 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 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

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.

```
1 for t = 1 to nt do
2    for x = 1 to nx do
3       for y = 1 to ny do
4          for z = 1 to nz do
5              A(t, x, y, z) ≡ u[t, x, y, z] = u[t-1, x, y, z] + \sum_{r=1}^{r=so/2} w_r \left( 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] \right);
6       foreach s in sources do // For every source
7          for i = 1 to np do // Get the points affected
8              xs, ys, zs = map(s, i) // through indirection
9              u[t, xs, ys, zs] += f(src(t,s)) // add their impact on the 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
- Source injection in a different iteration space
- 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
Methodology

• A negligible-cost scheme to precompute the source injection contribution.
• Align source injection data dependences to the grid
• 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

- **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.

```plaintext
1  for  t = 1 to 2  do
2       foreach  s in  sources  do
3           for  i = 1 to  np  do
4                   xs, ys, zs = map(s, i);
5                   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.

<table>
<thead>
<tr>
<th>Off-the-grid</th>
<th>Aligned</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{len(sources)})</td>
<td>(n_{\text{src}})</td>
</tr>
<tr>
<td>(\text{len(sources.coords)})</td>
<td>((n_{\text{src}}, 3))</td>
</tr>
<tr>
<td>(\text{len(sources.data)})</td>
<td>((n_{\text{src}}, nt))</td>
</tr>
</tbody>
</table>

**Listing 3:** Decomposing the source injection wavefields.

```python
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.

```plaintext
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
          SIMD? (AVX512)
          u[t, x, y, z2] += SM[x, y, z2] * src_dcmp[t, SID[x, y, z2]];
```
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 SID by cutting off zero z-slices

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

```plaintext
1  for t = 1 to nt do
2      for x = 1 to nx do
3          for y = 1 to ny do
4              for z = 1 to nz do
5                  A(t, x, y, z, s);
6              for z2 = 1 to nnz_mask[x][y] do
7                  I(t, x, y, z, s) ≡ { zind = Sp_SID[x, y, z2];
8                  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.

```c
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] + \sum_{r=0}^{s/2} w_r \left( 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] \right);
      
    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
    
  
```

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

```c
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]]; } 
```
Applying wave-front temporal blocking

- Aligning, automated in Devito DSL; TB with manual loop transformation
- The new source coordinates are aligned to the grid now
- 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.
Listing 6: The figure shows the loop structure after applying our proposed scheme.

```plaintext
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);
```
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

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

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

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

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

TABLE I: VM specification
Cache-aware roofline model

Space order:
- △4
- ○8
- □12

Broadwell, isotropic acoustic, $512^3$ grid points, 512ms
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).


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
Acknowledgements

Thanks to collaborators and contributors:
• Navjot Kukreja (Imperial College)
• John Washbourne (Chevron)
• Edward Caunt (Imperial College)

Thank you for your attention!
Questions?
References


Corner cases, increasing number of sources

Speed-up for increasing number of sources

- Sparse injection
- Dense injection
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 r11 = -2.84722222F*uref[t1][x + 8][y + 8][z + 8];
                    float r12 = 1.0F/dt;
                    float r13 = 1.0F/(dt*dt);
                    float r14 = -1.78571429e-3F*uref[t1][x + 8][y + 8][z + 8];
                    float r15 = 2.59683254e-2F*(uref[t1][x + 8][y + 8][z + 8] + uref[t1][x + 8][y + 8][z + 8] + uref[t1][x + 8][y + 8][z + 8] + uref[t1][x + 8][y + 8][z + 8])
                        + r14;  
                    float uref[t0][x + 8][y + 8][z + 8] = (r11*(r12*(-2.0F*uref[t1][x + 8][y + 8][z + 8])
                        + 2.59683254e-2F*(uref[t1][x + 8][y + 8][z + 8] + uref[t1][x + 8][y + 8][z + 8])
                        + uref[t1][x + 8][y + 8][z + 8])
                        + r14;  
                    float uref[t2][x + 8][y + 8][z + 8] = (r14 - 1.78571429e-3F*(uref[t1][x + 8][y + 8][z + 8]
                        + uref[t1][x + 8][y + 8][z + 8])
                        + r14;  
                    float uref[t3][x + 8][y + 8][z + 8] = (r14 - 1.78571429e-3F*(uref[t1][x + 8][y + 8][z + 8]
                        + uref[t1][x + 8][y + 8][z + 8])
                        + r14;  
                    float uref[t4][x + 8][y + 8][z + 8] = (r14 - 1.78571429e-3F*(uref[t1][x + 8][y + 8][z + 8]
                        + uref[t1][x + 8][y + 8][z + 8])
                        + r14;  
                    float uref[t5][x + 8][y + 8][z + 8] = (r14 - 1.78571429e-3F*(uref[t1][x + 8][y + 8][z + 8]
                        + uref[t1][x + 8][y + 8][z + 8])
                        + r14;  
                    float uref[t6][x + 8][y + 8][z + 8] = (r14 - 1.78571429e-3F*(uref[t1][x + 8][y + 8][z + 8]
                        + uref[t1][x + 8][y + 8][z + 8])
                        + r14;  
                    float uref[t7][x + 8][y + 8][z + 8] = (r14 - 1.78571429e-3F*(uref[t1][x + 8][y + 8][z + 8]
                        + uref[t1][x + 8][y + 8][z + 8])
                        + r14;  
                    float uref[t8][x + 8][y + 8][z + 8] = (r14 - 1.78571429e-3F*(uref[t1][x + 8][y + 8][z + 8]
                        + uref[t1][x + 8][y + 8][z + 8])
                        + r14);  
                }
            }
        }
    }
}
```
The generated C code - source injection

/* Begin section */
#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 ll_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 && ll_src_1 >= y_M - 1 && ll_src_2 >= z_M - 1 && ll_src_0 <= x_M + 1 && ll_src_1 <= y_M + 1 && ll_src_2 <= z_M + 1) ||
            (float r0 = 4.9801682821664F*(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[ttime][p_src];
        #pragma omp atomic update
        uref[p0][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 = 4.9801682821664F*(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[ttime][p_src];
        #pragma omp atomic update
        uref[p0][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 = 4.9801682821664F*(vp[ii_src_0 + 8][ii_src_2 + 8][ii_src_4 + 8]*vp[ii_src_0 + 8][ii_src_2 + 8][ii_src_4 + 8])*nx*ny*nz/(h_x*h_y*h_z) - nx*ny/(h_x*h_y) - ny*nz/(h_y*h_z) + nx/h_x + ny/h_y)*src[ttime][p_src];
        #pragma omp atomic update
        uref[p0][ii_src_0 + 8][ii_src_2 + 8][ii_src_4 + 8] += r2;
Algorithm 3: Source injection pseudocode.

1. for $t = 1$ to $nt$ do
   2. foreach $s$ in sources do
      3. # Find on the grid coordinates
         4. src_x_min = floor(src_coords[s][0], ox)
         5. src_x_max = ceil(src_coords[s][0], ox)
         6. src_y_min = floor(src_coords[s][1], oy)
         7. src_y_max = ceil(src_coords[s][1], oy)
         8. src_z_min = floor(src_coords[s][2], oz)
         9. src_z_max = ceil(src_coords[s][2], oz)
      10. # Compute weights
          11. px = f(src_coords[s][0], ox)
          12. py = f(src_coords[s][1], oy)
          13. pz = f(src_coords[s][2], oz)
      14. # Unrolled for 8 points
          15. if src_x_min, src_y_min, src_z_min in grid then
             16. $r_0 = v(src_x_min, src_y_min, src_z_min, src[t][s])$
             17. $u[t, src_x_min, src_y_min, src_z_min] += r_0$
          18. else if src_x_max, src_y_max, src_z_max in grid then
             19. $r_7 = v(src_x_max, src_y_max, src_z_max, src[t][s])$
             20. $u[t, src_x_max, src_y_max, src_z_max] += r_7$
Gpts/s for fixed tile size. (Sweeping block sizes)
Algorithm 3: Source injection pseudocode.

1. for $t = 1$ to $nt$ do
2.   foreach $s$ in sources do
3.     # Find on the grid coordinates
4.     src_x_min = floor(src_coords[s][0], ox)
5.     src_x_max = ceil(src_coords[s][0], ox)
6.     ...
7.     # Compute weights
8.     px = f(src_coords[s][0], ox)
9.     ...
10.    # Unrolled for 8 points ($2^3$, 3D case)
11.   if $src_x_min$, … in grid then
12.       $r0 = v(src_x_min, … src[t][s])$
13.       $u[t, src_x_min, …] += r0$
14.   else
15.       $r7 = v(src_x_max, … src[t][s])$
16.       $u[t, src_x_max, …] += r7$
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
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
    $r0 = v(src_x_min, ...src[t][s])$;
    $u[t, src_x_min, ...] += r0$
  
  # Unrolled loop for each affected point, compute injection part and add to field
  if $src_x_max, ...$ in grid then
    $r7 = v(src_x_max, ...src[t][s])$;
    $u[t, src_x_max, ...] += r7$
Is this buffered?
Missing injection?
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.

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

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

1 for t = 1 to nt do
2 for x = 1 to nx do
3   for y = 1 to ny do
4     for z = 1 to nz do
5       $A(t, x, y, z, s)$;
6   for z2 = 1 to nnz_mask[x][y] do
7     zind = Sp_SM[x, y, z];
8     u[t, x, y, z2] += SM[x, y, zind] * 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] + \sum_{r=1}^{s/2} w_r \left( 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] \right);
        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

```

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) \equiv \{ zind = Sp_SM[x, y, z2];
                u[t, x, y, z2] += SM[x, y, zind] * src_dcmp[t, SID[x, y, zind]]; \}
```
The transformation in Devito-DSL

```python
time
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])
```
# f : perform source injection on an empty grid

\[
f = \text{TimeFunction(name="f", grid=model.grid, space\_order=so, time\_order=2)}
\]

\[
src\_f = \text{src.inject(field=f.forward, expr=src * dt**2 / model.m)}
\]

\[
\text{op\_f = Operator([src\_f])}
\]

\[
\text{op\_f\_sum = op\_f.apply(time=3)}
\]

\[
nzinds = \text{np.nonzero(f.data[0])} \quad \# \text{nzinds is a tuple}
\]

\[
\text{eq0 = Eq(sp\_zi.symbolic\_max, nnz\_sp\_source\_mask[x, y] - 1, implicit\_dims=(time, x, y))}
\]

\[
\text{eq1 = Eq(zind, sp\_source\_mask[x, y, sp\_zi], implicit\_dims=(time, x, y, sp\_zi))}
\]

\[
\text{mask\_expr = source\_mask[x, y, zind] * save\_src[time, source\_id[x, y, zind]]}
\]

\[
\text{eq2 = Inc(usol.forward[t+1, x, y, zind], mask\_expr, implicit\_dims=(time, x, y, sp\_zi))}
\]

\[
\text{pde\_2 = model.m * usol.dt2 - usol.laplace + model.damp * usol.dt}
\]

\[
\text{stencil\_2 = Eq(usol.forward, solve(pde\_2, usol.forward))}
\]
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.

```plaintext
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);
      end for
    end for
  end for
end for
for z2 = 1 to nz do
  u[t, x, y, z2] += SM[x, y, z2] * src_dcmp[t, SID[x, y, z2]];
end for
```
FD grid + sparse off-the-grid

survey ship

source of shock waves (air gun)

hydrophones
Stencils are everywhere

- Vast literature on optimizing stencils…

- Parallelism, cache optimizations, accelerators

- From simplistic (1d-3pt)
  ...to wide and
  ...complex

Illustration of an Offshore Seismic Survey, Source: Kris Energy