Automated Temporal Blocking in the Devito DSL and Compiler framework

George Bisbas\textsuperscript{1,2}, Fabio Luporini\textsuperscript{3}, Mathias Louboutin\textsuperscript{4}, Rhodri Nelson\textsuperscript{2}, Gerard Gorman\textsuperscript{2,3}, Paul H.J. Kelly\textsuperscript{1}

\textsuperscript{1}Imperial College London, Dept. of Computing
\textsuperscript{2}Imperial College London, Dept. of Earth Sciences and Engineering
\textsuperscript{3}Devito Codes, UK
\textsuperscript{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?
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, $\approx O(\text{billions})$ 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
void kernel(...) {

    ...

    <impenetrable code with aggressive performance optimizations, manually applied, full-time human resources, less reproducibility, debugging nightmares>
    ...

}
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)
Introducing Devito

- **Open source** - MIT lic. - Try now!  
  [https://github.com/devitocodes/devito](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!)

Adapted from Aryaman Sharda Cueto et.al. (2022)
2D Heat diffusion modelling

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

\[u(0,y) = 0, \quad u(0,1) = 0, \quad u(x,0) = 0, \quad u(1,y) = 0\]

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.)
bright = Eq(u[t + 1, nx-1, y], 0.)
bc_top = Eq(u[t + 1, x, ny-1], 0.)
bbottom = 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

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

Write once, Run everywhere!
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 + 2][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] + r2*r4 + r2*u[t0][x + 2][y + 2][z + 3]) + 1.0e-1F;
                    }
                }
            }
        }
    }
}
Wavefront temporal blocking

Highest x-value that may be computed for t=4 due to the dependencies on t=3 values

Legend:
- Value already known
- Value being computed
- Value not yet known
- Value replaced in buffer
- Known value used as input

Overlapped temporal blocking

Only two buffered values for each point are kept in memory, the blue value (t-1) is being replaced by the green one (t+1)

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)

```c
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 int = 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));
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```
Experimental evaluation, low discretization orders

- 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

Table 4.1: Characteristics of CPU platforms used for benchmarking

<table>
<thead>
<tr>
<th>CPU characteristics</th>
<th>i7-10700KF</th>
<th>Gold 5218R</th>
<th>Gold 6230</th>
<th>EPYC 7742</th>
<th>E5-2640</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU(s)</td>
<td>16</td>
<td>80</td>
<td>40</td>
<td>256</td>
<td>32</td>
</tr>
<tr>
<td>Thread(s) per core</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Core(s) per socket</td>
<td>8</td>
<td>20</td>
<td>20</td>
<td>64</td>
<td>8</td>
</tr>
<tr>
<td>Socket(s)</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>NUMA node(s)</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>L1d cache: 256KiB</td>
<td>1.3MiB</td>
<td>32KiB</td>
<td>32KiB</td>
<td>512KiB</td>
<td></td>
</tr>
<tr>
<td>L1i cache: 256KiB</td>
<td>1.3MiB</td>
<td>32KiB</td>
<td>32KiB</td>
<td>512KiB</td>
<td></td>
</tr>
<tr>
<td>L2 cache: 2MiB</td>
<td>40MiB</td>
<td>1MiB</td>
<td>512KiB</td>
<td>4MiB</td>
<td></td>
</tr>
<tr>
<td>L3 cache: 16MiB</td>
<td>55MiB</td>
<td>27.5MiB</td>
<td>16MiB</td>
<td>40MiB</td>
<td></td>
</tr>
</tbody>
</table>
Experimental evaluation, higher discretization orders

- 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

**Table 4.1:** Characteristics of CPU platforms used for benchmarking
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

Join us, use Devito, work with us!
References


Appendix
...And works in tandem with all other Devito optimizations!

```c
#pragma omp parallel num_threads(nthreads)
{
    #pragma omp for collapse(2) schedule(dynamic,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)
                {
                    #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 + 1][z + 2] + r1*u[t0][-time + x + 3][-time + y + 2][z + 2] + r2*r4 + r2*u[t0][-time + x + 2][-time + y + 2][z + 2] + r2*u[t0][-time + x + 2][-time + y + 2][z + 2]);
                    }
                }
            }
        }
    }
}
```
Cache blocking optimizations in Devito

- **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)

GFLOps/s  
Data transfers between CPU and memory (GB)  
Elapsed Time (s)
Cache blocking optimizations in Devito

**Temporal blocking Methodology:**

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

For a given amount of available data, try to compute whatever can be computed!
1. A pass where time loop is blocked + another space loop level added

```c
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)
            {
                r4 = -2.0F * u[t0][x + 2][y + 2][z + 2];
                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

```c
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 + 2][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

```cpp
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);
                                                        } // for z = z_m; z <= z_M; z += 1)
                                                } // for y = y0_blk1; y <= MIN(MIN(y0_blk0 + y0_blk0_size - 1, time + y_M), y0_blk1 + y0_blk1_size - 1); y += 1)
                                        } // for x = x0_blk1; x <= MIN(MIN(x0_blk0 + x0_blk0_size - 1, time + x_M), x0_blk1 + x0_blk1_size - 1); x += 1)
                                    } // for 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 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_blk0 = y_m; y0_blk0 <= time_M - time_m + y_M; y0_blk0 += y0_blk0_size)
                        } // for int x0_blk0 = x_m; x0_blk0 <= time_M - time_m + x_M; x0_blk0 += x0_blk0_size)
                    } // for int time0_blk0 = time_m; time0_blk0 <= time_M; time0_blk0 += time0_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_blk0 = y_m; y0_blk0 <= time_M - time_m + y_M; y0_blk0 += y0_blk0_size)
    } // for int x0_blk0 = x_m; x0_blk0 <= time_M - time_m + x_M; x0_blk0 += x0_blk0_size)
} // for int time0_blk0 = time_m; time0_blk0 <= time_M; time0_blk0 += time0_blk0_size)
```
Works in tandem with all other Devito opts!

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

- 98 GFLOps/s, 643 s, 2371 GB on L1
- 103 GFLOps/s, 32 s
- 1093 GB on L1
- ~130 GFLOps/s, 33 s
- 167 GFLOps/s, 20 s
- 78 GFLOps/s, 42 s
- 1093 GB on L1

(More ops per data on L1)

(More ops per data on L1)

(+Spatial Blocking)

(+Wavefront Temporal Blocking)
Cache shares
Laplacian, discretization order 8, $512^3$ grid points, 512ms (Intel Skylake)

Highly optimized, spatially blocked, vectorized kernel

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

+ wavefront temporal blocking

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

<table>
<thead>
<tr>
<th>Memory Metrics</th>
<th>Impacts</th>
<th>Shares</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>6%</td>
<td>969.697GB</td>
</tr>
<tr>
<td>L2</td>
<td>11%</td>
<td>540.682GB</td>
</tr>
<tr>
<td>L3</td>
<td>13%</td>
<td>306.668GB</td>
</tr>
<tr>
<td>DRAM</td>
<td>68%</td>
<td>101.947GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Metrics</th>
<th>Impacts</th>
<th>Shares</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>13%</td>
<td>1023.946GB</td>
</tr>
<tr>
<td>L2</td>
<td>24%</td>
<td>573.586GB</td>
</tr>
<tr>
<td>L3</td>
<td>27%</td>
<td>323.449GB</td>
</tr>
<tr>
<td>DRAM</td>
<td>33%</td>
<td>24.925GB</td>
</tr>
</tbody>
</table>
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...

# 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 +
u[t0, x + 2, y + 3]/h_y**2 +
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_ON)>;
<C.Statement _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON)>;
<C.Line >
<List (0, 2, 0)>
<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(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)
{
    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 */
    }
}
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

```c
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 */
    }
}
```
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 + 1] = 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 */
    }
}
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;
                }
            } // #pragma omp parallel
        } // #pragma omp parallel
        STOP_TIMER(section0,timers)
        /* End section0 */
    }

    return 0;
}
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 devicerrm, 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 (devicerrm)
        return 0;
    }

Pipeline for each target: GPU/OpenACC
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](image1.png)

![A 3d-19pt stencil update](image2.png)

![Off-the-grid operators (Source injection/Receiver interpolation)](image3.png)
So = 4

A SYSTEM OF PARTIAL DIFFERENTIAL EQUATIONS

so = 12

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}}>
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

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

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.

Open-source, on top of latest Devito!

---

**TABLE I: VM specification**

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

---

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, -2.0F*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>++]

---

Optimized! 😊😊😊
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 $\Rightarrow$ 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.

```python
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

<table>
<thead>
<tr>
<th>Off-the-grid</th>
<th>Aligned</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{len}(\text{sources}) )</td>
<td>( n_{\text{src}} )</td>
</tr>
<tr>
<td>( \text{len}(\text{sources.coords}) )</td>
<td>( (n_{\text{src}}, 3) )</td>
</tr>
<tr>
<td>( \text{len}(\text{sources.data}) )</td>
<td>( (n_{\text{src}}, nt) )</td>
</tr>
<tr>
<td>( \text{len}(\text{sources.coords}) )</td>
<td>( n_{\text{aff.pts}} )</td>
</tr>
<tr>
<td>( \text{len}(\text{sources.data}) )</td>
<td>( (n_{\text{aff.pts}}, 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));
```
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
    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 \textit{nnz\_mask}
- Reduce the size of SID by cutting off zero z-slices

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure.png}
\caption{Stencil kernel update with reduced size iteration space for source injection.}
\end{figure}

\begin{lstlisting}
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 \( \text{nnz\_mask}[x][y] \) do
        \( I(t, x, y, z, s) \equiv \{ \text{zind} = \text{Sp\_SID}[x, y, z2]; \}
        \( u[t, x, y, z2] += \text{src\_dcmp}[t, \text{SID}[x, y, \text{zind}]]; \}
\end{lstlisting}
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] + ∑_{r=so/2}^{r=so} 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
  
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

- 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.

```c
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)
Corner cases, increasing number of sources

Speed-up for increasing number of sources

- **Speed-up**
- **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) {
            #pragma omp simd aligned(damp, uref, vp:32)
            for (int z = z_m; z <= z_M; z += 1) {
                float r14 = -2.04722222f*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][z+8]*vp[x+8][y+8][z+8][z+8]);
                uref[t0][x+8][y+8][z+8][z+8] = (r11*(-r12*(-2.0f*uref[t1][x+8][y+8][z+8][z+8] +
                             uref[t2][x+8][y+8][z+8][z+8])) + r13*(damp[x+1][y+1][z+1]*uref[t1][x+8][y+8][z+8][z+8]) +
                             (r14 - 1.78571429e-3f*(uref[t1][x+8][y+8][z+8][z+8] +
                             uref[t2][x+8][y+8][z+8][z+8] + uref[t1][x+8][y+8][z+8][z+8] + uref[t1][x+8][y+8][z+8][z+8] +
                             uref[t1][x+8][y+8][z+8][z+8]) + 2.0e-1f*(uref[t1][x+8][y+8][z+8][z+8] + uref[t1][x+8][y+8][z+8][z+8]) +
                             (r14 - 1.78571429e-3f*(uref[t1][x+8][y+8][z+8][z+8] +
                             uref[t1][x+8][y+8][z+8][z+8] + uref[t1][x+8][y+8][z+8][z+8] + uref[t1][x+8][y+8][z+8][z+8] +
                             uref[t1][x+8][y+8][z+8][z+8]) + 2.0e-1f*(uref[t1][x+8][y+8][z+8][z+8] + uref[t1][x+8][y+8][z+8][z+8]) +
                             (r14 - 1.78571429e-3f*(uref[t1][x+8][y+8][z+8][z+8] +
                             uref[t1][x+8][y+8][z+8][z+8] + uref[t1][x+8][y+8][z+8][z+8] + uref[t1][x+8][y+8][z+8][z+8] +
                             uref[t1][x+8][y+8][z+8][z+8]))/(h_x*h_x));
            }
        }
    }
}
```
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 tl_src_0 = (int)(floor((-o_x + src_coords[p_src][0])/h_x));
    int tl_src_1 = (int)(floor((-o_y + src_coords[p_src][1])/h_y));
    int tl_src_2 = (int)(floor((-o_z + src_coords[p_src][2])/h_z)) + 1;
    int tl_src_3 = (int)(floor((-o_x + src_coords[p_src][0])/h_x)) + 1;
    int tl_src_4 = (int)(floor((-o_y + src_coords[p_src][1])/h_y)) + 1;
    int tl_src_5 = (int)(floor((-o_z + src_coords[p_src][2])/h_z)) + 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 (tl_src_0 >= x_m - 1 && tl_src_1 >= y_m - 1 && tl_src_2 >= z_m - 1 && tl_src_0 <= x_M + 1 && tl_src_1 <= y_M + 1 && tl_src_2 <= z_M + 1)
    {
        float r0 = 4.49016082216644F*(vp[tl_src_0 + 8][tl_src_1 + 8][tl_src_2 + 8]*vp[tl_src_0 + 8][tl_src_1 + 8][tl_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][tl_src_0 + 8][tl_src_1 + 8][tl_src_2 + 8] += r0;
    }
    if (tl_src_0 >= x_m - 1 && tl_src_1 >= y_m - 1 && tl_src_3 >= z_m - 1 && tl_src_0 <= x_M + 1 && tl_src_1 <= y_M + 1 && tl_src_3 <= z_M + 1)
    {
        float r1 = 4.49016082216644F*(vp[tl_src_0 + 8][tl_src_1 + 8][tl_src_3 + 8]*vp[tl_src_0 + 8][tl_src_1 + 8][tl_src_3 + 8])*(px*py*pz/(h_x*h_y*h_z) - px*py/(h_x*h_z) - py*pz/(h_y*h_z) + pz/h_z)*src[time][p_src];
        #pragma omp atomic update
        uref[t0][tl_src_0 + 8][tl_src_1 + 8][tl_src_3 + 8] += r1;
    }
    if (tl_src_0 >= x_m - 1 && tl_src_2 >= z_m - 1 && tl_src_4 >= y_m - 1 && tl_src_0 <= x_M + 1 && tl_src_2 <= z_M + 1 && tl_src_4 <= y_M + 1)
    {
        float r2 = 4.49016082216644F*(vp[tl_src_0 + 8][tl_src_4 + 8][tl_src_2 + 8]*vp[tl_src_0 + 8][tl_src_4 + 8][tl_src_2 + 8])*(nx*ny*nz/(h_x*h_v*h_z) - nx*ny/(h_x*h_v) - ny*nz/(h_y*h_z) + ny/h_y)*src[time][p_src];
        #pragma omp atomic update
        uref[t0][tl_src_0 + 8][tl_src_4 + 8][tl_src_2 + 8] += r2;
    }
}
```
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 loop for each affected point, compute injection part and add to field
  if src_x_min, src_y_min, src_z_min in grid then
    $r_0 = v(src_x_{min}, src_y_{min}, src_z_{min}, src[t][s])$
    $u[t, src_x_{min}, src_y_{min}, src_z_{min}] += r_0$
  
  if src_x_max, src_y_max, src_z_max in grid then
    $r_7 = v(src_x_{max}, src_y_{max}, src_z_{max}, src[t][s])$
    $u[t, src_x_{max}, src_y_{max}, src_z_{max}] += r_7$
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.     :
15. if $src_x_max$, ... in grid then
16.     |\ r7 = v(src_x_max, ... src[t][s]);
17.     |\ u[t, src_x_max, ...] += r7)
18. }
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
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 (2D, 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
Dependency violation

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.

\[
\text{for } t = 1 \text{ to } nt \text{ do }
\]
\[
\text{for } x = 1 \text{ to } nx \text{ do }
\]
\[
\text{for } y = 1 \text{ to } ny \text{ do }
\]
\[
\text{for } z = 1 \text{ to } nz \text{ do }
\]
\[
A(t, x, y, z) \equiv u[t, x, y, z] = u[t-1, x, y, z] + \sum_{r=1}^{s0/2} w_r \left[ u[t-1, x - r, y, z] + u[t-1, x + r, y, z] + u[t-1, x, y - r, z] + u[t-1, x, y + r, z] + u[t-1, x, y, z - r] + u[t-1, x, y, z + r] \right];
\]
\[
\text{foreach } s \text{ in sources do }
\]
\[
\text{for } i = 1 \text{ to } np \text{ do }
\]
\[
xs, ys, zs = \text{map}(s, i);
\]
\[
u[t, xs, ys, zs] += f(src(t, s))
\]

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

\[
\text{for } t = 1 \text{ to } nt \text{ do }
\]
\[
\text{for } x = 1 \text{ to } nx \text{ do }
\]
\[
\text{for } y = 1 \text{ to } ny \text{ do }
\]
\[
\text{for } z = 1 \text{ to } nz \text{ do }
\]
\[
A(t, x, y, z, s);
\]
\[
z2 = 1 \text{ to } \text{nnz_mask}[x][y] \text{ do }
\]
\[
\text{zind} = \text{Sp_SM}[x, y, z];
\]
\[
u[t, x, y, z2] += \text{SM}[x, y, zind] \times \text{src_dcmp}[t, \text{SID}[x, y, zind]];
\]
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] + \sum_{r=so/2}^{r=sr/2} \mathcal{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]);
```

```python
# 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 nznz_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]]; }
```
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

```python
# 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

# 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.

```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
        u[t, x, y, z2] += SM[x, y, z2] * src_dcmp[t, SID[x, y, z2]];
  |
```