

A shared compilation stack for distributed-memory parallelism in stencil DSLs

G. Bisbas^{*1}, A. Lydike^{*2}, E. Bauer^{*2}, N. Brown^{*2},
M. Fehr², L. Mitchell, G. Rodriguez-Canal², M. Jamieson²,
P. H.J. Kelly¹, M. Steuwer³, T. Grosser⁴

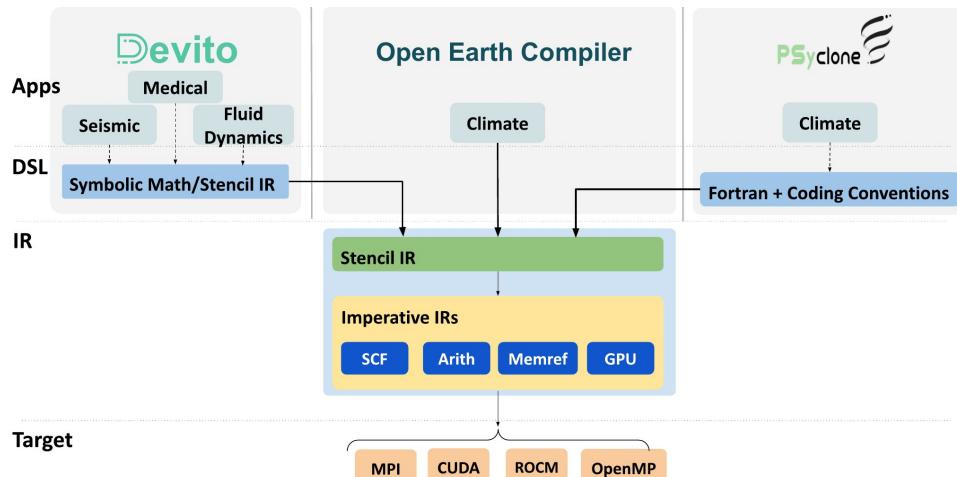
***authors contributed equally**

¹Imperial College London, UK

²The University of Edinburgh, UK

³Technische Universität Berlin, Germany

⁴University of Cambridge, UK



The problem: Monolithic Domain-specific languages

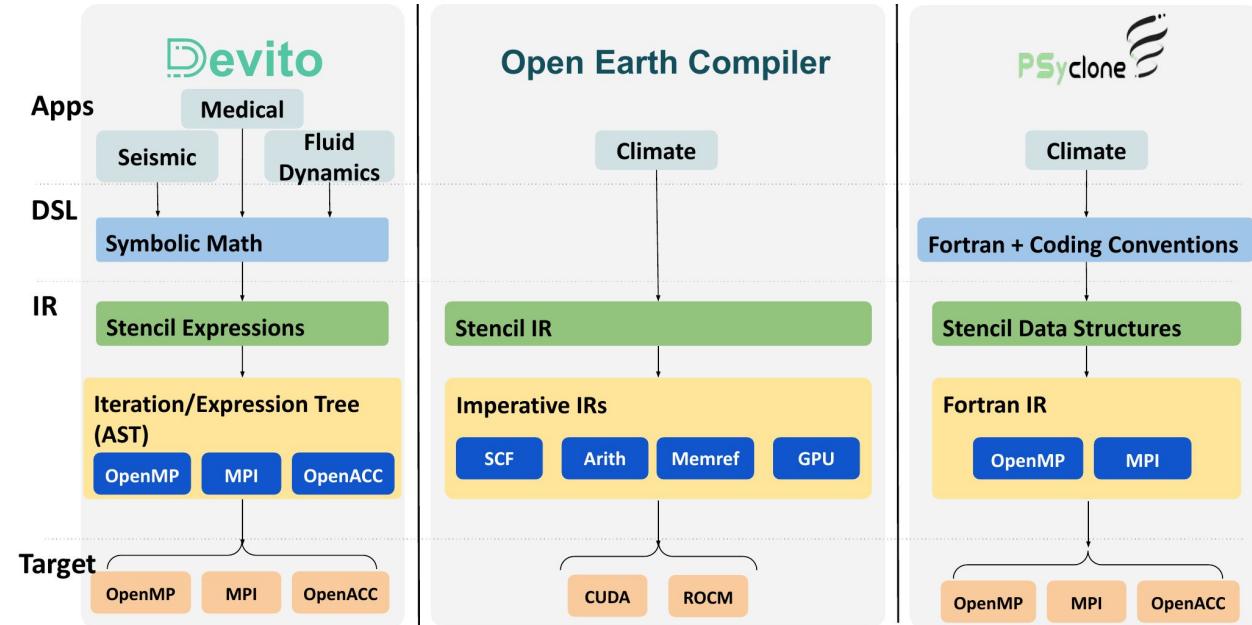
Tailored to their domain, but actually lots of common generic concepts!

- ✓ Performance
- ✓ Productivity
- ✓ Portability

but...

Technical challenges:

- ✗ Independent/Siloed
- ✗ Lack of code reuse
- ✗ Separate
- ✗ Short lifespan



Societal challenges:

- ✗ Disjoint communities
- ✗ Lack of knowledge transfer

The problem: Monolithic Domain-specific languages

Tailored to their domain, but actually lots of common generic concepts!

- ✓ Performance
- ✓ Productivity
- ✓ Portability

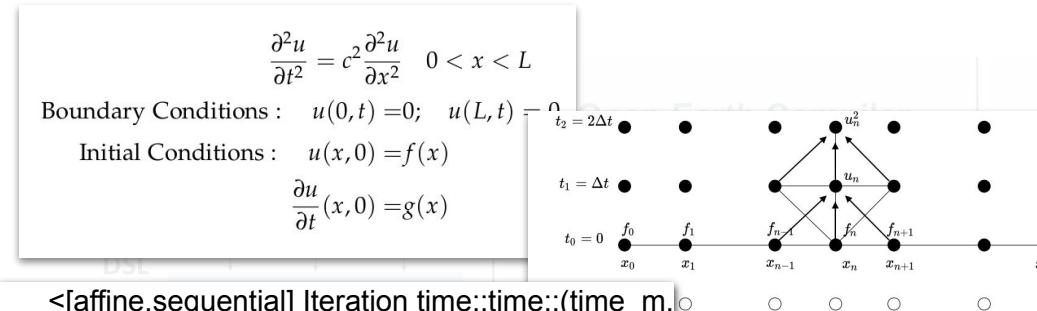
but...

Technical challenges:

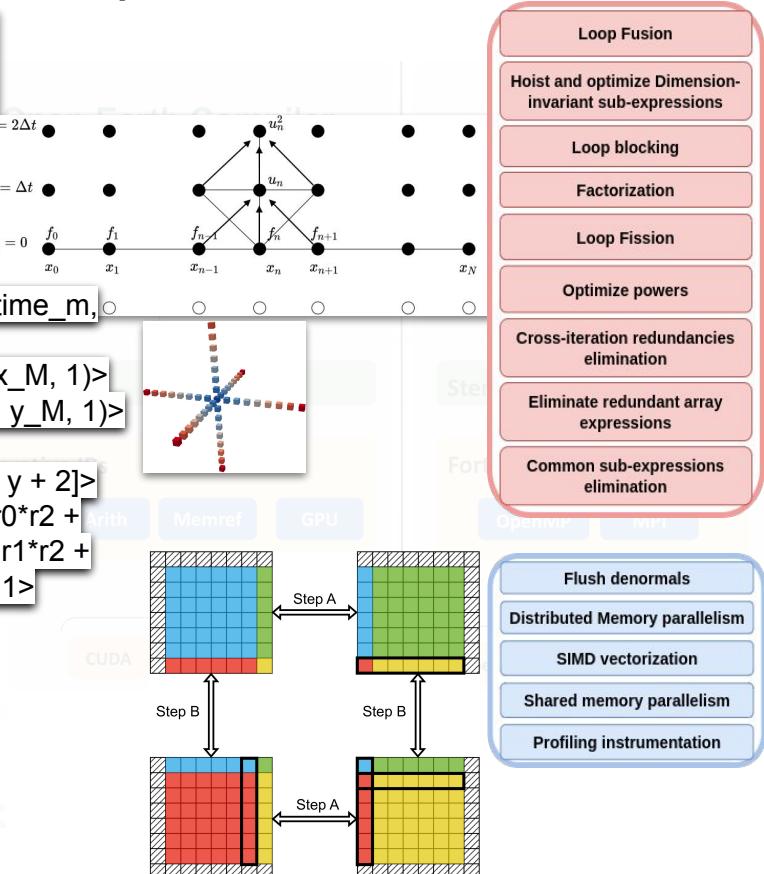
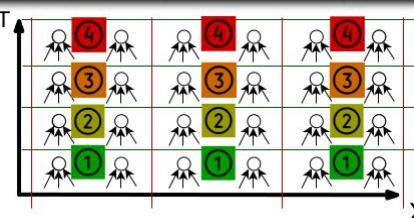
- ✗ Independent/Siloed
- ✗ Lack of code reuse
- ✗ Separate
- ✗ Short lifespan

Societal challenges:

- ✗ Disjoint communities
- ✗ Lack of knowledge transfer

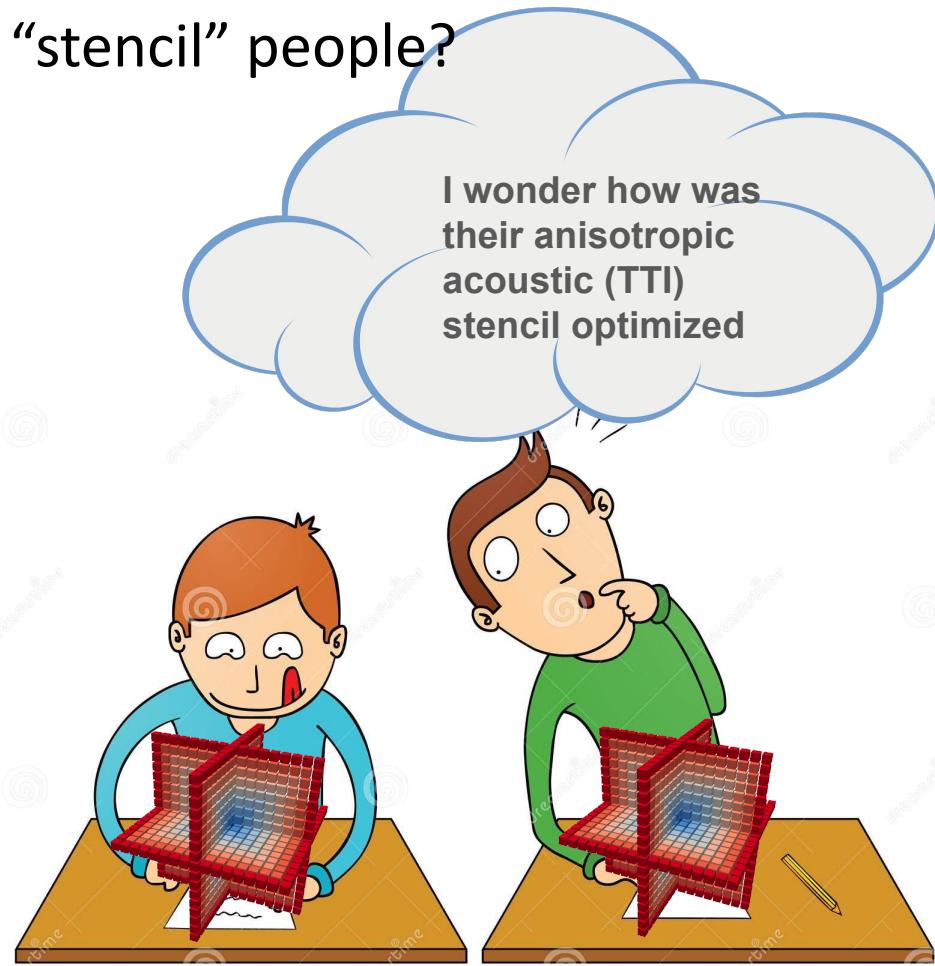


```
<[affine,sequential] Iteration time::time::(time_m,  
time_M, 1)>  
<[affine,parallel] Iteration x::x::(x_m, x_M, 1)>  
<[affine,parallel] Iteration y::y::(y_m, y_M, 1)>  
<ExpressionBundle (2)>  
<Expression r2 = -2.0*u[t0, x + 2, y + 2]>  
<Expression u[t1, x + 2, y + 2] = r0*r2 +  
r0*u[t0, x + 1, y + 2] + r0*u[t0, x + 3, y + 2] + r1*r2 +  
r1*u[t0, x + 2, y + 1] + r1*u[t0, x + 2, y + 3] + 1>
```



Have you ever asked these Qs to “stencil” people?

- What **tiling strategy** are you using for cache efficiency?
- Are your OpenMP parallel loops using dynamic **scheduling**?
- What is your **MPI strategy** for running on X machine
- What is your **process pinning** strategy?
- Are you using **temporal blocking**? How?
- Yes, but is this good for my “wide” stencils?
- How do you handle **boundary conditions**?
- **GPUs:** OpenACC or OpenMP offloading or CUDA?
- Gpts/s, Gpts/Watt, Gpts/\$£ ?
- What about your solver ?
 - Hand-written?
 - DSL + codegen?
 - Template-based?
 - Hybrid approaches?



We propose: A shared compilation stack for HPC in stencil DSLs

We propose to use compiler technology for it!



MLIR for the win! (HPC 🤪 ?)

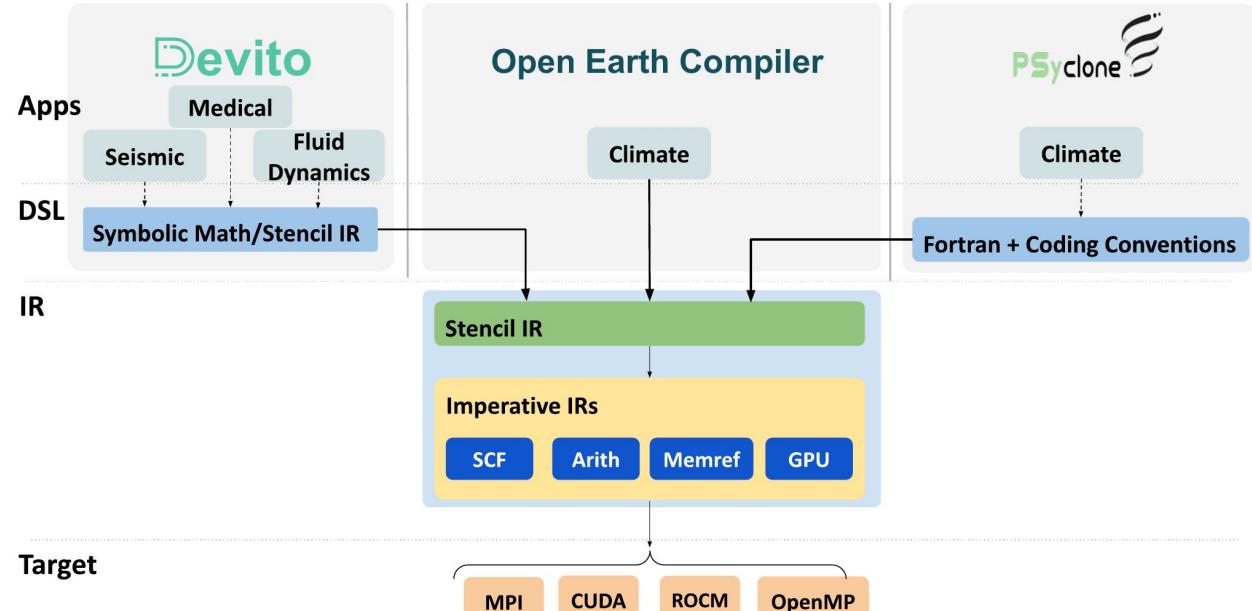
- ✓ Performance
- ✓ Productivity
- ✓ Portability

Technical benefits:

- ✓ Composability
- ✓ Code Reuse
- ✓ Interoperability
- ✓ Longevity

Societal benefits:

- ✓ Connected communities
- ✓ Extensive knowledge transfer



Contributed infrastructure:

★ A set of HPC-specific compilation components focusing on FD-stencil computations:

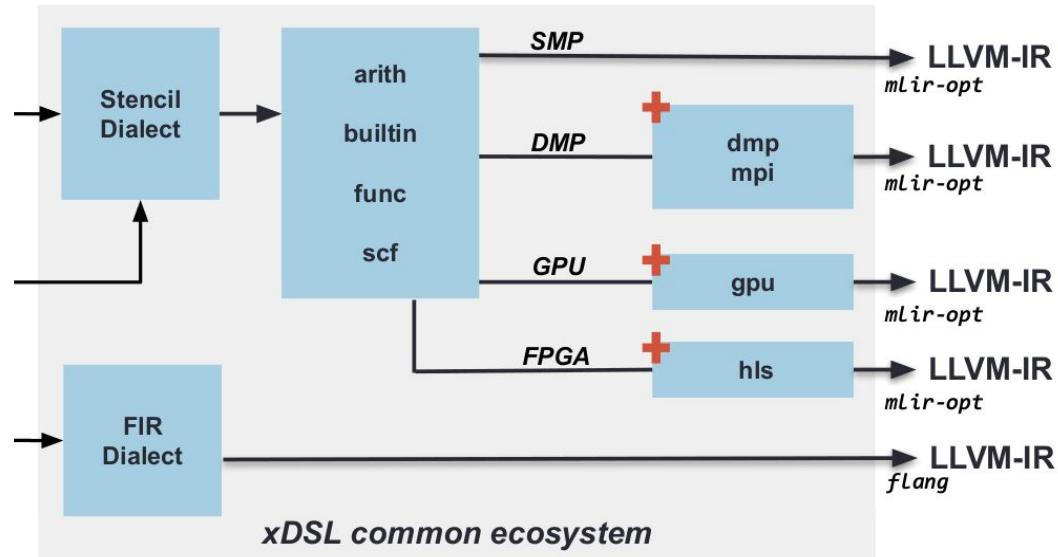
- “dmp” dialect for automated domain decomposition
- “mpi” dialect for message passing
(✓ Upstreamed to MLIR !)

★ A shared compilation stack for Devito and Psyclone:

- Based on SSA concepts
- Utilizing  xDSL /MLIR

★ Performance evaluation shows:

- ✓ Highly competitive DMP solvers
- ✓ Support for new targets and hardware
- ✓ Improved performance for a number of cases



xDSL: A Python-native SSA Compiler Framework

- ✓ SSA-based IRs
- ✓ SSA + regions concept
- ✓ Mix predefined IRs
- ✓ Add custom IRs
- ✓ Connect with MLIR/LLVM
- ✓ Benefit from Python's productivity
- ✓ Open-source/CI/CD/codecov
- ✓ Active contributor community
- ✓ Join us on <https://xdsl.zulipchat.com/>



About

A Python Compiler Design Toolkit

- Readme
- View license
- Activity
- Custom properties
- 204 stars
- 18 watching
- 57 forks

Report repository

Releases 28

v0.19.0 (Latest)
3 weeks ago

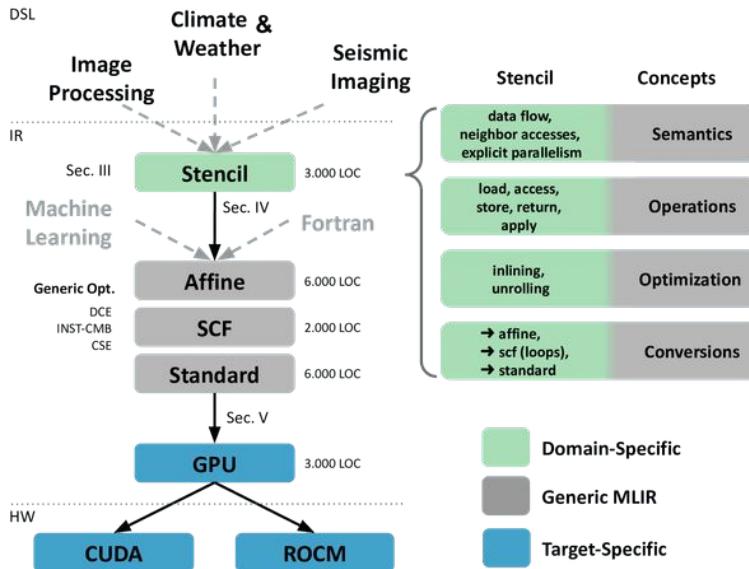
+ 27 releases

Contributors 54



+ 40 contributors

The Open Earth Compiler: the ‘stencil’ dialect



```
1  %source = stencil.load(%114) : (!field<[0,128]xf64>
2          -> !temp<?xf64>
3
4  %out = stencil.apply(%arg = %source : !temp<?xf64>
5          -> !temp<?xf64> {
6      %l = stencil.access %arg[-1] : f64
7      %c = stencil.access %arg[0] : f64
8      %r = stencil.access %arg[1] : f64
9      // %v = %l + %r - 2.0 * %c
10     stencil.return %v : f64
11
12    stencil.store %out to %target([1]:[127])
13        : !temp<?xf64> to !field<[0,128]xf64>
```

Listing 1. Example MLIR for 1-dimensional 3-point Jacobi stencil.

- ✓ Updated, ported to xDSL
- ✓ Extended to multi-node

Gysi et.al, Domain-Specific Multi-Level IR Rewriting for GPU:
The Open Earth Compiler for GPU-accelerated Climate (2021), ACM TACO

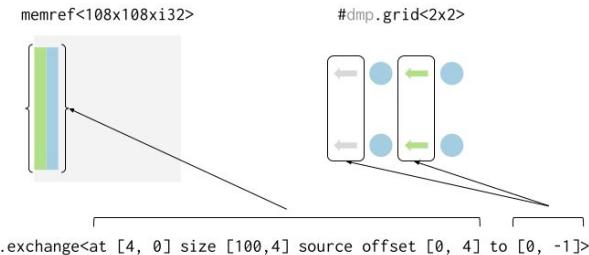
<https://github.com/spcl/open-earth-compiler/>



The ‘dmp’ dialect

```
1 dmp.swap(%data)
2 {
3     "grid" = #dmp.grid<2x2>,
4     "swaps" = [
5         #dmp.exchange<at [4, 0] size [100, 4]
6                         source offset [0, 4] to [0, -1]>,
7         #dmp.exchange<at [4, 104] size [100, 4]
8                         source offset [0, -4] to [0, 1]>
9     ]
10 } : (memref<108x108xf32>) -> ()
```

Listing 2. A high-level declarative expression of a data subsection exchange from some buffer.



- ✓ High-level halo exchanges
- ✓ Describe communication patterns
- ✓ Rectangular data subsections

The ‘mpi’ dialect

- ✓ Message-passing IR
- ✓ Lowered to MPI library calls
- ✓ Upstreamed to MLIR! (thanks to A. Lydike)

- Operations

- [mpi.comm_rank](#) (`mpi::CommRankOp`)
- [mpi.error_class](#) (`mpi::ErrorClassOp`)
- [mpi.finalize](#) (`mpi::FinalizeOp`)
- [mpi.init](#) (`mpi::InitOp`)
- [mpi.recv](#) (`mpi::RecvOp`)
- [mpi.retval_check](#) (`mpi::RetvalCheckOp`)
- [mpi.send](#) (`mpi::SendOp`)

- Attributes

- [MPI_ErrorClassEnumAttr](#)

- Types

- [RetvalType](#)

mpi.recv (`mpi::RecvOp`) ¶

Equivalent to `MPI_Recv(ptr, size, dtype, dest, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE)`

Syntax:

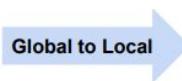
```
operation ::= `mpi.recv` `(` $ref `, ` $tag `, ` $rank `)` attr-dict `::` type($ref)
` , ` type($tag) ` , ` type($rank)(`->` type($retval))`?
```

`MPI_Recv` performs a blocking receive of `size` elements of type `dtype` from rank `dest`. The `tag` value and communicator enables the library to determine the matching of multiple sends and receives between the same ranks.

Lowering from `stencil` to `mpi`

```
Stencil level IR  
  
%source = stencil.load(%114) : (!field<[0,128]xf64>  
    -> !temp<?xf64>)  
%out = stencil.apply(%arg = %source : !temp<?xf64>)  
    -> !temp<?xf64> {  
    %l = stencil.access %arg[-1] : f64  
    %c = stencil.access %arg[0] : f64  
    %r = stencil.access %arg[1] : f64  
    // %v = %l + %r - 2.0 * %c  
    stencil.return %v : f64  
}  
  
stencil.store %out to %target([1]:[127])
```

1 127
Global Domain



DMP level IR

DMP to MPI

MPI level IR

```
%ref = builtin.unrealized_conversion_cast %114 :  
    !field<[0,64]xf64> to memref<64xf64>  
dmp.swap(%ref) {  
    "grid" = #dmp.grid<2>,  
    "swaps" = [  
        #dmp.exchange<at [0] size [1]  
            source offset [1] to [-1]>,  
        #dmp.exchange<at [64] size [1]  
            source offset [-1] to [1]>  
    ]  
} : (memref<64xf64>) -> ()  
%source = stencil.load(%114) ...  
%out = stencil.apply(%source) ...  
stencil.store %out to %target([1]:[64])
```

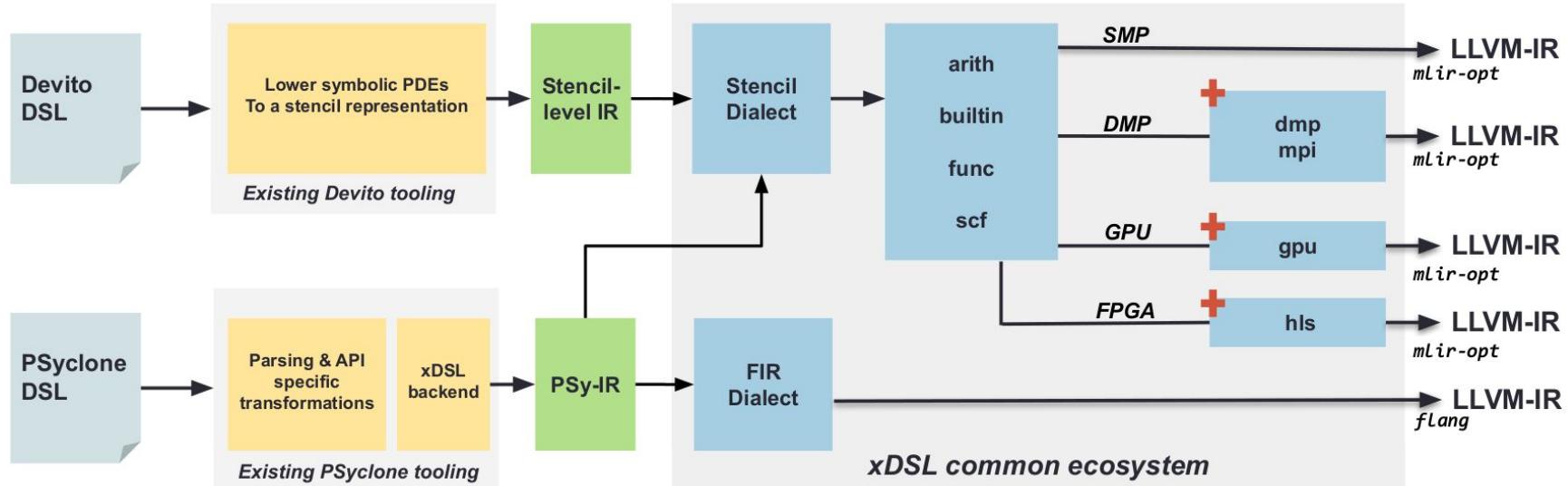


Local Domains with halo exchanges highlighted

```
%rank = mpi.comm_rank : i32  
// First swap communication calls  
%dest = arith.add %rank, %minus_one : i32  
%is_in_bounds = arith.cmpi sge, %dest, %zero  
scf.if %is_in_bounds {  
    %view = memref.subview %ref[0][1][1] : memref<64xf64>  
        to memref<1xf64>  
    // copy data into send buffer and set up communication  
    // (omitted for clarity)  
    mpi.isend %sptr, %count, %dtype, %dest, %tag, %send_req  
    mpi.irecv %rptr, %count, %dtype, %dest, %tag, %recv_req  
}  
// Second swap  
// ...  
mpi.waitall %requests, %four // synchronization barrier  
// First swap copy back  
scf.if %is_in_bounds {  
    %view = memref.subview %ref[1][1][1] : memref<64xf64>  
        to memref<1xf64>  
    memref.copy %recv_buffer_1 to %view  
}  
// Second swap copy back  
// Lowered stencil comes here
```

Data being operated on
Shape and halo information
Communication-related information

The shared compilation stack for DMP in stencil DSLs

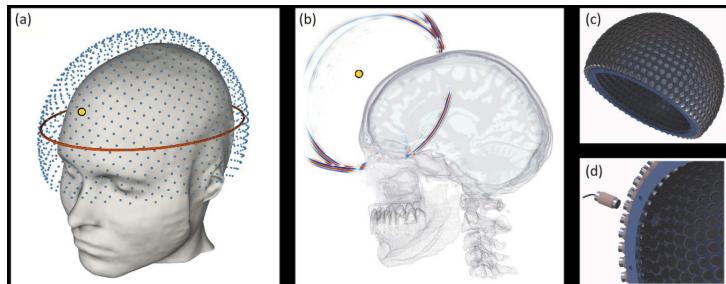
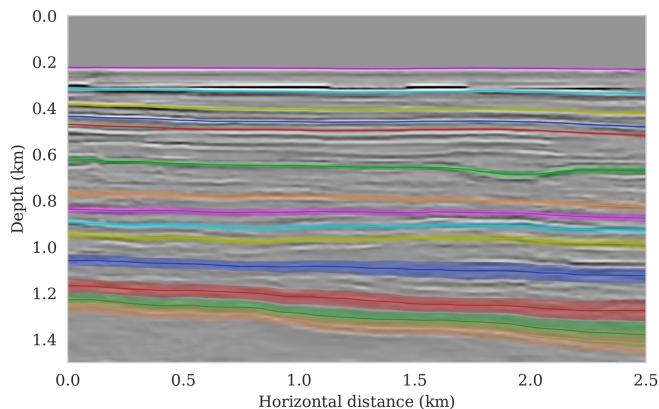


- ✓ Unlocked optimizations
- ✓ Unlocked multi-node CPU
- ✓ Unlocked other backends (FPGA, CUDA)

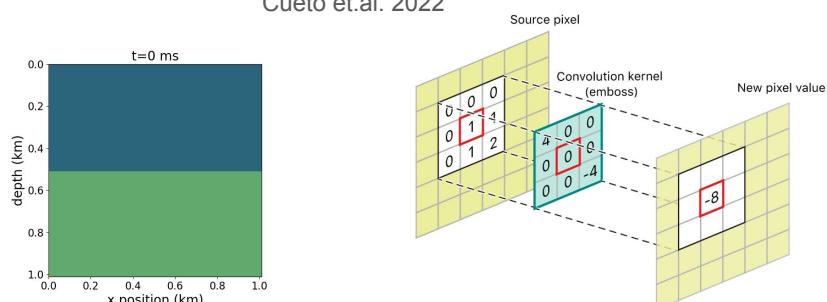
✓ Competitive or better performance with an order of 1000s LoC saved!

A deeper look: The example of Devito

- Devito is a DSL and compiler framework for FD-stencil computations
- Rich API for supporting way more than that!
- 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)

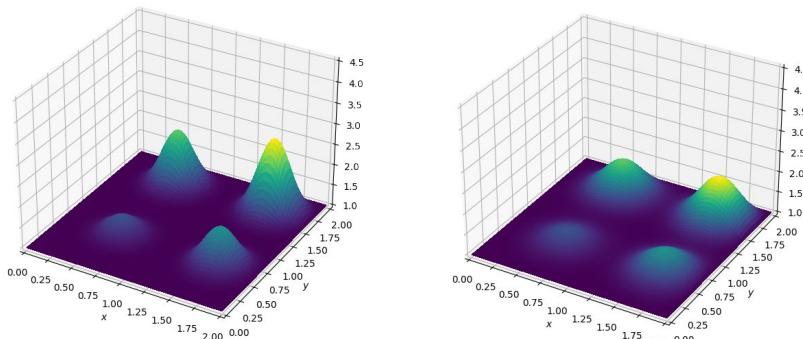
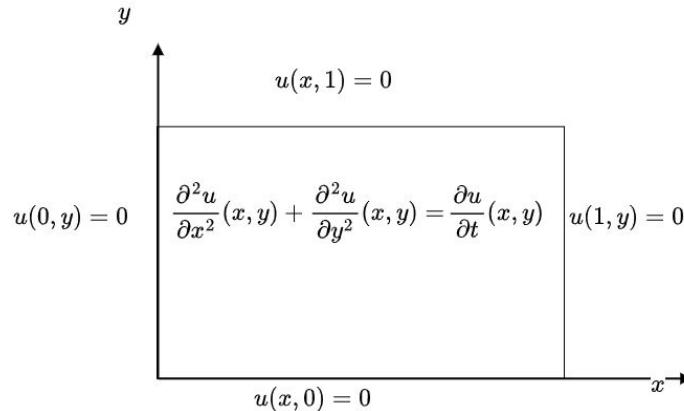


Cueto et.al. 2022



An example from textbook maths to Devito DSL

2D Heat diffusion modelling



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

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

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

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

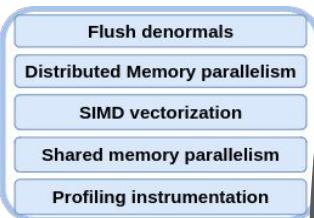
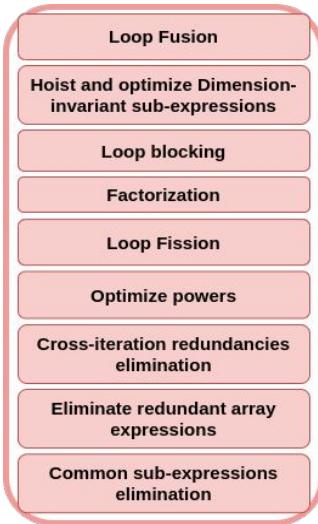
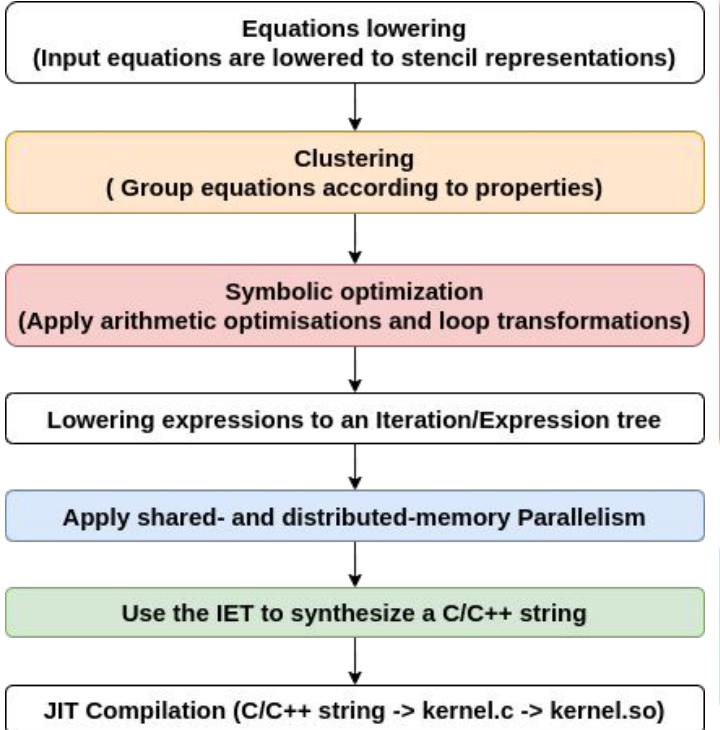
# Define boundary conditions
x, y = grid.dimensions
t = grid.stepping_dim

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

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

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

A deeper look: The example of Devito

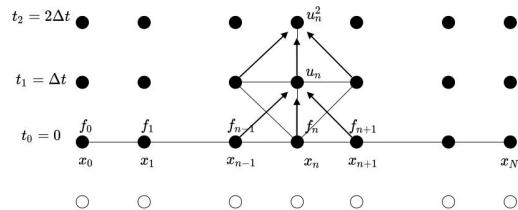


$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \quad 0 < x < L$$

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

Initial Conditions : $u(x, 0) = f(x)$

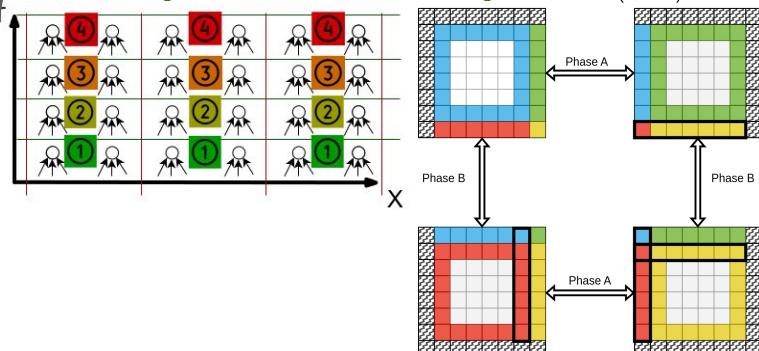
$$\frac{\partial u}{\partial t}(x, 0) = g(x)$$



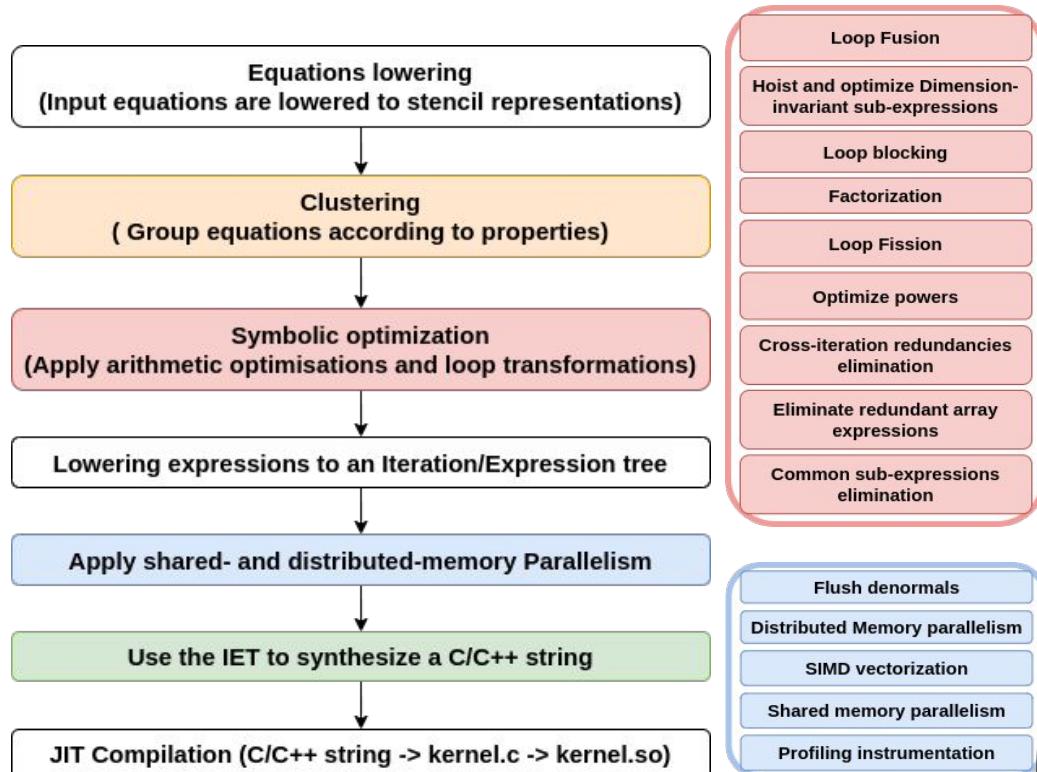
Groups of expressions, Cluster-level (Optimized)

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

Basic MPI mode (4 ranks)



A deeper look: The example of Devito

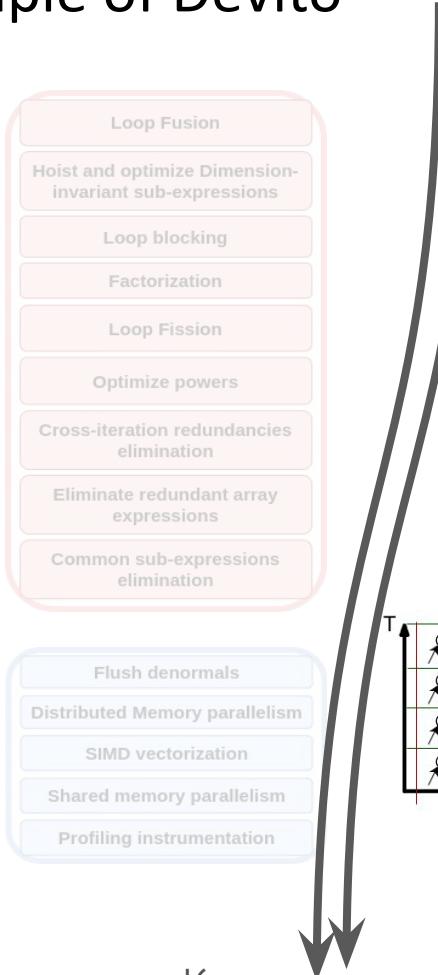
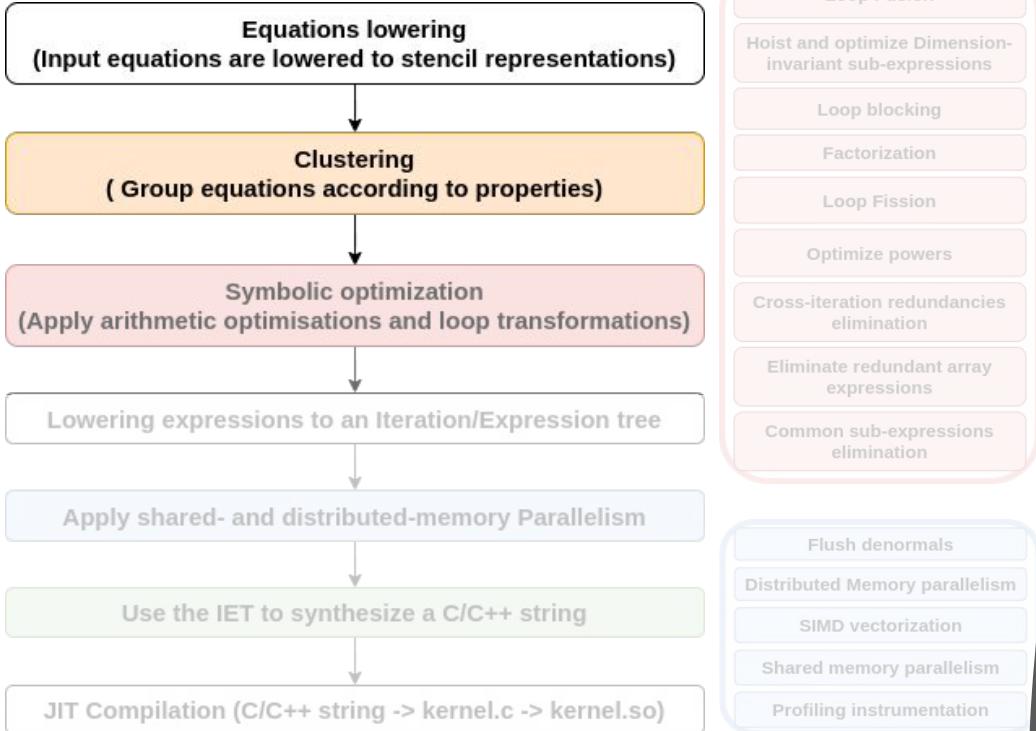


+ advanced combinations of them!
+ heuristics to tune them more!

Write once,
Run everywhere!

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

A deeper look: The example of Devito

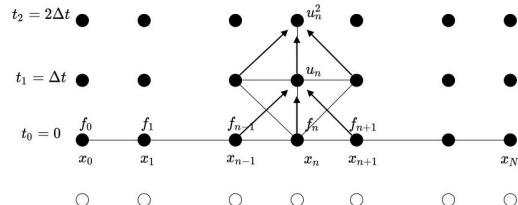


$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \quad 0 < x < L$$

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

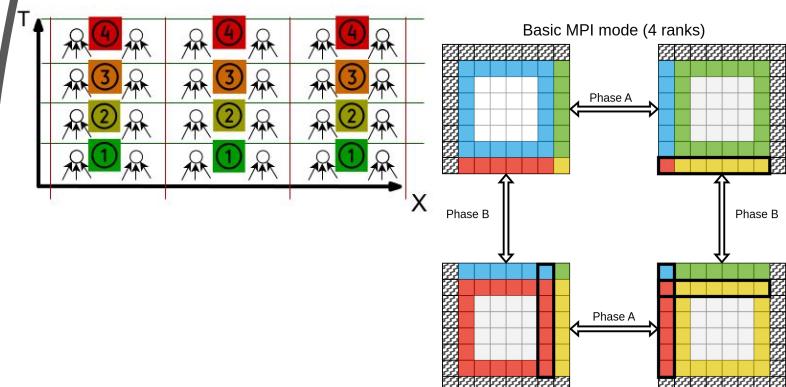
Initial Conditions : $u(x, 0) = f(x)$

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

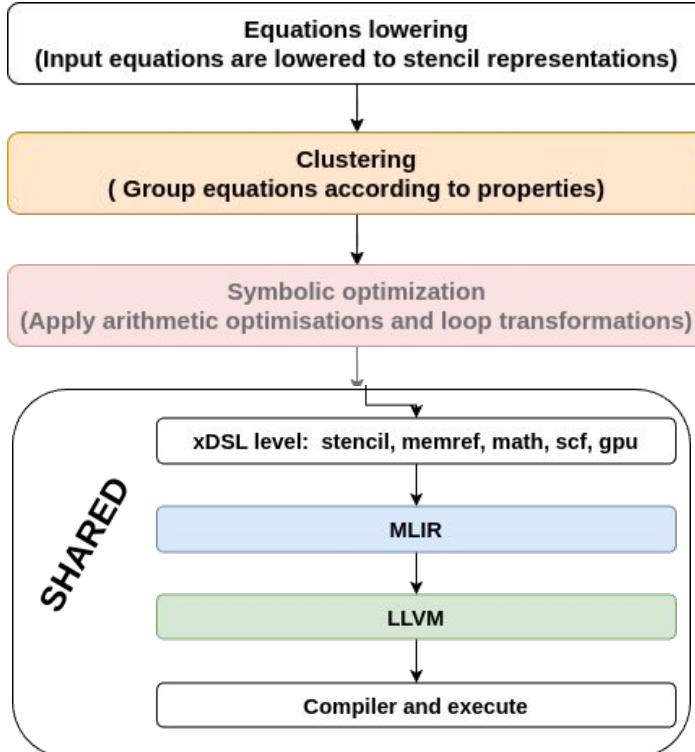


Groups of expressions, Cluster-level (Optimized)

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



A deeper look: The example of Devito

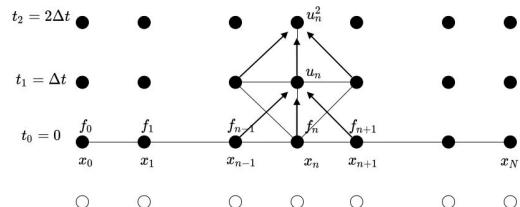


$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \quad 0 < x < L$$

Boundary Conditions : $u(0, t) = 0$; $u(L, t) = 0$

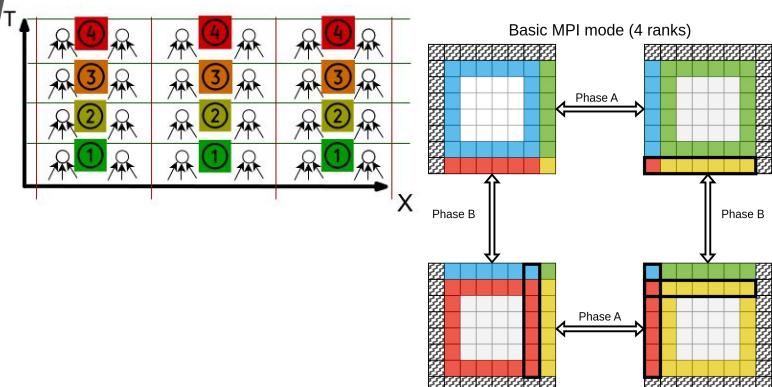
Initial Conditions : $u(x, 0) = f(x)$

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



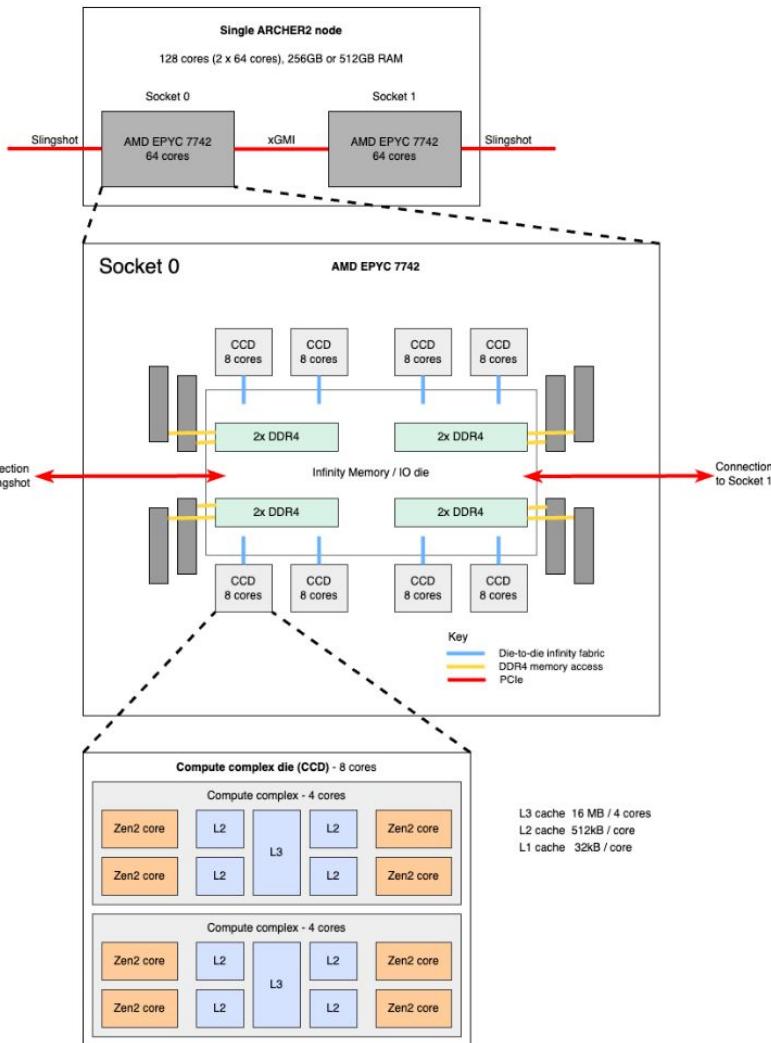
Groups of expressions, Cluster-level (Optimized)

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

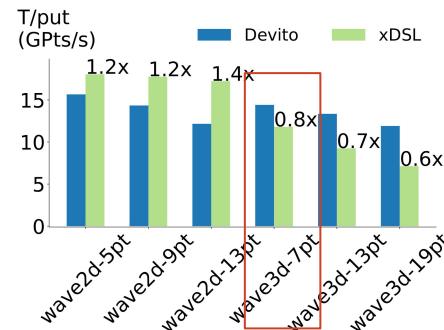
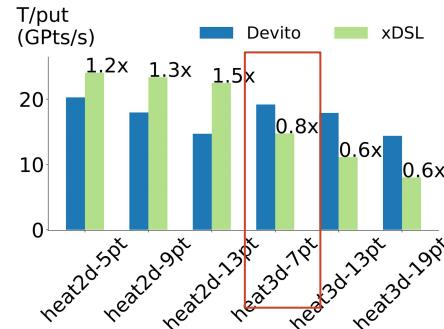


hardware

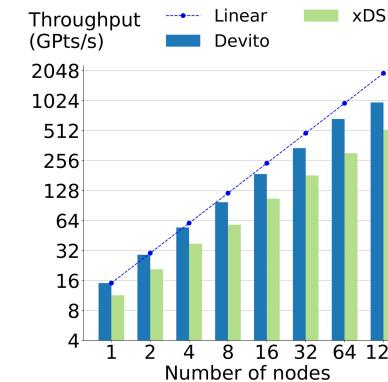
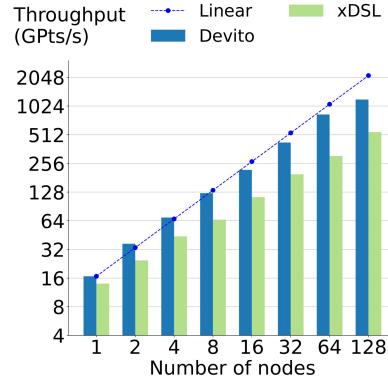
Component	Details
Processor	2x AMD Zen2 (Rome) EPYC 7742, 64-core, 2.25 Ghz
Cores per node	128
NUMA structure	8 NUMA regions per node (16 cores per NUMA region)
Memory per node	256 GB (standard), 512 GB (high memory)
Memory per core	2 GB (standard), 4 GB (high memory)
L1 cache	32 kB/core
L2 cache	512 kB/core
L3 cache	16 MB/4-cores
Vector support	AVX2
Network connection	2x 100 Gb/s injection ports per node



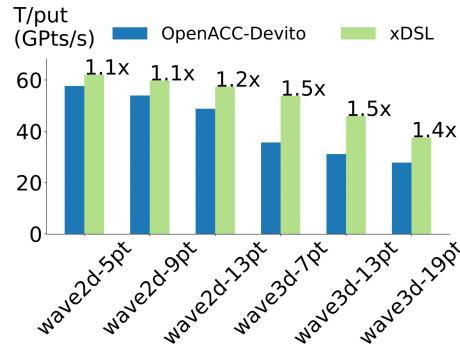
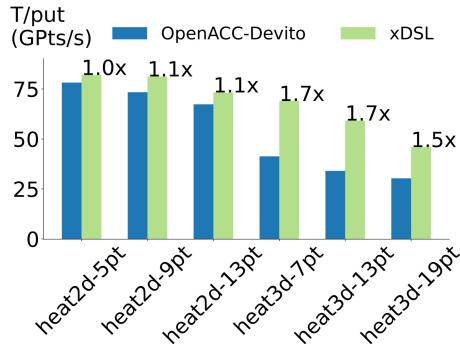
Performance evaluation: vs Devito Optimized



Single-node AMD EPYC 7742,
8 MPI ranks x 16 OpenMP threads,
 16384^2 (2D) and 1024^3 (3D)



Heat (top) and wave (bottom), 3D-7pt, multi-node strong scaling up to 128 nodes, total of 16384 cores.

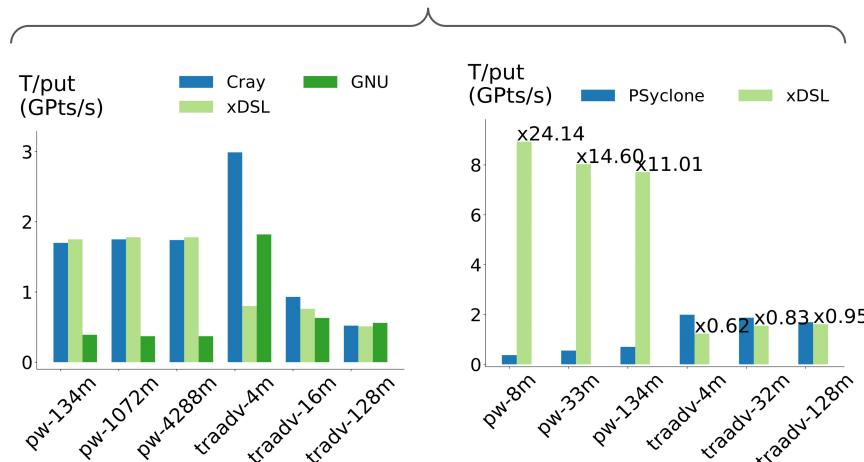


xDSL adds support for CUDA, outperforming Devito's OSS support for OpenACC, running on V100-SXM2-16GB (Volta).

Performance evaluation: PSyclone

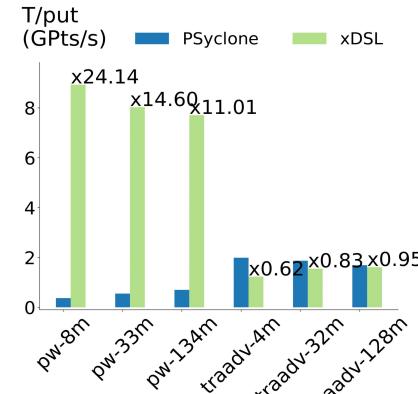


Piacsek and Williams (PW) advection and NEMO tracer advection (traadv) kernels



Single-node AMD EPYC 7742 throughput, PSyclone target code compiled with Cray and GNU compilers against xDSL-PSyclone.

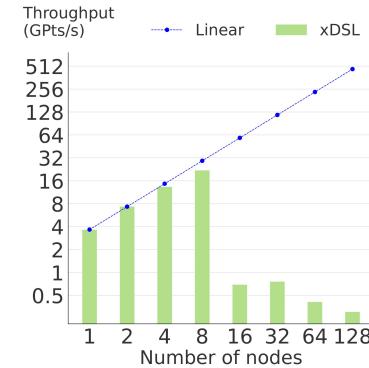
xDSL-PSyclone code matches Cray code and significantly outperforms GNU code for PW advection.



Single-node Cirrus NVIDIA Tesla V100-SXM2-16GB throughput, PSyclone NVIDIA GPU code against xDSL-PSyclone GPU code.

xDSL-PSyclone significantly outperforms PSyclone NVIDIA for PW advection due to data allocation approach (explicit device memory allocation for xDSL-PSyclone).

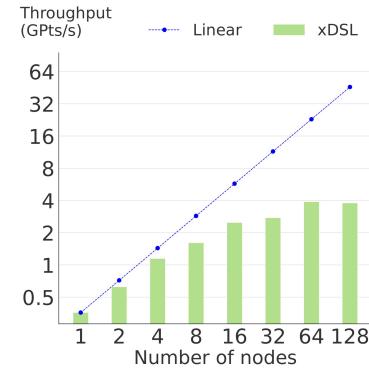
PW advection kernel



Multi-node strong scaling of problem size [256,256,128], scaling up to 128 nodes, total of 16384 cores.

Suffers scaling effects at 8 nodes due to small global problem size.

traadv kernel



Multi-node strong scaling of problem size [512,512,128], scaling up to 128 nodes, total of 16384 cores.

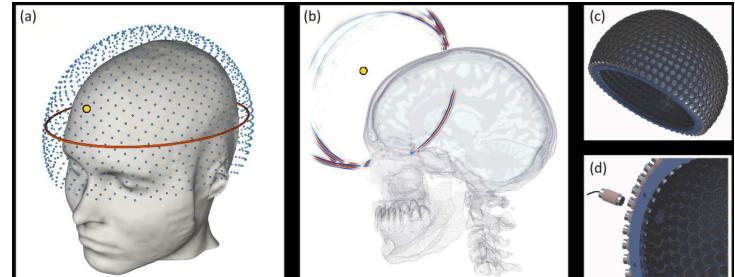
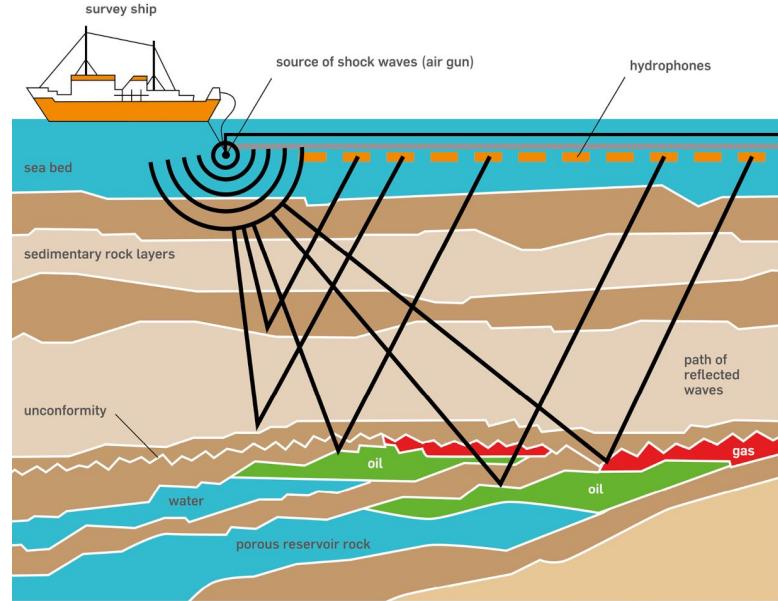
2D decomposition strategy limits strong scaling.

Roadmap:

- **“Beta” Support for Elastic wave propagation, more complex kernel**

J. Virieux (1986). "P-SV wave propagation in heterogeneous media: Velocity-stress finite-difference method." GEOPHYSICS, 51(4), 889-901.
<https://doi.org/10.1190/1.1442147>

- **Building the blocks for Full Waveform Inversion:**
 - Source injection/Receiver interpolation
- Support for some exotic accelerators (CS-2)
- Improve performance analysis
 - Bridge after Devito optimizations
- Use more value semantics



Summary, limitations and future work

★ A set of HPC-specific compilation components:

- “dmp” dialect for automated domain decomposition
- “mpi” dialect for message passing
(✓ Upstreamed to MLIR !)

● Limitations and Future work

- Performance analysis
- More computation/communication schemes to be evaluated

★ A shared compilation stack for **Devito** and **Psyclone**:

- Based on SSA concepts
- Utilizing xDSL/MLIR

► Future work

- Extend ‘dmp’ and ‘mpi’ dialects
- Support more wave propagators (e.g. isotropic elastic/ anisotropic TTI)
- Building blocks for real-life simulation benchmarks
- Exotic accelerators

★ Performance evaluation shows:

- ✓ Highly competitive or improved solvers
- ✓ Support for new targets, and hardware



Reach out!

- Find us on <https://xDSL.zulipchat.com/>

Building abstractions is great!
Sharing them is better!

Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) grants **EP/W007789/1** and **EP/W007940/1**.

The authors thank the xDSL, Devito, and PsyClone communities for their comments and discussions.

Appendix

The **Devito** compiler automatically applies lots of optimisations

The **XYZ** compiler automatically applies lots of optimisations

The **XYZ** compiler automatically applies lots of **SILOED** optimisations

Do not reinvent (?) the wheel, but improve it!



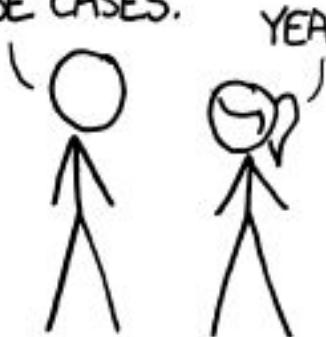
Disclaimer

- I do not claim to be an MLIR expert myself, but rather an application developer who sees its strong potential and wants to access its treasure of benefits.

HOW STANDARDS PROLIFERATE: (SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)

SITUATION:
THERE ARE
14 COMPETING
STANDARDS.

14?! RIDICULOUS!
WE NEED TO DEVELOP
ONE UNIVERSAL STANDARD
THAT COVERS EVERYONE'S
USE CASES.



YEAH!

SOON:

SITUATION:
THERE ARE
15 COMPETING
STANDARDS.

MPICH support

Motivated by the distribution on our target platform (ARCHER2), we support the \emph{mpich} implementation of the MPI standard. To make use of MPI, it is usually required to include the implementations C header file, a notion not supported by MLIR.\@ Instead, we extract magic values from our library's header file and substitute them for e.g.\ datatype constants during the lowering process. This makes our provided MPI lowering specific to the \emph{mpich} library, but this strategy can be adapted to other MPI libraries like OpenMPI.\@ Furthermore, there is an ongoing effort to provide one standard MPI ABI, greatly simplifying this problem \citet{hammond2023mpi}.

Contributions: A shared compilation stack for DMP in stencil DSLs

★ A set of HPC-specific compilation components:

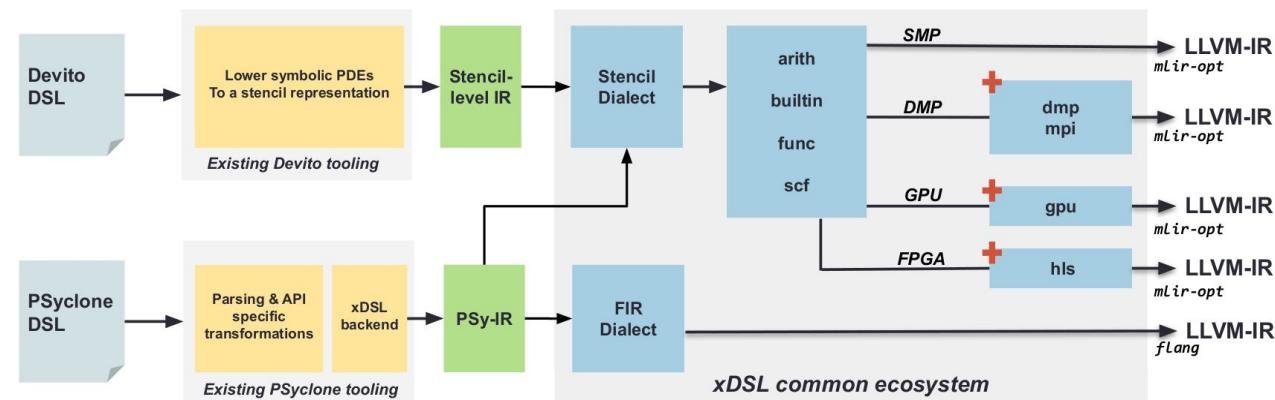
- “dmp” dialect for automated domain decomposition
- “mpi” dialect for message passing (□ Upstreamed to MLIR !)

★ A shared compilation stack for Devito and Psyclone:

- Based on SSA concepts
- Utilizing  /MLIR

★ Performance evaluation shows:

- Highly competitive DMP solvers
- Support for new targets
- Support for new hardware
- Improved performance for a number of cases



-> Full-paper presentation at the technical sessions: time TBA
-> Poster at Sunday 28th of April (18:00-21:00)

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

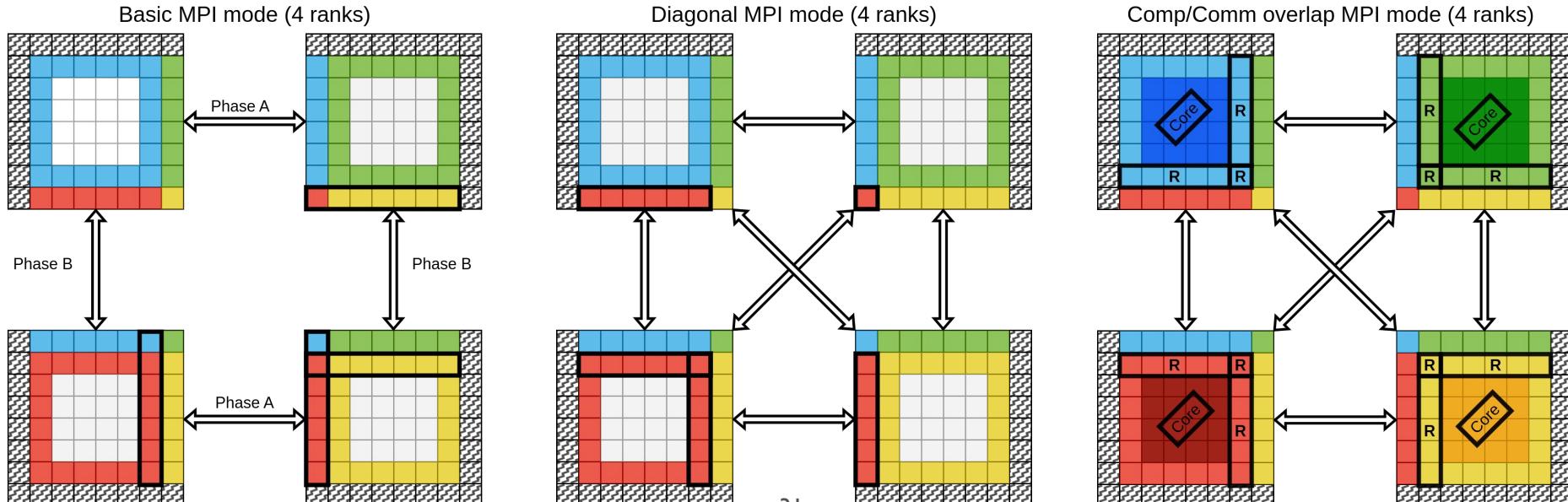
```
void kernel(...) {
    ...
    <impenetrable code with
    aggressive performance
    optimizations, manually
    applied, full-time human
    resources, less
    reproducibility, debugging
    nightmares>
    ...
}
```

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

Write once, run everywhere!

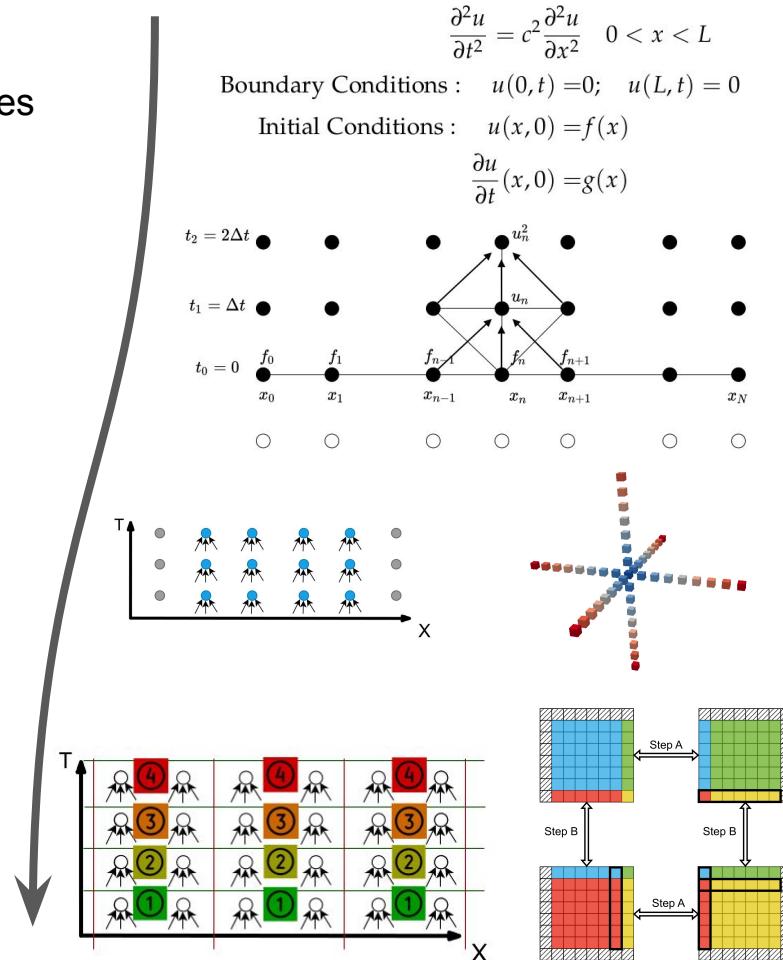
Devito offers automated MPI-openmp code generation, taking advantage of several optimised communication/computation patterns

User only has to use : “ `DEVITO_MPI=<mode> mpirun -n 2 python my_devitodescript.py` ”



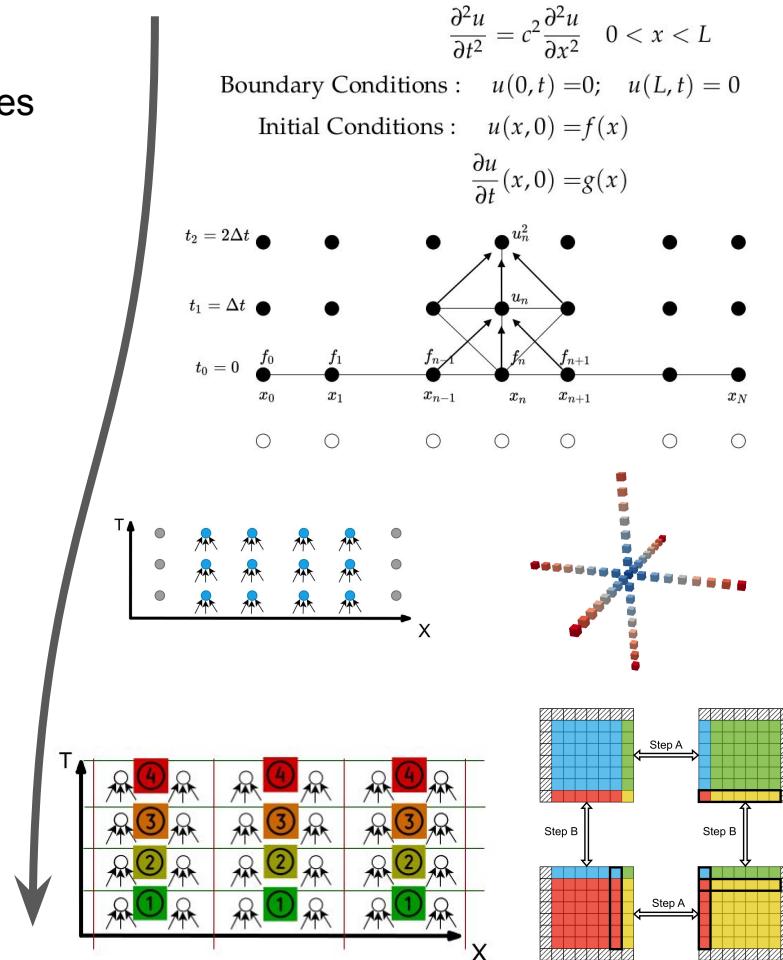
Our motivation:

- A common ecosystem for building Domain Specific Languages (DSLs)
- Lower the barrier to entry in developing DSLs and aims to improve the end-user experience by affording a mature and well supported ecosystem based upon MLIR and LLVM.
- People keep reinventing the wheel (almost!)
- Better abstractions, more accessible HPC
- FD-stencil computations: one of the seven dwarfs of HPC



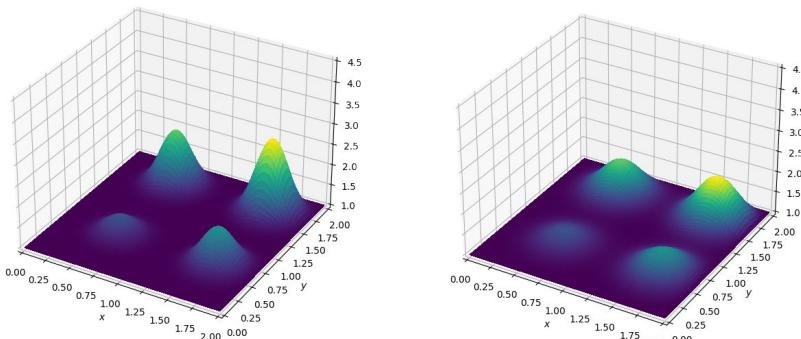
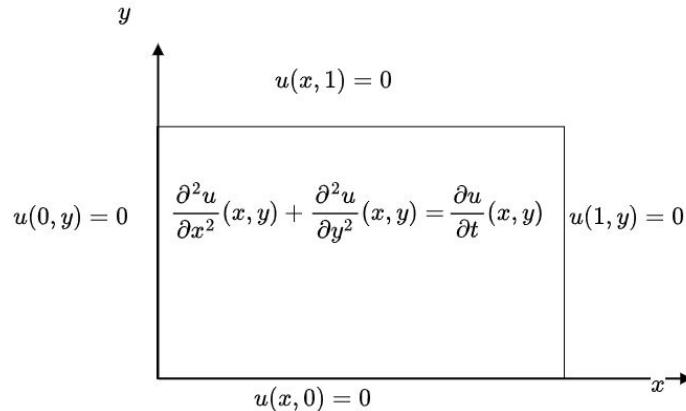
Our motivation:

- A common ecosystem for building Domain Specific Languages (DSLs)
- Lower the barrier to entry in developing DSLs and aims to improve the end-user experience by affording a mature and well supported ecosystem based upon MLIR and LLVM.
- People keep reinventing the wheel (almost!)
- Better abstractions, more accessible HPC
- FD-stencil computations: one of the seven dwarfs of HPC



An example from textbook maths to via Devito DSL

2D Heat diffusion modelling



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

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

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

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

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

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

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

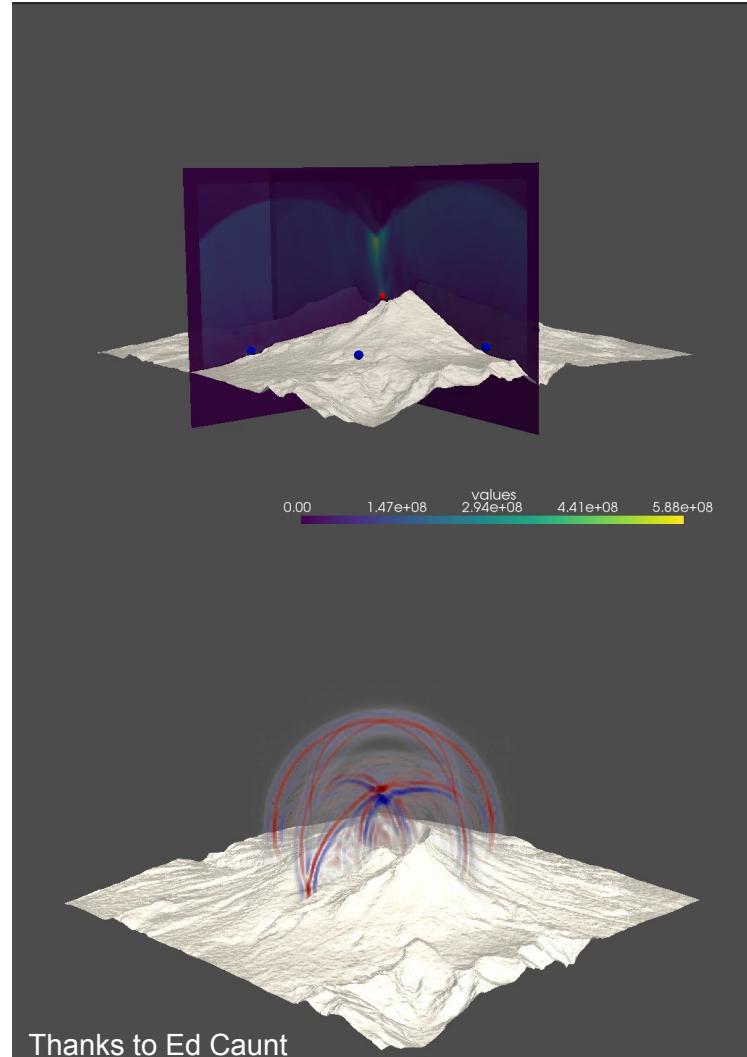
# Compute for 3 timesteps
op.apply(time_M=3, dt=0.1)
```

Devito's API is much richer though...

- Any PDE simulations
- Boundary conditions
- Sparse off-grid operations (interpolations)
- Subdomains
- Immersed boundaries (WIP)

Examples available with:

- *CFD* (convection/diffusion/cavity flow/shallow waters (tsunami) /Darcy flow)
- *Wave propagators* ((Visco-)Acoustic/Elastic, TTI)
- *Seismic/Medical Imaging* (FWI/RTM)
- *Finance*



Thanks to Ed Caunt

Math-related optimisations -- Reducing OI/AI of stencil kernels

- **Isotropic Acoustic**

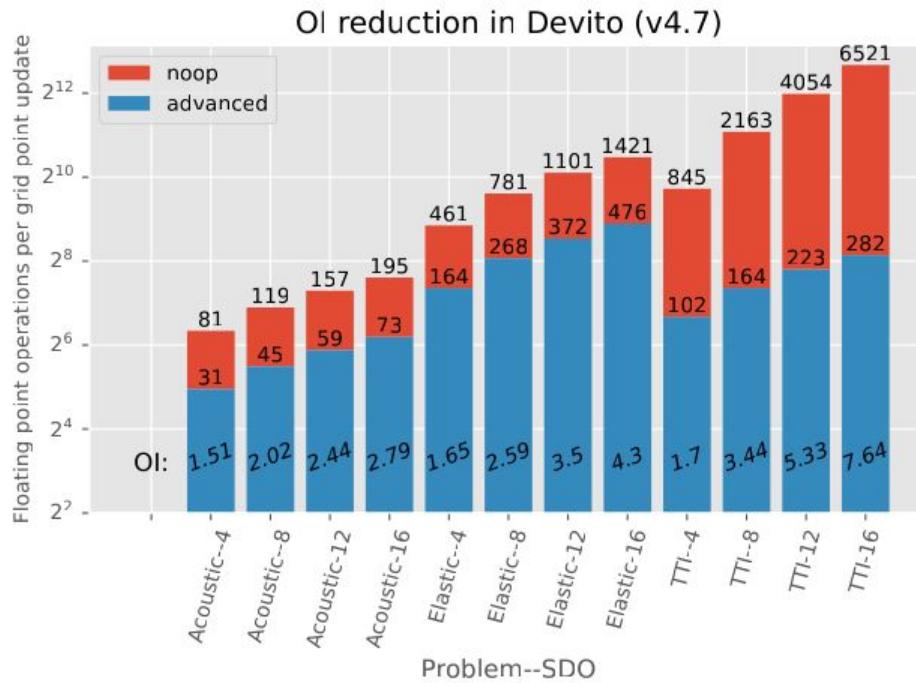
Generally known, single scalar PDE,
laplacian like, low cost

- **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,
Zhang-Louboutin variation)**

Industrial applications, rotated
laplacian, coupled system of two
scalar PDEs, several variations based
on variable or constant density



Not the typical memory-bw bound stencils!



Significantly reduced operational intensity!