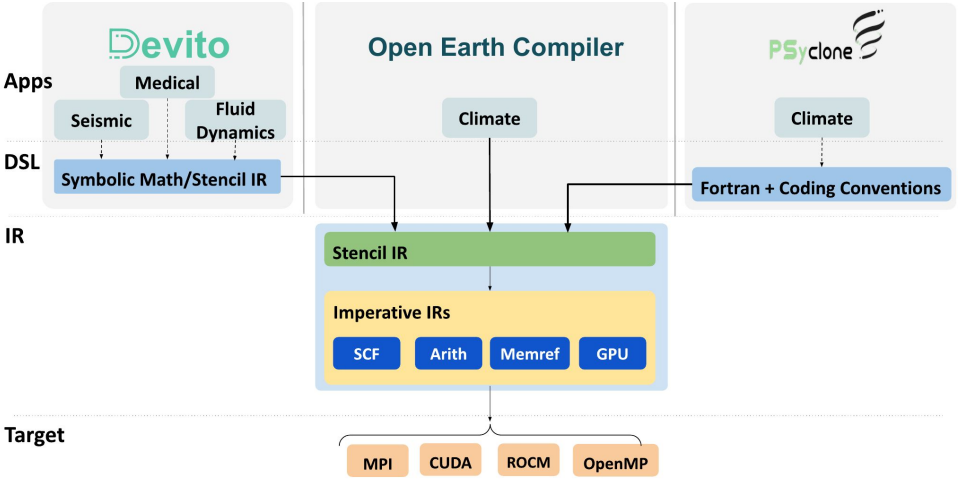
A shared compilation stack for distributed-memory parallelism in stencil DSLs

G. Bisbas*[1], A. Lydike*[2], E. Bauer*[2], N. Brown*[2],
M. Fehr[2], L. Mitchell, G. Rodriguez-Canal[2], M. Jamieson[2],
P. H.J. Kelly[1], M. Steuwer[3], T. Grosser[4]

***authors contributed equally**

[1]*Imperial College London, UK*
[2]*The University of Edinburgh, UK*
[3]*Technische Universität Berlin, Germany*
[4]*University of Cambridge, UK*



The problem: Monolithic Domain-specific languages

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

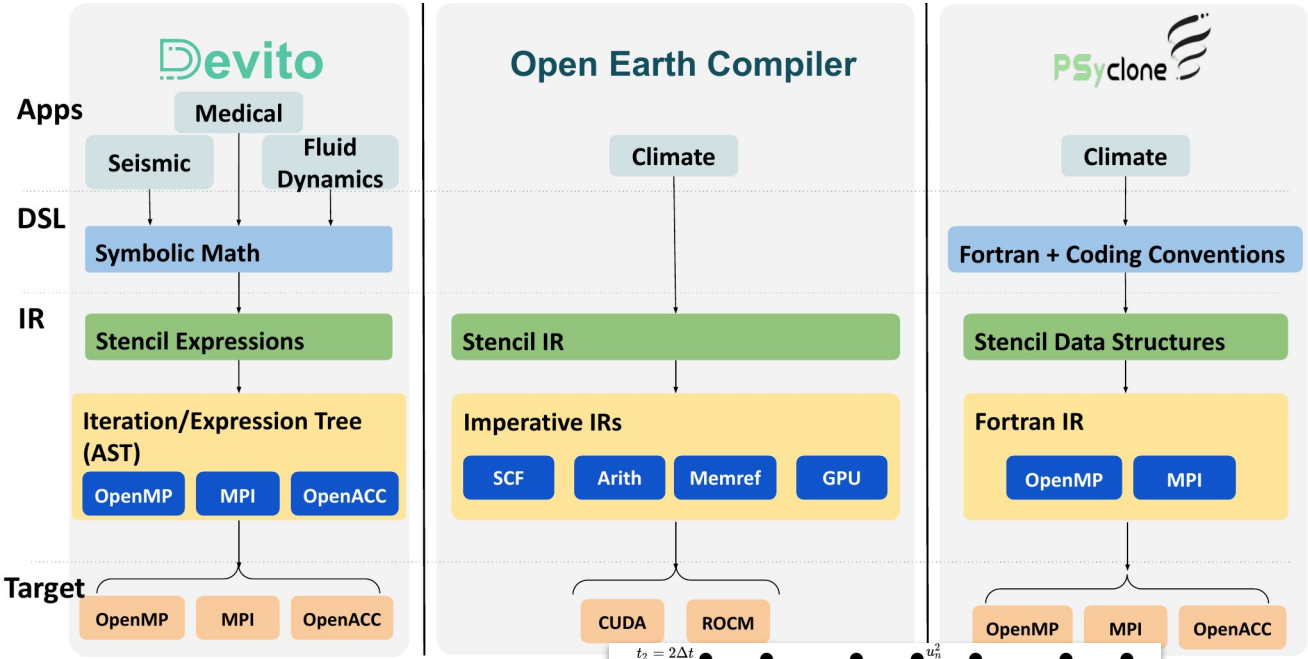
✔ Performance
✔ Productivity
✔ Portability
but…

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:

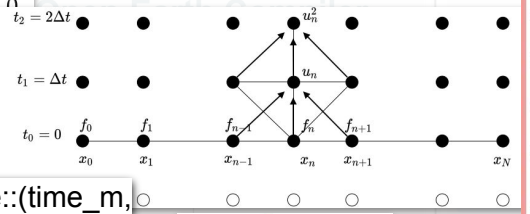
- ✘ Disjoint communities
- ✘ Lack of knowledge transfer

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

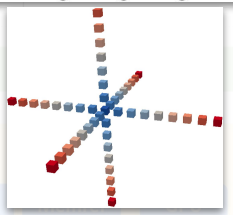
Boundary Conditions : $u(0,t) = 0; \quad u(L,t) = 0$

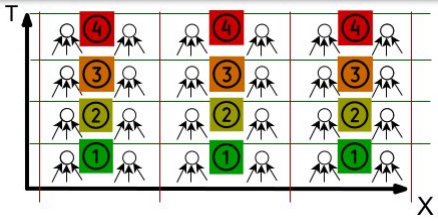
Initial Conditions : $u(x,0) = f(x)$

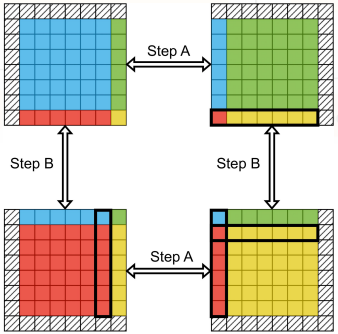
$$\frac{\partial u}{\partial t}(x,0) = g(x)$$




```
<[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>
```





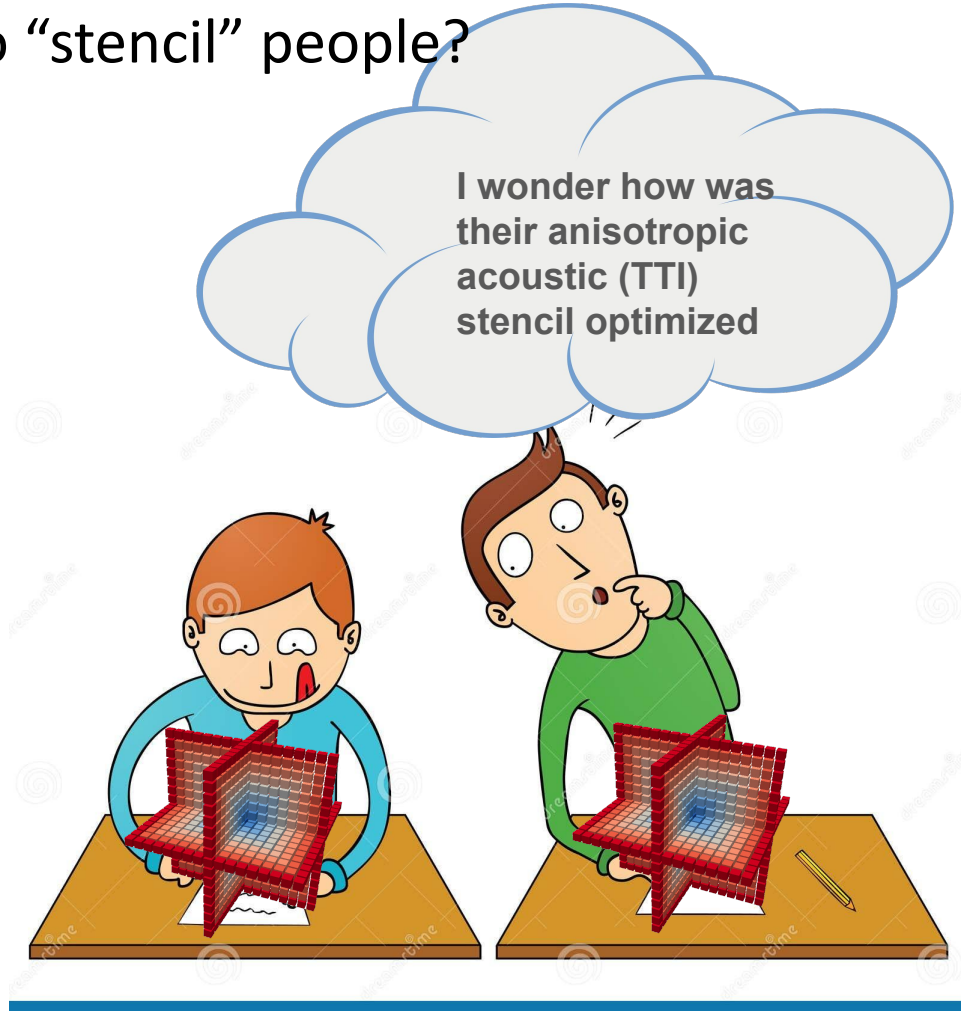


- Loop Fusion
- Hoist and optimize Dimension-invariant sub-expressions
- Loop blocking
- Factorization
- Loop Fission
- Optimize powers
- Cross-iteration redundancies elimination
- Eliminate redundant array expressions
- Common sub-expressions elimination

- Flush denormals
- Distributed Memory parallelism
- SIMD vectorization
- Shared memory parallelism
- Profiling instrumentation

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

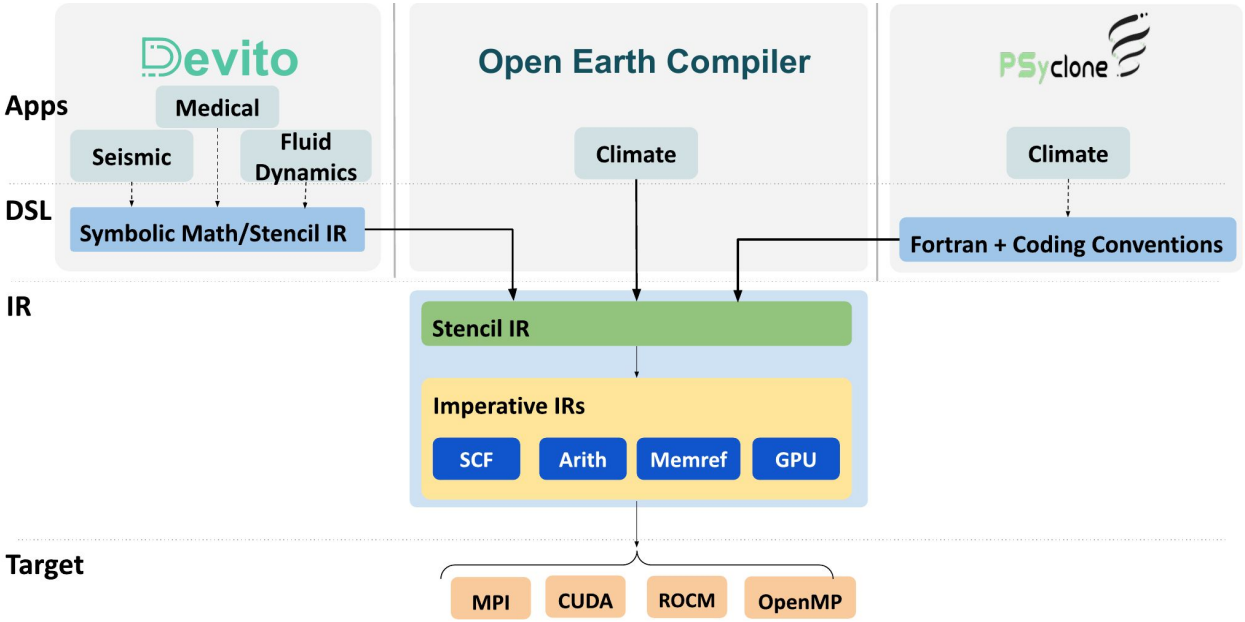
- What **tiling strategy** are you using for cache efficiency?
- Are your OpenMP parallel loops using dynamic **scheduling**?
- What is your **MPI strategy** for running on X machine
- What is your **process pinning** strategy?
- Are you using **temporal blocking**? How?
- Yes, but is this good for my “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:

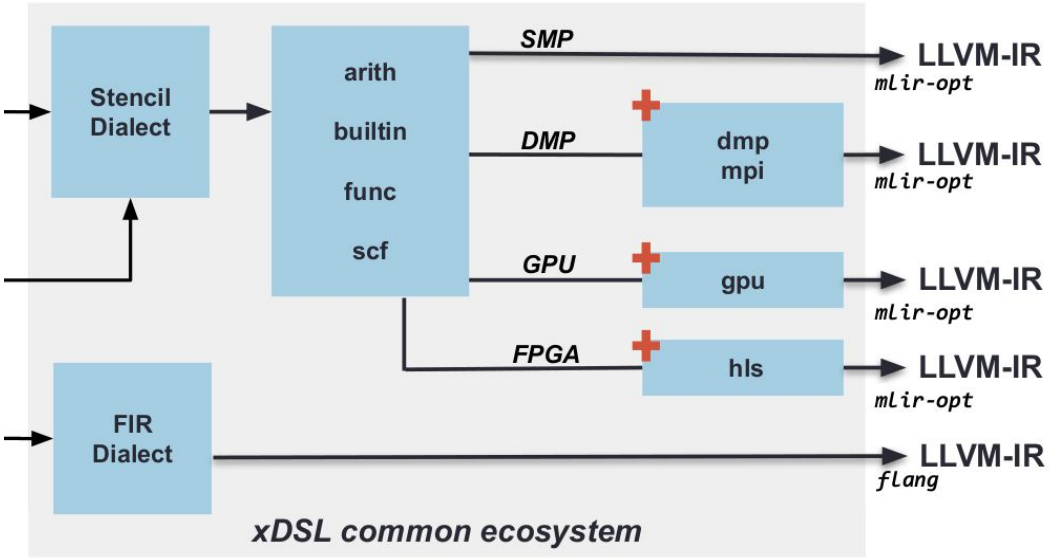
⭐**A set of HPC-specific compilation components focusing on FD-stencil computations:**

- “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

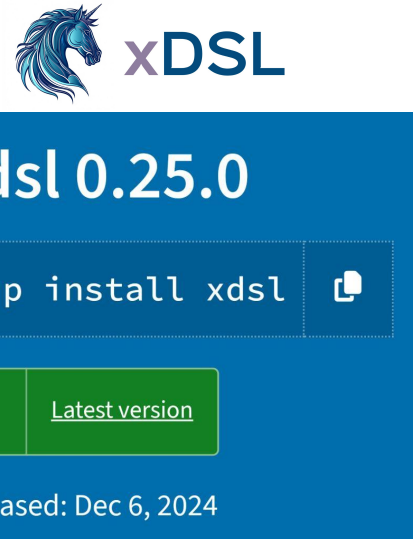
xDSL: A Python-native SSA Compiler Framework

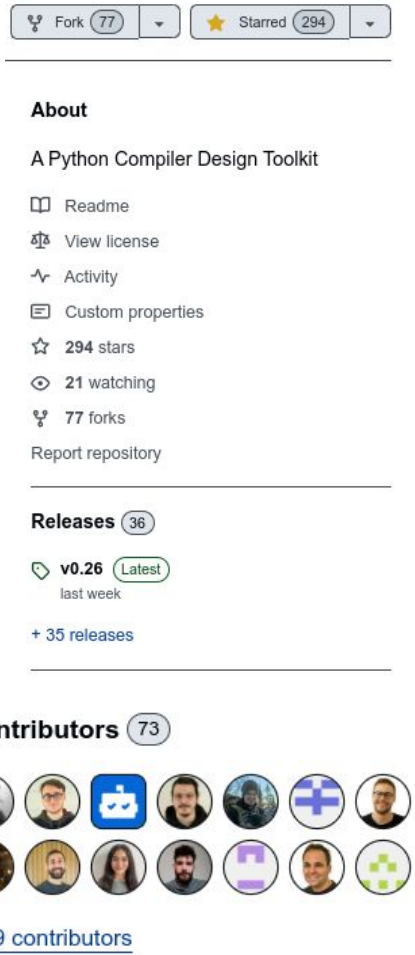
- ✔ SSA-based IRs
- ✔ SSA + regions concept

- ✔ Mix predefined IRs
- ✔ Add custom IRs

- ✔ Connect with MLIR/LLVM
- ✔ Benefit from Python's productivity

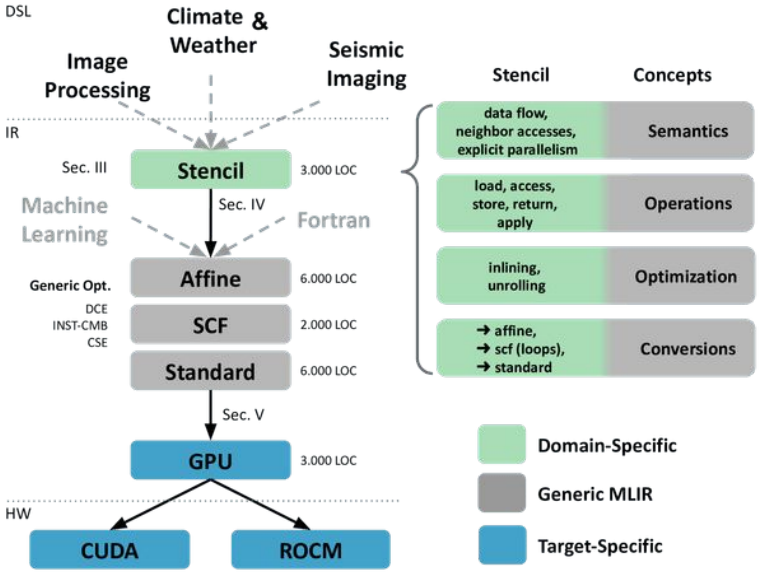
- ✔ Open-source/CI/CD/codecov
- ✔ Active contributor community
- ✔ Join us on https://xdsl.zulipchat.com/





M. Fehr, M. Weber, C. Ulmann, A. Lopoukhine, M. Lücke, T. Degioanni, M. Steuwer, T. Grosser, Sidekick compilation with xDSL. (2023), arXiv:2311.07422

The Open Earth Compiler: the ‘stencil’ dialect




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

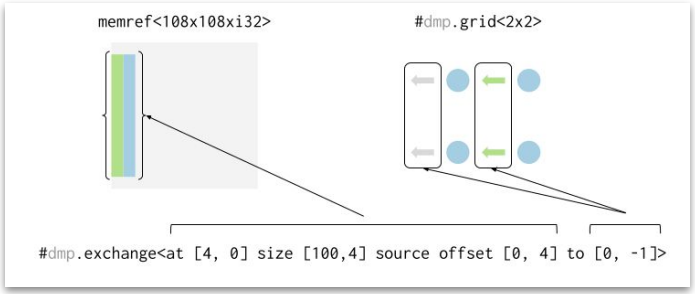
Gysi et.al, Domain-Specific Multi-Level IR Rewriting for GPU:
The Open Earth Compiler for GPU-accelerated Climate (2021), ACM TACO

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

The ‘dmp’ dialect

```
dmp.swap(%data)
{
  "grid" = #dmp.grid<2x2>,
  "swaps" = [
    #dmp.exchange<at [4, 0] size [100, 4]
                  source offset [0, 4] to [0, -1]>,
    #dmp.exchange<at [4, 104] size [100, 4]
                  source offset [0, -4]  to [0, 1]>
  ]
} : (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)^)?
```

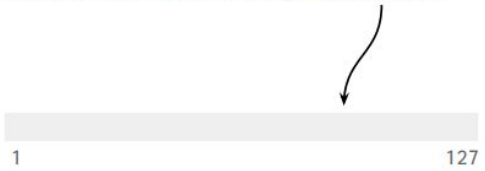
MPI_Recv performs a blocking receive of `size` elements of type `dtype` from rank `dest`. The `tag` value and communicator enables the library to determine the matching of multiple sends and receives between the same ranks.

Lowering from `stencil` to `mpi`

Stencil level IR → Global to Local → DMP level IR → DMP to MPI → MPI level IR

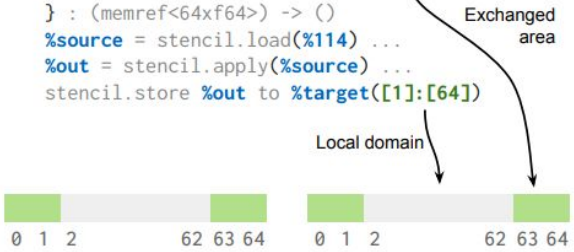
```
%source = stencil.load(%114) : (!field<[0,128]xf64>)
                               -> !temp<?xf64>
%out = stencil.apply(%arg = %source : !temp<?xf64>)
                  -> !temp<?xf64> {
  %l = stencil.access %arg[-1] : f64
  %c = stencil.access %arg[0] : f64
  %r = stencil.access %arg[1] : f64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

stencil.store %out to %target([1]:[127])
```



Global Domain

```
%ref = builtin.unrealized_conversion_cast %114 :
    !field<[0,64]xf64> to memref<64xf62>
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])
```

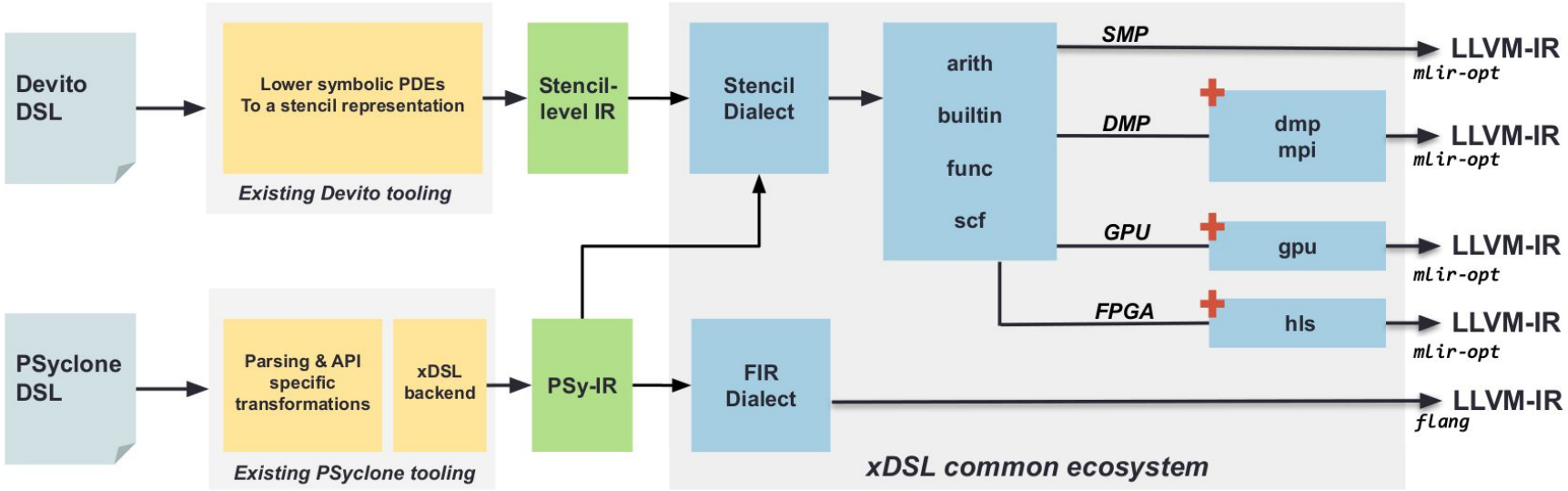


Local Domains with halo exchanges highlighted

```
%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



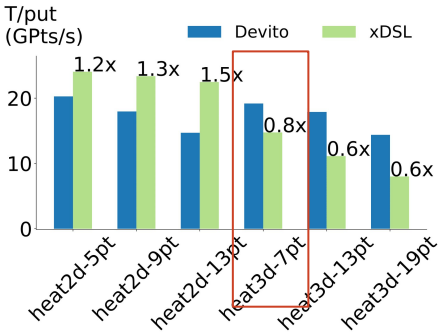
✔ Unlocked optimizations

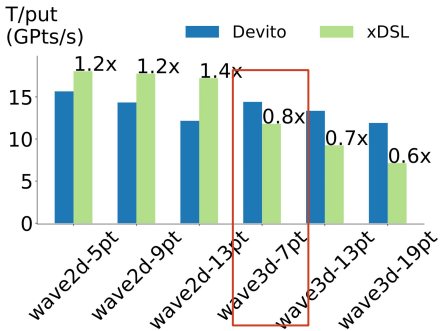
✔ Unlocked multi-node CPU

✔ Unlocked other backends (FPGA, CUDA)

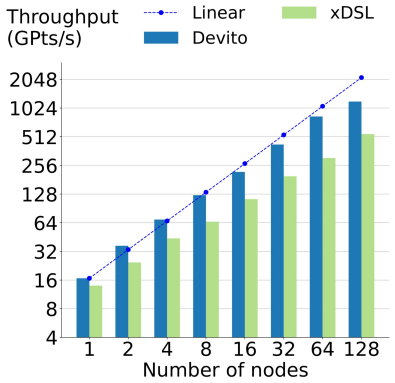
✔ Competitive or better performance with an order of 1000s LoC saved!

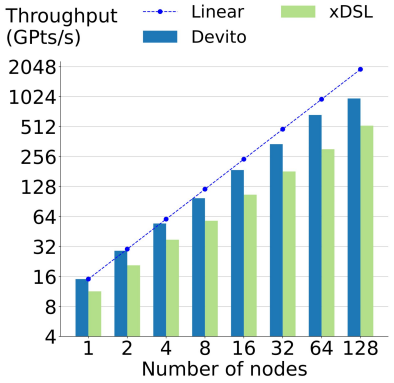
Performance evaluation: vs Devito Optimized



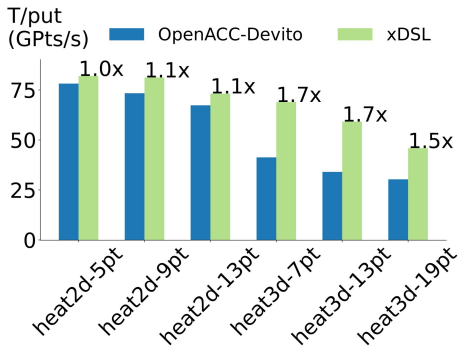


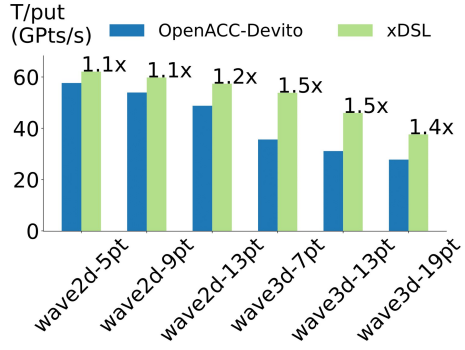
Single-node AMD EPYC 7742,
8 MPI ranks x 16 OpenMP threads,
16384^2 (2D) and 1024^3 (3D)





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

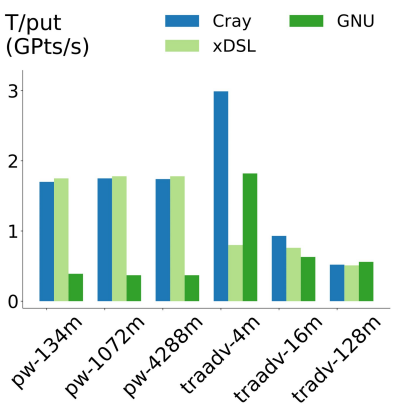


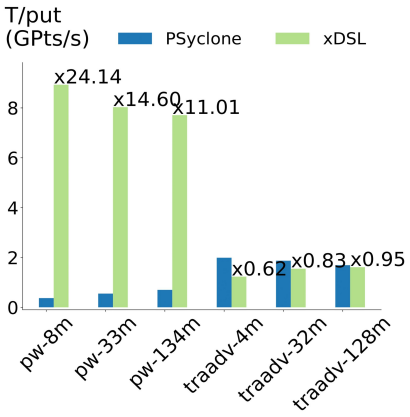


xDSL adds support for CUDA, outperforming
Devito's OSS support for OpenACC, running on
V100-SXM2-16GB (Volta).

Performance evaluation: PSyclone

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





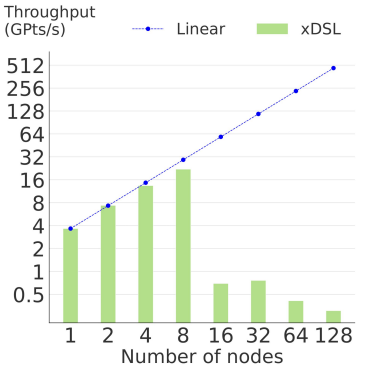
Single-node AMD EPYC 7742 throughput, PSyclone target code compiled with Cray and GNU compilers against xDSL-PSyclone.

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).

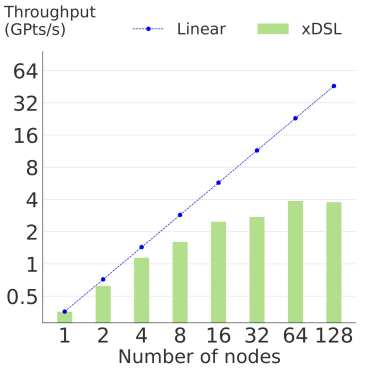
PW advection kernel



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.

Limitations and possible extensions:

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

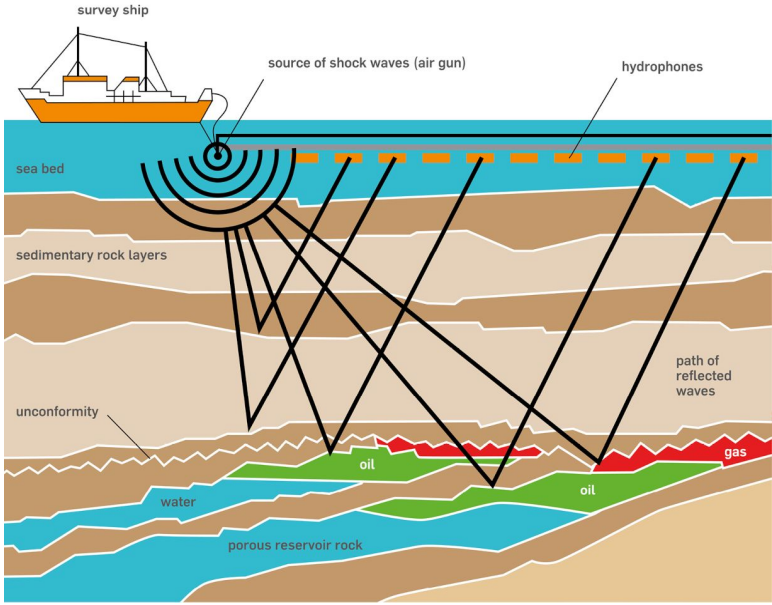
J. Virieux (1986). ”P-SV wave propagation in heterogeneous media: Velocity-stress finite-difference method.” GEOPHYSICS, 51(4), 889-901. https://doi.org/10.1190/1.1442147

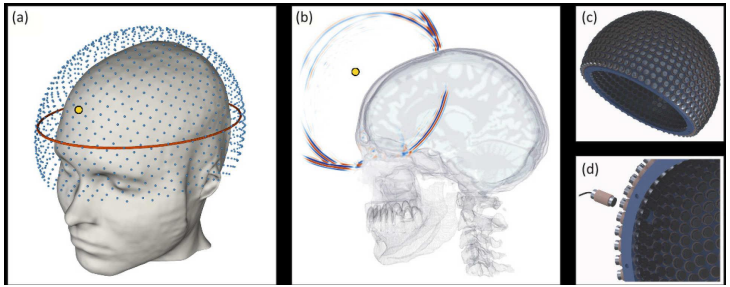
- **Building the blocks for Full Waveform Inversion:**
  - **Source injection/Receiver interpolation**

- Support for some exotic accelerators (CS-2)

- Improve performance analysis
  - Bridge after Devito optimizations

- Use more value semantics





Cueto et.al. 2022

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 !)

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

- Based on SSA concepts
- Utilizing xDSL/MLIR

⭐**Performance evaluation shows:**

✔Highly competitive or improved solvers

✔Support for new targets, and hardware

Building abstractions is great!
Sharing them is better!

🟠 **Limitations and Future work**

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

⏩ **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

🙋 **Reach out!**

- Find us on https://xdsl.zulipchat.com/
- Add some resources?

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.