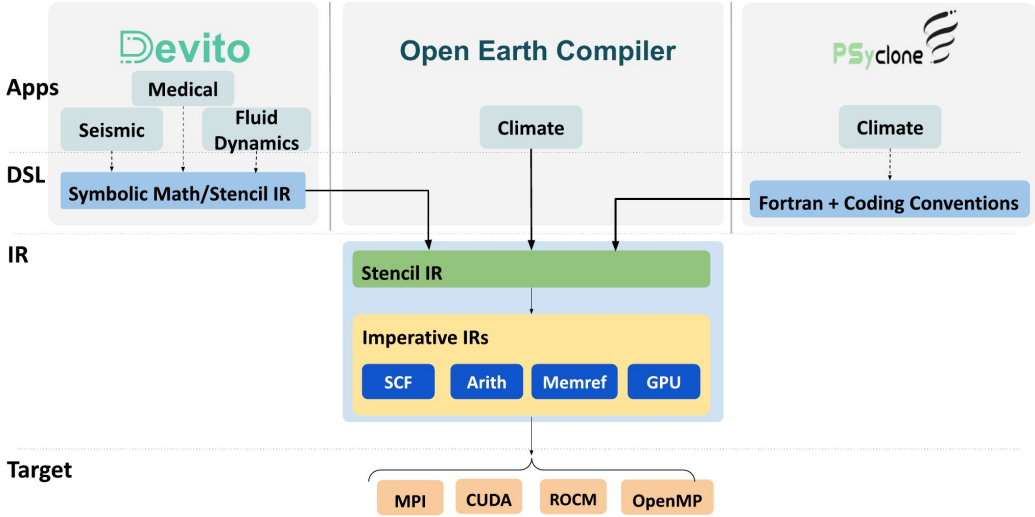
A shared compilation stack for distributed-memory parallelism in stencil DSLs

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*



ASPLOS

The problem: Monolithic Domain-specific languages

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

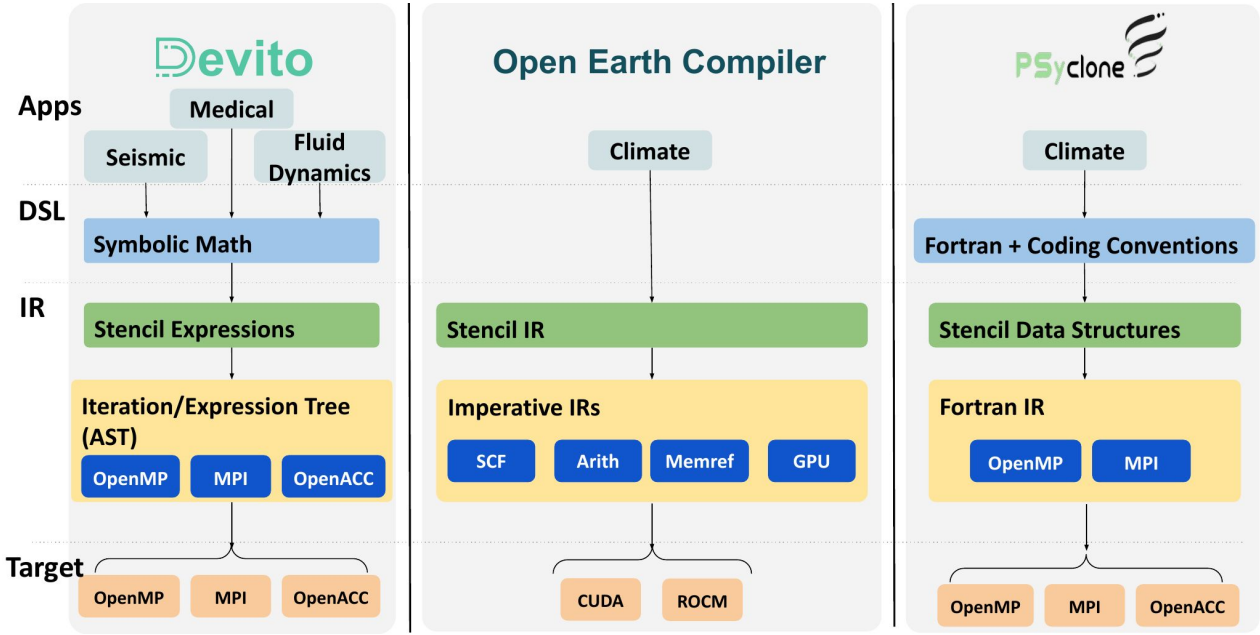
✔ Performance
✔ Productivity
✔ Portability
but…

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:

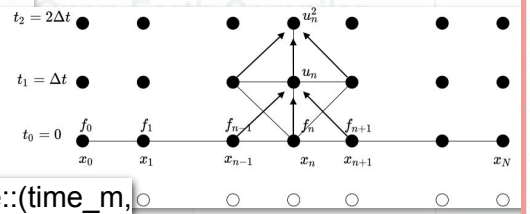
- ✘ Disjoint communities
- ✘ Lack of knowledge transfer

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

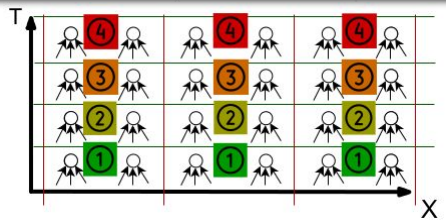
$$\text{Boundary Conditions}: \quad u(0,t) = 0; \quad u(L,t) = 0$$

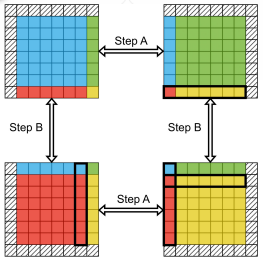
$$\text{Initial Conditions}: \quad u(x,0) = f(x)$$

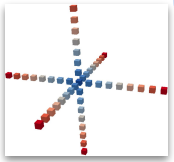
$$\frac{\partial u}{\partial t}(x,0) = g(x)$$




```
<[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

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

We propose to use compiler technology for it!  **MLIR for the win! (HPC 🤔 ?)**

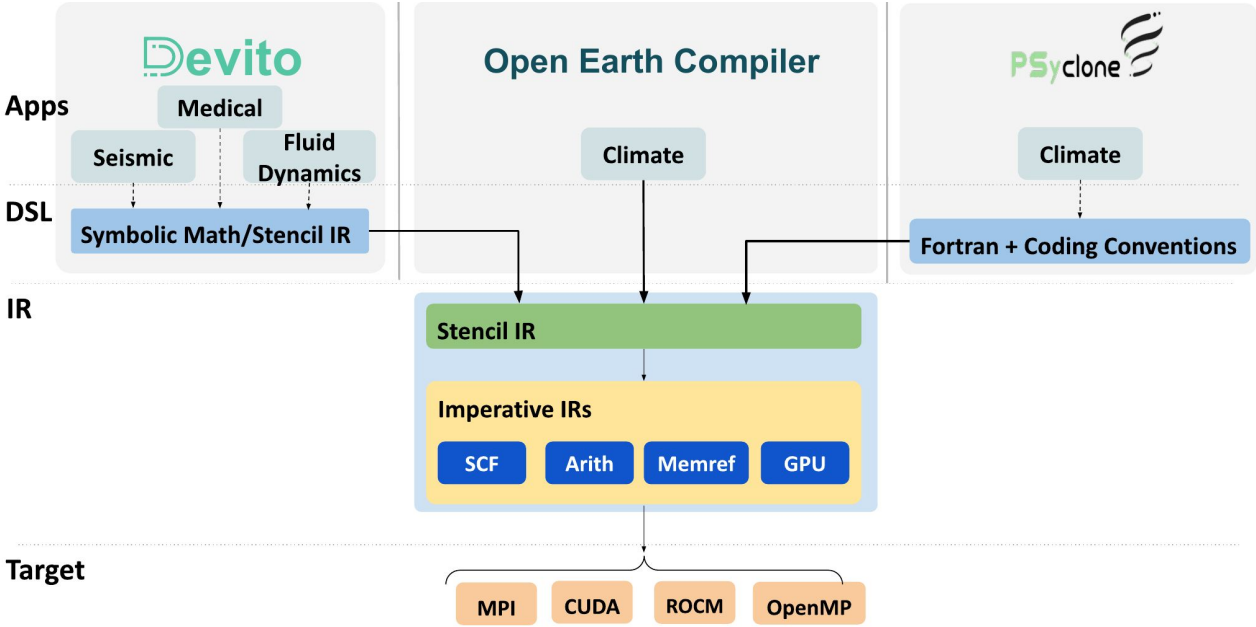
- ✔ Performance
- ✔ Productivity
- ✔ Portability

Technical benefits:

- ✔ Composability
- ✔ Code Reuse
- ✔ Interoperability
- ✔ Longevity

Societal benefits:

- ✔ Connected communities
- ✔ Extensive knowledge transfer



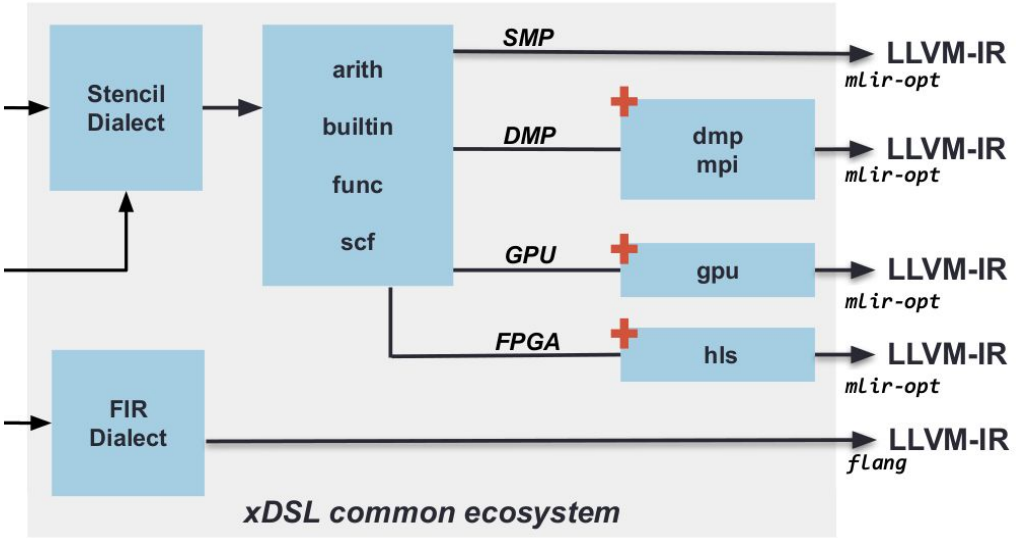
Contributions:

⭐**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

README License

CI - Python-based Testing passing | pypi package 0.17 | codecov 90% | chat on zulip

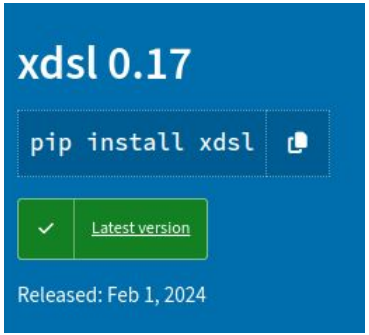
xDSL: A Python-native SSA Compiler Framework

- ✔ 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
192 stars
18 watching
54 forks
Report repository

Releases 26

v0.17 Latest
on Feb 1

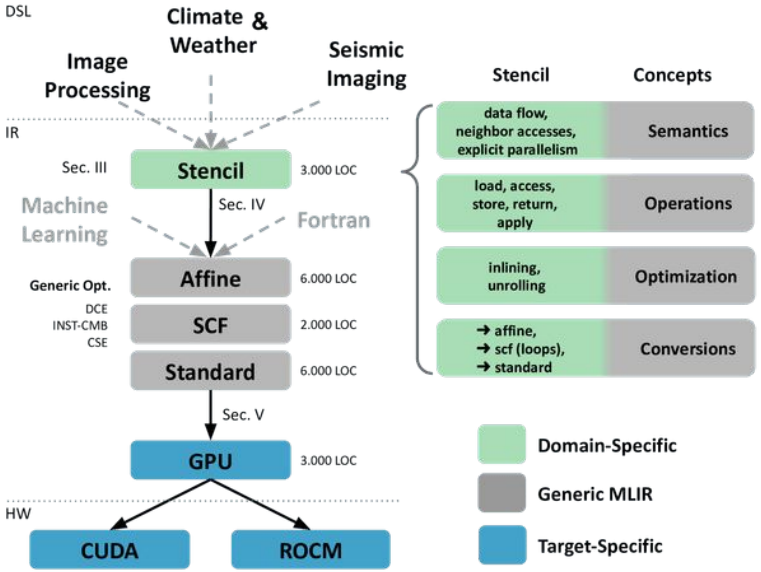
+ 25 releases

Contributors 54

+ 40 contributors

Fehr et.al, 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

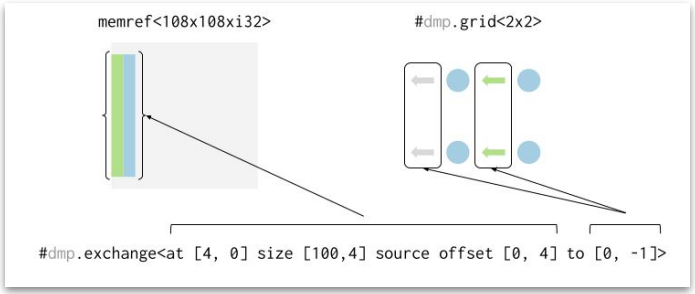
Gysi et.al, Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-accelerated Climate (2021), ACM TACO

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

The ‘dmp’ and ‘mpi’ dialects

```
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

'mpi' Dialect

This dialect models the Message Passing Interface (MPI), version 4.0. It is meant to serve as an interfacing dialect that is targeted by higher-level dialects. The MPI dialect itself can be lowered to multiple MPI implementations and hide differences in ABI. The dialect models the functions of the MPI specification as close to 1:1 as possible while preserving SSA value semantics where it makes sense, and uses `memref` types instead of bare pointers.

This dialect is under active development, and while stability is an eventual goal, it is not guaranteed at this juncture. Given the early state, it is recommended to inquire further prior to using this dialect.

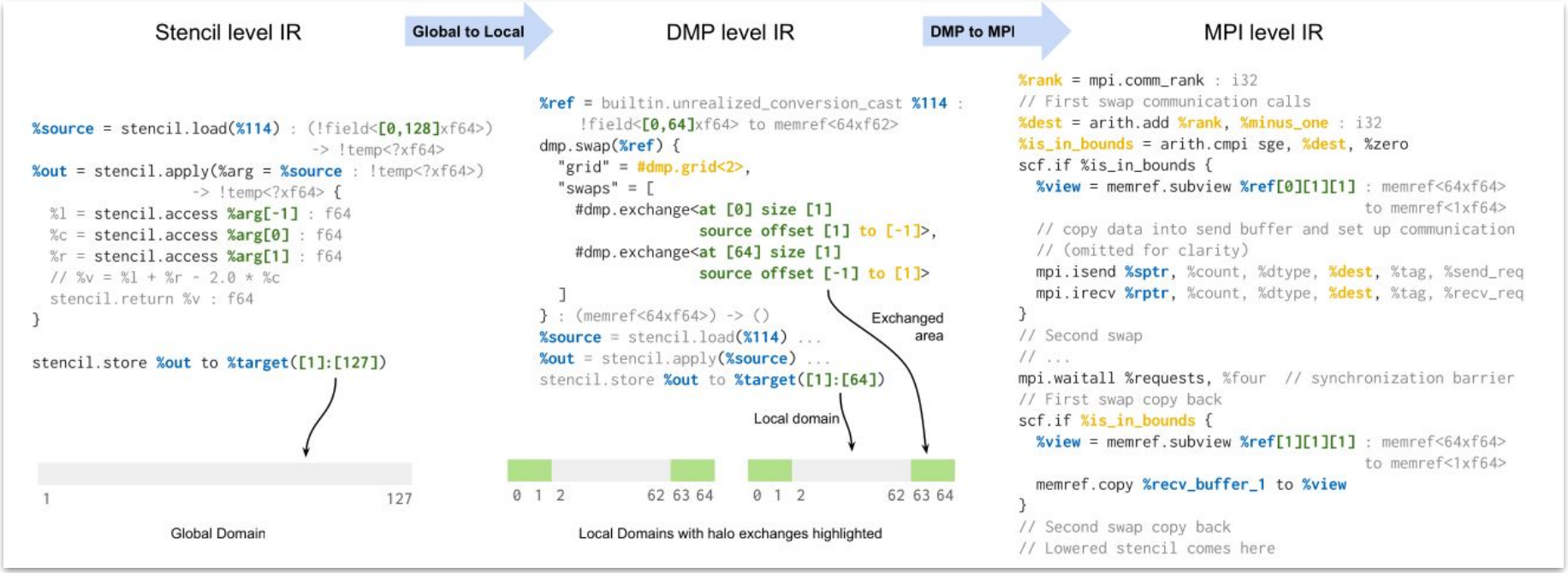
For an in-depth documentation of the MPI library interface, please refer to official documentation such as the OpenMPI online documentation.

- 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

- ✔ Message-passing IR
- ✔ Lowered to MPI library calls
- ✔ 🥷 Upstreamed to MLIR!

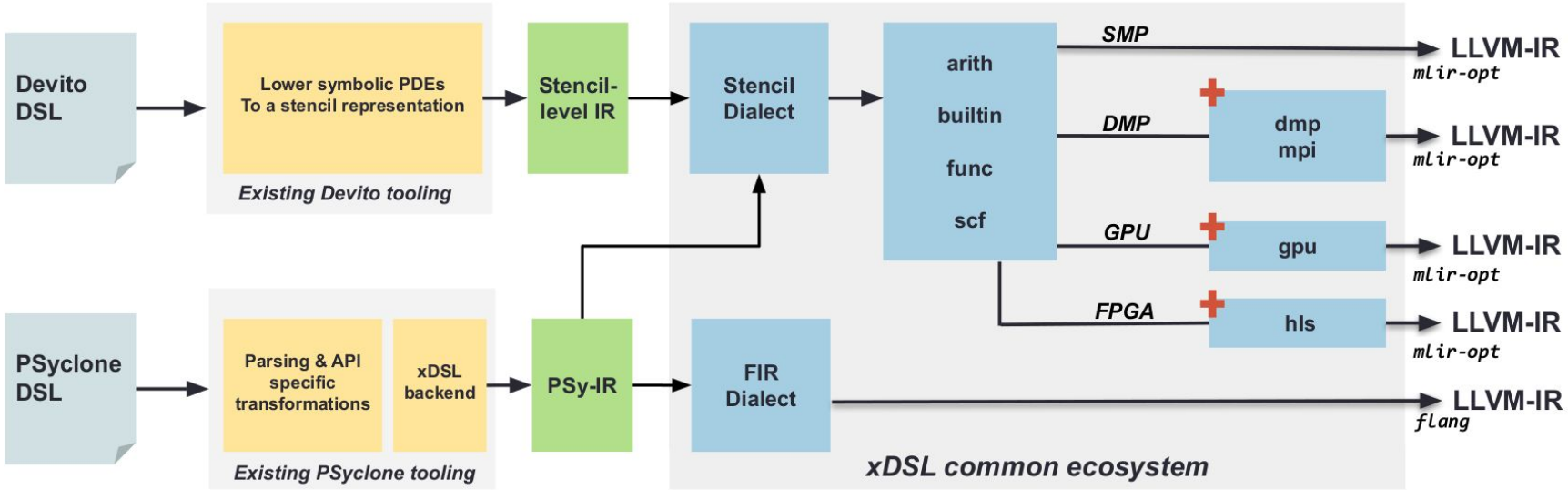
(Look for Anton Lydike, our in-house expert!)

Lowering from `stencil` to `mpi`



We highlight the data being operated on, shape and halo information, and communication-related information. This showcases how we can enrich the IR with relevant information to perform efficient rewrites at every level of abstraction.

The shared compilation stack for DMP in stencil DSLs



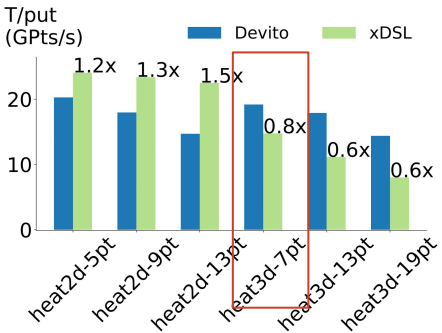
✔ Unlocked optimizations

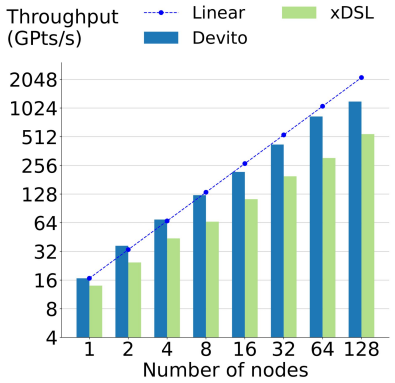
✔ Unlocked multi-node CPU

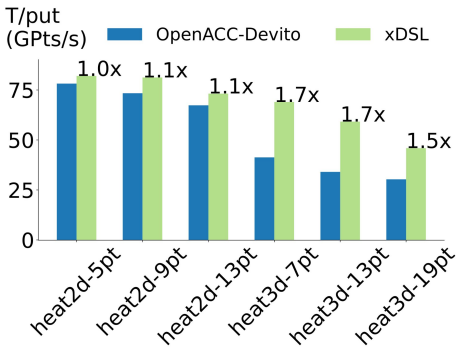
✔ Unlocked other backends (FPGA, CUDA)

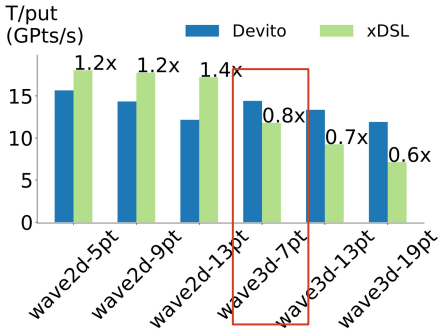
✔ Competitive or better performance with an order of 1000s LoC saved!

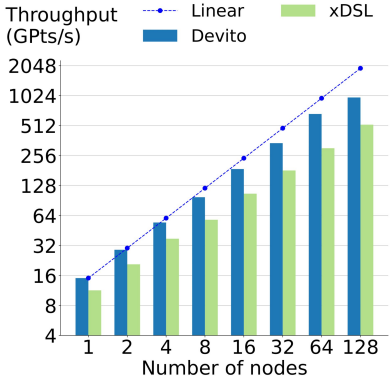
Performance evaluation: Devito

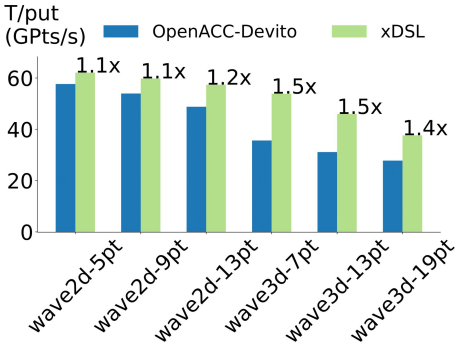












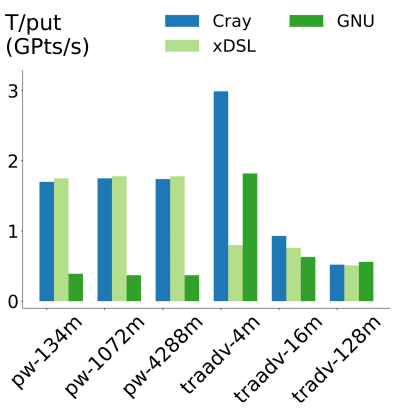
Single-node AMD EPYC 7742,
8 MPI ranks x 16 OpenMP threads,
16384^2 (2D) and 1024^3 (3D)

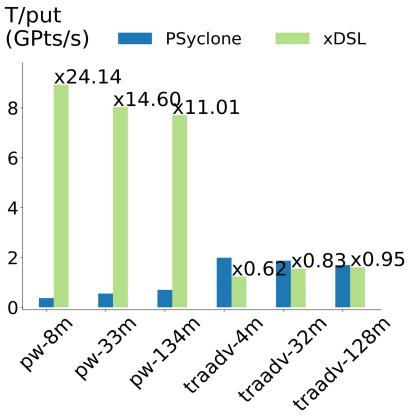
Heat (top) and wave (bottom), 3D-7pt,
multi-node strong scaling up to 128 nodes, total
of 16384 cores.

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





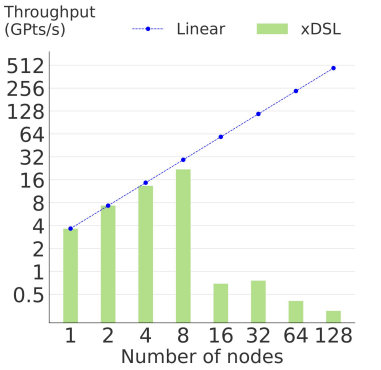
Single-node AMD EPYC 7742 throughput, PSyclone target code compiled with Cray and GNU compilers against xDSL-PSyclone.

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).

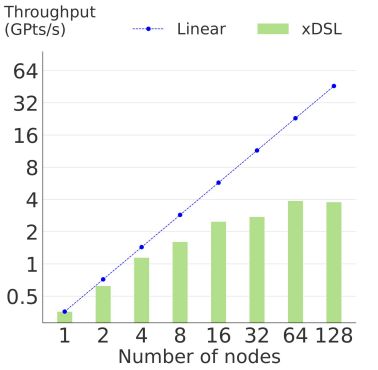
PW advection kernel



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.

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 use cases/wave propagators (e.g. isotropic elastic/ anisotropic TTI)
- Real-life simulation benchmarks

🙋 **Reach out!**

- Around with Anton Lydike, happy for a chat!
- Find us on https://xdsl.zulipchat.com/

Appendix

Acknowledgements

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

The authors would like to thank the shepherd and the reviewers for their comments and allocated time towards improving this work.

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

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!

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.\@ Futhermore, there is an ongoing effort to provide one standard MPI ABI, greatly simplifying this problem \citep{hammond2023mpi}.

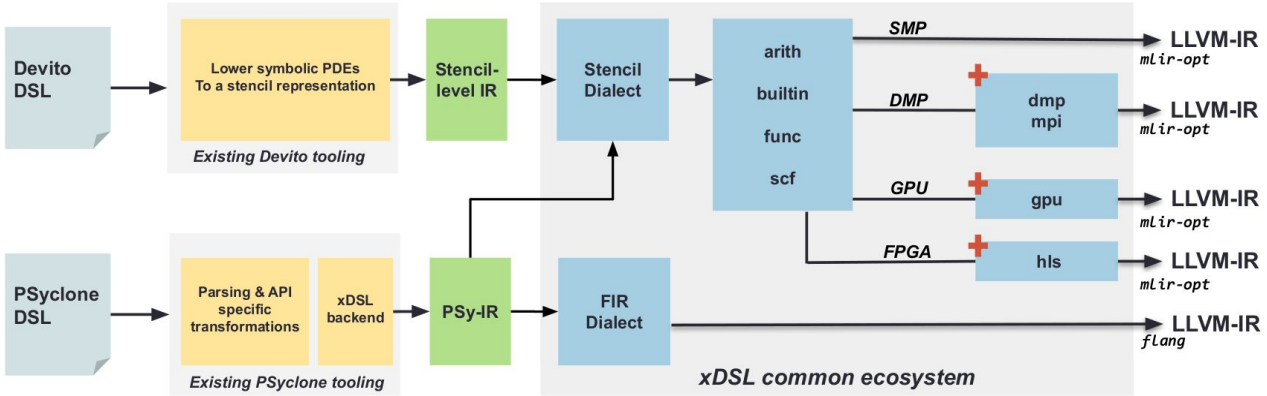
Contributions: A shared compilation stack for DMP in stencil DSLs

⭐**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 xDSL /MLIR



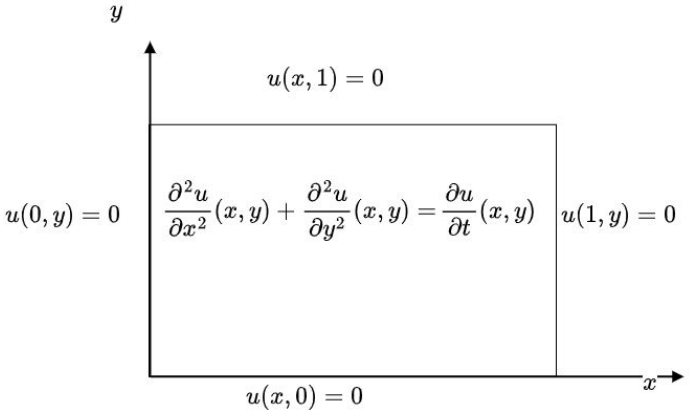
⭐Performance evaluation shows:

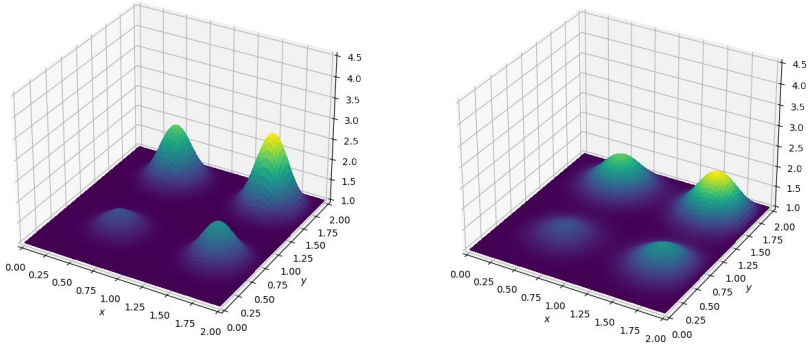
- Highly competitive DMP solvers
- Support for new targets
- Support for new hardware
- Improved performance for a number of cases

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

An example from textbook maths to via Devito DSL

2D Heat diffusion modelling






```python
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)
```

- 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_devitoscript.py ”

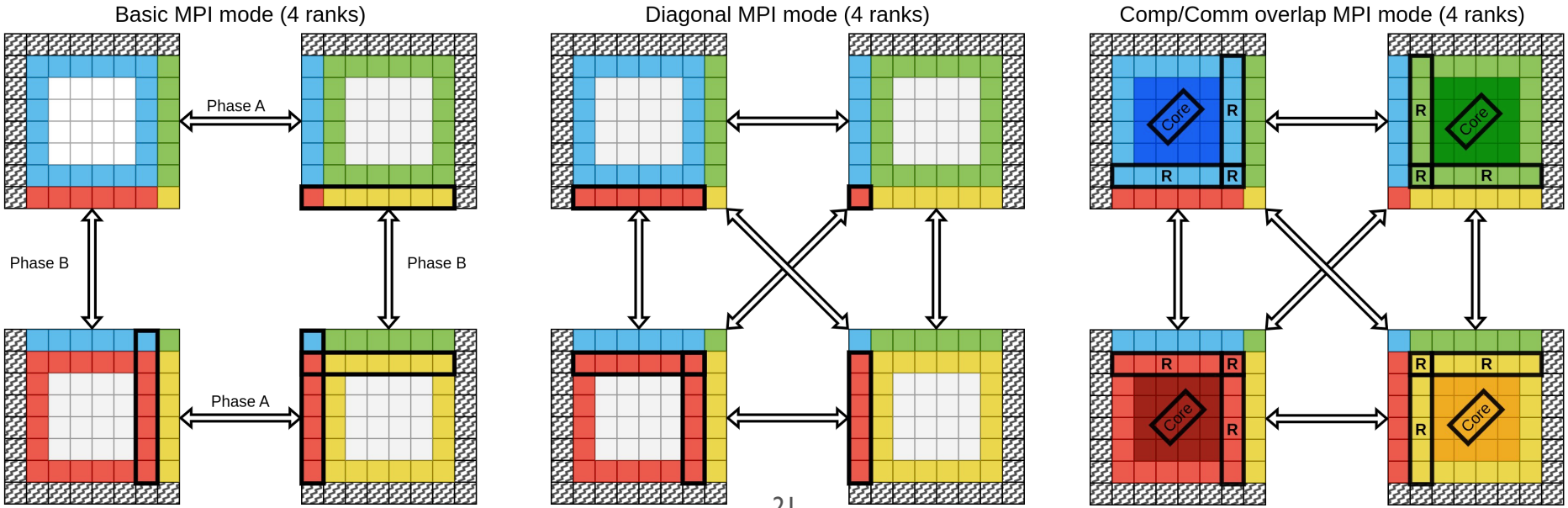
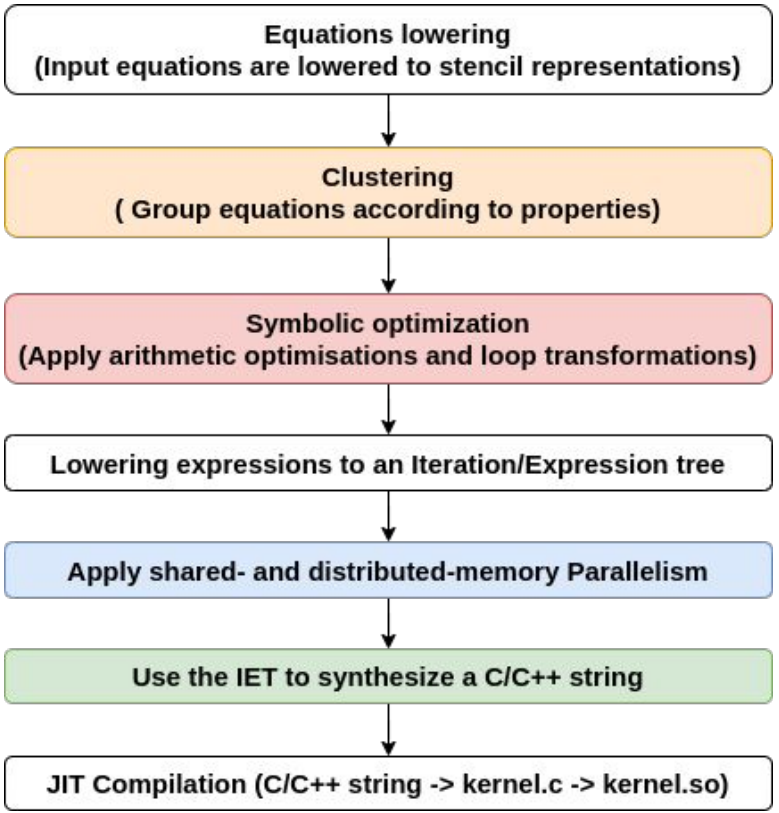
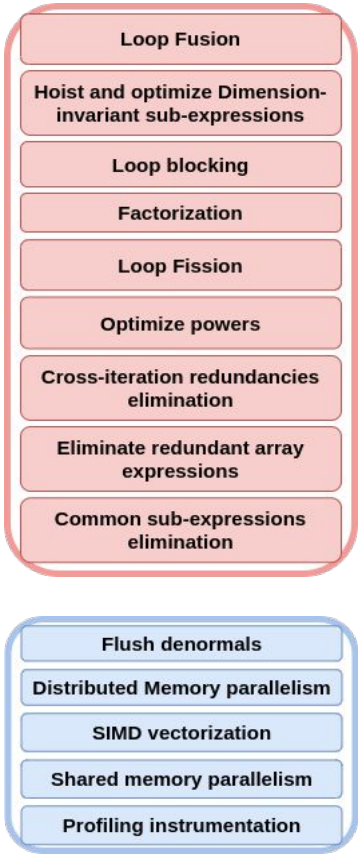


Figure style influenced from Li et.al, ICPP 2021

The Devito compiler automatically applies lots of optimisations





+ 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

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

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

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

$$\text{Initial Conditions}: \quad u(x,0) = f(x)$$

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