Making HPC more accessible: Effective HPC programming via domain specific abstractions

Nick Brown
n.brown@epcc.ed.ac.uk

Emilien Bauer
emilien.bauer@ed.ac.uk

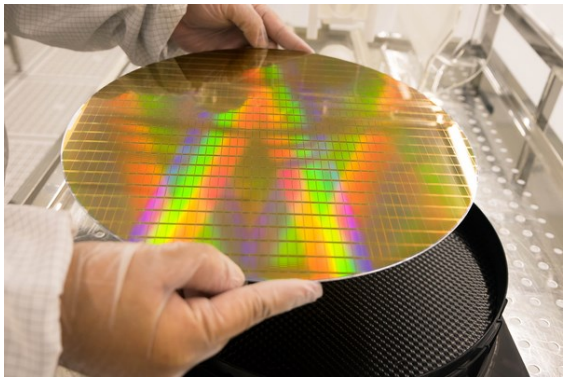
epcc

THE UNIVERSITY of EDINBURGH
informatics

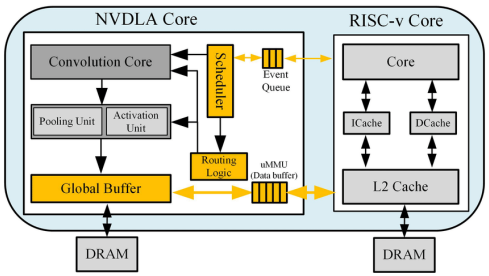
THE UNIVERSITY OF EDINBURGH

The challenge

- Writing parallel code that can exploit present day supercomputers is extremely hard and requires highly specialist skills



- But this is going to get even more difficult as we move further into the exascale era
- It is no longer tenable to directly leverage serial languages and add in our own parallelism (e.g. MPI, CUDA, vectorisation etc)



ExCALIBUR

Domain Specific Languages to the rescue!

GridTools

OP-DSL

- Raise the abstraction level so the programmer can provide a high level description of their algorithm that the compiler can then exploit to make tricky, low level decisions around parallelism

Saiph

PSyclone

- *Languages* is a poor term, *abstractions* is far better

ANTAREX 10^18

NMODL
Code Generation Framework

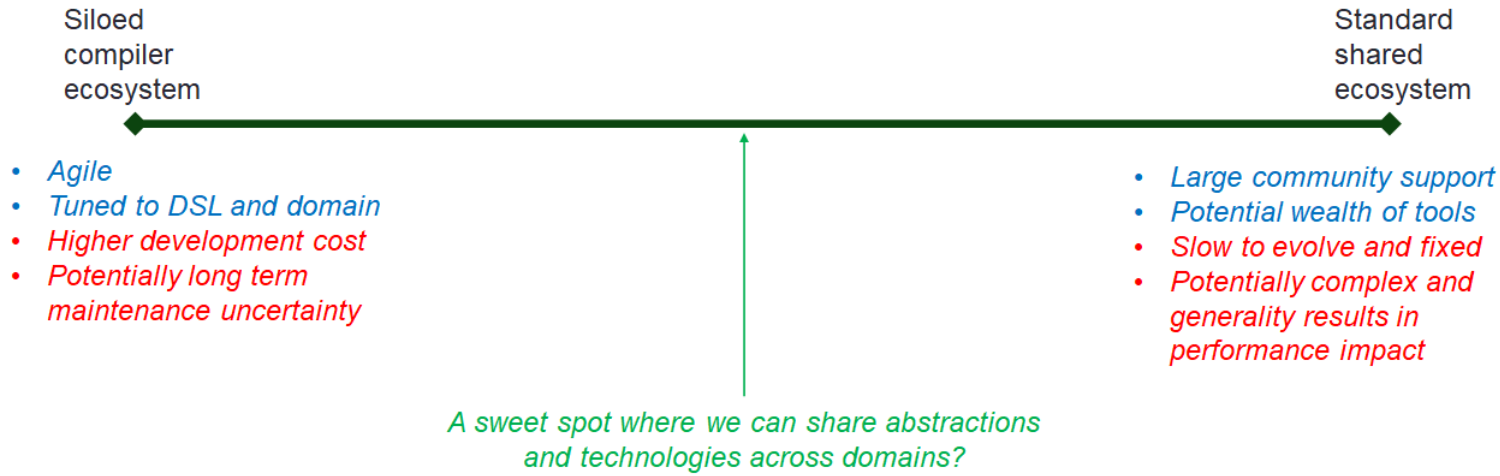
CLAW

Halide
a language for fast, portable computation on images and tensors

ExCALIBUR

Breaking down silos

- The elephant in the room is that these are all heavily siloed and reinvent the wheel
  - Requires significant development effort from the DSL designers
  - Risk for users (e.g. will the DSL be maintained in the future?)
  - Challenges supporting new architectures



There is therefore a sweet spot in the middle, where we gain the best of both worlds

Step in MLIR and LLVM

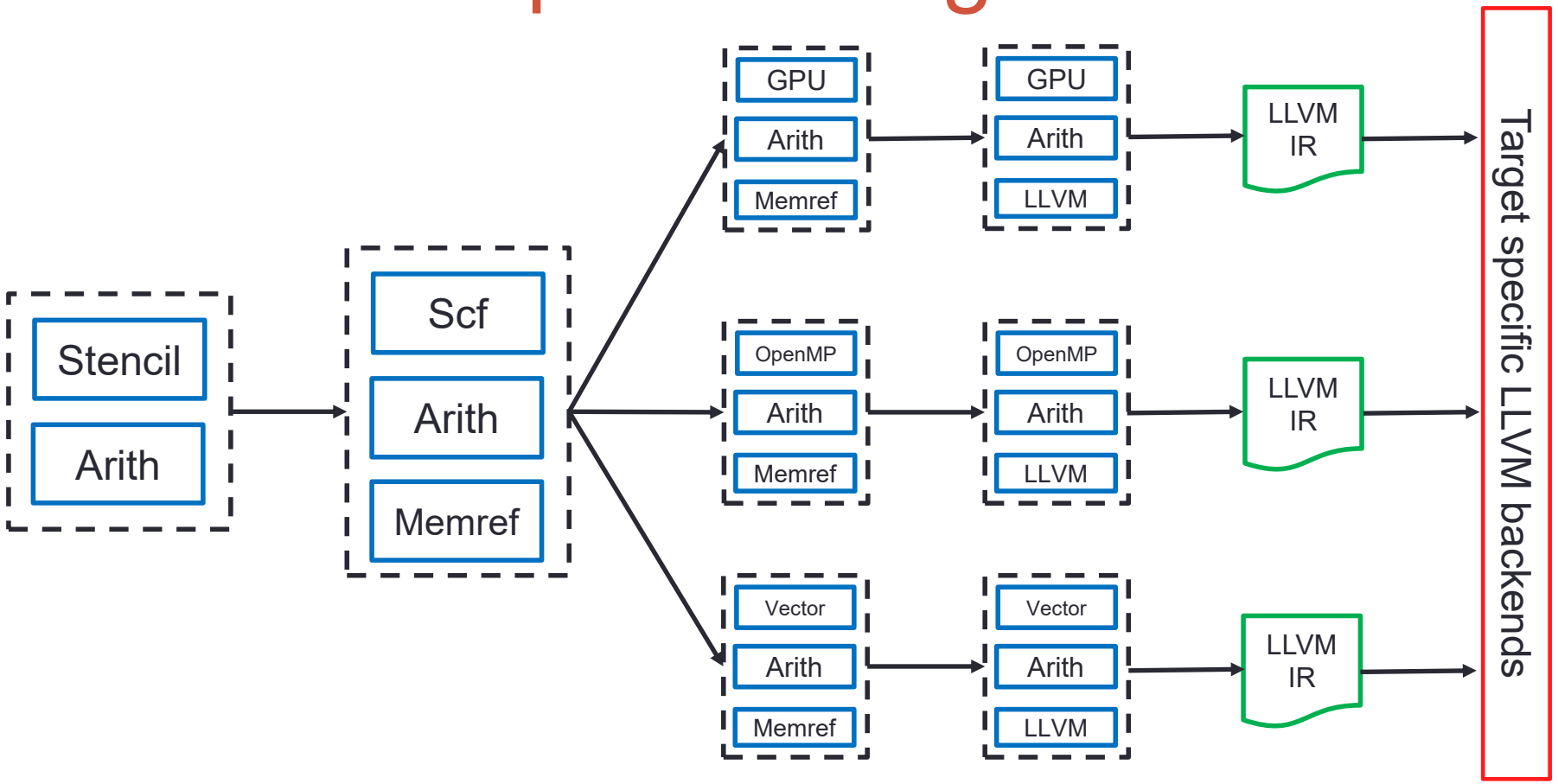
- LLVM is the ubiquitous compiler framework that has been around for over 20 years
  - In addition to providing its own compilers, AMD, Intel and Arm compilers are all built on-top of LLVM, as is the Cray C/C++ compiler and AMD Xilinx's FPGA HLS technology.

- MLIR was developed by Google in 2020 and since 2021 has been part of the main LLVM repository

MLIR

- At its core MLIR is a framework for developing different types of Intermediate Representations (IR) at different levels
- Numerous (IR) dialects and transformations are provided which enables lowering between these
- Can add your own easily
- A big community has grown up

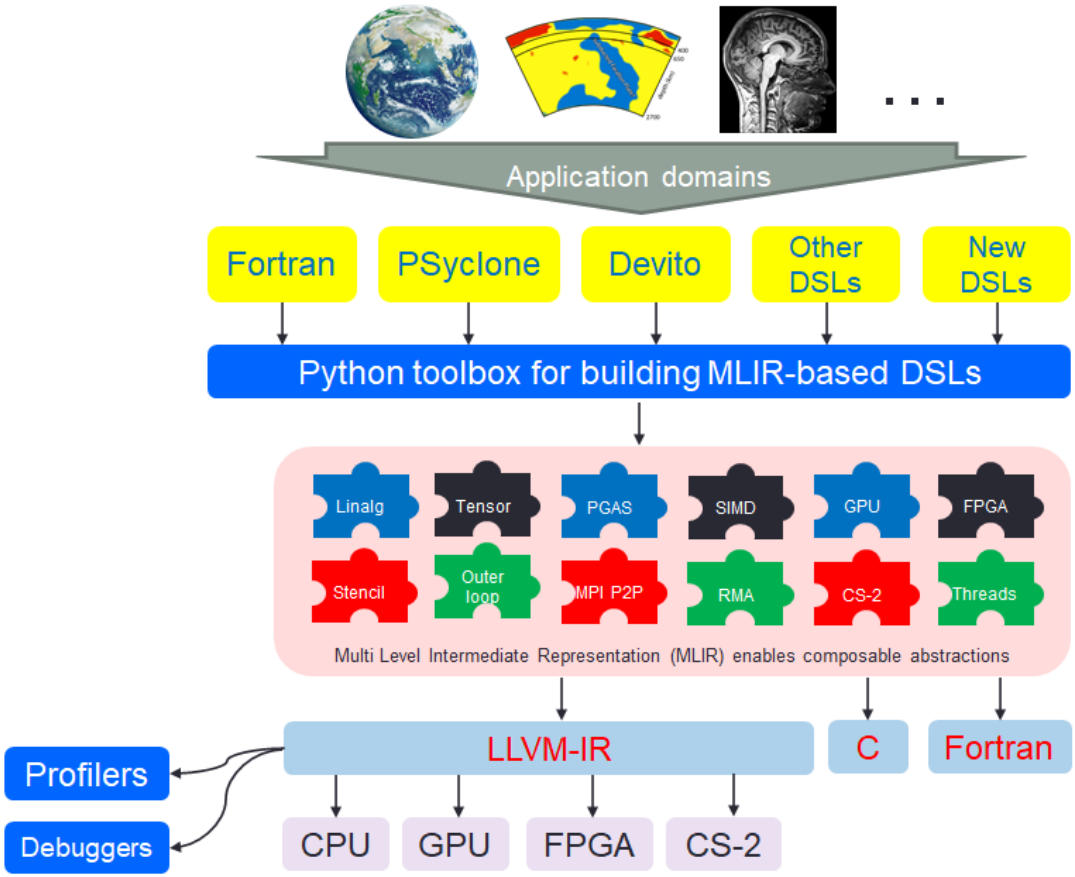
MLIR example lowering



- But MLIR is written in C++ and using specialist Tablegen configuration format for dialects, MLIR is esoteric and requires a steep learning curve

xDSL: A Python toolkit for MLIR

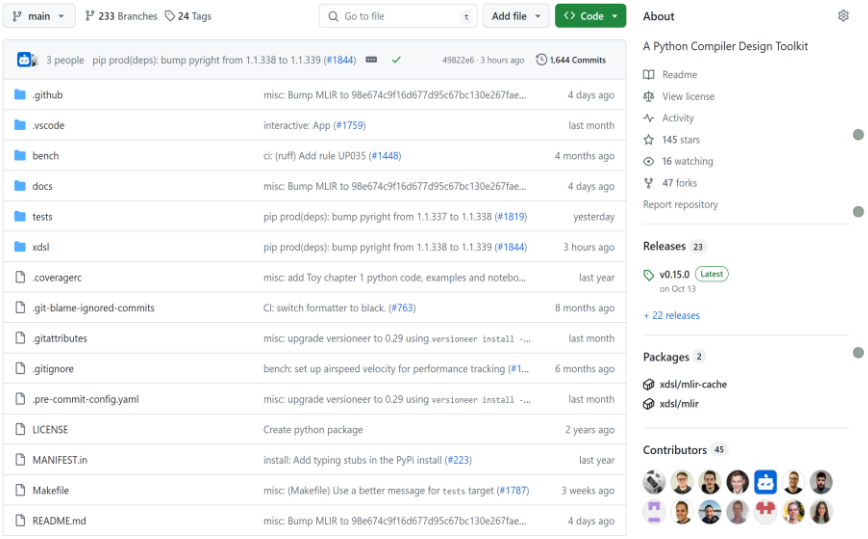
- Python toolkit for MLIR that enables high productivity development of dialects and transformations



- Contains existing MLIR dialects & transformations and we are adding HPC focussed ones too
- Whole load of other things also, such as an MLIR interpreter and Python frontend

EXCALIBUR

xDSL



https://github.com/xdslproject/xdsl

xDSL

- Makes experimenting with MLIR trivial
- Can go between xDSL and MLIR, leveraging transformations in both
- For our DSL purposes also means that a DSL can be a thin abstraction layer on-top of xDSL which provides a wealth of dialects and transformations that will ultimately drive MLIR/LLVM

intel

CINECA

ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA A.D. 1088

Google

ETH zürich

Numba

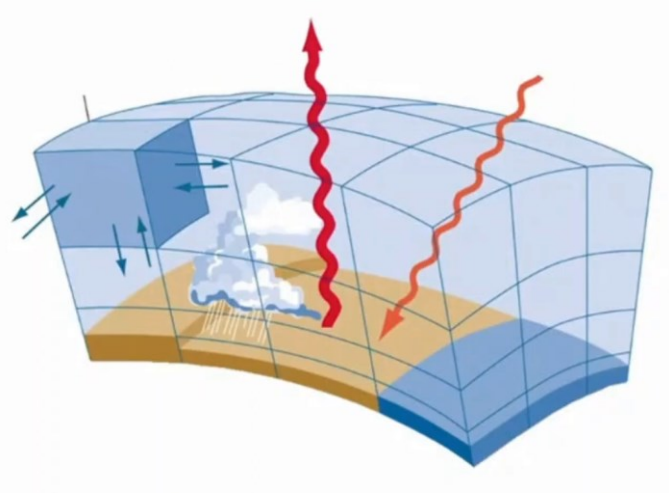
TECHNISCHE UNIVERSITÄT DRESDEN

CIUK Theme: Productive supercomputing

- Making HPC More Accessible
  1. For HPC developers as they can more easily leverage supercomputers by using Domain Specific Languages
  2. For DSL developers as these are now a thin abstraction layer atop a common, well supported, ecosystem

Domain Specific Compilation

- The Open Earth Compiler project from ETH Zurich used MLIR for domain specific compilation of stencil codes
- Successfully leveraged MLIR's qualities to leverage high-level information and reach high throughput on GPUs



ETH zürich

EXCALIBUR

First targets

PSyclone

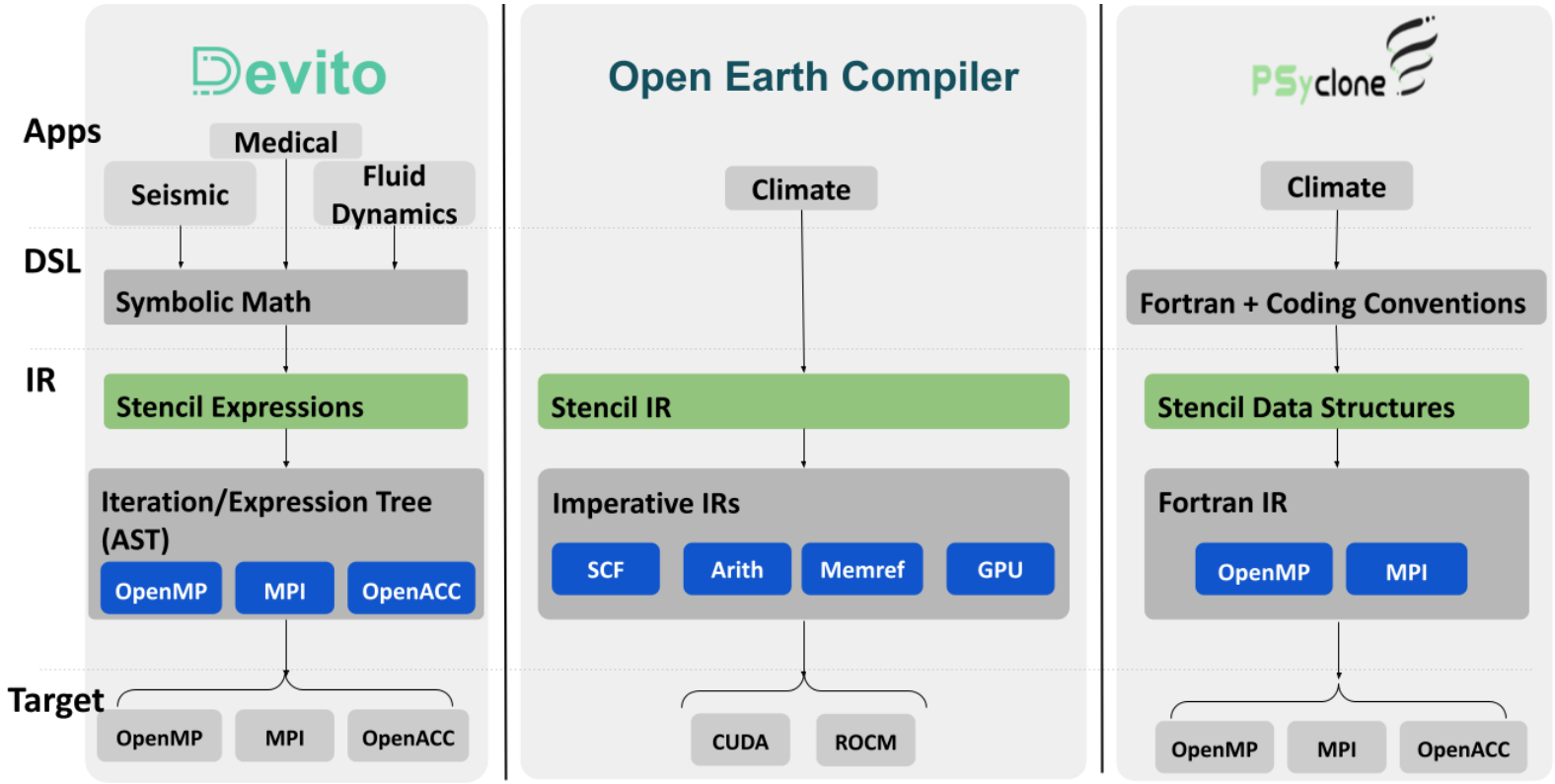
- Climate simulation
- Discovers stencil code in Fortran
- Apply Domain Specific optimizations
- Generates MPI, OpenMP, OpenACC code

Devito

- Seismic and fluid simulation, medical imagery
- Generates stencil code from Python PDEs
- Apply Domain Specific optimizations
- Generates MPI, OpenMP, OpenACC code

ExCALIBUR

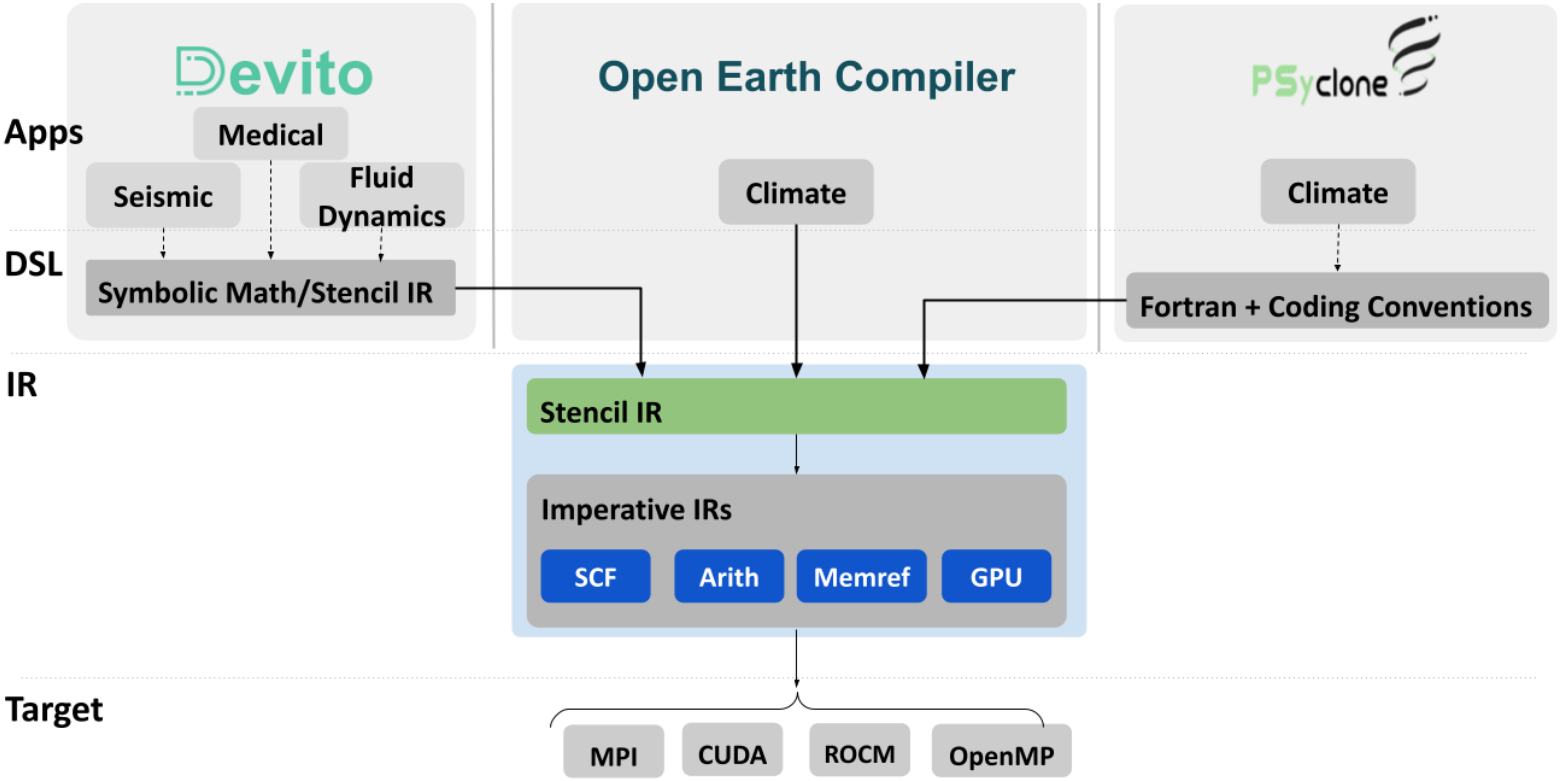
The broken silos



- Everything below the DSL layers is reinvented wheels

The sweet spot

xDSL

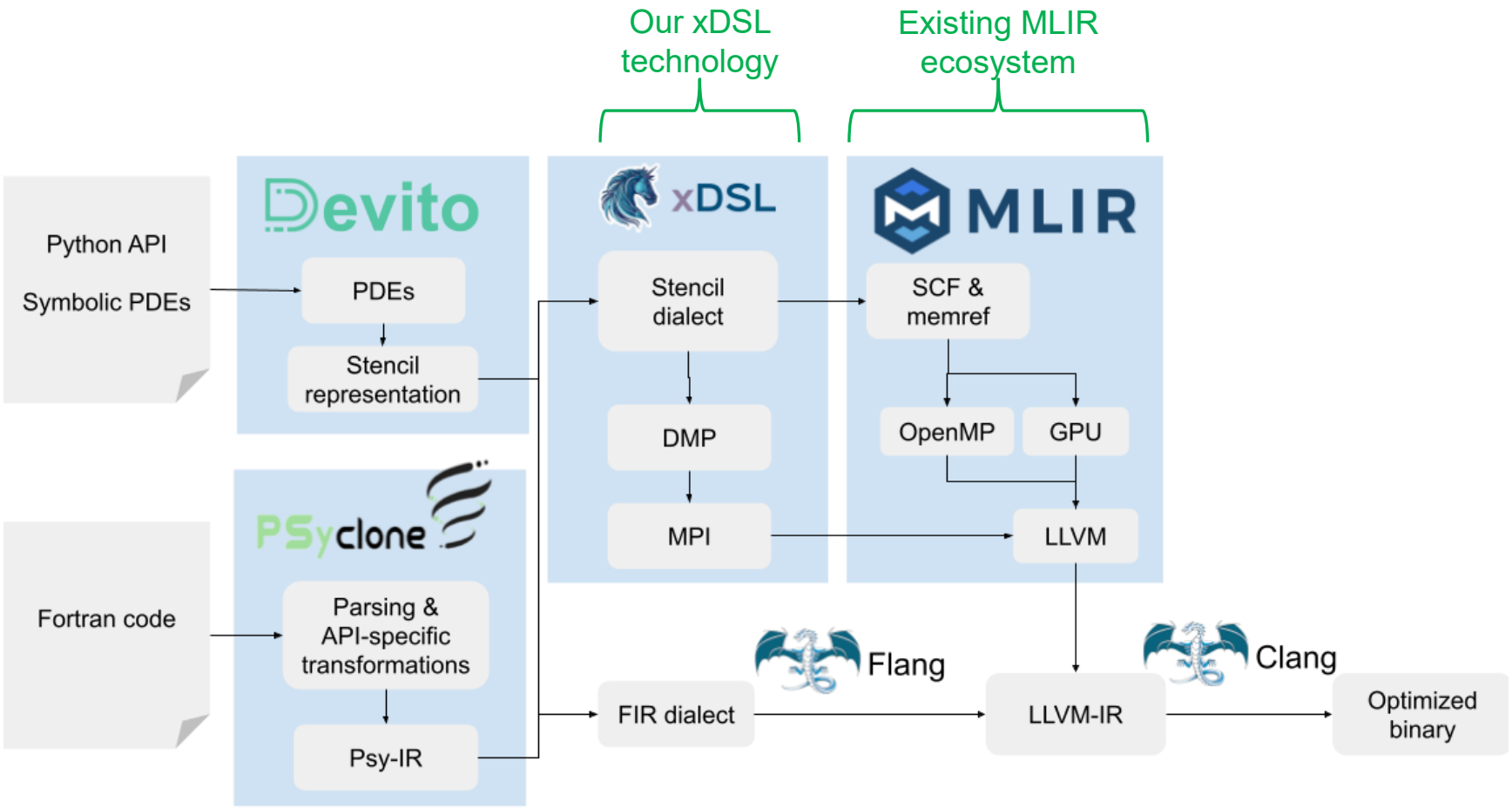


Sharing infrastructure:

- Implementation and Maintenance cost is spread across projects
- Everyone gets all benefits
- Can still be driven by specific needs

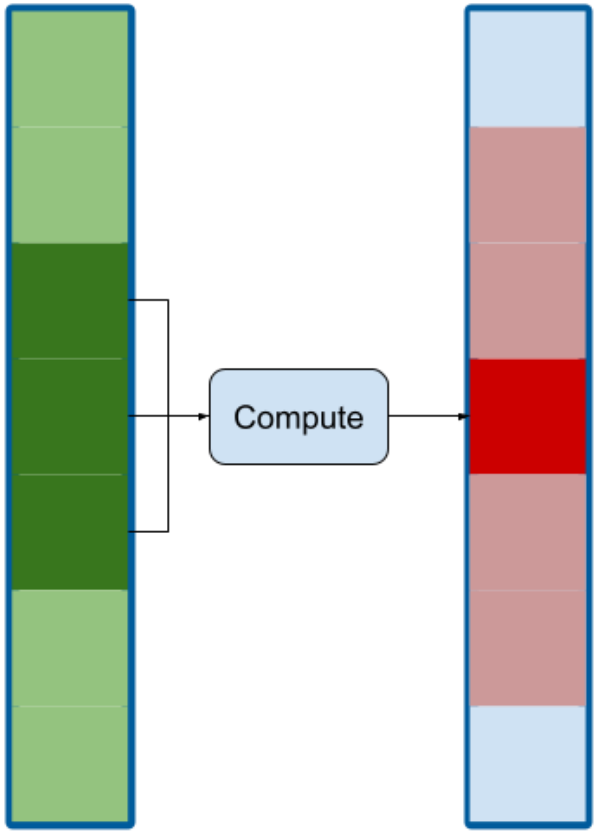
ExCALIBUR

The sweet spot

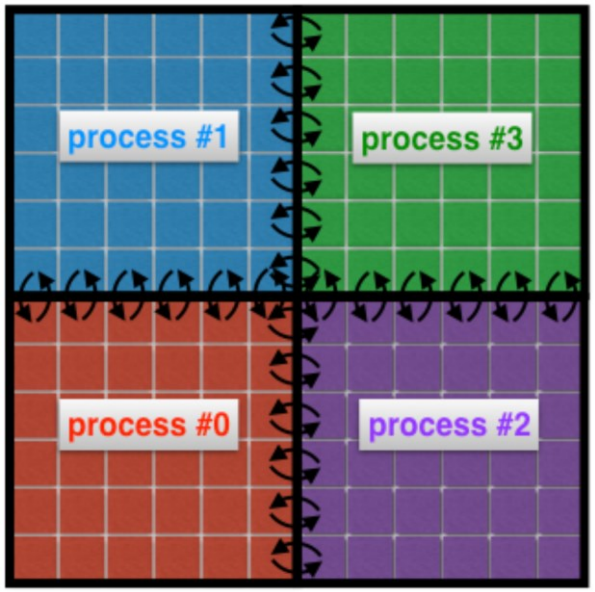


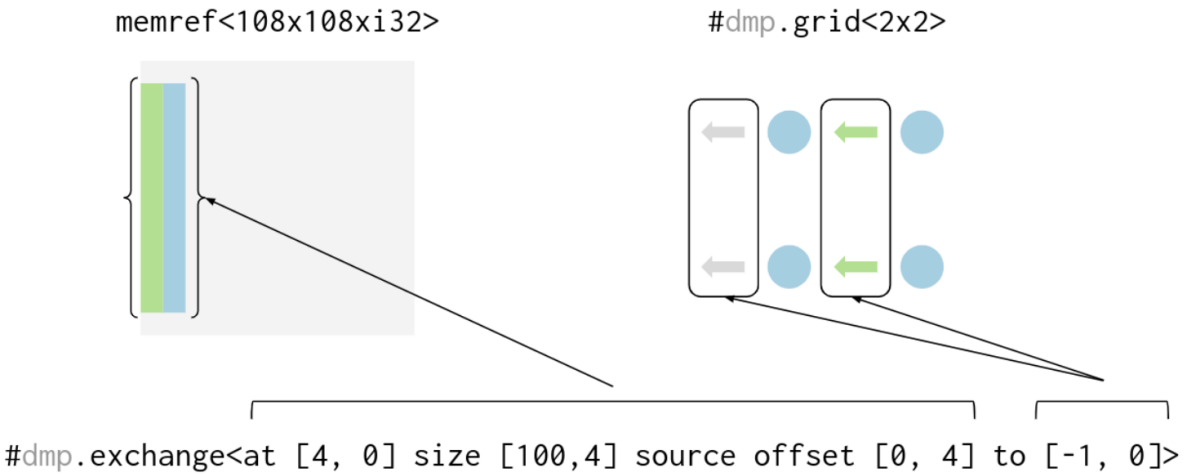
A flexible abstraction

```
  %input = stencil.load(%input_buffer) :
   (!field<[0,7]xf64>)-> !temp<?xf64>
  %out = stencil.apply(%arg = %input : !temp<?xf64>)-
> !temp<?xf64> {
    %l = stencil.access %arg[-1] : f64
    %c = stencil.access %arg[0] : f64
    %r = stencil.access %arg[1] : f64

    // %v = Some arbitrary computation

    stencil.return %v : f64
  }

  stencil.store %out to %target([1]:[6])
       : !temp<?xf64> to !field<[0,128]xf64>
```



High-level distribution



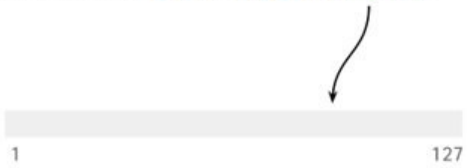


Halo exchange is a simple idea, let's *keep it* simple

High-level distribution

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

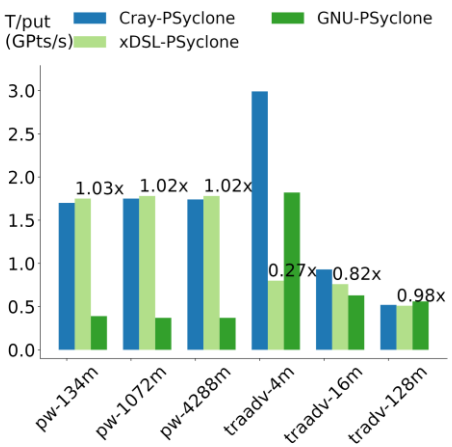
```
%ref = builtin.unrealized_conversion_cast %114 :
    !field<[0,64]xf64> to memref<64xf62>
dmp.swap(%ref) {
  "grid" = #dmp.grid<2>,
  "swaps" = [
    #dmp.exchange<at [0] size [1]
                  source offset [1] to [-1]>,
    #dmp.exchange<at [64] size [1]
                  source offset [-1] 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

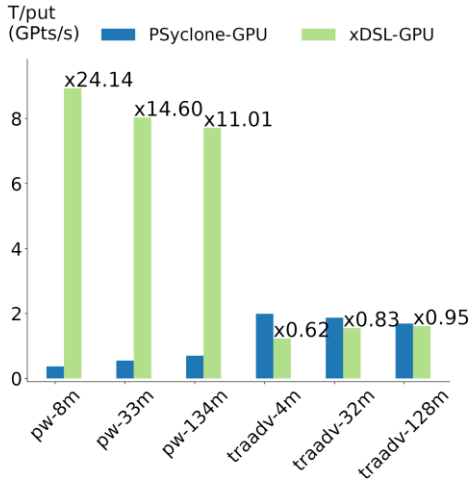
```
%rank = mpi.comm_rank : i32
// First swap communication calls
%dest = arith.add %rank, %minus_one : i32
%is_in_bounds = arith.cmpi sge, %dest, %zero
scf.if %is_in_bounds {
  %view = memref.subview %ref[0][1][1] : memref<64xf64>
                                      to memref<1xf64>
  // copy data 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
```

Performance of PSyclone & Devito

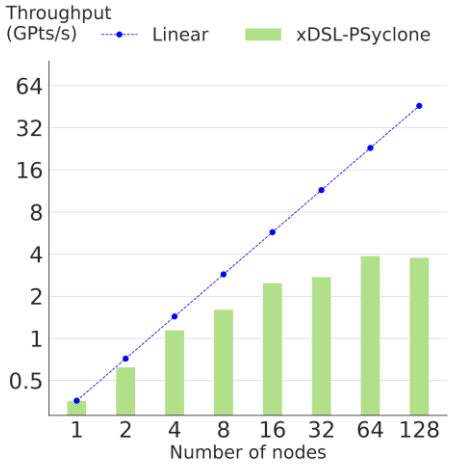
Single-node on ARCHER2

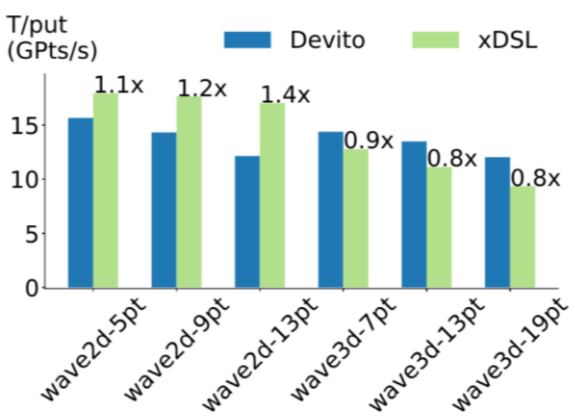


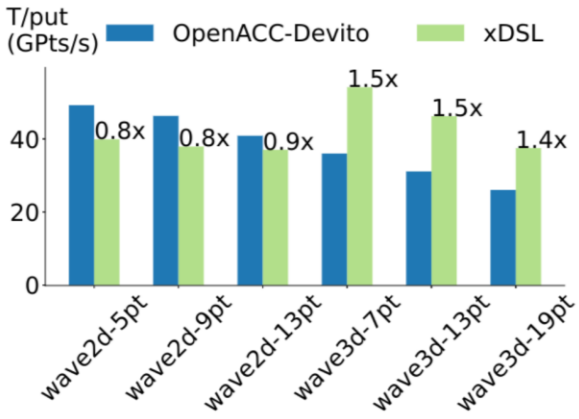
GPU on Cirrus (V100)

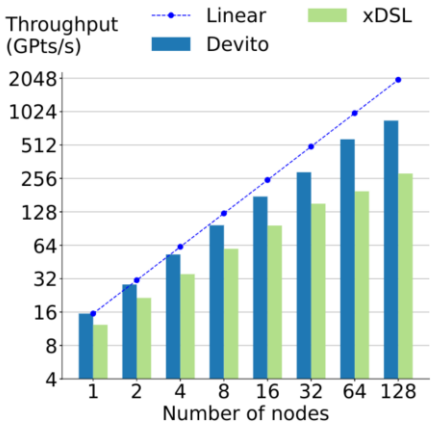


Strong scaling on ARCHER2





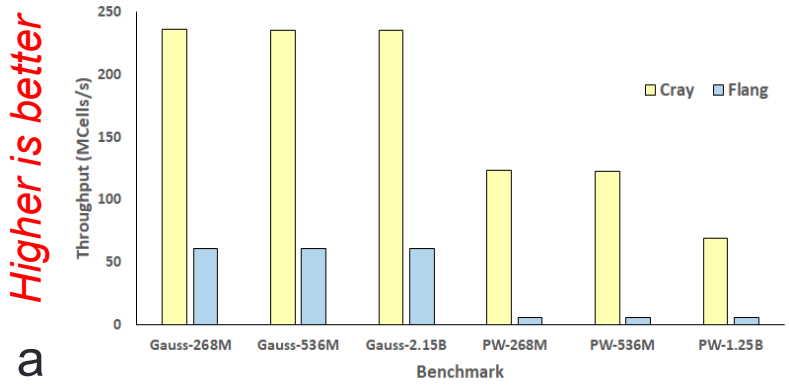


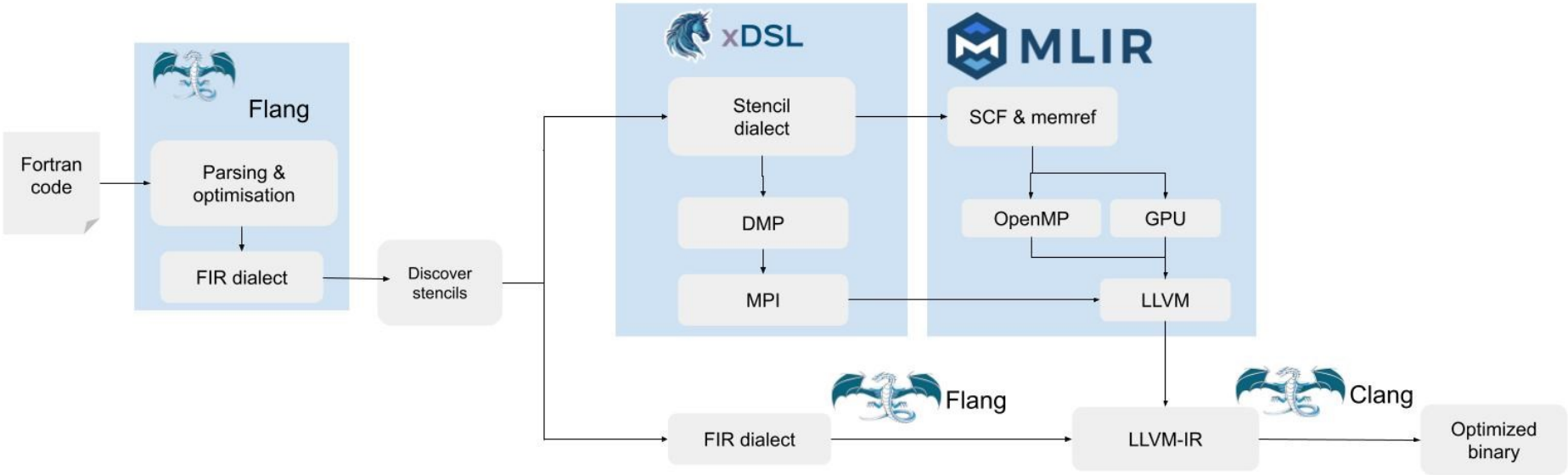


Higher is better, PSyclone top row & Devito bottom row

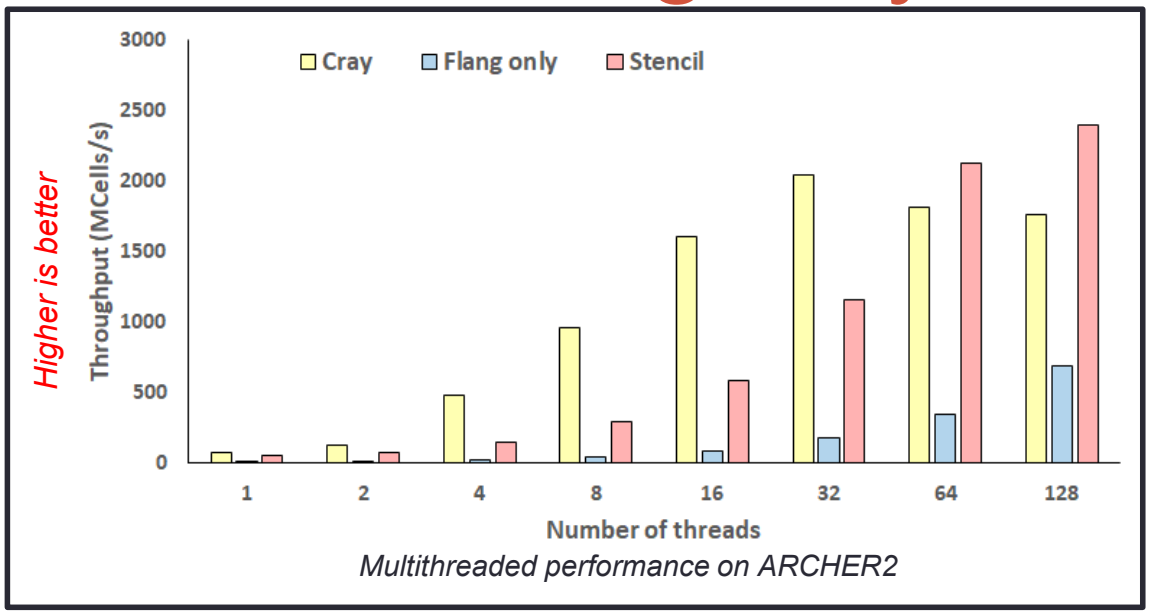
Integration with Flang: Beyond DSLs

- Performance falls short of Cray compiler for our stencil benchmarks (on a single core of ARCHER2, HPE Cray EX)
  - Our theory was that we can gain a performance improvement by combining with domain specific optimisations

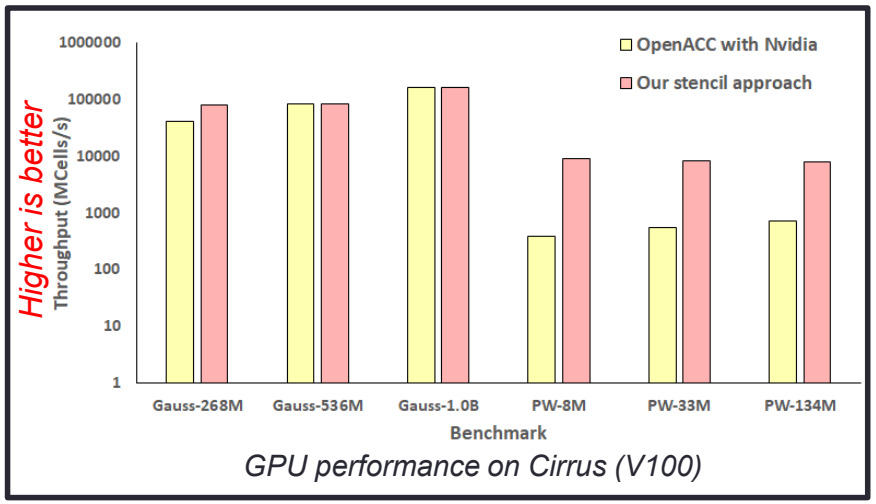




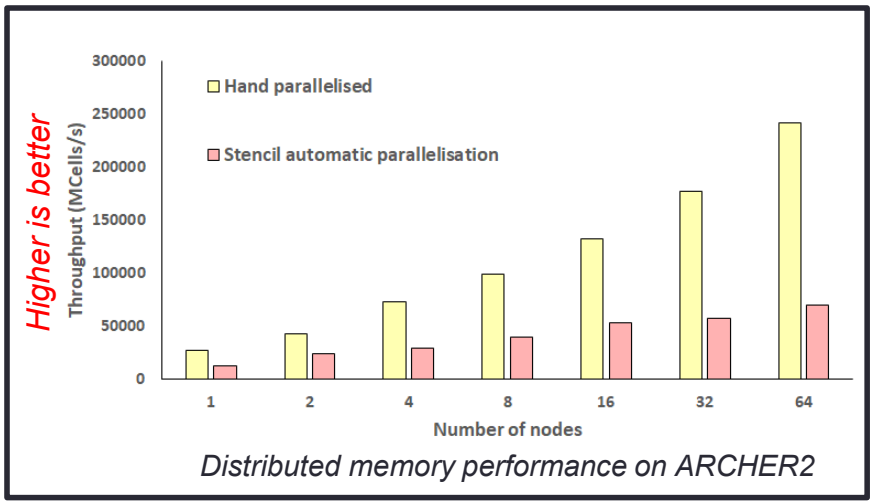
Integration with Flang: Beyond DSLs



Multithreaded performance on ARCHER2



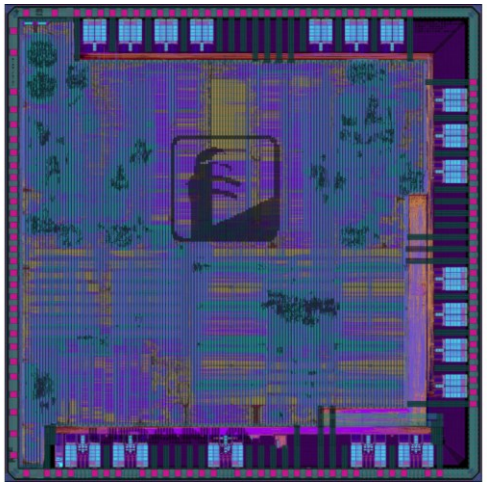
GPU performance on Cirrus (V100)



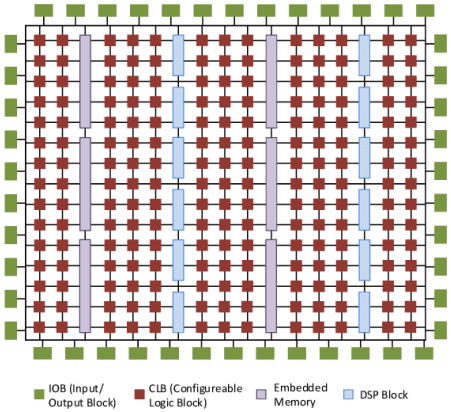
Distributed memory performance on ARCHER2

Auto-optimisation for new architectures

Field Programmable Gate Arrays (FPGAs)



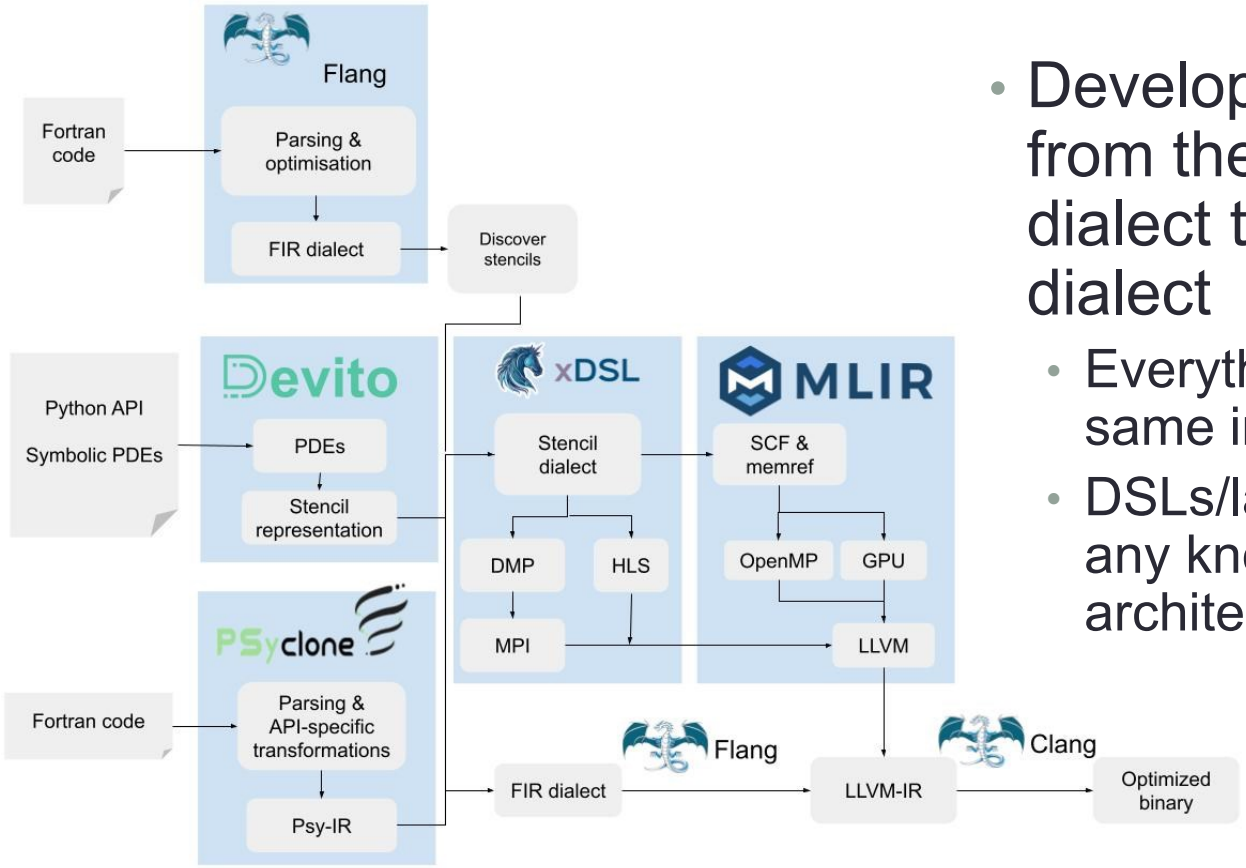
RISC-V high core-count accelerator chip



- Very different algorithm layout on FPGAs from the Von Neumann counterpart
  - Requires significant experience, expertise and time to port codes to the architecture
  - Using our existing infrastructure and domain specific abstractions, can we automatically optimise algorithms for FPGAs?
    - So there is a single, unchanged, Von Neumann version driving them?

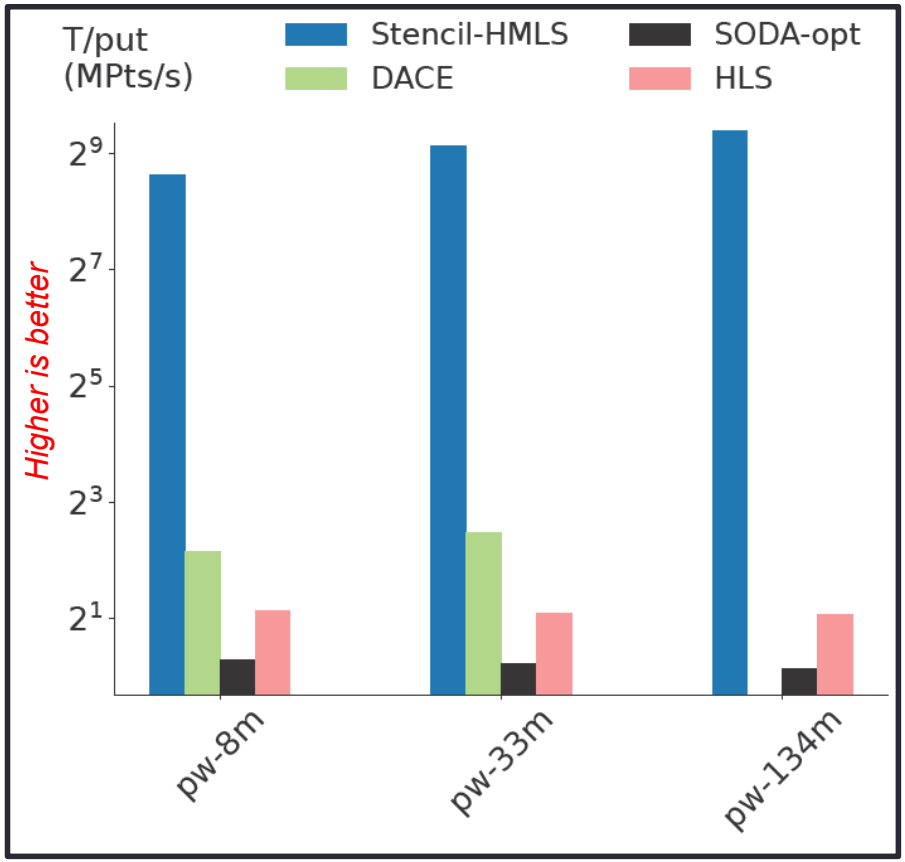
Automatic optimisation for FPGAs

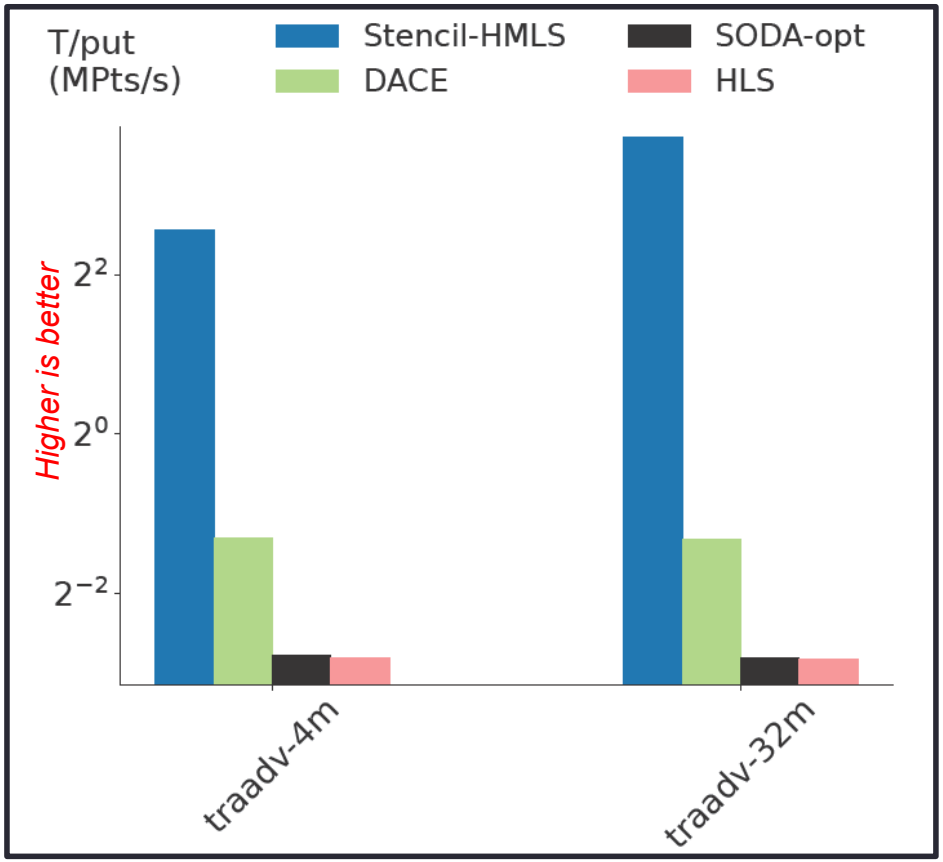
- AMD Xilinx already have an LLVM backend
  - We added a new High Level Synthesis (HLS) MLIR dialect that then lowers to IR compatible with AMD Xilinx's backend



- Developed transformations from the existing stencil dialect to this new HLS dialect
  - Everything else remains the same in the compiler pass
  - DSLs/languages don't need any knowledge of the target architecture

Automatic optimisation for FPGAs





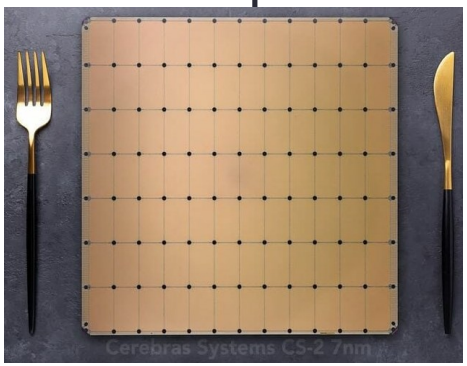
- On an AMD Xilinx U280 FPGA
- For PW advection, our approach is between 90 and 100 times faster than DaCE
- For tracer advection, our approach is between 14 and 21 times faster than DaCE

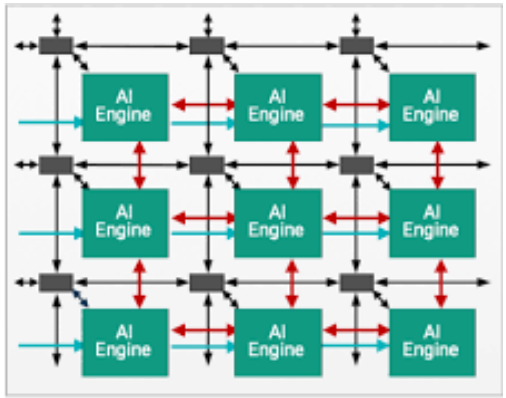
EXCALIBUR

Conclusions and next steps…

- We can't keep reinventing the wheel when it comes to compiler infrastructure for DSLs
- LLVM and MLIR are a strong alternative for sharing
  - We have developed the xDSL Python framework to lower the barrier to entry and offer key HPC components so that the ecosystems supports HPC workloads
- A lot of potential for bringing domain specific abstractions into existing languages, and we should be investing in Flang

- To date our focus has been on stencils, are now generalising this to other patterns





ExCALIBUR

Nick Brown

Emilien Bauer

Anton Lydike

- https://xdsl.dev
- https://github.com/xdslproject/xdsl
- https://xdsl.zulipchat.com/