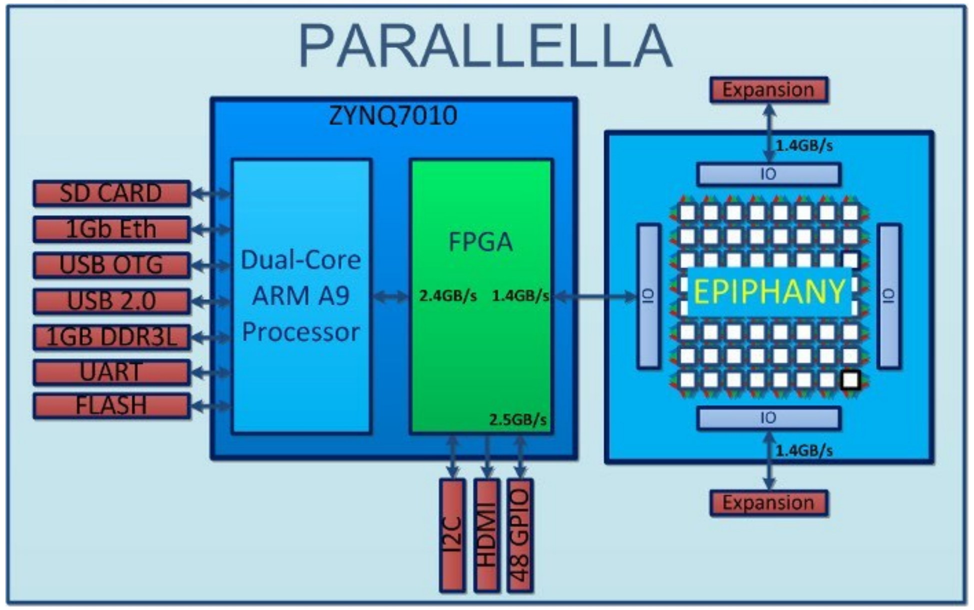
PERFORMANCE PORTABILITY ON NOVEL ENERGY EFFICIENT ARCHITECTURES: FEASIBLE OR FANTASY?

Nick Brown, EPCC

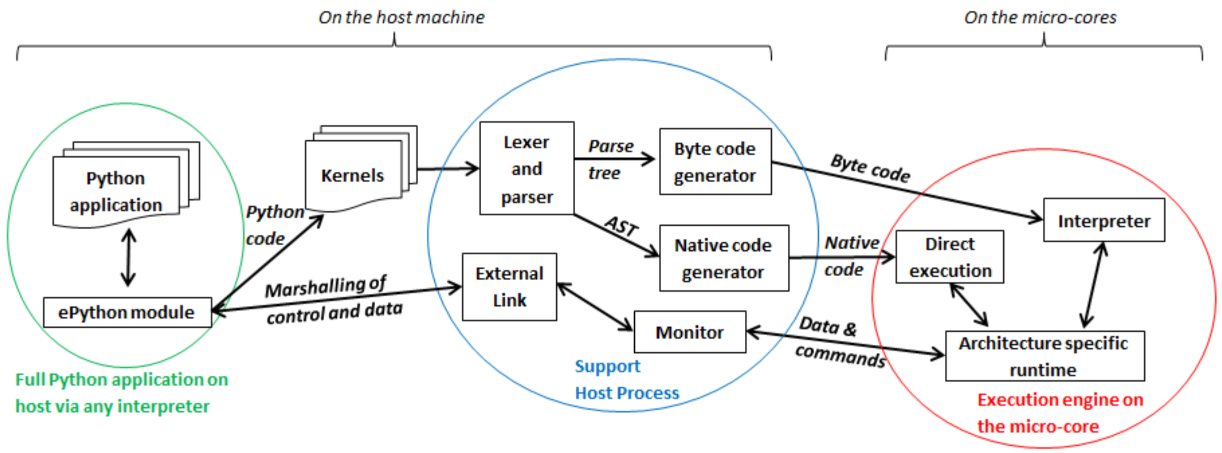
n.brown@epcc.ed.ac.uk

Talking about work done by lots of people including Gabriel Rodriguez-Canal, Jake Davies, David Kacs, Nicolai Stawinoga, Tobias Grosser, Sasha Lopoukhine, Anton Lydike, Emilien Bauer, George Bisbas

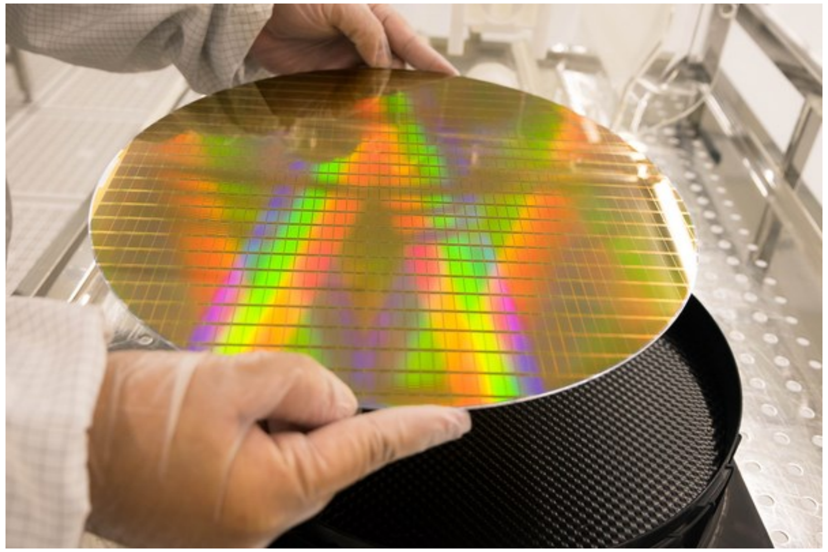
Back to 2014…..

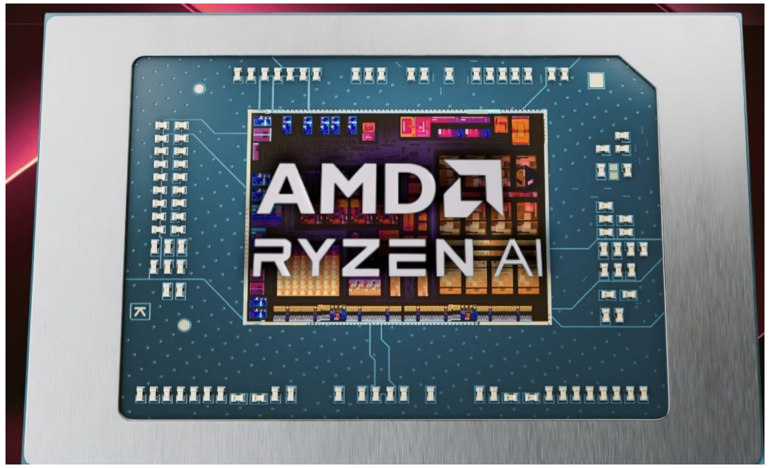


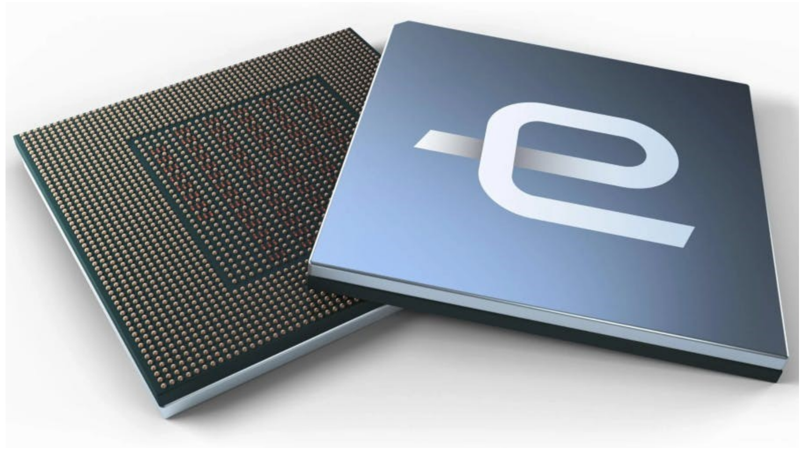
- My first involvement with novel, energy efficient, architectures was over a decade ago
  - And I learnt a very valuable lesson!
- Two main challenges
  - Required a specific killer app/use-case
  - Need to be easy to program

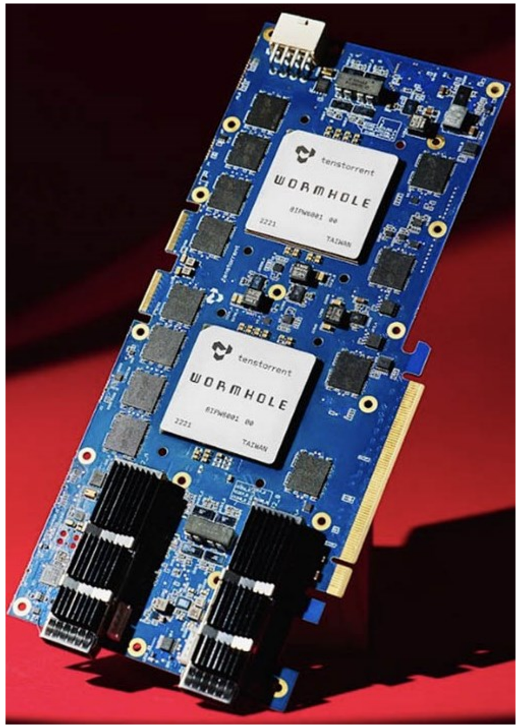


Times change









- AI/ML (theoretically) solves these challenges
  - A very strong killer app, not just for customers but also investors!
  - To provide Tensorflow / PyTorch only a small number of fundamental operations needed to be supported

But what about if we want to use these for scientific computing?

The challenge of connecting one world with another….

Software

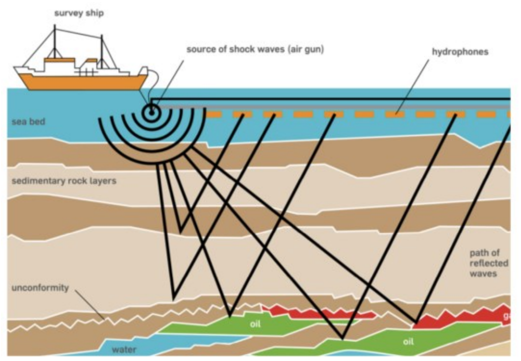
TensorFlow tvm Firedrake PSyclone flang

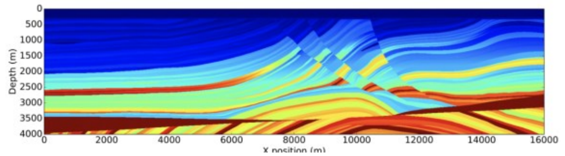
How to compile specialised software across a range of hardware?

General-purpose **Hardware** Specialised

Devito

Imperial College London





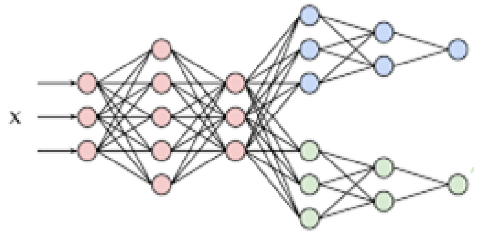
- < 50,000 lines of code
- Compiler implemented in Python
- Uses three IRs to compile
- Applies many optimisations, e.g. for loops and parallelism
- Support for GPUs & distributed memory, no support for hardware accelerators

PSyclone

UKRI Science and Technology Facilities Council

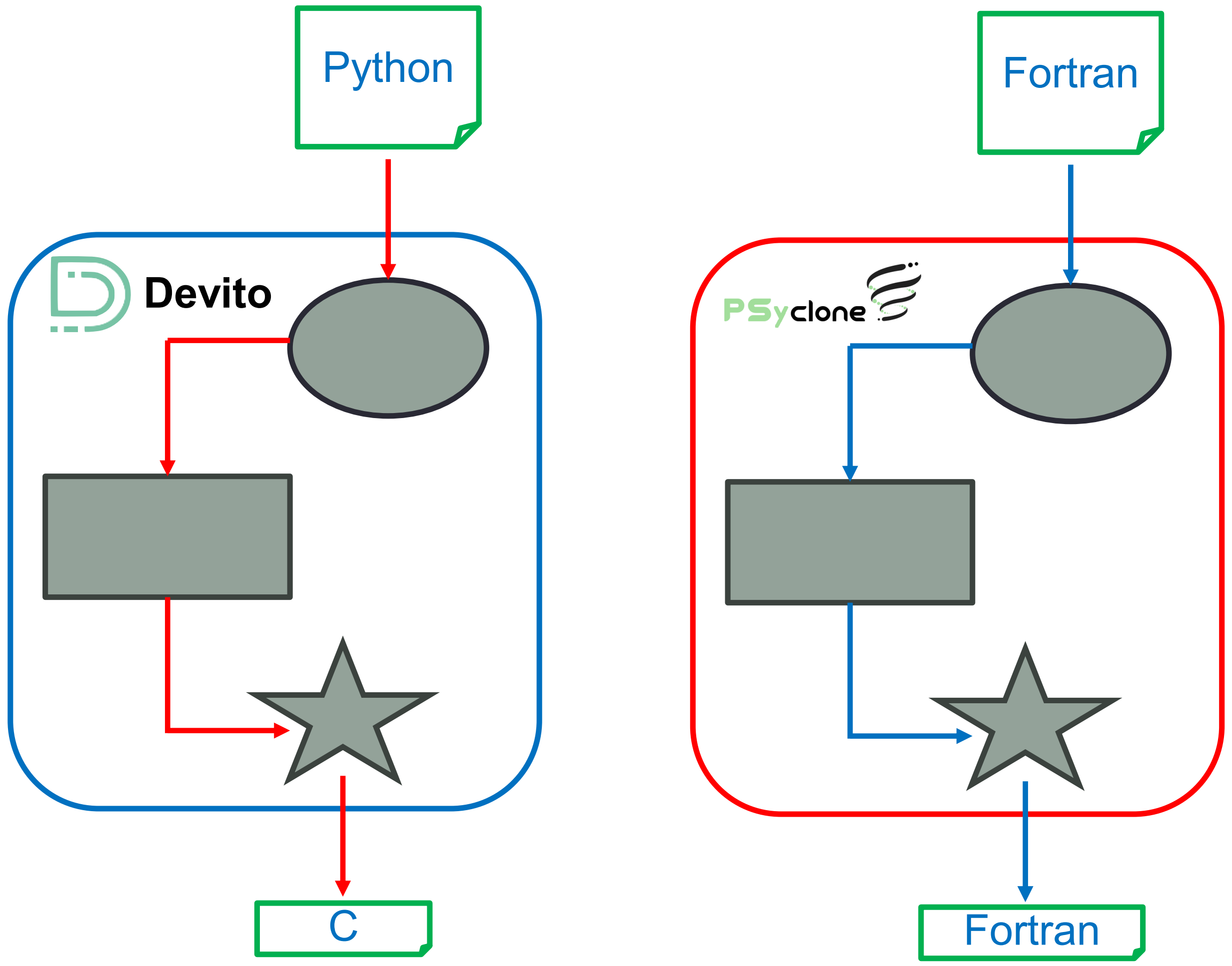
- Approx. 50,000 lines of code
- Programmer writes code in Fortran
- Compiler implemented in Python with lots of optimisations
- Uses one IRs to compile
- Support for GPUs and distributed memory, no support for hardware accelerators

TensorFlow



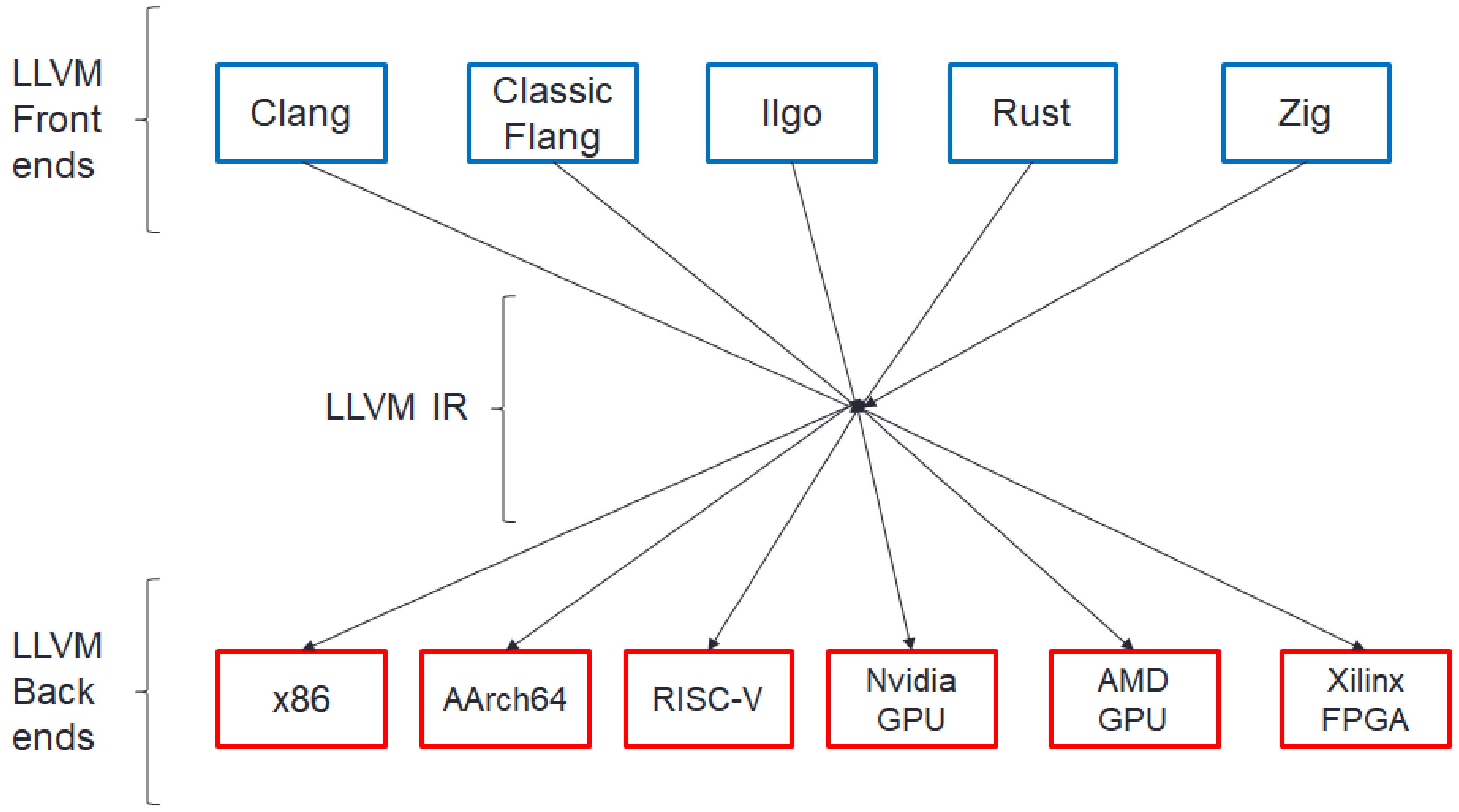
- > 2,500,000 lines of code
- Compiler implemented in Python & C++
- Uses two IRs with > 500 different types of expressions
- Applies many classical loop optimizations
- Great Performance & Support for custom hardware: TPU

A problem of siloing



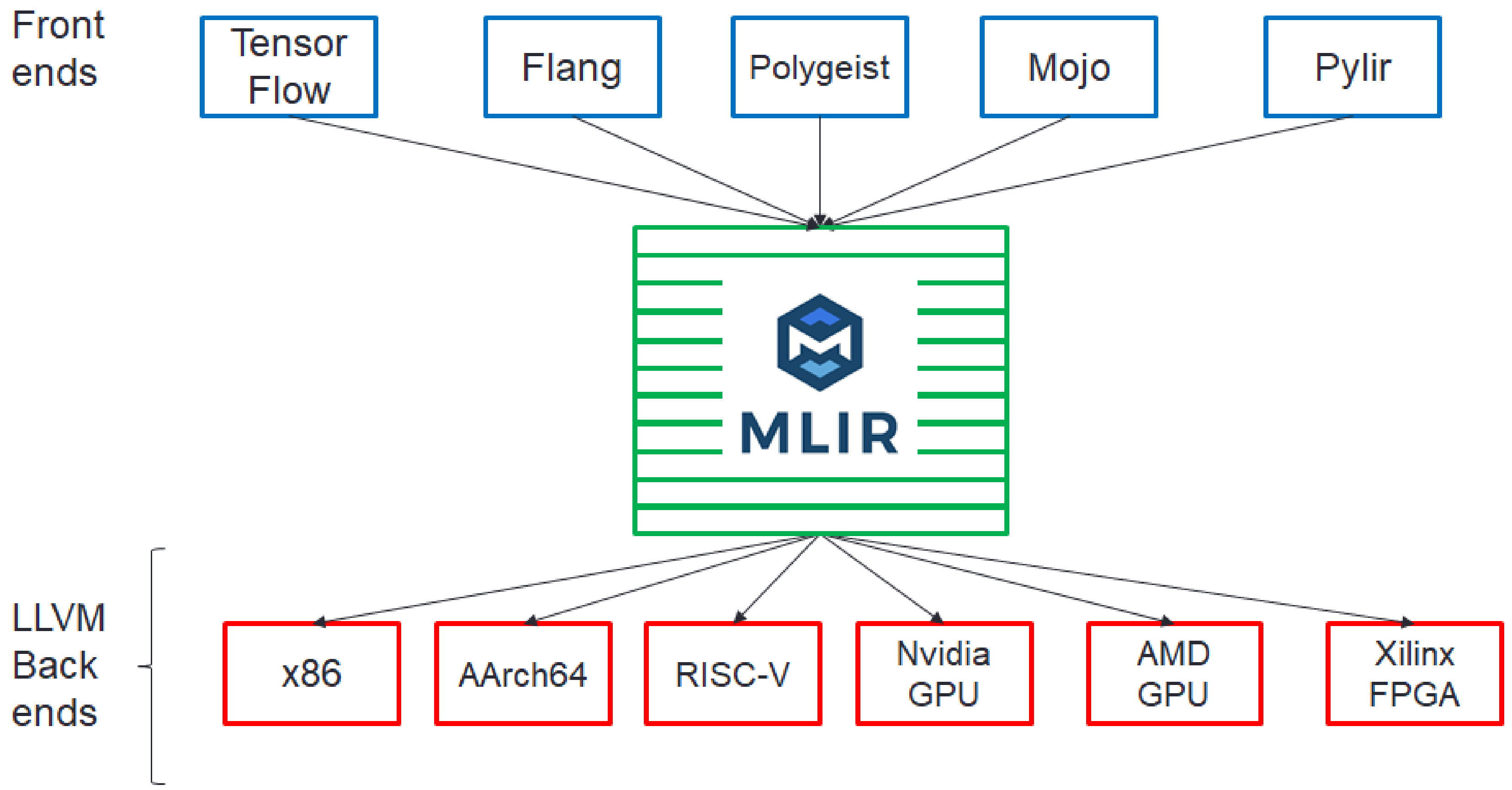
- Entirely separate compilation stacks
  - However, lots of similar activities being undertaken and duplication, even though specifics are different
- Everybody loses here!
  - Risk for users, will my DSL compiler still be around in 5 years and support the latest supercomputers?
  - DSL developers must invest lots of effort to build and maintain their stacks
    - Especially when then targeting new architectures

TensorFlow had a different idea….



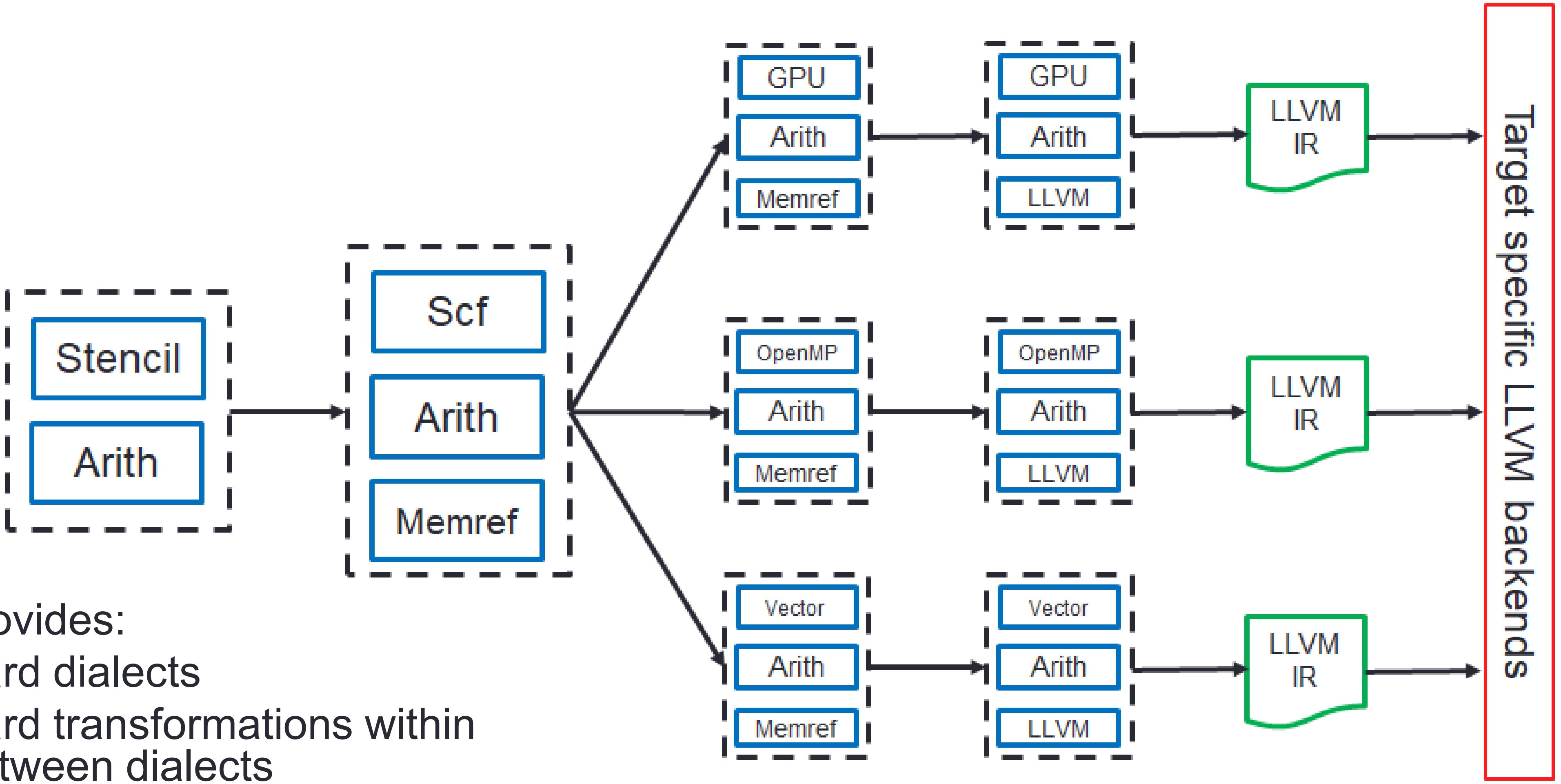
- Sensible to use LLVM due to all the backends for a variety of architectures
- But LLVM-IR is low level
- Front ends again have duplication between them

TensorFlow had a different idea….



- MLIR is a subproject of LLVM
- Comprises dialects and transformations
  - And a framework for developing these
- Lower between dialects, and these can be mixed
- Entry point can be much higher level

Example MLIR lowering



MLIR provides:

- Standard dialects
- Standard transformations within and between dialects
- Framework for bespoke dialects & transformations

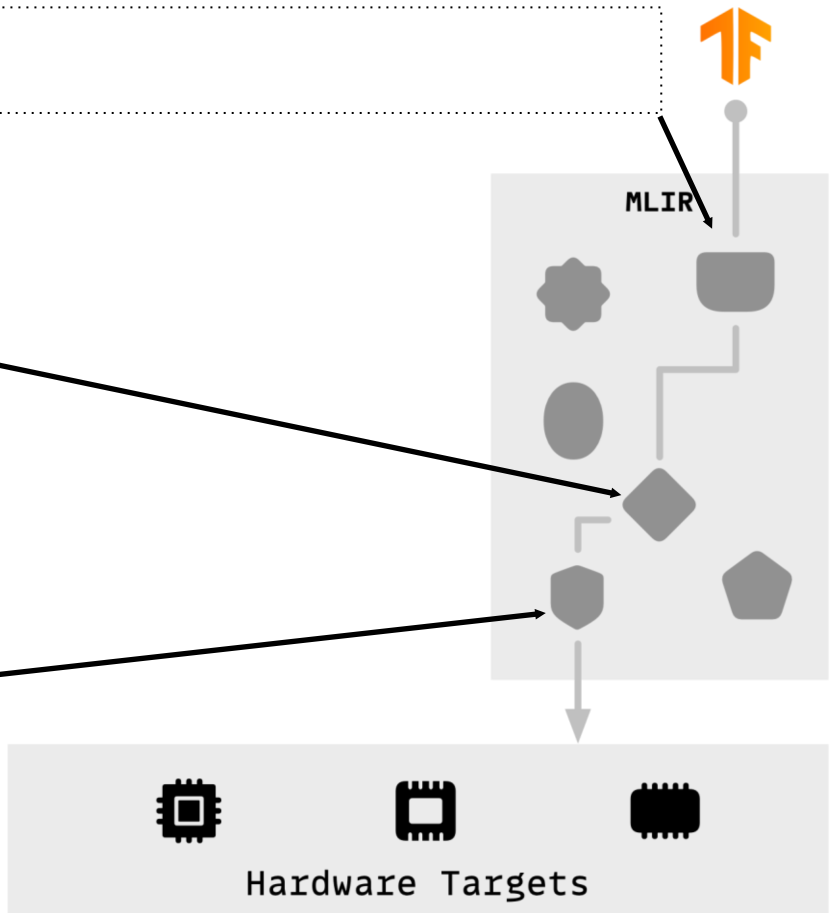
MLIR — Multi-Level Intermediate Representation

Progressive Lowering from Application Domain to Hardware

```
%x = tf.Conv2d(%input, %filter) {strides: [1,1,2,1], padding: "SAME", dilations: [2,1,1,1]}
    : (tensor<*xf32>, tensor<*xf32>) -> tensor<*xf32>
```

```
affine.for %i = 0 to %n {
 …
 %sum  = addf %a, %b : f32
 …
}
```

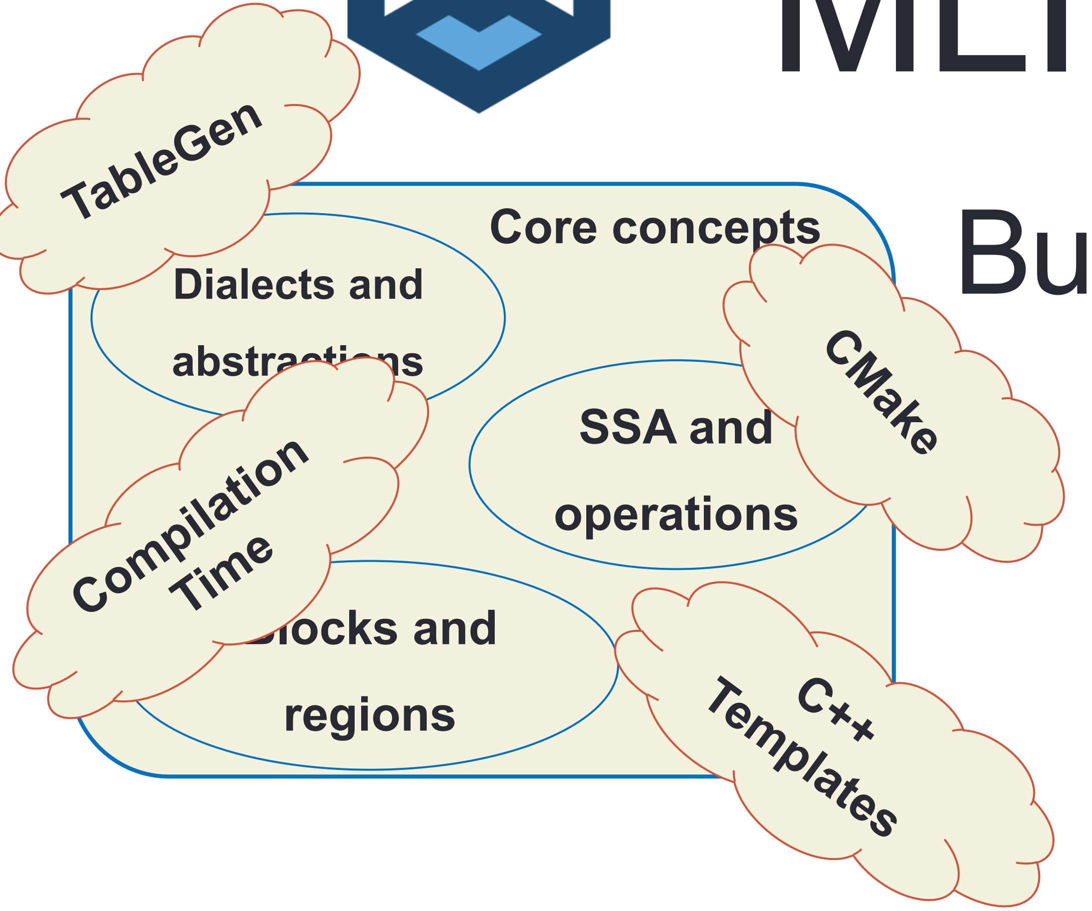
```
gpu.launch(%gx,%gy,%c1,%lx,%c1,%c1) {
 ^bb0(%bx: index, %by: index, %bz: index,
    %tx: index, %ty: index, %tz: index,
    %num_bx: index, %num_by: index, %num_bz: index,
    %num_tx: index, %num_ty: index, %num_tz: index)
 …
 %sum  = addf %a, %b : f32
 …
}
```



MLIR is really nice!

MLIR is really nice!

But it’s very complicated



xDSL: Raising productivity with MLIR

xDSL is a Python implementation of MLIR concepts

- Focus on *approachability*
- *Reuse* of existing concepts implemented in a simpler way
- *Expands* on MLIR concepts
- Making compiler frameworks *interoperable*
- Over 600,000 downloads on PyPI

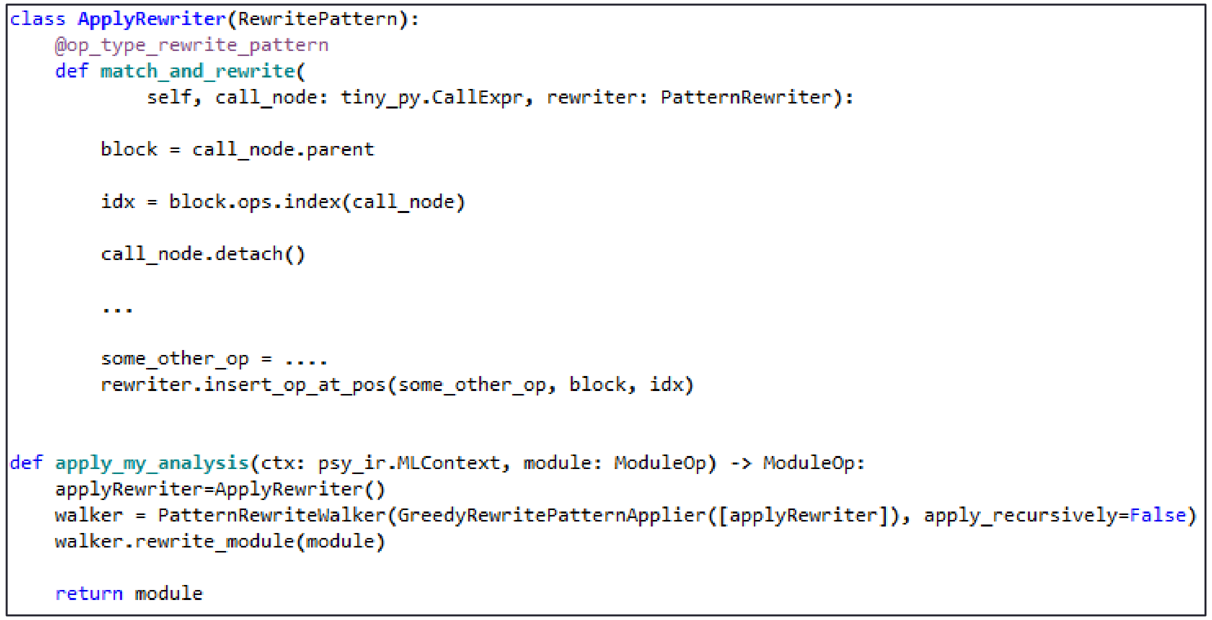
https://xdsl.dev
https://github.com/xdslproject/xdsl

ExCALIBUR 10

epcc

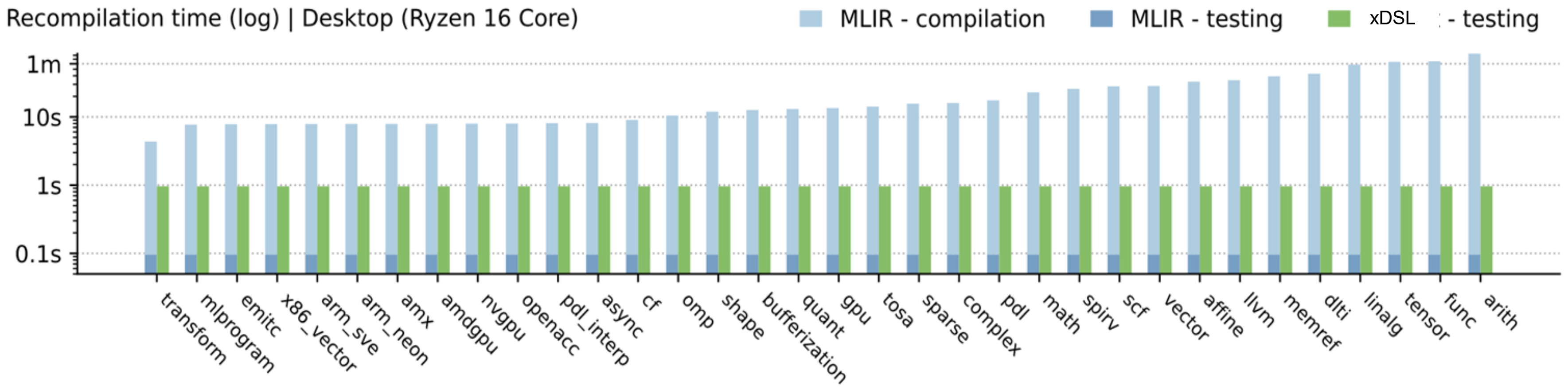
THE UNIVERSITY OF EDINBURGH

Benefits: Programmer productivity & development time



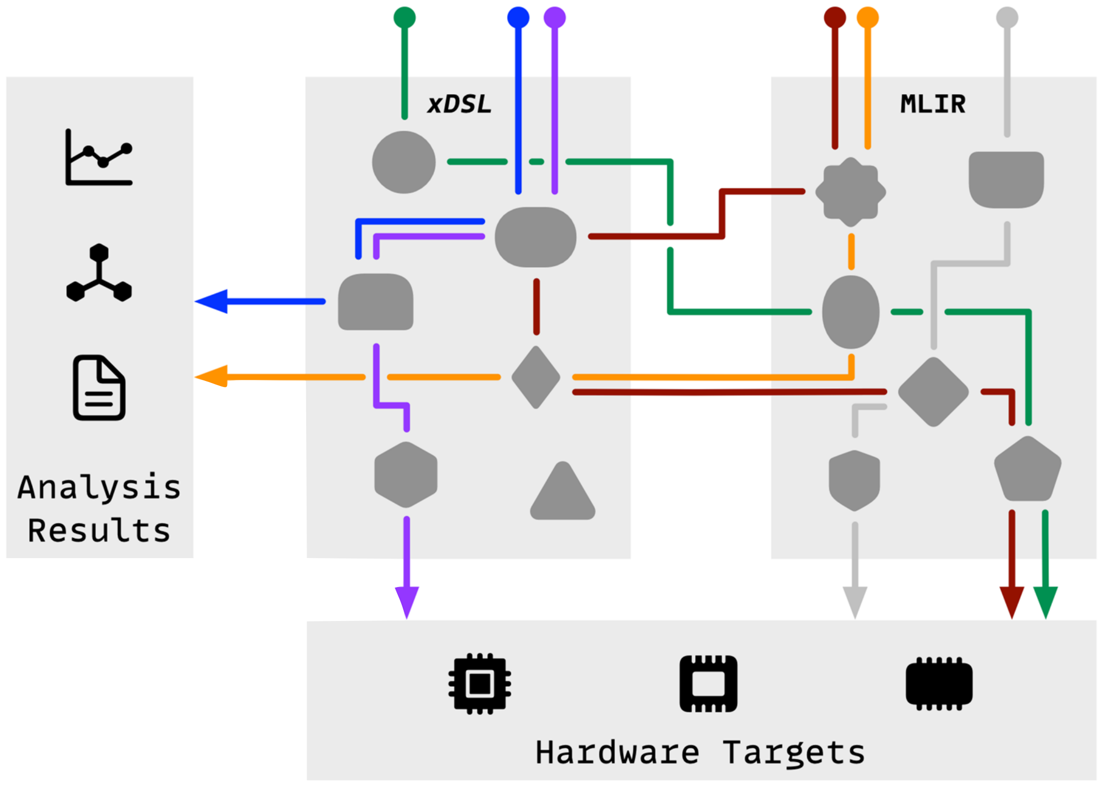
```python
class ApplyRewriter(RewritePattern):
    @op_type_rewrite_pattern
    def match_and_rewrite(
            self, call_node: tiny_py.CallExpr, rewriter: PatternRewriter):

        block = call_node.parent

        idx = block.ops.index(call_node)

        call_node.detach()

        ...

        some_other_op = ....
        rewriter.insert_op_at_pos(some_other_op, block, idx)


def apply_my_analysis(ctx: psy_ir.MLContext, module: ModuleOp) -> ModuleOp:
    applyRewriter=ApplyRewriter()
    walker = PatternRewriteWalker(GreedyRewritePatternApplier([applyRewriter]), apply_recursively=False)
    walker.rewrite_module(module)

    return module
```

- Same underlying concepts as MLIR with operations, regions, blocks, attributes etc
- But expressed in Python



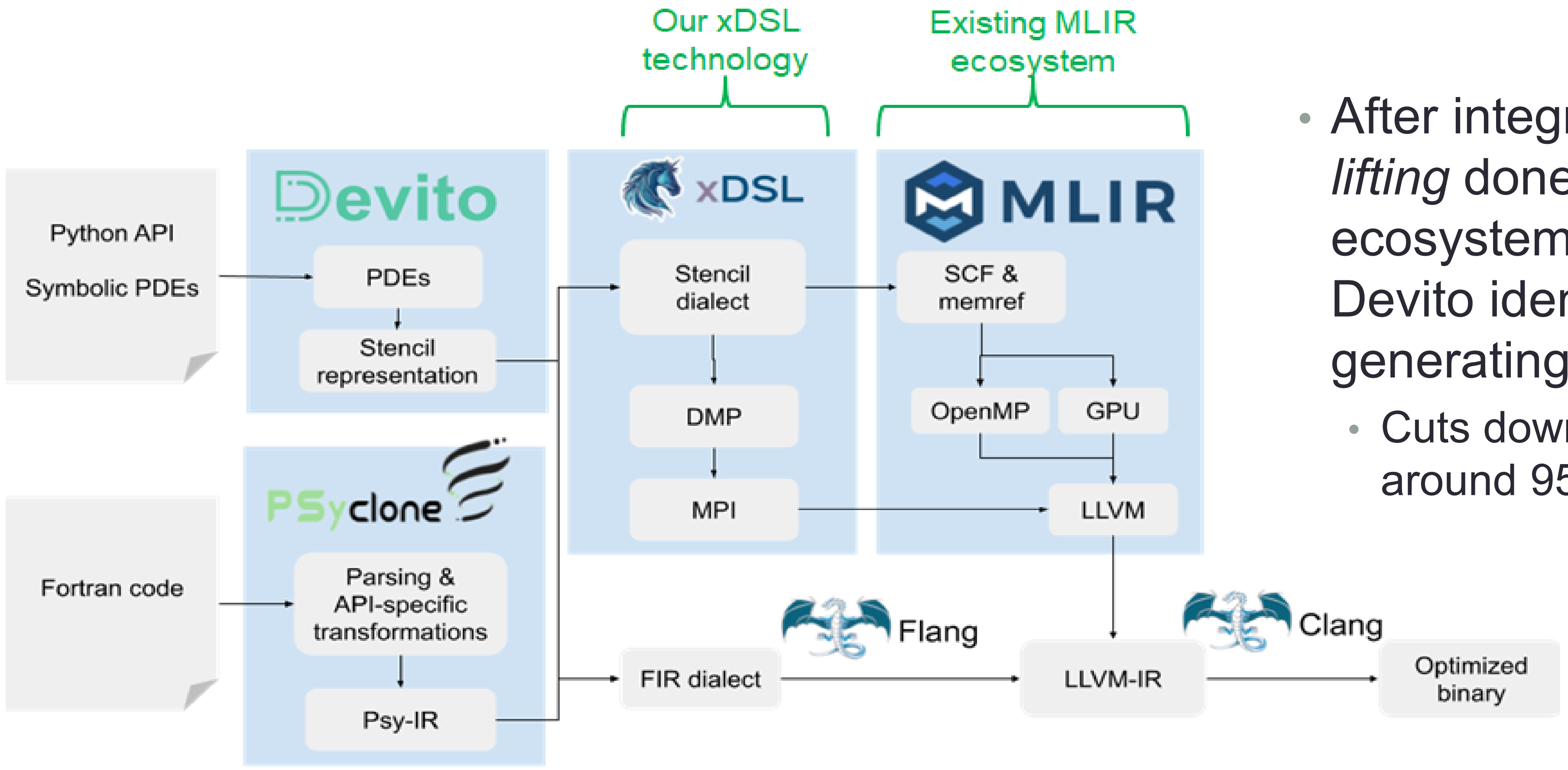
xDSL: a ***Sidekick*** to MLIR

Making the MLIR ecosystem accessible and extensible from Python



- A Python native compilation framework
- Fast prototyping of new MLIR concepts and ideas
  - Such as new dialects and transformations
    - Such as the new MLIR MPI dialect
- Integration of Python-based DSLs into MLIR

A common DSL ecosystem



- After integration, majority of *heavy lifting* done in the xDSL common ecosystem, with PSyclone and Devito identifying stencils and generating IR in the stencil dialect
  - Cuts down PSyclone codebase by around 95%

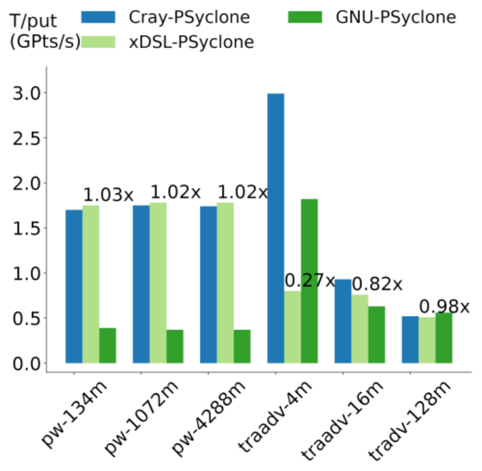
- Lots of foundational work on new dialects and transformations
  - For instance, the MPI dialect, which has now been merged into MLIR

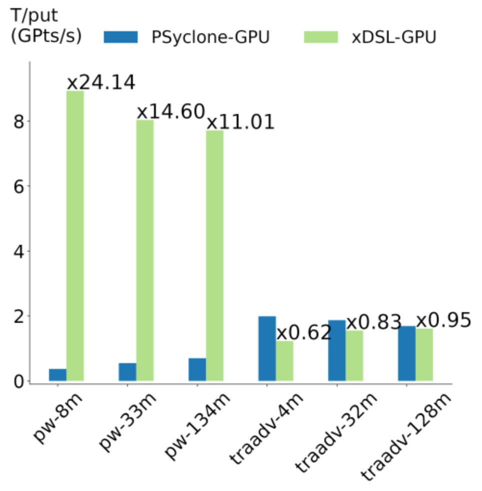
Single-node on ARCHER2

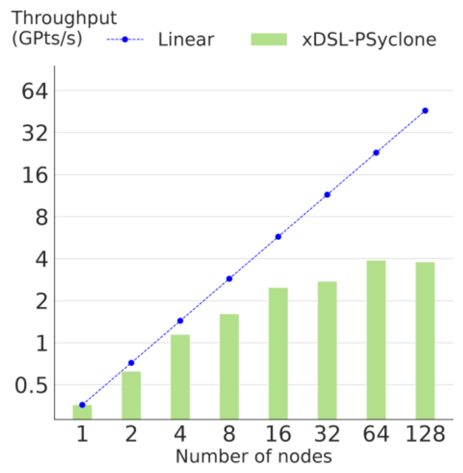
GPU on Cirrus (V100)

Strong scaling on ARCHER2

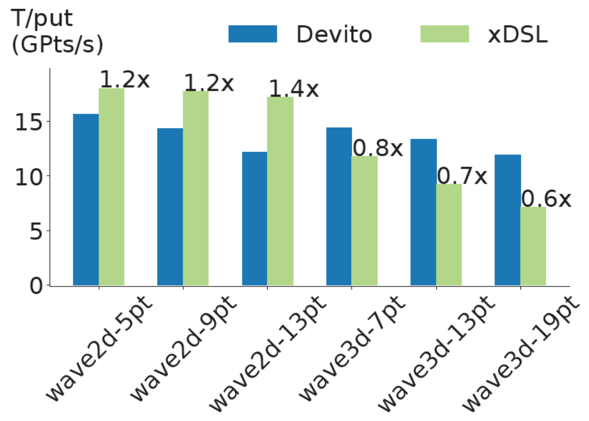
PSyclone

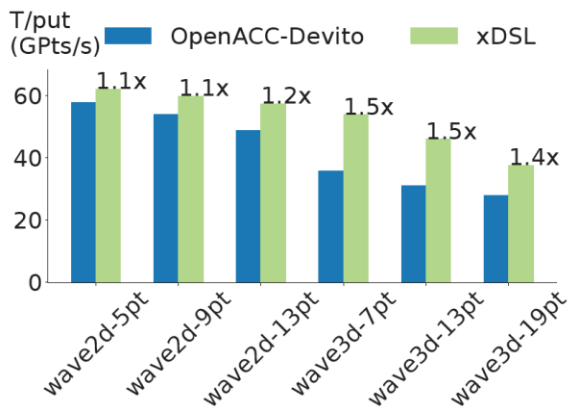


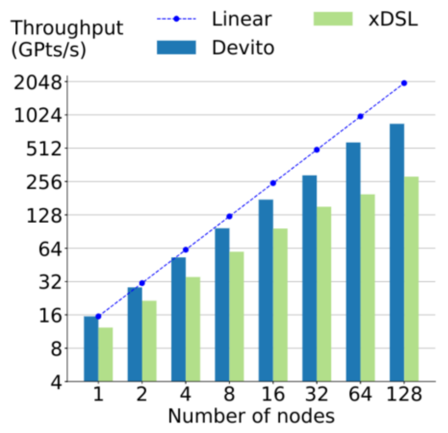




Devito



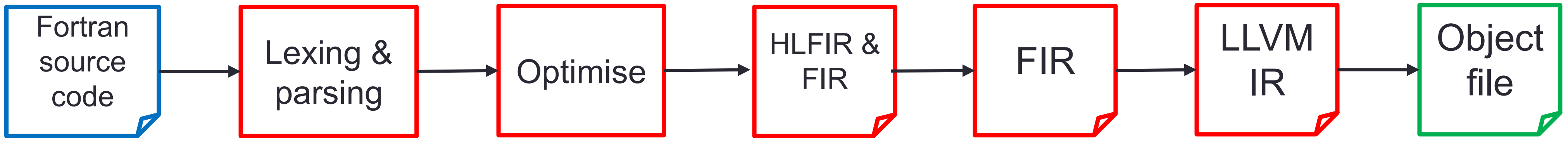




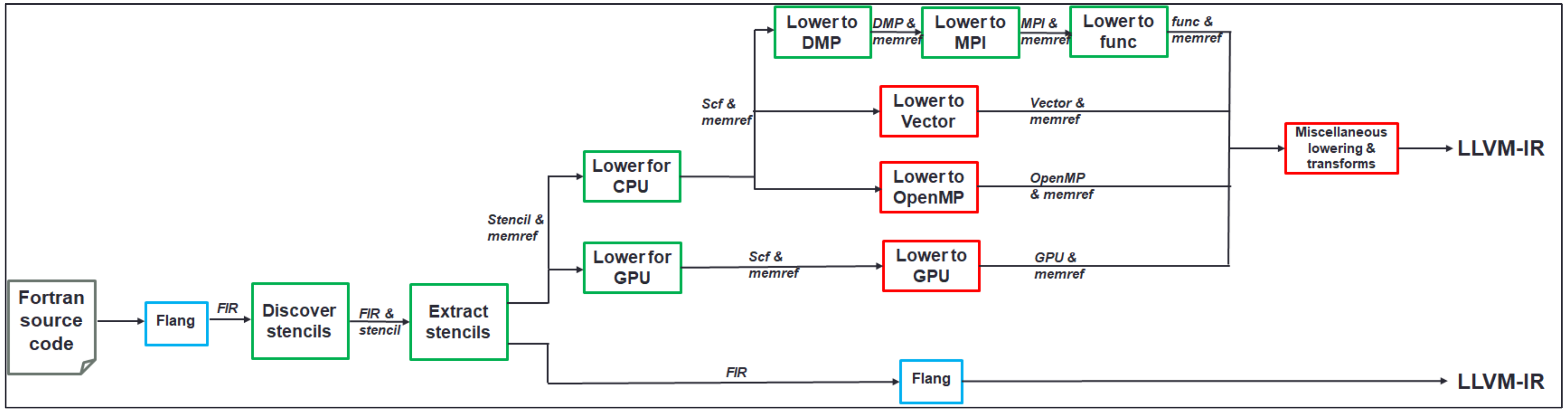
https://arxiv.org/pdf/2404.02218.pdf

epcc

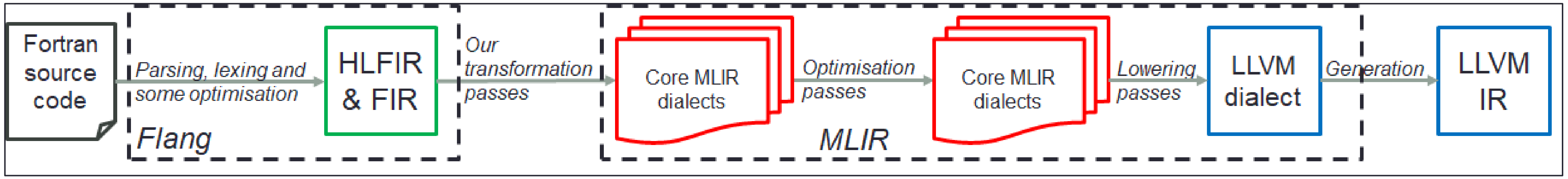
Driving via Flang



1.



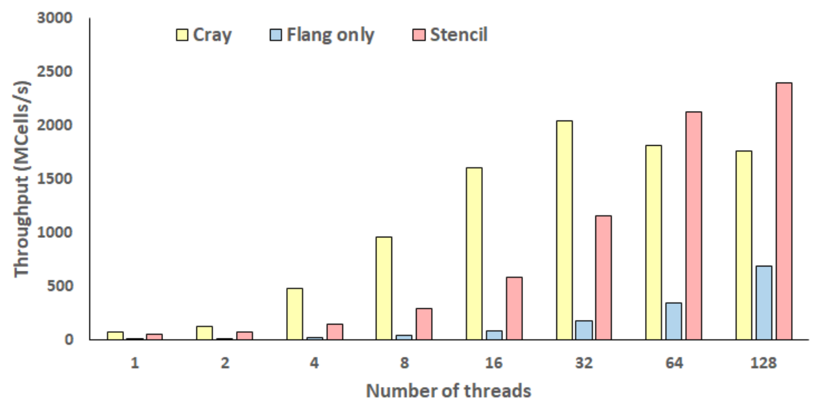
2.



Performance benefits on CPU and GPU….

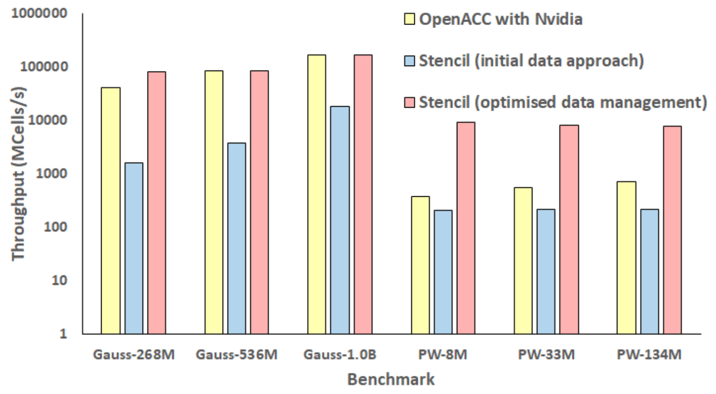
Stencil flow

Multi-threaded



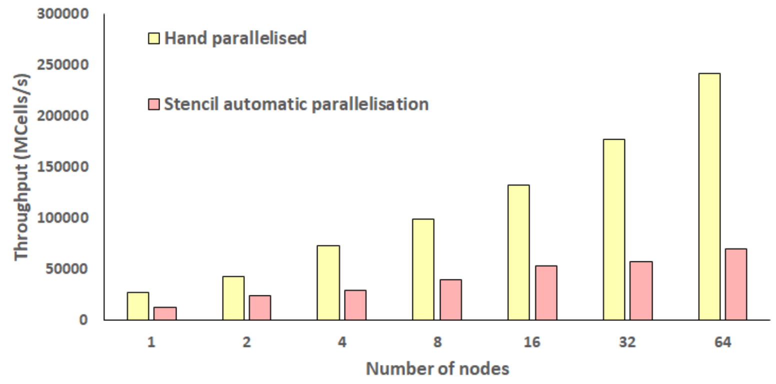
PW advection benchmark with 2.1 billion grid points

GPU



On V100 GPU, 22.11 of Nvidia SDK

Distributed memory



Gauss Seidel benchmark with global problem size 17 billion grid cells

https://arxiv.org/pdf/2310.01882

General flow

	Benchmark Runtime (s)			
Benchmark	**Our approach**	**Flang v20**	**Cray**	**GNU**
ac	10.23	11.89	8.67	31.43
linpk	5.43	6.24	5.79	4.81
nf	10.69	10.29	7.72	7.43
test_fpu	72.41	110.80	32.56	76.99
tfft	52.33	48.90	61.65	115.86
jacobi	249.08	277.67	109.89	232.62
pw-advection	86.47	205.33	47.28	192.05
tra-adv	124.72	141.95	79.38	116.71

Running over a single core of ARCHER2, a Cray-EX (AMD Rome)

Operation	**Our approach runtime (s)**		**Flang v20 runtime (s)**
	serial	**threaded**	**serial**
transpose	214.48	40.75	272.38
matmul	43.12	11.85	45.71
dotproduct	0.81	-	2.70
sum	1.63	-	1.65

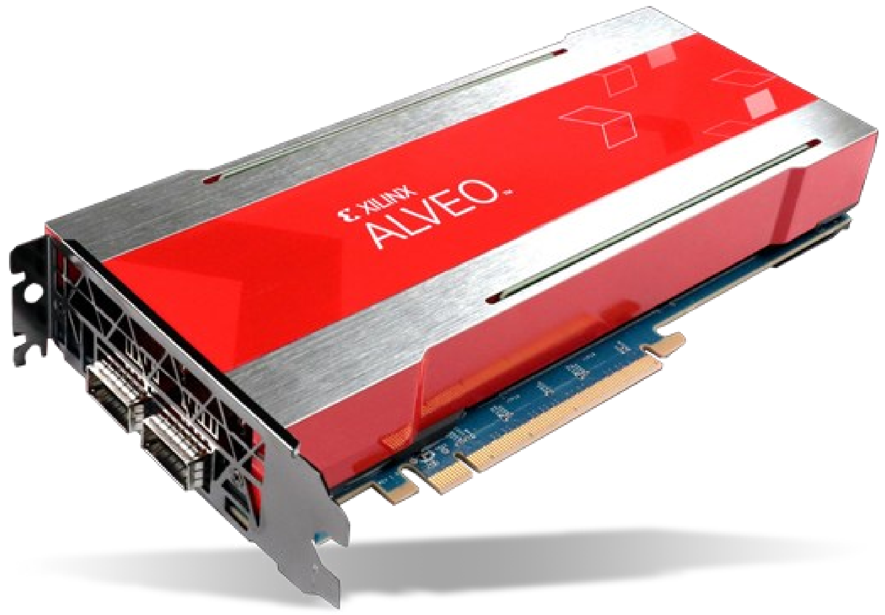
Running on ARCHER2, a Cray-EX (AMD Rome)

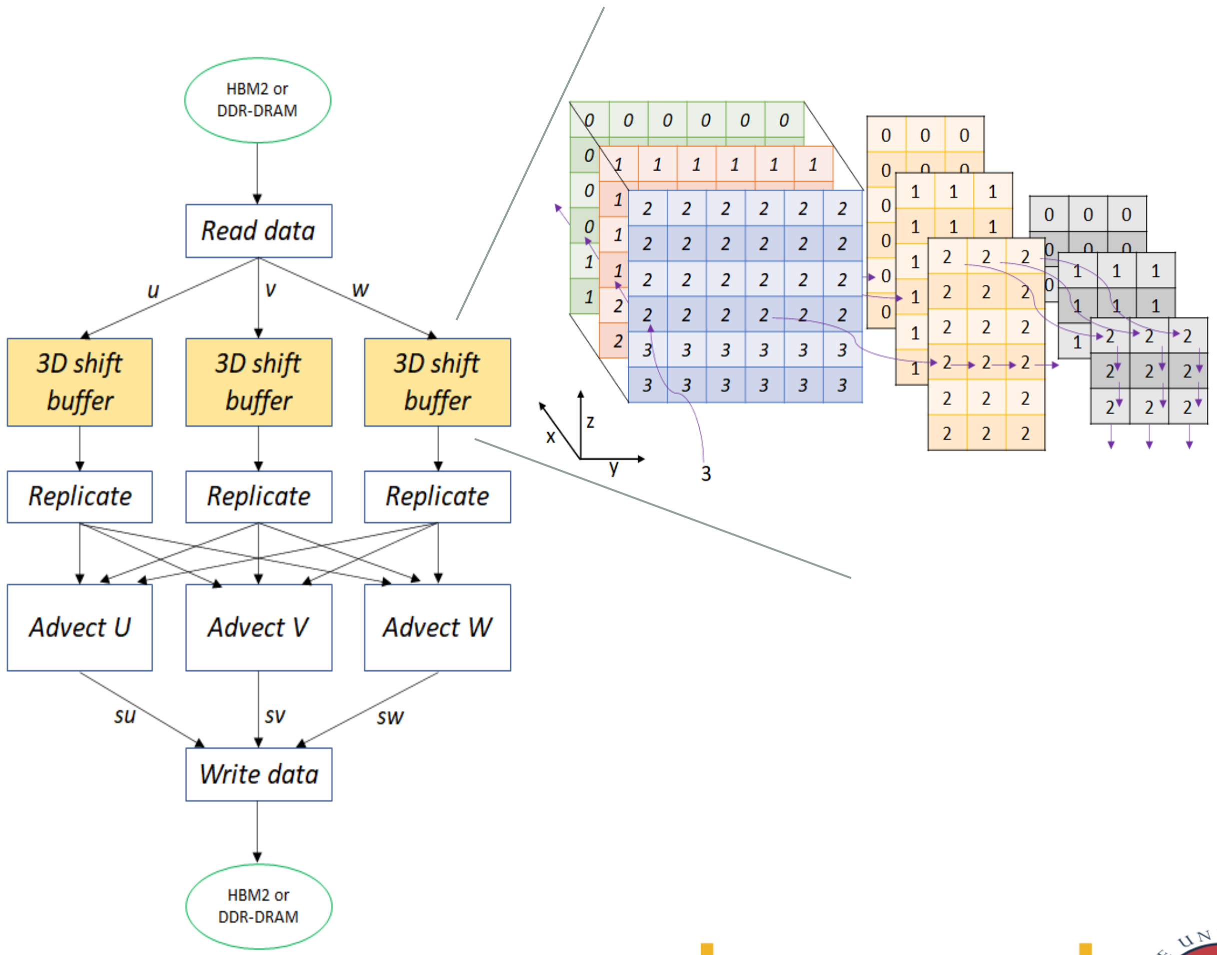
https://arxiv.org/pdf/2409.18824

epcc

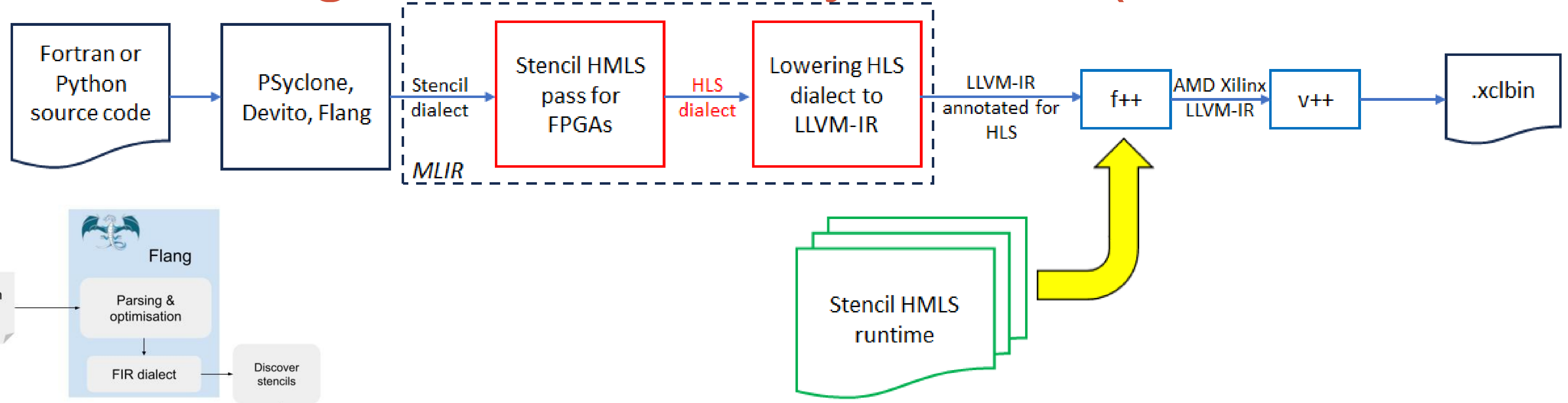
THE UNIVERSITY OF EDINBURGH

Targeting FPGAs for stencils using MLIR

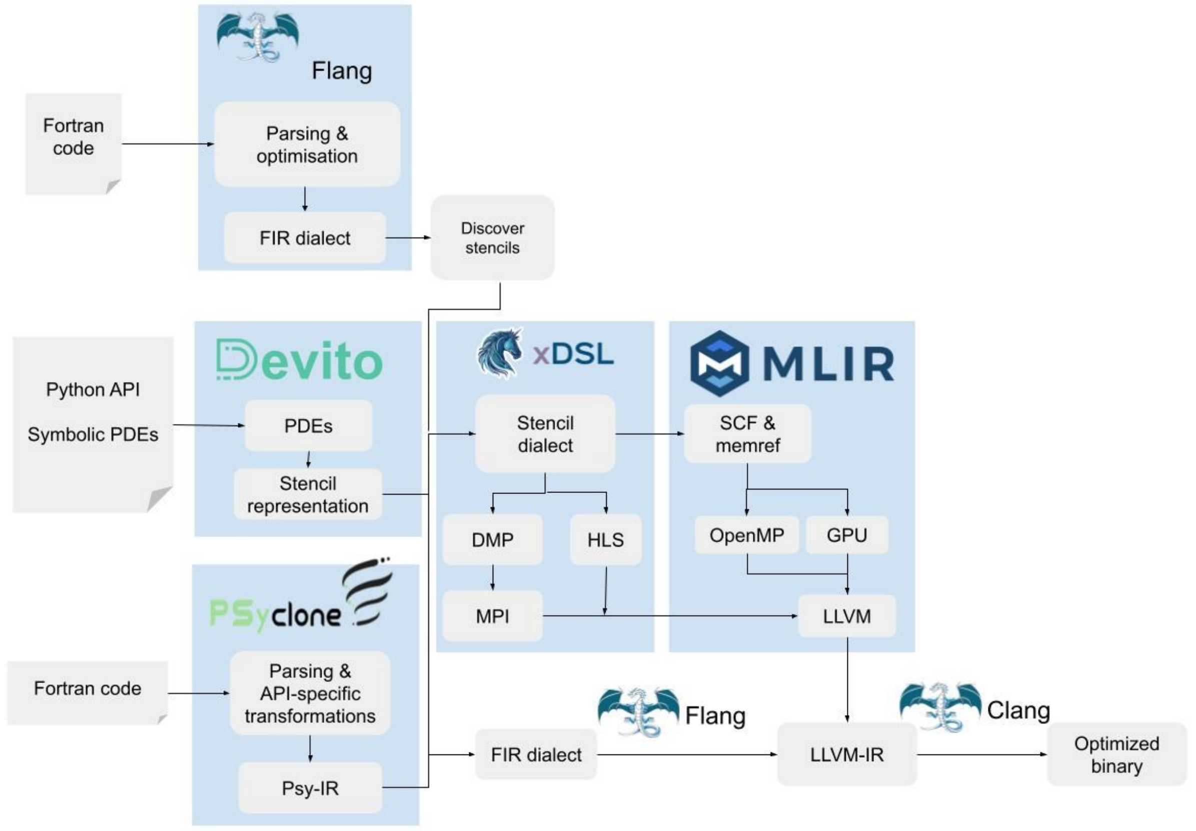




Stencil High Multi-Level Synthesis (Stencil-HMLS)



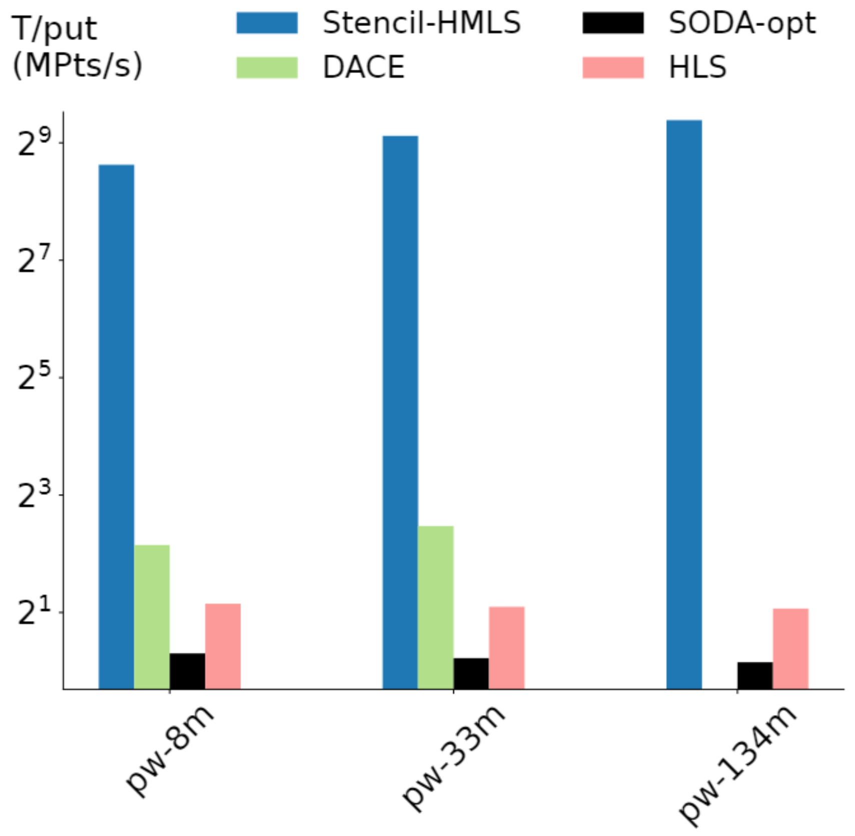
Fortran or Python source code
PSyclone, Devito, Flang
Stencil dialect
Stencil HMLS pass for FPGAs
HLS dialect
Lowering HLS dialect to LLVM-IR
MLIR
LLVM-IR annotated for HLS
f++
AMD Xilinx LLVM-IR
v++
.xclbin
Stencil HMLS runtime



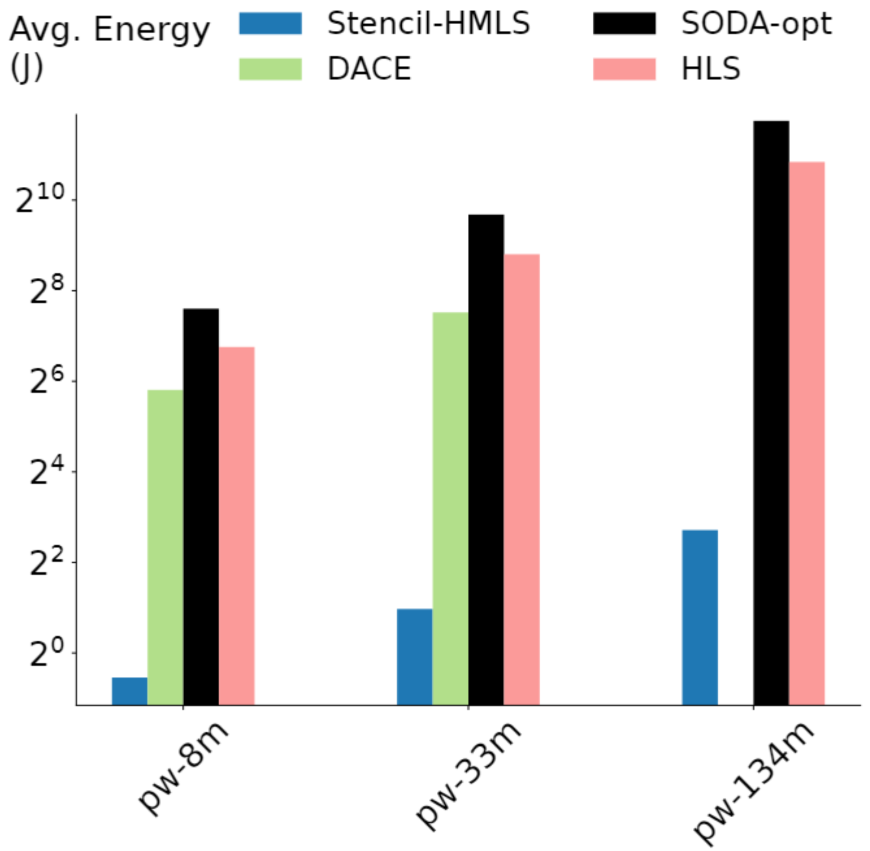
- Using the existing flow and dialects, we lower the stencil dialect (and others) to the HLS dialect
- Ultimately means that stencil codes written in any language can target FPGAs

Performance

(Higher is better)



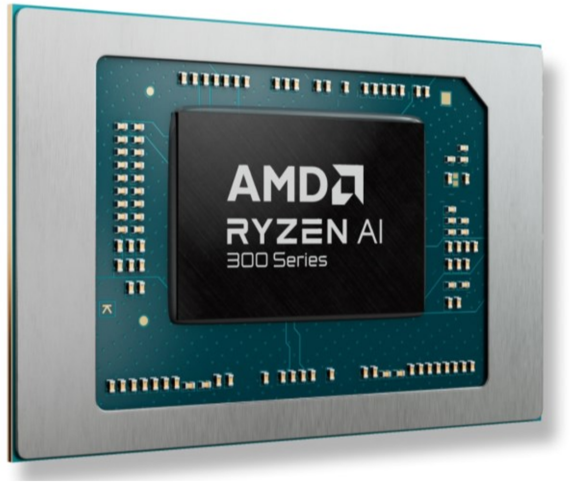
(Lower is better)



- For this benchmark, our approach is between 90 and 100 times faster than DaCE
- Lots more details at https://arxiv.org/pdf/2310.01914

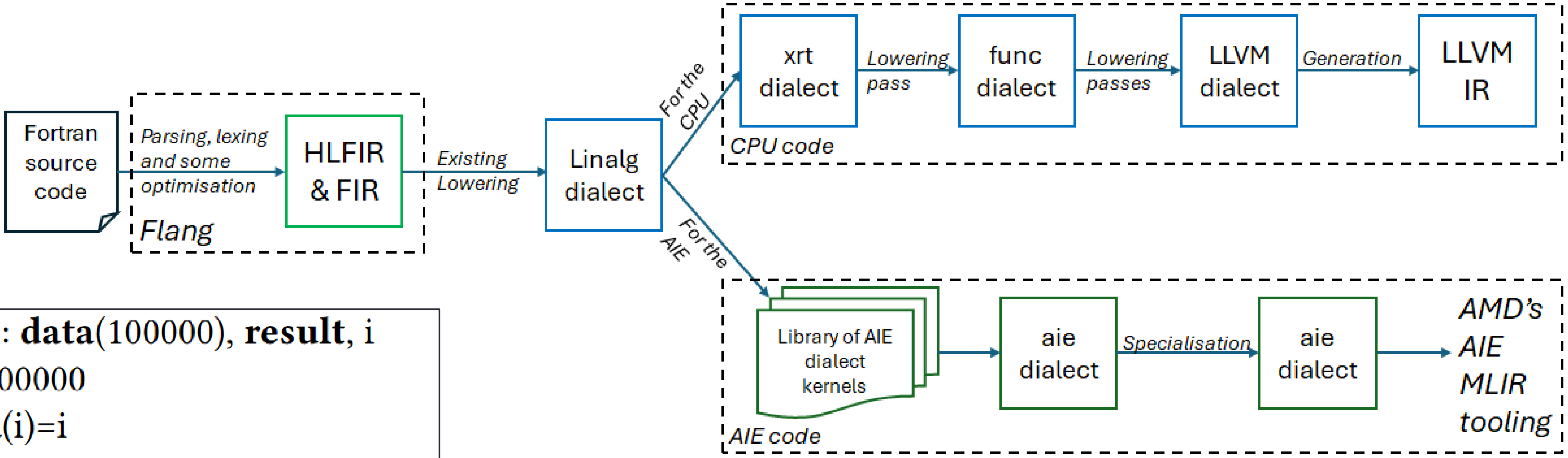
AMD’s AI Engines (AIEs)

- In late 2023 AMD released their Ryzen AI CPU, which contains their Neural Processor Unit which is a marketing term for an array of AIEs
- Very interesting, as a much more attractive proposition if these are already inside a CPU



- Current models of Ryzen AI contain an array of 20 AIEs, each AIE-ML contains 64KB and has five memory tiles each of 512KB
  - However Int32 and FP32 support have been removed compared to the AIEs in Versal, with BF16 provided instead
    - Int32 and FP32 are emulated so can be run on the NPU
- Direct programming via kernels in C++ using API and Riallto Python framework for the dataflow graph

Extending to seamlessly offload on the AMD's AIEs



```
integer :: data(100000), result, i
do i=1, 100000
    data(i)=i
end do
result=sum(data)
```

Data type	CPU runtime (us)	NPU first runtime (us)	NPU subsequent runtime (us)
int16	5473	2572	1353
int32	14032	2635*	1503*
bfloat16	815194	2626	1357
float32	17566	3901*	1471*

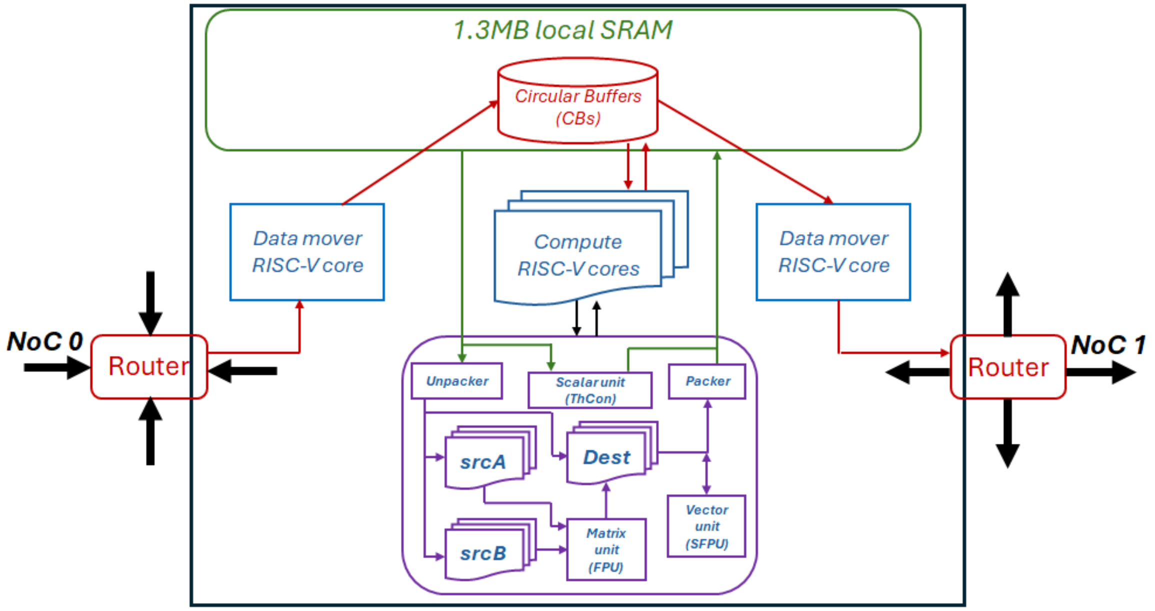
Table 4: Runtime (in microseconds) of *matmul* Fortran intrinsic with a problem size of 256x256x512 elements

- This is all hidden from the programmer, they simply recompile their code and the intrinsics will be run on the Ryzen-AI's AIE array if appropriate

https://arxiv.org/pdf/2502.10254

Tenstorrent Tensix architecture

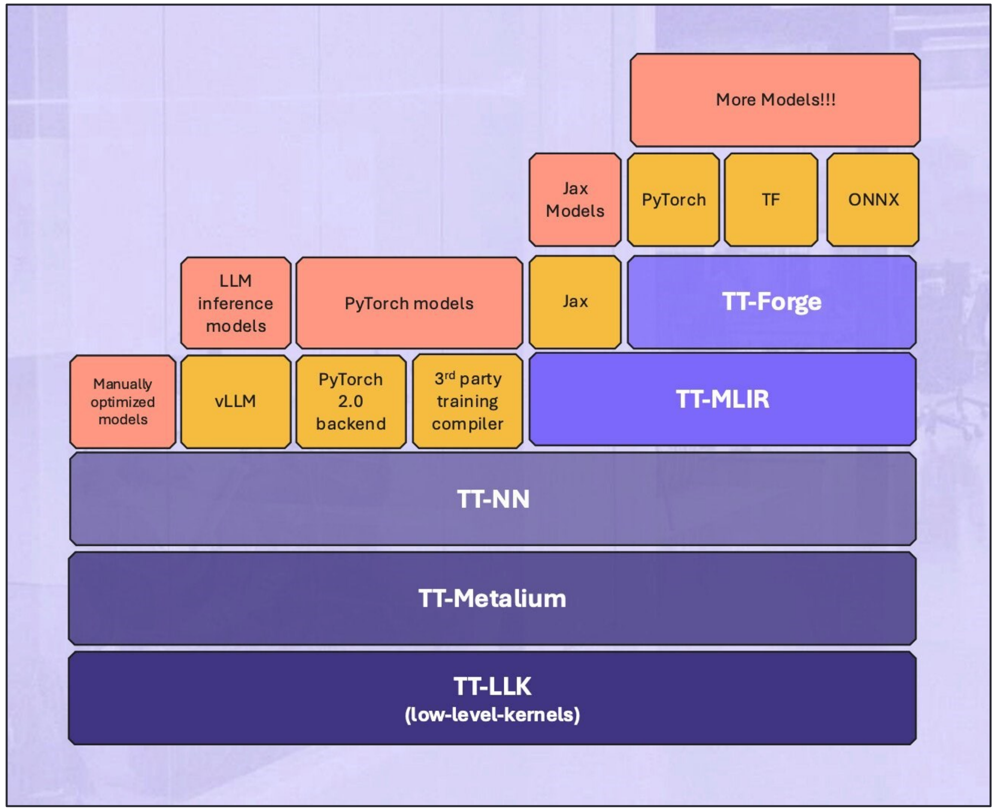
- Range of PCIe accelerator cards
- On their original Grayskull accelerator we demonstrated comparable performance to a 24-core Platinum CPU for a stencil benchmark, but at five times less energy

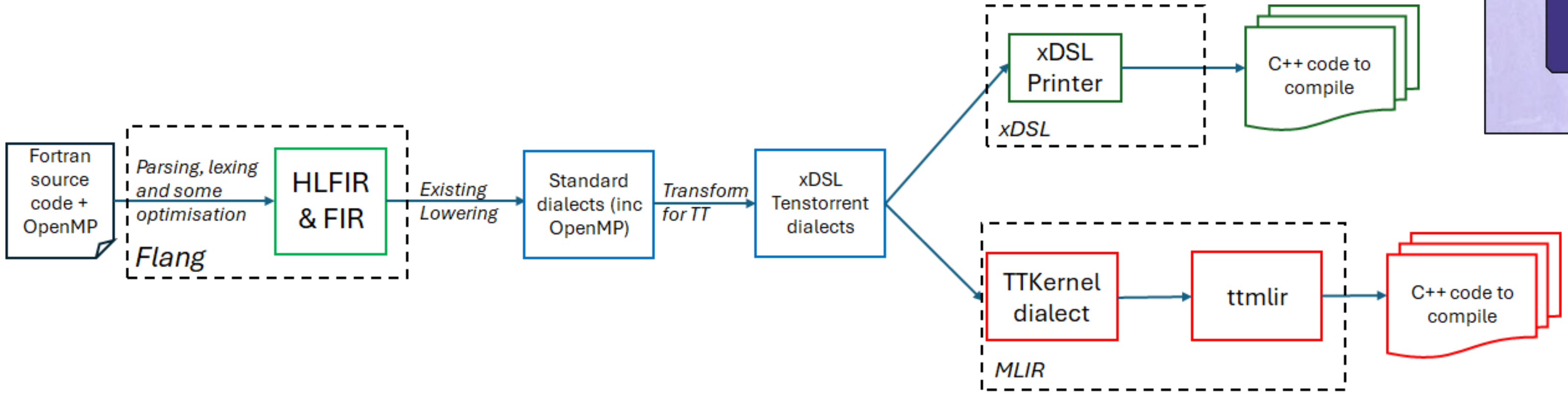


- Architecture designed for AI, but the decoupling of memory from compute (and wide matrix & vector units) have significant potential for HPC
  - But people won't port their codes to their bespoke C++ based API!

Driving Tenstorrent from Fortran via MLIR

- They have a nice software stack approach
  - TT-NN are all the primatives required for AI/ML workloads, built upon the generalised TT-Metalium framework which itself calls into low level kernels
    - Can hook code into the stack at any level
    - Crucially, they have support for MLIR which can either call TT-NN kernels or directly drive the TT-Metalium layer
  - All entirely open with Tenstorrent committing to working in an *open first* fashion





OpenMP + Fortran on Tenstorrent

```
subroutine saxpy(a, x, y, n)
  ...
  do i=1, n
    y(i) = a * x(i) + y(i)
  end do
end subroutine
```

Just normal Fortran, this will run on the CPU

But imagine we had a Wormhole card plugged into the machine……

OpenMP + Fortran on Tenstorrent

```
subroutine saxpy(a, x, y, n)
  ...
  !$omp target
  do i=1, n
    y(i) = a * x(i) + y(i)
  end do
  !$omp end target
end subroutine
```

Will offload this loop to a target device (in this case the Tenstorrent Wormhole).

Without any further information, *target* will run the loop on a single data movement baby RISC-V core. With data movement and kernel launch all happening seamlessly.

OpenMP + Fortran on Tenstorrent

```
subroutine saxpy(a, x, y, n)
  ...
  !$omp target simd simdlen(32)
  do i=1, n
    y(i) = a * x(i) + y(i)
  end do
  !$omp end target simd
end subroutine
```

simdlen is optional and will set the tile size (a sensible default is used if omitted)

The simd directive tells the compiler that the loop iterations are independent and can be vectorised.

Here we offload this to the Tensix unit via the compute core. This is very different than the previous code, as iteration data is tiled by data in core, fed to compute core, and results read back by data out core and written to DDR

OpenMP + Fortran on Tenstorrent

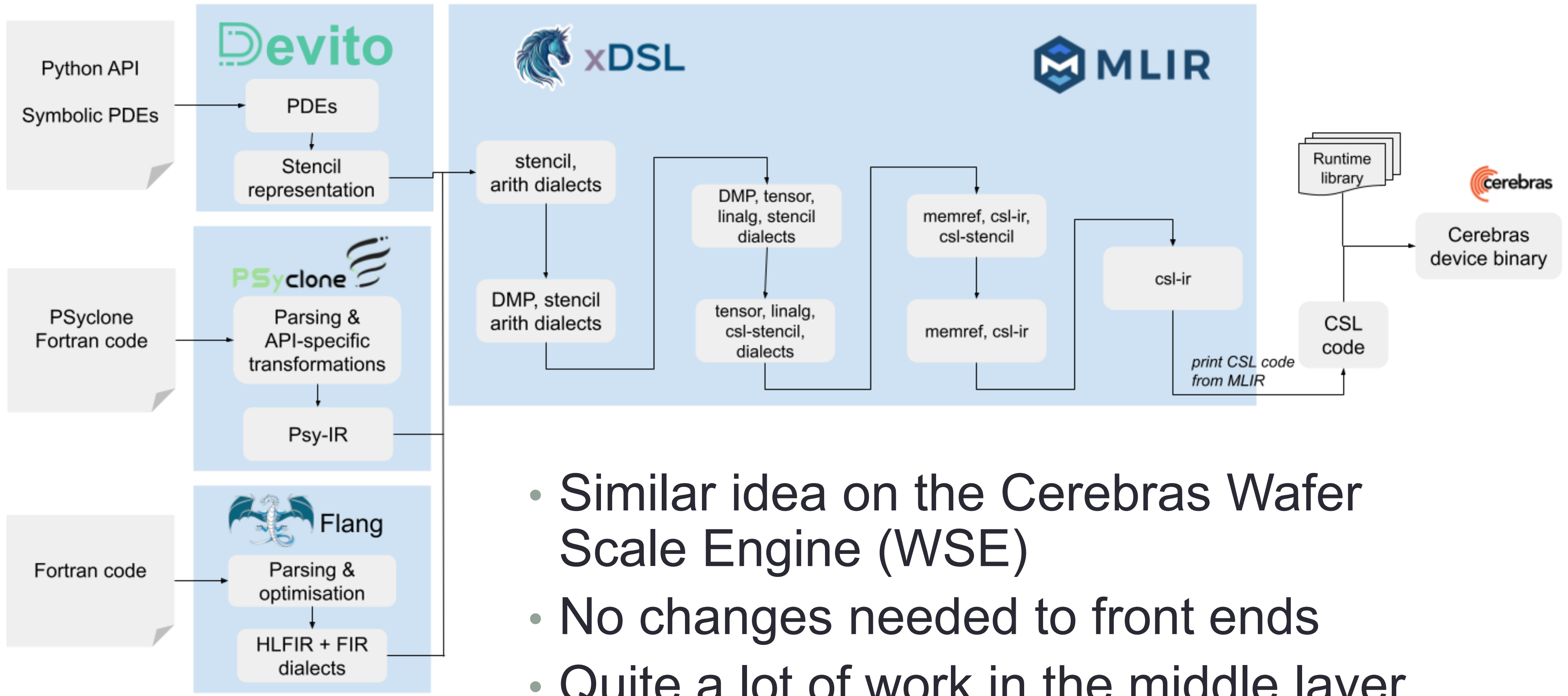
```
subroutine saxpy(a, x, y, n)
  ...
  !$omp target parallel do num_threads(10) simd
  do i=1, n
    y(i) = a * x(i) + y(i)
  end do
  !$omp end target parallel do simd
end subroutine
```

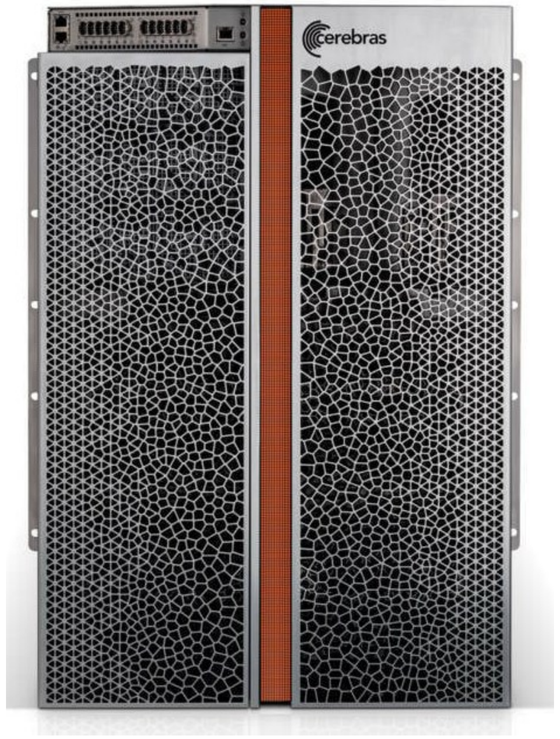
Until this point were running on a single Tensix core, the parallel do directive parallelises loop iterations across Tensix cores. So now here we are running over all Tensix cores of the Wormhole, with the SIMD directive driving the Tensix unix of each.

Multiple levels of parallelism, across Tensix cores in the device and SIMD within each core.

num_threads is optional, sets the number of tensix units to parallelise over (if omitted runs on all)

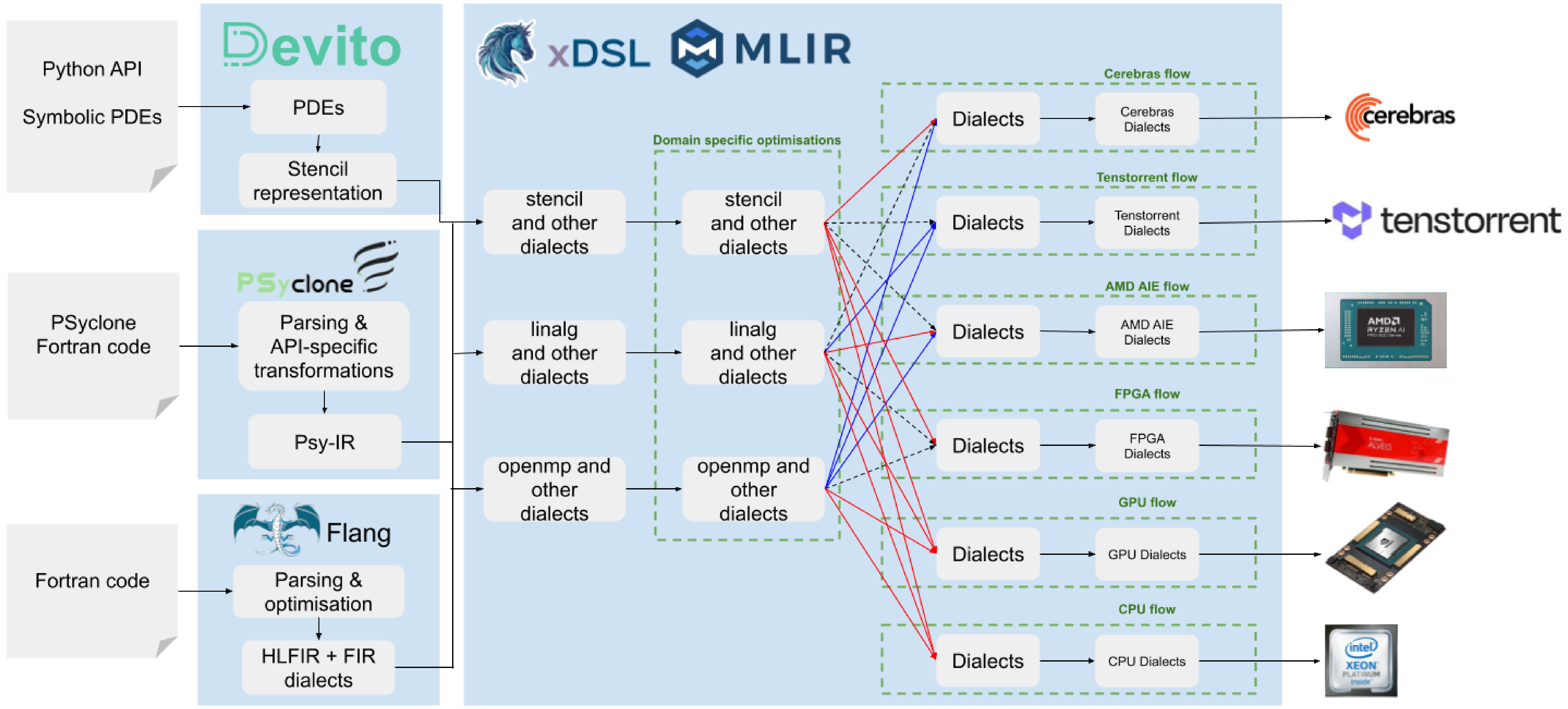
Cerebras CS-3 flow





- Similar idea on the Cerebras Wafer Scale Engine (WSE)
- No changes needed to front ends
- Quite a lot of work in the middle layer, ultimately generating CSL code

Programming is the problem, MLIR enables the solution



Summary

- Largely driven by AI workloads, it's a very exciting time in the hardware world
  - Given the ever increasing demand for performance and sustainability, we need new options
- But this is all for nothing if we can not program such machines!
- We must bring the hardware to the programmers, rather than expecting them to change their code & algorithms

- MLIR/xDSL provides new possibilities for compiler optimisations
  - Arguably, is the method to help us address some of the grand challenges of parallel programming
  - Lots of discussion about *green software engineering,* arguably this all should be handled by the compiler as programmers won't change their code

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