LOWERING THE BARRIERS TO PROGRAMMING FPGAS AND AIES FOR HPC

Nick Brown (EPCC), with lots of help from Gabriel Rodriguez-Canal

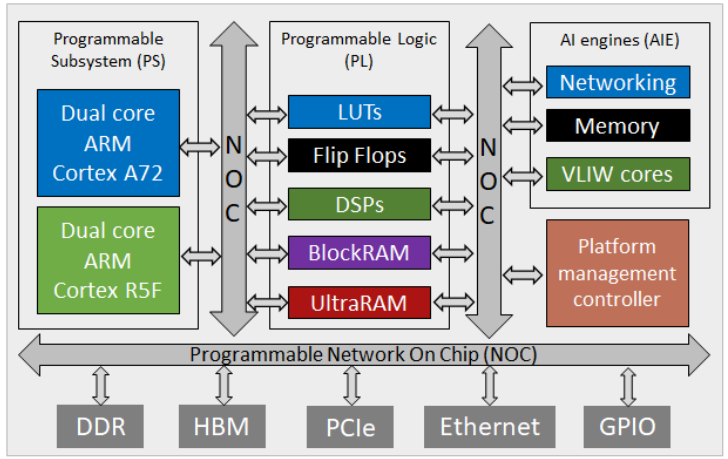
n.brown@epcc.ed.ac.uk

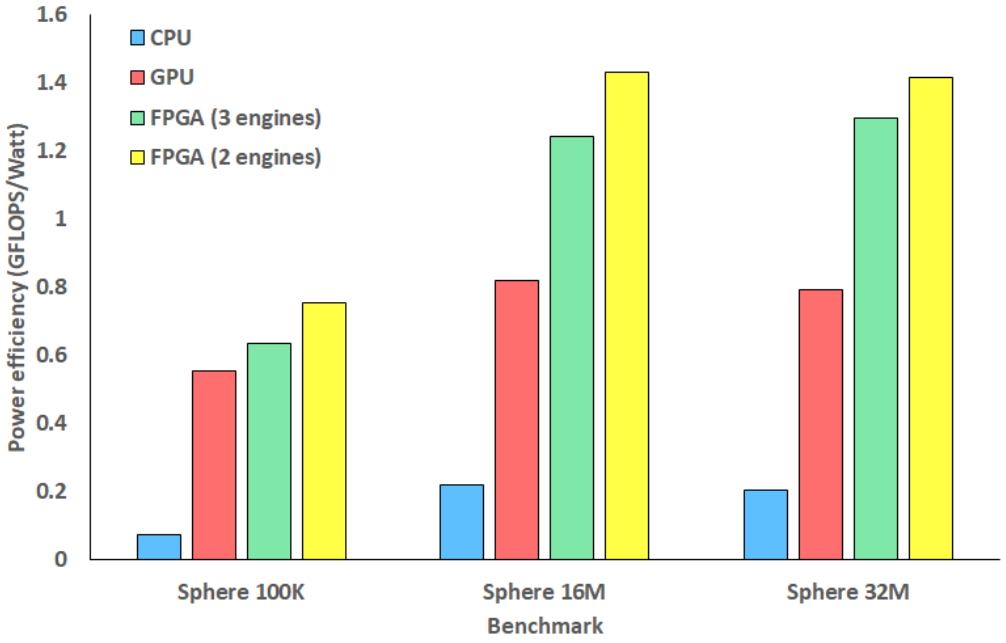
epcc

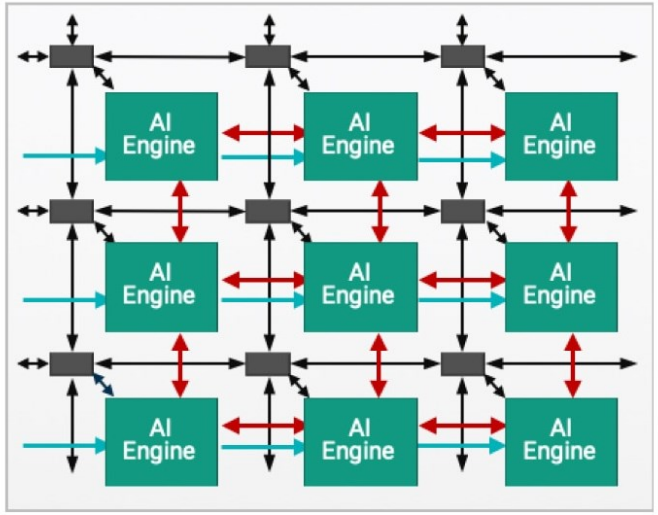
THE UNIVERSITY OF EDINBURGH

FPGAs and AI Engines have lots of potential

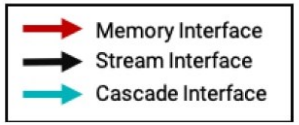
- If you pick the right algorithm, then FPGAs can be successful – especially when considering energy
- AI Engines (AIEs) are also interesting
  - Recently included in Ryzen AI CPUs





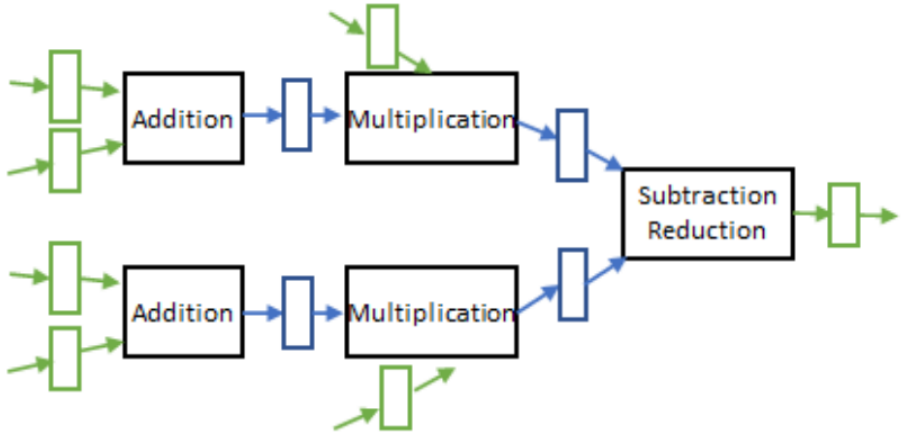


Flexible Interconnect



But programming these is (really) hard!

- Arguably, annotating HLS pragmas is the easy bit
  - Although can be more complex if the code is initially in Fortran as many HPC codes are
  - The challenge is restructuring the algorithm to suit a dataflow architecture which can be a very time consuming process!

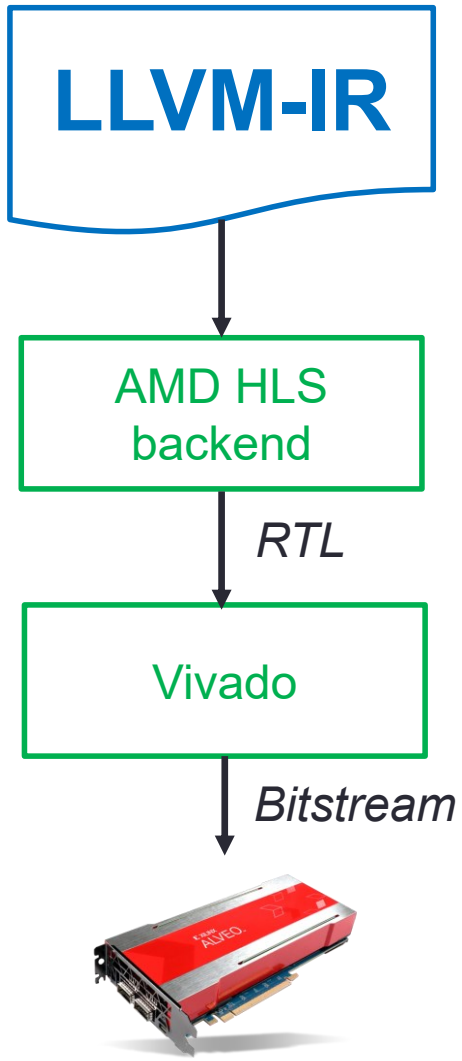


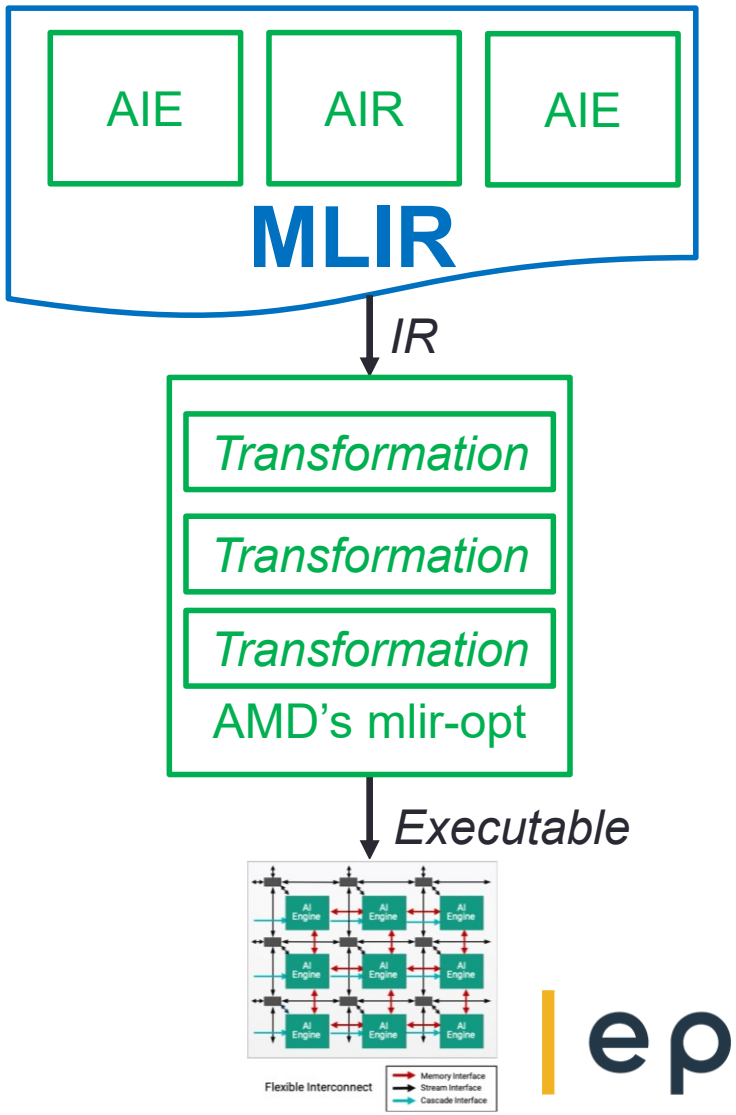
Description	Runtime (ms)	% of CPU
24 cores Xeon CPU	61.72	-
Initial FPGA port	15714.99	0.39%
Optimised II	1508.60	4.09%
Optimised dataflow	284.04	21.73%
Indirect access on host	48.19	128.08%
Threaded result handling	26.67	231.42%

- Ultimately, scientific computing programmers won't do this at large
- They just want to concentrate on their science and get results as quickly as possible

epcc | THE UNIVERSITY OF EDINBURGH

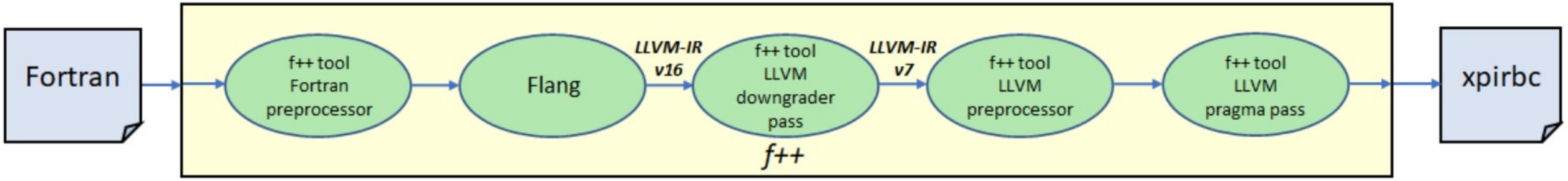
AMD have opened up their toolchains



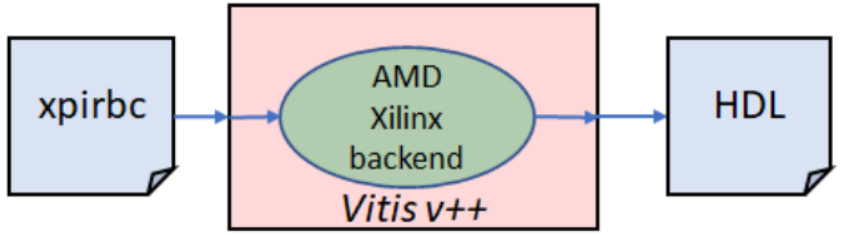


Initial integration: Flang HLS

- Some incompatibilities between Flang and the backend:
  - AMD's version 7 of LLVM, HLS pragmas must be lowered to AMD Xilinx specific LLVM-IR metadata or directives, HLS streams managed via custom IR primitives from the HLS Clang frontend



- Preprocessor identifies HLS pragmas and converts these to function calls. The downgrader then identifies these and replaces with corresponding IR
  - Followed this approach as it requires no changes to Flang
- Streams provided in HLS by C++ template library
  - Use a module approach, the preprocessor determines type and size and generates this



Flang HLS: Performance

- Benchmarked with TeaLeaf suite
- Found that C and Fortran HLS performance is fairly comparable, although there are differences for individual kernels
- Similar optimisation opportunities, although individual differences

Benchmark	Alveo U280 FPGA	
	HLS C Runtime(s)	HLS Fortran Runtime(s)
cg_calc_w_norxy	0.200	0.222
ppcg_calc_rrn (partial sums)	0.630	0.631
ppcg_calc_rrn (dataflow)	0.820	0.535
Synthetic Streaming	0.131	0.132

Lots more detail at https://arxiv.org/pdf/2308.13274

Benchmark	Problem size	Alveo U280 FPGA		Xeon Platinum CPU	
		HLS C Runtime(s)	HLS Fortran Runtime(s)	C Runtime(s)	Fortran Runtime(s)
calc_2norm	480000	0.056	0.038	0.002	0.089
calc_residual	480000	0.196	0.215	0.008	0.087
cg_calc_p	480000	0.315	0.319	0.002	0.003
cg_calc_ur	480000	0.781	0.947	0.015	0.034
cg_calc_w	480000	0.198	0.201	0.009	0.006
cg_calc_w_norxy	480000	0.317	0.299	0.008	0.005
cg_init	480000	0.214	0.214	0.007	0.007
cheby_init	480000	0.850	0.845	0.025	0.093
cheby_iterate	480000	0.628	1.072	0.016	0.083
common_init	480000	0.711	0.631	0.025	0.093
field_summary	480000	0.105	0.054	0.008	0.095
finalise	480000	0.003	0.055	≈0.000	0.107
generate_chunk	480000	0.369	0.898	0.008	0.096
initialise_chunk	480000	0.116	0.143	≈0.000	0.048
jacobi_solve	480000	0.192	0.230	0.018	0.001
ppcg_calc_rrn	39677401	3.871	3.866	0.184	0.203
ppcg_calc_zrnorm	480000	0.059	0.059	0.002	0.086
ppcg_init	480000	0.343	0.509	0.009	0.109
ppcg_init_sd	480000	0.227	0.188	0.005	0.069
ppcg_inner	480000	0.754	0.727	≈0.000	0.106
ppcg_inner_norxy	480000	0.748	0.704	≈0.000	0.098
ppcg_pupdate	480000	0.043	0.044	0.002	0.084
ppcg_store_r	480000	0.043	0.044	0.002	0.086
ppcg_update_z	480000	0.043	0.044	0.002	0.099
set_field	480000	0.043	0.044	0.002	0.087
tea_block_init	480000	0.616	0.615	0.005	0.020
tea_block_solve	480000	0.098	0.267	≈0.000	0.026
tea_diag_init	480000	0.083	0.083	0.002	0.019
tea_diag_solve	480000	0.083	0.099	0.002	0.019
update_halo	480000	0.788	0.889	0.011	0.059
update_halo_cell	480000	0.001	0.001	≈0.000	0.008
update_in_halo_bt	480000	0.016	0.016	≈0.000	0.067
update_in_halo_cell_bt	480000	0.002	0.002	≈0.000	0.004
update_in_halo_cell_lr	480000	0.049	0.080	0.001	0.006
update_in_halo_lr	480000	0.874	0.898	0.011	0.067

On an Alveo U280

Flang HLS: Utilisation

- Benchmarked with TeaLeaf suite
- Similar optimisation opportunities, although individual differences

Benchmark	HLS kernel usage C/Fortran (% total resources) LUTs	FFs	BRAM	DSP
cg_calc_w _norxy	6.13 / 5.76	6.32 / 5.13	3.17 / 3.17	0.74 / 0.74
	(-6.04%)	(-18.83%)	(0.00%)	(0.00%)
ppcg_calc_rrn (partial sums)	0.65 / 0.60	0.34 / 0.30	0.40 / 0.40	0.19 / 0.22
	(-7.69%)	(-11.76%)	(0.00%)	(15.79%)
ppcg_calc_rrn (dataflow)	0.64 / 0.78	0.34 / 0.40	0.40 / 0.40	0.19 / 0.39
	(21.88%)	(17.65%)	(0.00%)	(105.26%)
Synthetic Streaming	0.29 / 0.22	0.10 / 0.08	0.15 / 0.10	0.00 / 0.00
	(-24.13%)	(-20.00%)	(-33.33%)	-

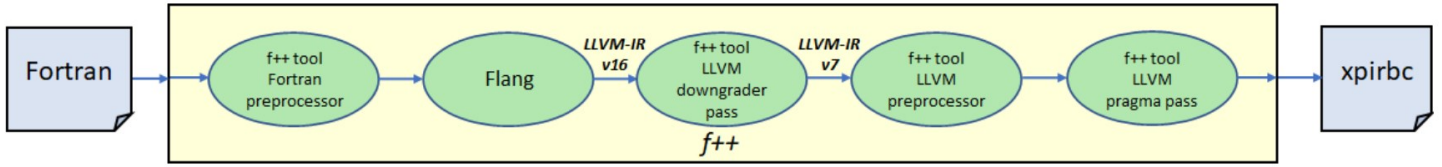
Lots more detail at https://arxiv.org/pdf/2308.13274

Kernel	HLS C % usage BRAM	LUT	FF	DSP	HLS Fortran % usage BRAM	LUT	FF	DSP
calc_2norm	0.10	0.32	0.14	0.19	0.10	0.43	0.16	0.16
calc_residual	0.10	0.50	0.30	0.32	0.10	0.64	0.35	0.29
cg_calc_p	0.10	0.52	0.22	0.32	0.10	0.49	0.22	0.27
cg_calc_ur	0.10	2.24	1.20	1.30	0.10	2.88	1.23	1.00
cg_calc_w	0.10	0.69	0.36	0.42	0.10	0.80	0.39	0.32
cg_calc_w_norxy	0.10	0.60	0.31	0.32	0.10	0.75	0.37	0.32
cg_init	0.10	1.82	1.09	1.26	0.10	2.19	0.95	0.81
cheby_init	0.10	2.15	1.31	1.43	0.10	2.50	1.19	0.98
cheby_iterate	0.10	2.31	1.36	1.46	0.10	2.85	1.31	1.10
common_init	0.10	2.29	1.42	1.13	0.10	2.82	1.53	0.79
field_summary	0.10	0.44	0.24	0.22	0.10	0.56	0.26	0.27
finalise	0.10	0.22	0.11	0.10	0.10	0.30	0.13	0.03
generate_chunk	0.10	1.05	0.54	0.32	0.10	1.63	0.74	0.70
initialise_chunk	0.10	0.91	0.33	0.12	0.10	1.07	0.46	0.22
jacobi_solve	0.10	0.63	0.37	0.35	0.10	0.85	0.44	0.32
ppcg_calc_rrn	0.10	0.37	0.17	0.19	0.10	0.45	0.20	0.23
ppcg_calc_zrnorm	0.10	0.35	0.15	0.19	0.10	0.46	0.18	0.16
ppcg_init	0.10	1.95	1.15	1.33	0.10	2.39	1.04	0.84
ppcg_init_sd	0.10	0.54	0.26	0.22	0.10	0.68	0.30	0.20
ppcg_inner	0.10	2.88	1.61	1.66	0.10	3.62	1.70	1.46
ppcg_inner_norxy	0.10	2.55	1.43	1.46	0.10	3.16	1.43	1.14
ppcg_pupdate	0.10	0.23	0.09	0.07	0.10	0.31	0.11	0.03
ppcg_store_r	0.10	0.23	0.09	0.07	0.10	0.31	0.11	0.03
ppcg_update_z	0.10	0.23	0.09	0.07	0.10	0.31	0.11	0.03
set_field	0.10	0.23	0.09	0.07	0.10	0.31	0.11	0.03
tea_block_init	0.10	0.78	0.35	0.48	0.10	0.89	0.35	0.35
tea_block_solve	0.10	1.23	0.82	0.93	0.10	1.30	0.65	0.61
tea_diag_init	0.10	0.26	0.38	0.07	0.10	0.36	0.42	0.07
tea_diag_solve	0.10	0.24	0.12	0.16	0.10	0.39	0.16	0.12
update_halo	0.20	1.95	0.44	0.20	0.74	2.76	0.65	0.31
update_halo_cell	0.20	1.35	0.32	0.20	0.20	1.40	0.40	0.31
update_in_halo_bt	0.74	1.48	0.34	0.13	0.74	1.67	0.42	0.17
update_in_halo_cell_bt	0.10	0.40	0.15	0.13	0.10	0.56	0.23	0.17
update_in_halo_cell_lr	0.10	0.39	0.18	0.13	0.10	0.50	0.22	0.13
update_in_halo_lr	0.74	1.52	0.35	0.13	0.74	1.63	0.38	0.13

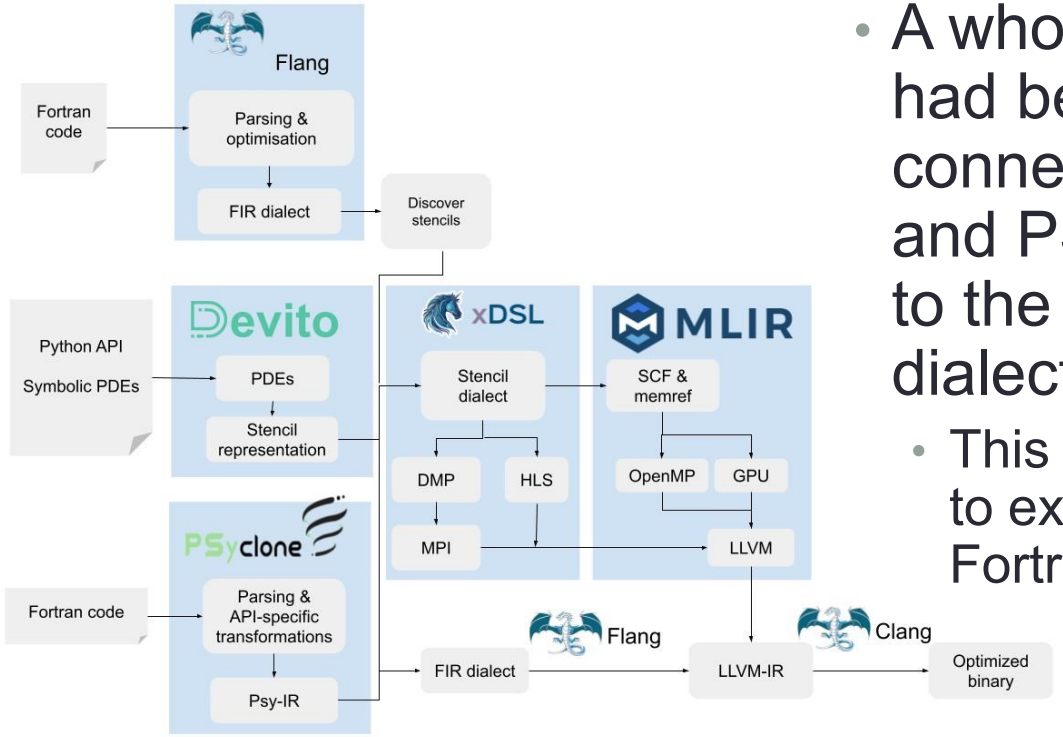
On an Alveo U280

Extracting domain specific patterns

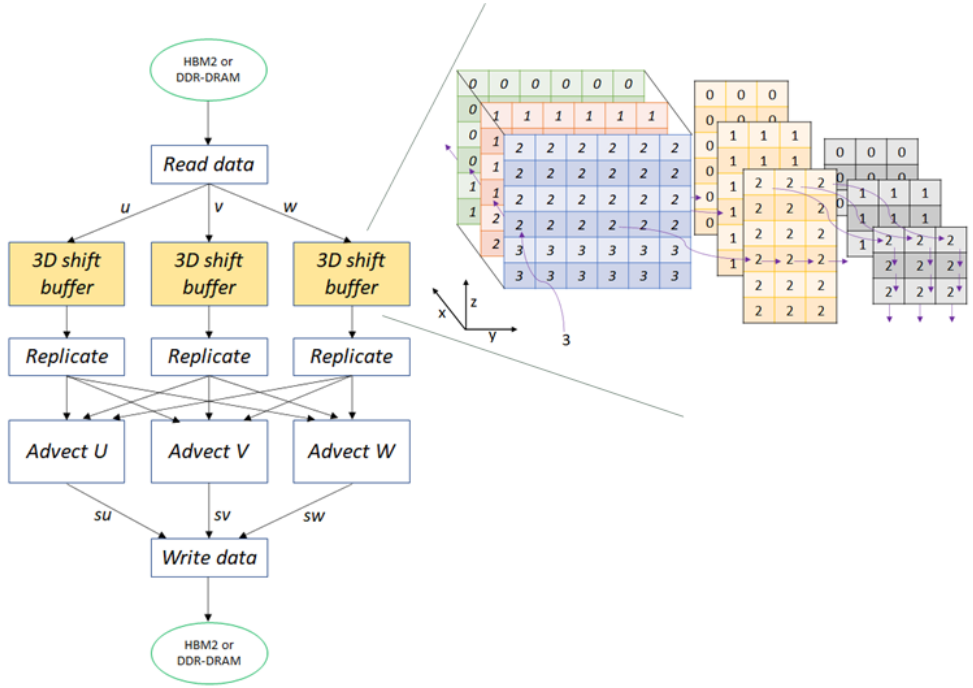
- The approach we took meant that this wasn't tied to Fortran



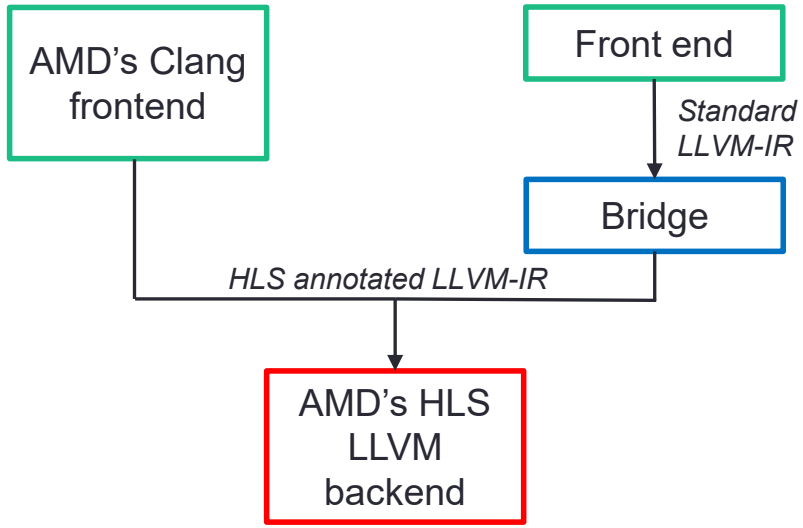
- We could feed in any LLVM-IR to our downgrading pass, including that generated from MLIR



- A whole load of work had been done to connect the Devito and PSyclone DSLs to the MLIR stencil dialect
  - This was then extended to extract stencils from Fortran code



A new HLS dialect




```
hls.axi_protocol(%protocol) : (i32) -> hls.axi_protocol
hls.streamtype(%elem_type) : (f64) -> hls.streamtype
```

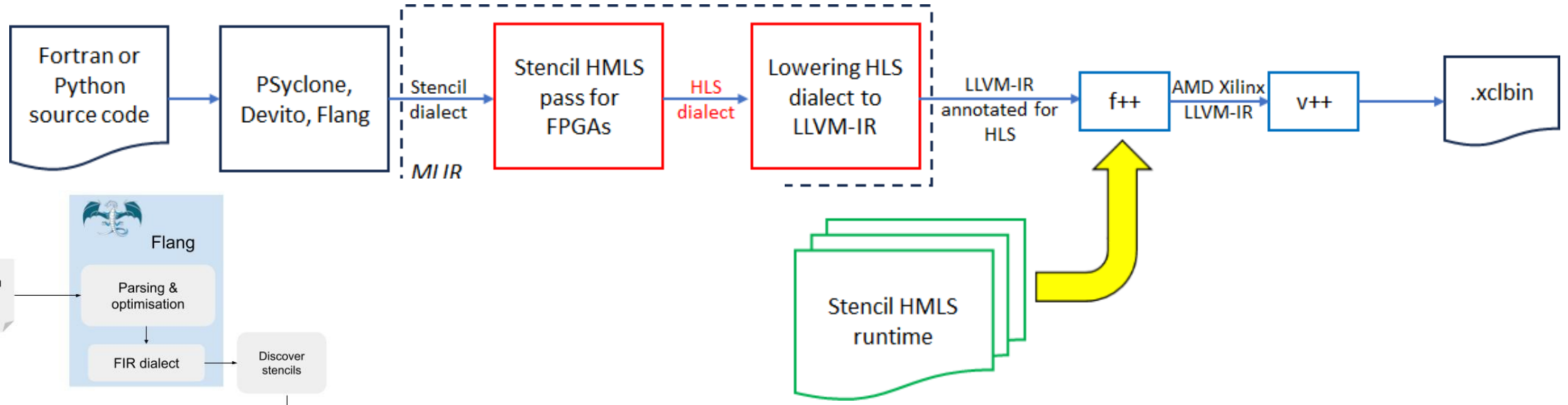
New types

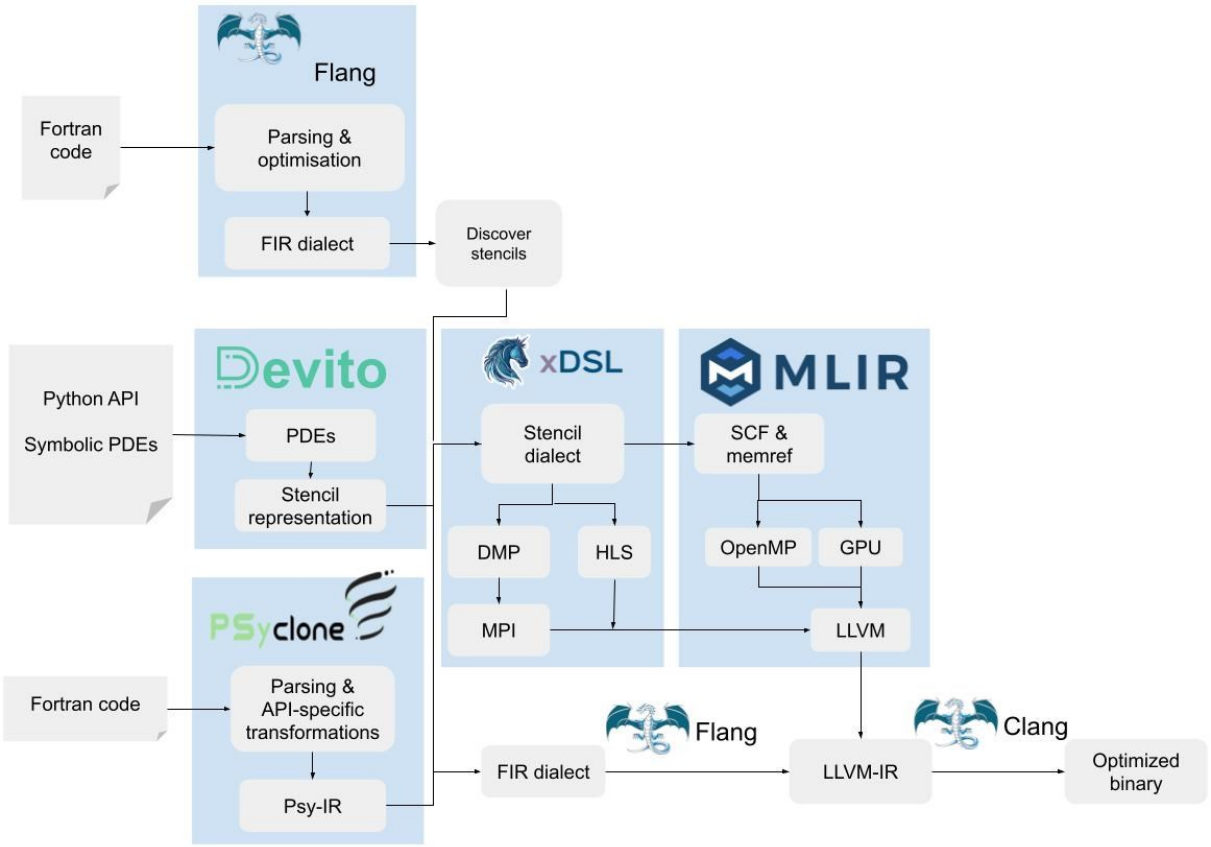
```
hls.interface(%protocol, %bundle) : (hls.axi_protocol,
    str) -> ()
hls.pipeline(%ii) : (i32) -> ()
hls.unroll(%factor) : (i32) -> ()
hls.array_partition() : () -> ()
hls.dataflow() : () -> ()
hls.create_stream(%elem_type : f64)
hls.read(%stream) : hls.streamtype -> (f64)
hls.write(%stream, %elem) : (hls.streamtype, f64) -> ()
hls.empty(%stream) : (hls.streamtype) -> i1
hls.full(%stream) : (hls.streamtype) -> i1
```

New operations

- There isn't an HLS dialect, so we developed one based on the abstractions provided in the AMD C++ HLS front-end
- Then developed transformations to lower this dialect down to the form that is understandable by our existing connection
  - Following the same sort of approach that we used for Flang, where HLS directives are represented by function calls and these are then replaced by the corresponding IR

Our stencil HLS flow

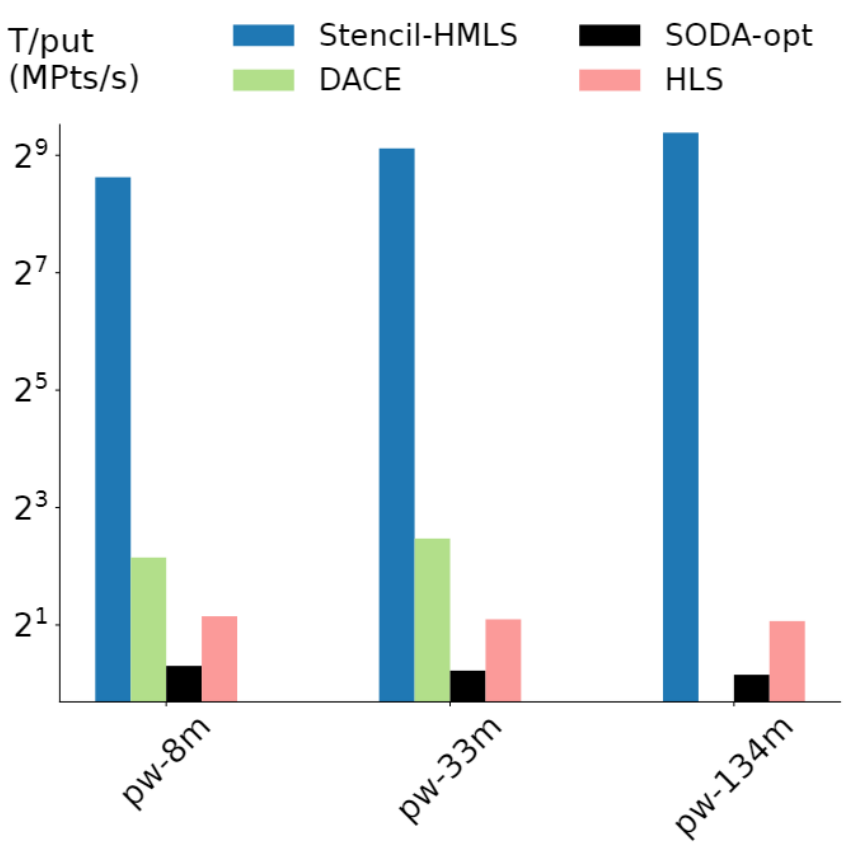




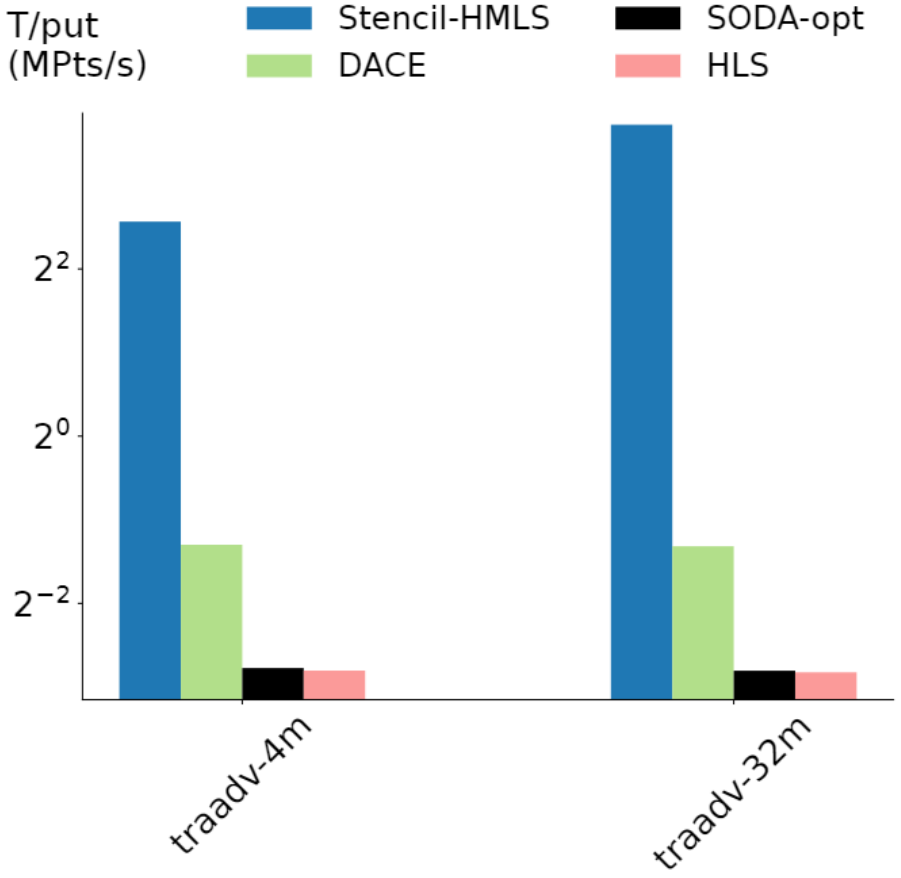
- Using the existing flow and dialects, we lower the stencil dialect (and others) to the HLS dialect
- Ultimately means that stencil codes written in any language can target FPGAs
- Just had to implement the HLS dialect and transformations, using the rest of the ecosystem

Performance

(Higher is better)



On an Alveo U280



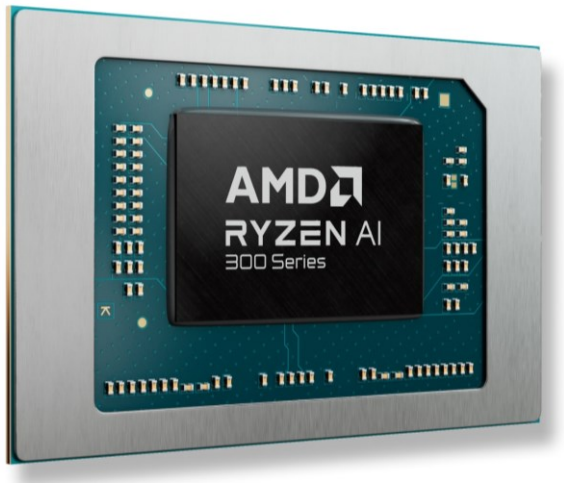
On an Alveo U280

- More details at https://arxiv.org/pdf/2310.01914

epcc

THE UNIVERSITY OF EDINBURGH

AMD's AI Engines (AIEs)

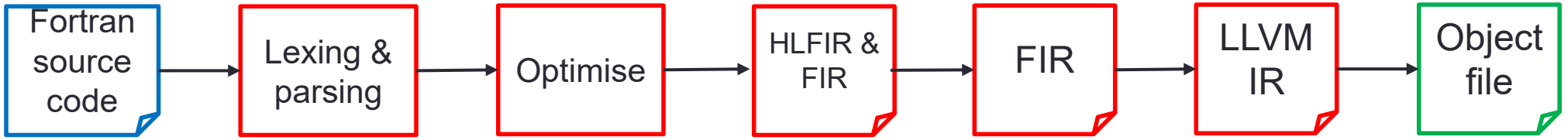


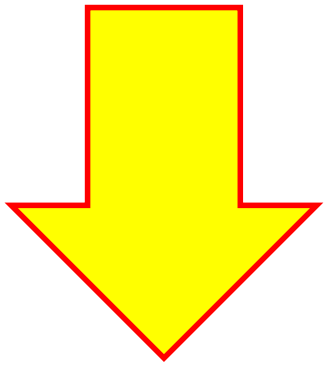
- In late 2023 AMD released their Ryzen AI CPU, which contains their Neural Processor Unit which is a marketing term for an array of AIEs
- Very interesting, as a much more attractive proposition if these are already inside a CPU

- 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

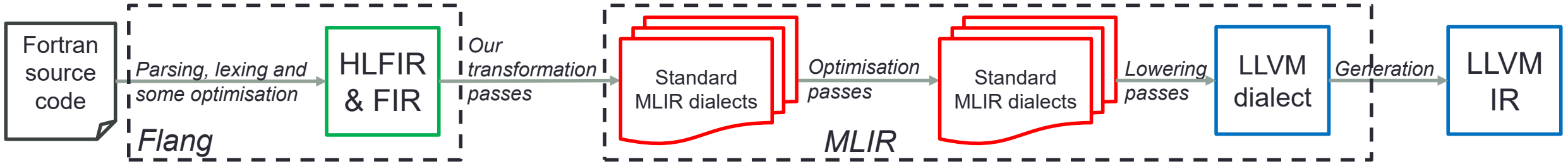
- Direct programming via kernels in C++ using API and Riallto Python framework for the dataflow graph

Work on Flang to common MLIR dialects a foundation





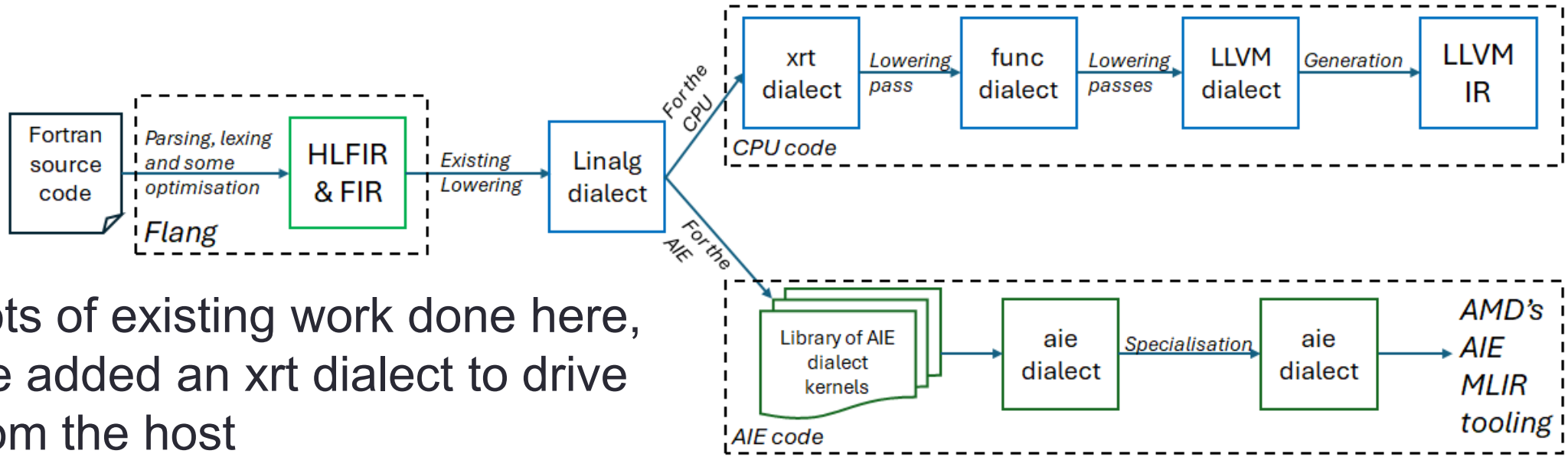
To develop a transformation that operates on HLFIR & FIR based IR, transforming this into dialects that are present in standard MLIR and then using standard MLIR transformation and optimisation passes to generate LLVM-IR



epcc

THE UNIVERSITY OF EDINBURGH

Extending to seamlessly offload on the AIEs



- Lots of existing work done here, we added an xrt dialect to drive from the host

- AMD have a Python interface to their AIE MLIR dialects and lots of examples
  - We pre-generate lots of kernels, and add placeholders for constants and types
  - These are all added to a library, when mapping the Fortran intrinsic our approach picks the appropriate MLIR, fills in the placeholders and forwards to AMD's *aie-opt* tool

Driving from Fortran

- The idea is that 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

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

- For simple reduction based intrinsics such as *sum*, *prod*, *maxval*, *minval* the CPU core tends to be faster for int32 and int16, although for bf16 and fp32 the AIEs tend to outperform the CPU
- For *matmul* the AIEs are around five times faster than the CPU core
  - *However the caveat is that these are currently small matrix sizes due to what can fit in the memory tile's memory across the NPU*
  - *This is likely a limit of the matrix multiplication kernel we are using and not the general approach however*

Conclusions

- MLIR is powerful and enables us to do some cool things that addresses a major challenge for adoption
  - It's especially great that AMD have invested in MLIR for the AIEs, and I think this opens up a whole host of opportunities

xDSL

- It's easy to get started with MLIR!
  - xDSL is a Python based compiler framework that is 1:1 compatible with MLIR
  - Enables fast prototyping and exploration of the underlying ideas
  - https://xdsl.dev

epcc

THE UNIVERSITY OF EDINBURGH