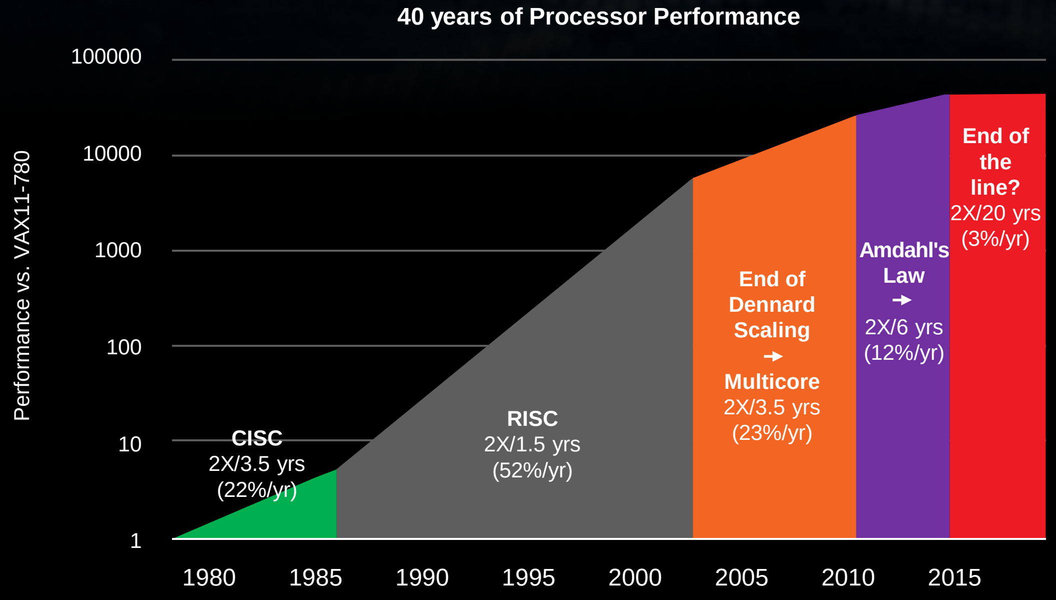
MLIR Compilers for Heterogeneous Computing

Stephen Neuendorffer
Fellow

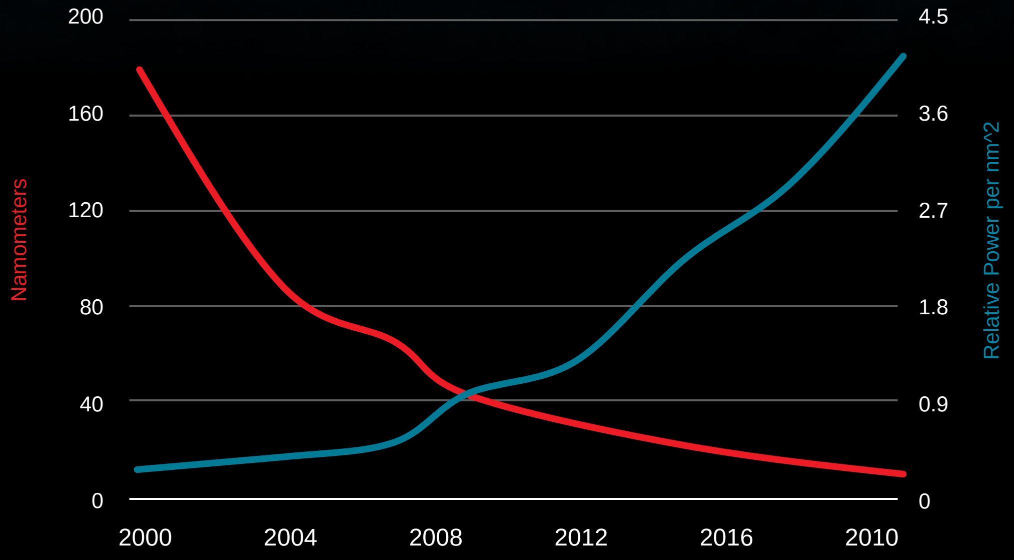
November 14, 2022

Context

End of Growth of Single Program Speed?



Technology & Power: Dennard Scaling



A New Golden Age for
Computer Architecture

→

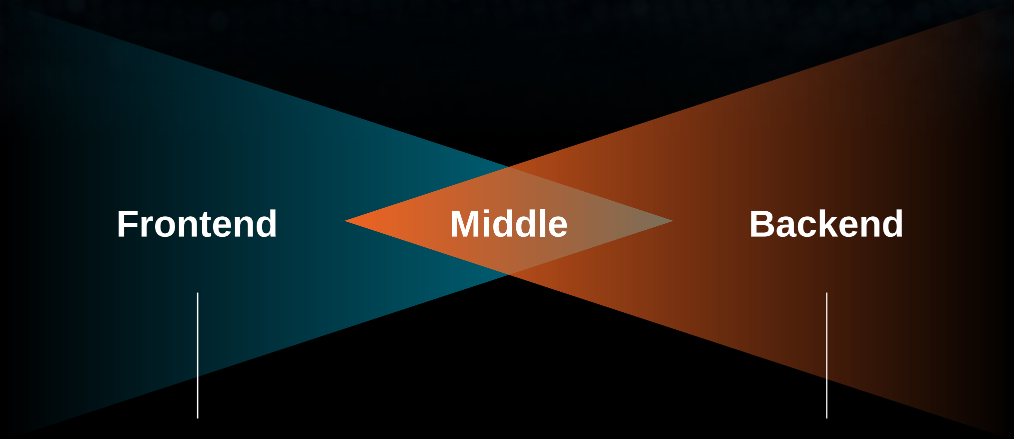
A New Golden Age for
Compilers

MLIR: Multi-Level Intermediate Representation

Next generation open source compiler infrastructure

- LLVM core project

Well positioned to support this new golden age!



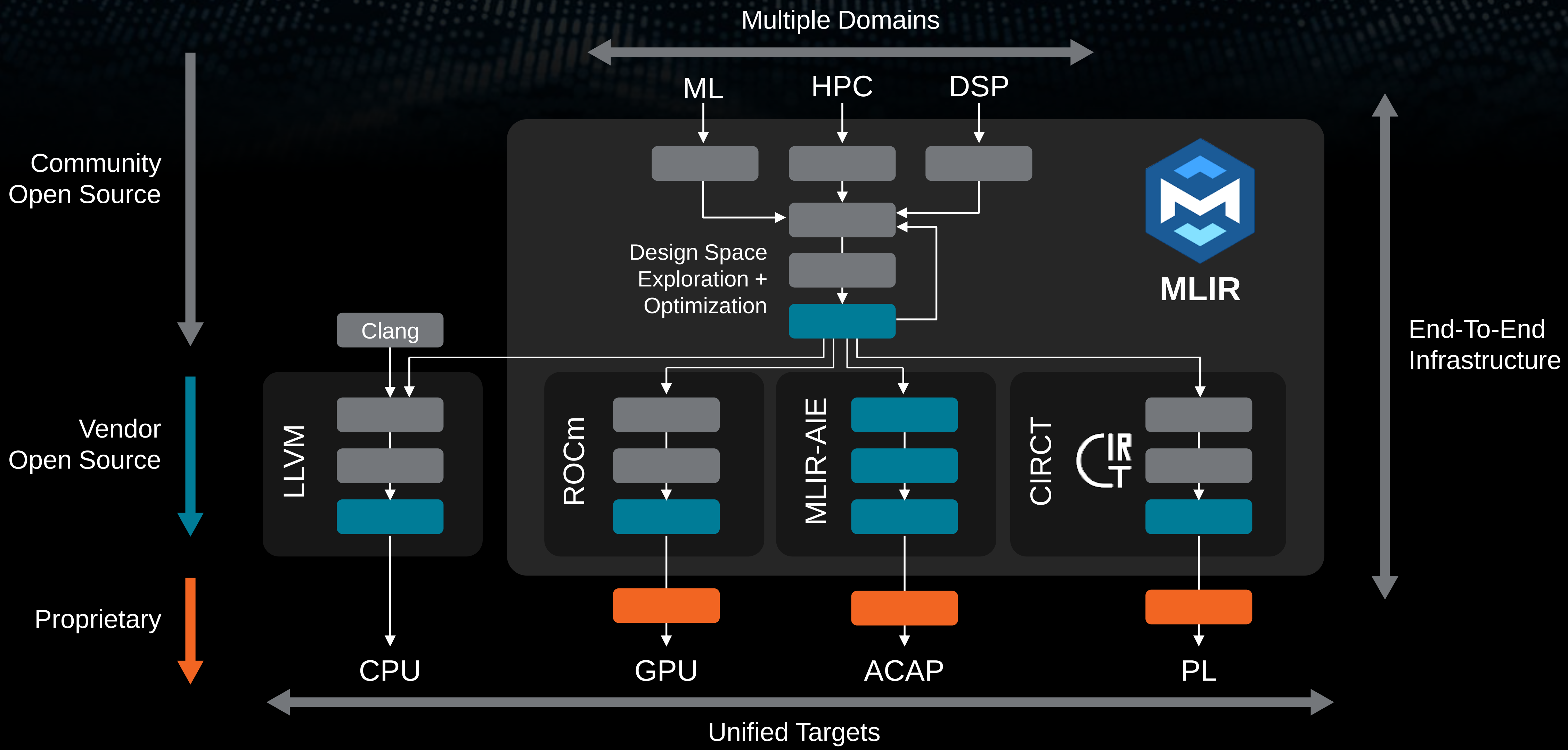
Multiple open-source frontend languages (not just ML!)

Exascale Fortran, Tensorflow, pytorch -> C, HIP/CUDA, SYCL

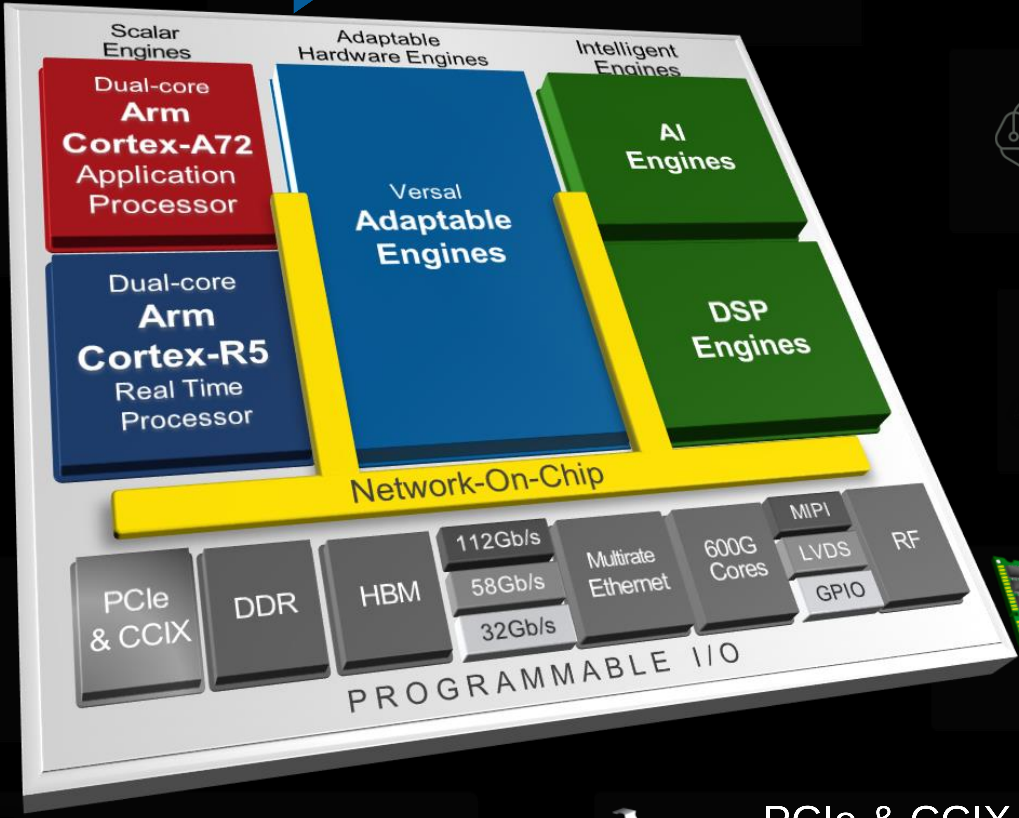
Multiple open-source backends

LLVM, GPU, ACAP, Programmable Logic

Future Heterogeneous Programming



Versal ACAP Architecture



Adaptable Engines
2X compute density

Scalar Engines
- Platform Control
- Edge Compute

AI Engines
- AI Compute
- Diverse DSP workloads

Protocol Engines
- Integrated 600G cores
- 4X encrypted bandwidth

Network-on-Chip
- Guaranteed Bandwidth
- Enables SW Programmability

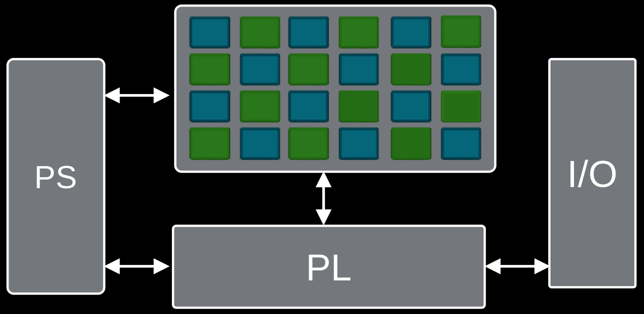
Programmable I/O
- Any interface or sensor
- Includes 4.2Gb/s MIPI

DDR Memory
- 3200-DDR4, 3200-LPDDR4
- 2X bandwidth/pin

Transceivers
- Broad range, 25G →112G
- 58G in mainstream devices

PCIe & CCIX
- 2X PCIe & DMA bandwidth
- Cache-coherent interface to accelerators

AI Engine: Array Architecture



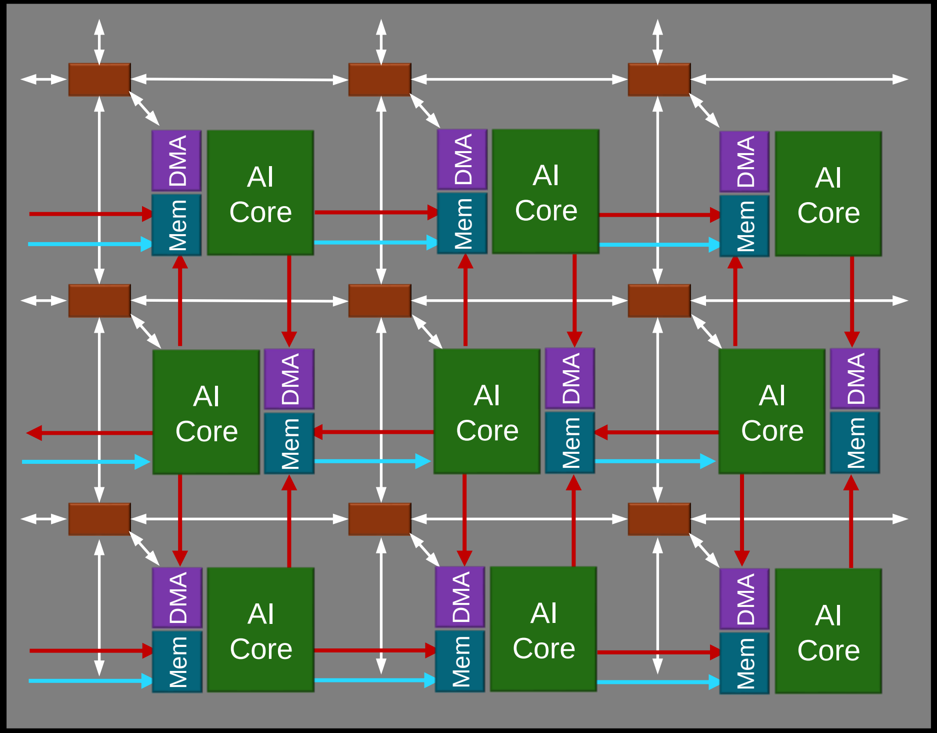
Array of AI Engines

- Increase in compute, memory and communication bandwidth

Modular and scalable architecture

- More tiles = more compute
- Up to 400 per device
  - Versal AI Core VC1902 device

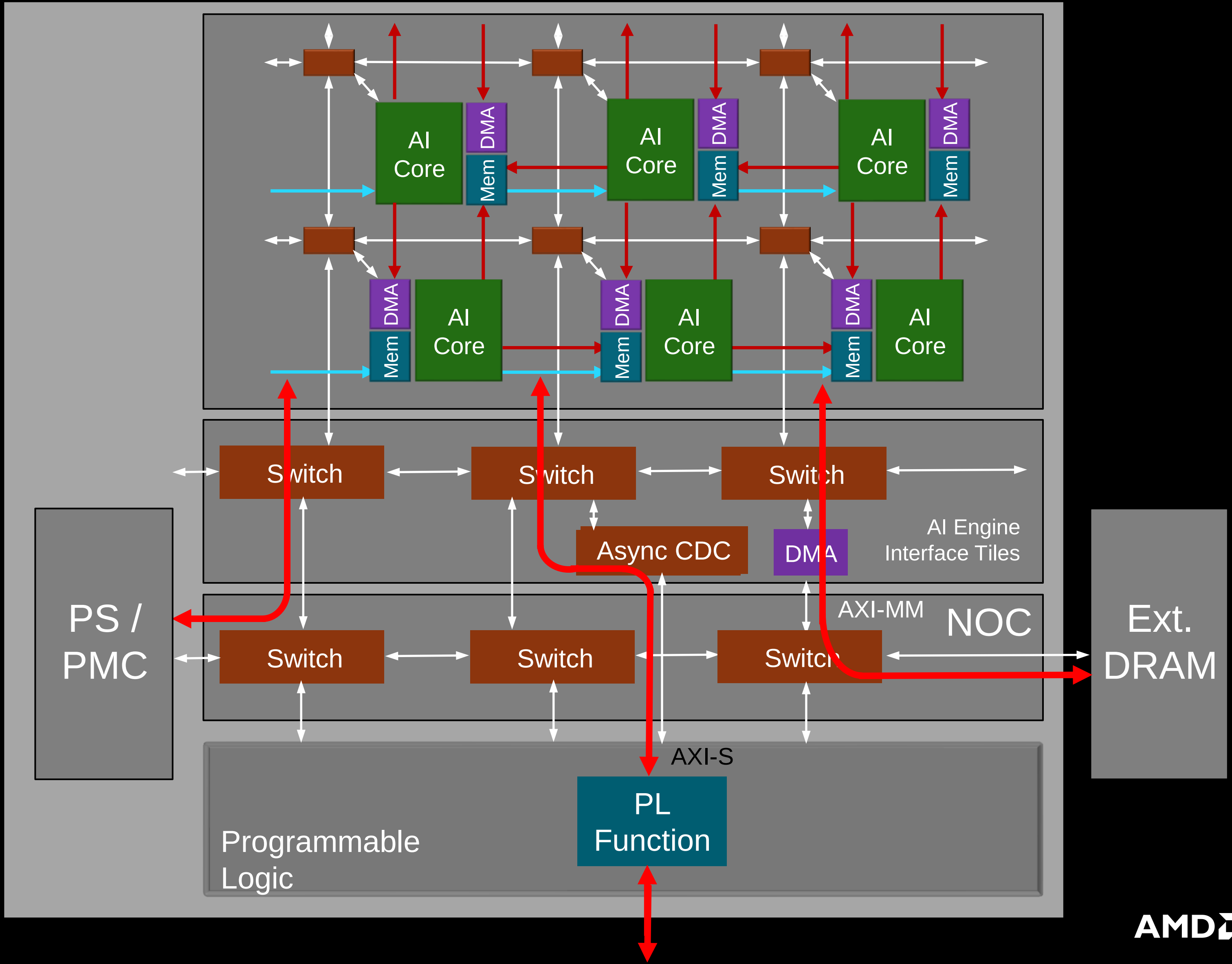
Distributed memory hierarchy
Maximize memory bandwidth



Deterministic Performance & Low Latency

AI Engine Integration with Versal ACAP

- TB/s of Interface Bandwidth
  - AI Engine to Programmable Logic
  - AI Engine to NOC
- Leveraging NOC connectivity
  - PS manages Config / Debug / Trace
  - AI Engine to DRAM (no PL req'd)



Basic MLIR Concepts

- LLVM done better
  - Textual + Binary Syntax
  - Efficient, Modern C++ Framework
- Fundamentally Extensible
- Lots of 'Batteries Included'
  - Math, Loops, and Tensor *Dialects*
  - Many Optimizations
- Lots of Connections
  - ML Frontends, LLVM Backends

```
module {
  %foo, %bar = myDialect.myOp(%biz, %baz) {
    %1 = math.add(%biz, %baz) : i32
   ^bb1:
    myDialect.myOtherOp(%1)
   ^bb2:
    myDialect.goto ^bb1
  }
  myDialect.myGraphOp() {
    %1 = myDialect.myOp(%2)
    %2 = myDialect.myOp(%1)
  }
}
```

Basic MLIR Concepts

- *Operations* have *Operands* and *Results*
  - *MLIR is generic*

```
module {
    %foo, %bar =
        myDialect.myOp(%biz, %baz) {
    %1 = std.add(%biz, %baz) : i32
   ^bb1:
    myDialect.myOtherOp(%1)
   ^bb2:
    myDialect.goto ^bb1
  }
  myDialect.myGraphOp() {
    %1 = myDialect.myOp(%2)
    %2 = myDialect.myOp(%1)
  }
}
```

Basic MLIR Concepts

- *Operations* have *Operands* and *Results*
  - *MLIR is generic*
- *Operations* live in *Dialects*
  - *MLIR is extensible*

```
module {
    %foo, %bar =
      myDialect.myOp(%biz, %baz) {
    %1 = std.add(%biz, %baz) : i32
   ^bb1:
    myDialect.myOtherOp(%1)
   ^bb2:
    myDialect.goto ^bb1
  }
  myDialect.myGraphOp() {
    %1 = myDialect.myOp(%2)
    %2 = myDialect.myOp(%1)
  }
}
```

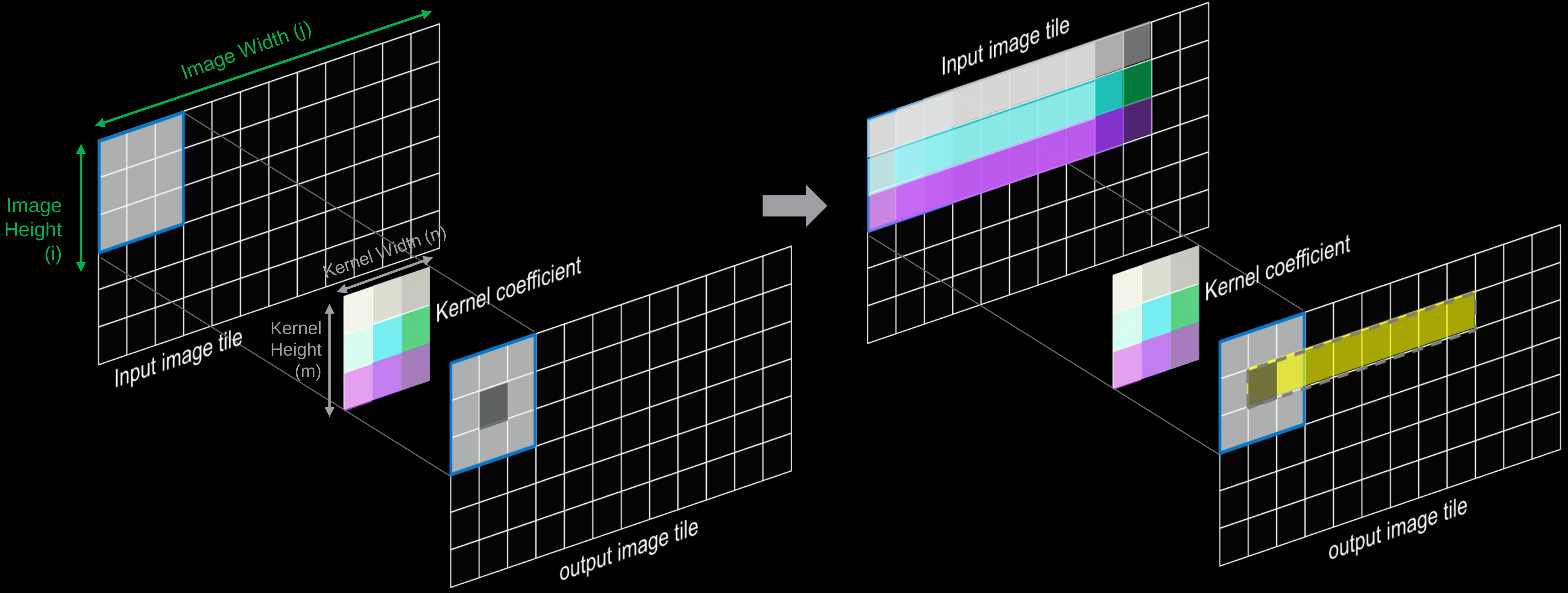
Basic MLIR Concepts

- *Operations* have *Operands* and *Results*
  - *MLIR is generic*
- *Operations* live in *Dialects*
  - *MLIR is extensible*
- *Operations* can contain *Regions*
  - *MLIR is hierarchical*

```
module {
    %foo, %bar =
       myDialect.myOp(%biz, %baz) {
    %1 = std.add(%biz, %baz) : i32
   ^bb1:
    myDialect.myOtherOp(%1)
   ^bb2:
    myDialect.goto ^bb1
  }
  myDialect.myGraphOp() {
    %1 = myDialect.myOp(%2)
    %2 = myDialect.myOp(%1)
  }
}
```

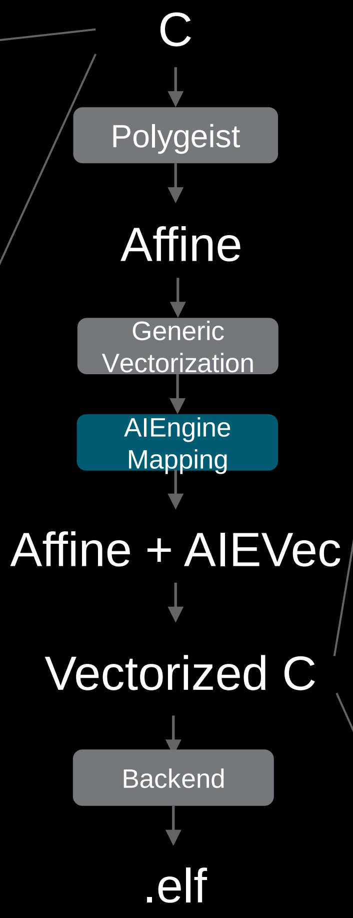
Single Core Performance

Vectorization



Automatic Vectorization in MLIR

```
void conv2d(int img_in[18][272],
            int kernel[3][3],
            int img_out[16][256]) {
 for(int r = 0; r < 16; r++)
   for(int c = 0; c < 256; c++) {
     int acc = 0;
     for(int i = 0; i < 3; i++)
       for(int j = 0; j < 3; j++)
         acc += img_in[r+i][c+j] *
         kernel[i][j];
     img_out[r][c] = acc;
   }
}
```



```
for (size_t v19 = v16; v19 < v17; v19 += v18)
    chess_prepare_for_pipelining
    chess_loop_range(255, 256) {
      v8int32 v20 = *(v8int32 *)(v3 + 2046*v11+v19);
      v21 = upd_w(v21, 0, *(v8int32 *)(v1 + 2048*v11+v19));
      v8acc80 v22 = lups(v20, 0);
      v22 = lmac8(v22, v21, 0, 0x76543210, v6, 0, 0x00000000);
      size_t v23 = 1;
      size_t v24 = v19 + v23;
      v21 = upd_w(v21, 1, *(v8int32 *)(v1 + 2048*v11+v24 + 7));
      v22 = lmac8(v22, v21, 1, 0x76543210, v6, 1, 0x00000000);
      v22 = lmac8(v22, v21, 2, 0x76543210, v6, 2, 0x00000000);
      v25 = upd_w(v25, 0, *(v8int32 *)(v1 + 2048*v13+v19));
      v22 = lmac8(v22, v25, 0, 0x76543210, v6, 3, 0x00000000);
      v25 = upd_w(v25, 1, *(v8int32 *)(v1 + 2048*v13+v24 + 7));
      v22 = lmac8(v22, v25, 1, 0x76543210, v6, 4, 0x00000000);
      v22 = lmac8(v22, v25, 2, 0x76543210, v6, 5, 0x00000000);
      v26 = upd_w(v26, 0, *(v8int32 *)(v1 + 2048*v15+v19));
      v22 = lmac8(v22, v26, 0, 0x76543210, v6, 6, 0x00000000);
      v26 = upd_w(v26, 1, *(v8int32 *)(v1 + 2048*v15+v24 + 7));
      v22 = lmac8(v22, v26, 1, 0x76543210, v6, 7, 0x00000000);
      v22 = lmac8(v22, v26, 2, 0x76543210, v7, 0, 0x00000000);
      *(v8int32 *)(v3 + 2046*v11+v19) = srs(v22, 0);
    }
```

https://xilinx.github.io/mlir-aie/AIEVectorization

Affine Loops in MLIR

```
func @conv2d(%arg0: memref<18x272xi32>, %arg1: memref<3x3xi32>, %arg2: memref<16x256xi32>) {
   %c0_i32 = arith.constant 0 : i32
   affine.for %arg3 = 0 to 16 {
     affine.for %arg4 = 0 to 256 {
       %0 = affine.for %arg5 = 0 to 3 iter_args(%arg6 = %c0_i32) -> (i32) {
         %1 = affine.for %arg7 = 0 to 3 iter_args(%arg8 = %arg6) -> (i32) {
           %2 = affine.load %arg0[%arg3 + %arg5, %arg4 + %arg7] : memref<18x272xi32>
           %3 = affine.load %arg1[%arg5, %arg7] : memref<3x3xi32>
           %4 = arith.muli %2, %3 : i32
           %5 = arith.addi %arg8, %4 : i32
           affine.yield %5 : i32
         }
         affine.yield %1 : i32
       }
       affine.store %0, %arg2[%arg3, %arg4] : memref<16x256xi32>
     }
   }
   return
 }
```

Mapping to Target-specific Operations

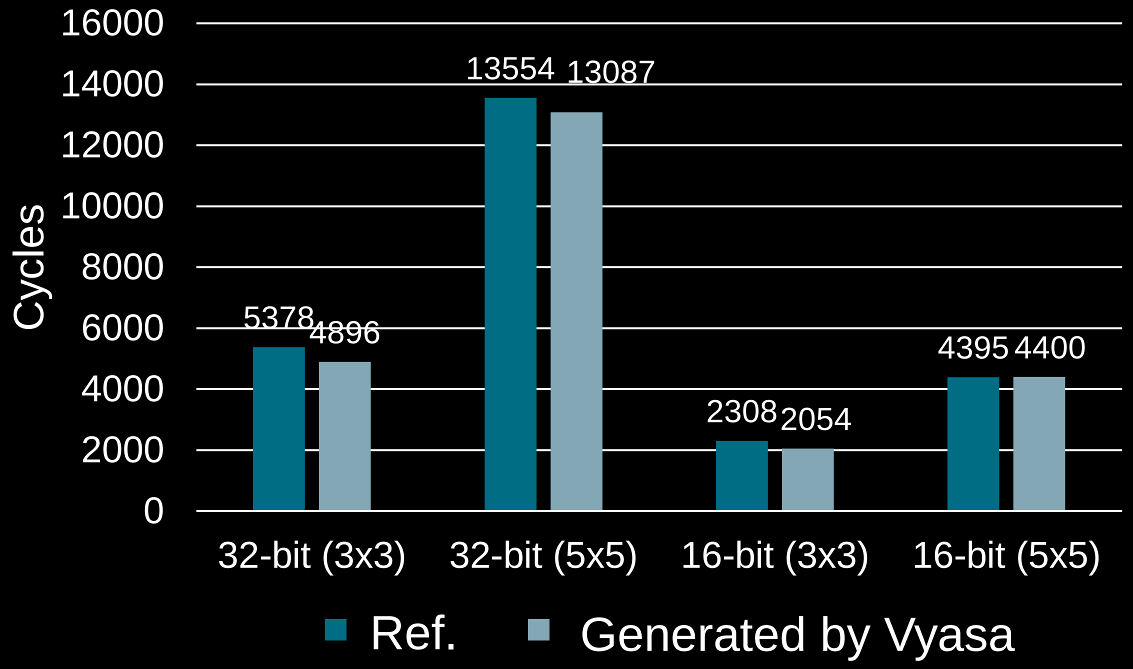
```
%41 = affine.apply #map2(%arg3)
%42 = affine.apply #map2(%arg4)
%43 = vector.transfer_read %arg0[%41, %42], %c0_i32 : memref<18x272xi32>, vector<8xi32>
%44 = vector.transfer_read %arg1[%c2, %c2], %c0_i32 {permutation_map = #map0} : memref<3x3xi32>, vector<8xi32>
%45 = arith.muli %43, %44 : vector<8xi32>
%46 = arith.addi %40, %45 : vector<8xi32>
vector.transfer_write %46, %arg2[%arg3, %arg4] : vector<8xi32>, memref<16x256xi32>
```

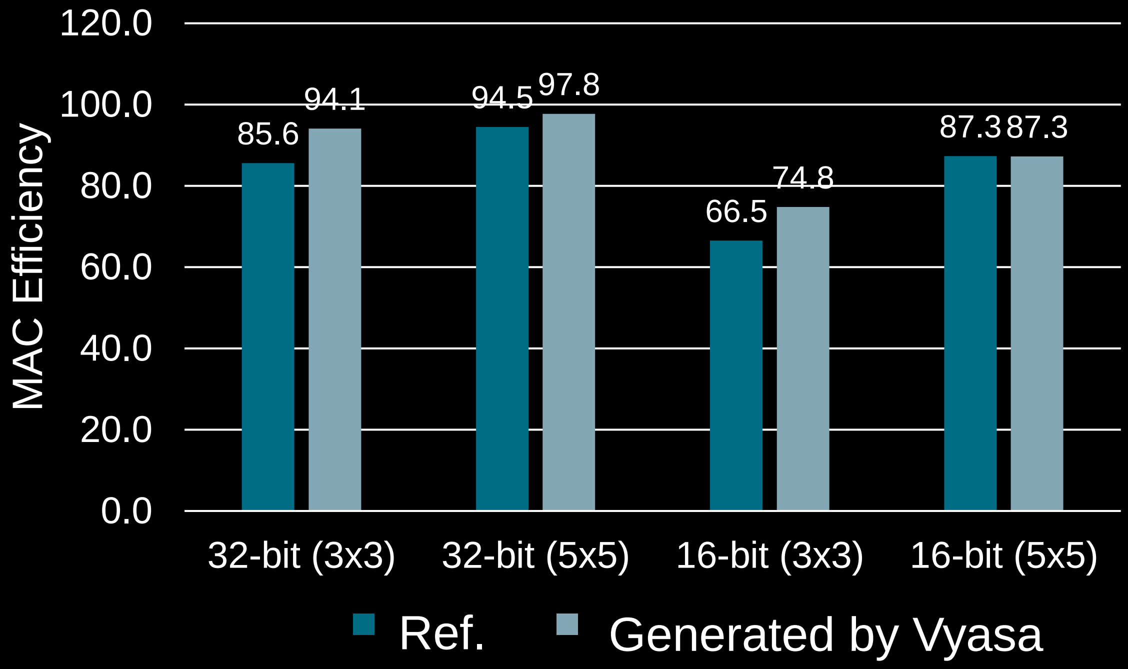
⬇

```
%17 = aievec.upd %arg0[%3, %6], %15 {index = 1 : i8, offset = 224 : si32} : memref<18x272xi32>, vector<16xi32>
%19 = aievec.mac %17, %1, %18 {xoffsets = "0x76543210", xstart = "2", zoffsets = "0x00000000", zstart = "0"} :
         vector<16xi32>, vector<8xi32>, !aievec.acc<8xi80>
%20 = aievec.srs %19 {shift = 0 : i8} : !aievec.acc<8xi80>, vector<8xi32>
vector.transfer_write %20, %arg2[%arg3, %arg4] {in_bounds = [true]} : vector<8xi32>, memref<16x256xi32>
```

Comparison with Reference Implementation

Output image size: 256x16, Filter sizes: 3x3 and 5x5

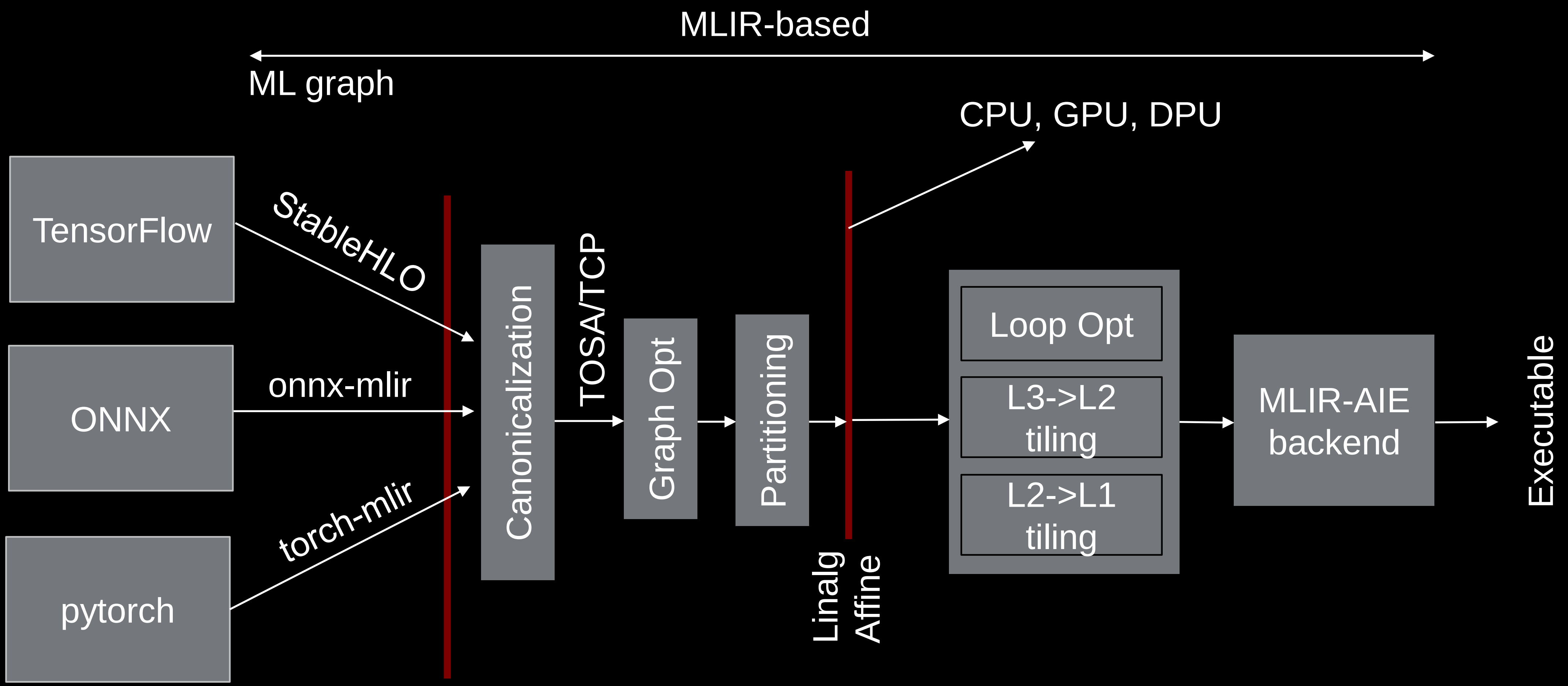




"Vyasa: A High-Performance Vectorizing Compiler for Tensor Convolutions on the Xilinx AI Engine" - HPEC 2020

Machine Learning

Machine Learning in MLIR



See also: https://iree-org.github.io/iree/

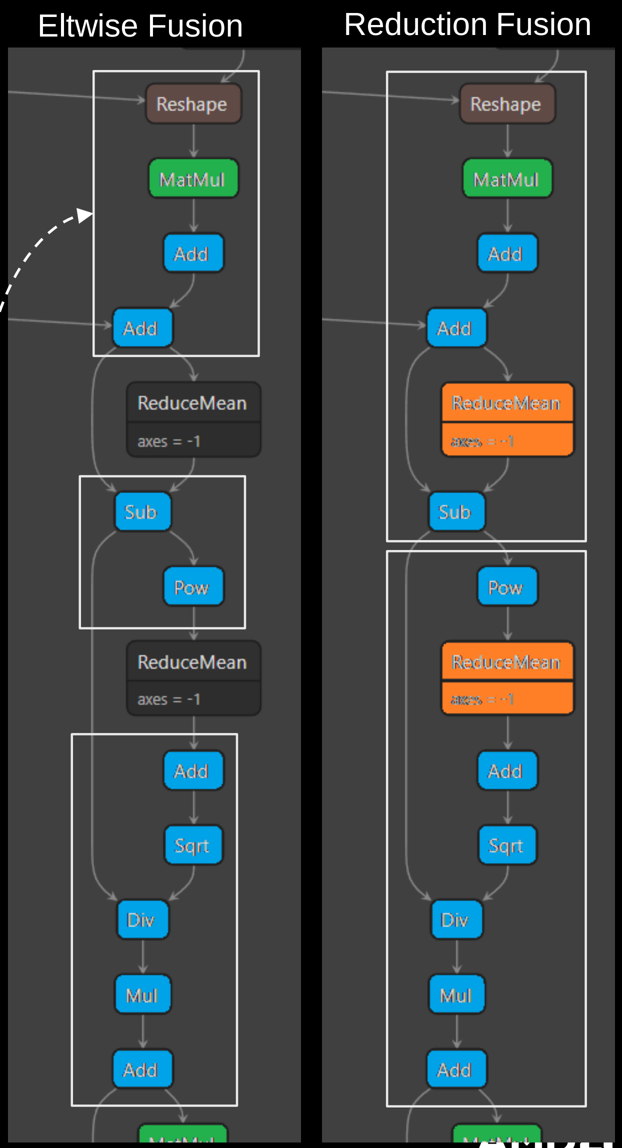
ML Graphs

```
    %105 = "tosa.const"() {value = dense<0.000000e+00> : tensor<512xf32>} : () -> tensor<512xf32>
    %106 = "tosa.transpose"(%arg0, %101) : (tensor<1x3x224x224xf32>, tensor<4xi32>) -> tensor<1x224x224x3xf32>
    %107 = "tosa.conv2d"(%106, %96, %100) {dilation = [1, 1], pad = [3, 3, 3, 3], stride = [2, 2]} :
(tensor<1x224x224x3xf32>, tensor<64x7x7x3xf32>, tensor<64xf32>) -> tensor<1x112x112x64xf32>
    %108 = "tosa.transpose"(%107, %102) : (tensor<1x112x112x64xf32>, tensor<4xi32>) -> tensor<1x64x112x112xf32>
    %109 = "tosa.sub"(%108, %94) : (tensor<1x64x112x112xf32>, tensor<1x64x1x1xf32>) -> tensor<1x64x112x112xf32>
```

```
%3 = linalg.conv_2d_nchw_fchw {dilations = dense<1> : vector<2xi64>, strides = dense<2> : vector<2xi64>}
         ins(%0, %cst_3 : tensor<1x3x230x230xf32>, tensor<64x3x7x7xf32>)
         outs(%2 : tensor<1x64x112x112xf32>) -> tensor<1x64x112x112xf32>
%5 = linalg.generic {indexing_maps = [...], iterator_types = ["parallel", "parallel", "parallel", "parallel"]}
         ins(%3, %cst_6, %cst_7, %cst_4, %cst_5 : tensor<1x64x112x112xf32>, tensor<64xf32>, tensor<64xf32>,
                                                        tensor<64xf32>, tensor<64xf32>)
         outs(%3 : tensor<1x64x112x112xf32>) {
    ^bb0(%arg1: f32, %arg2: f32, %arg3: f32, %arg4: f32, %arg5: f32, %arg6: f32):
      %123 = arith.truncf %cst_2 : f64 to f32
      %124 = arith.addf %arg5, %123 : f32
      %125 = math.rsqrt %124 : f32
      %126 = arith.subf %arg1, %arg4 : f32
      %127 = arith.mulf %126, %125 : f32
      %128 = arith.mulf %127, %arg2 : f32
      %129 = arith.addf %128, %arg3 : f32
      linalg.yield %129 : f32
} -> tensor<1x64x112x112xf32>
```

Graph Optimization and Partitioning

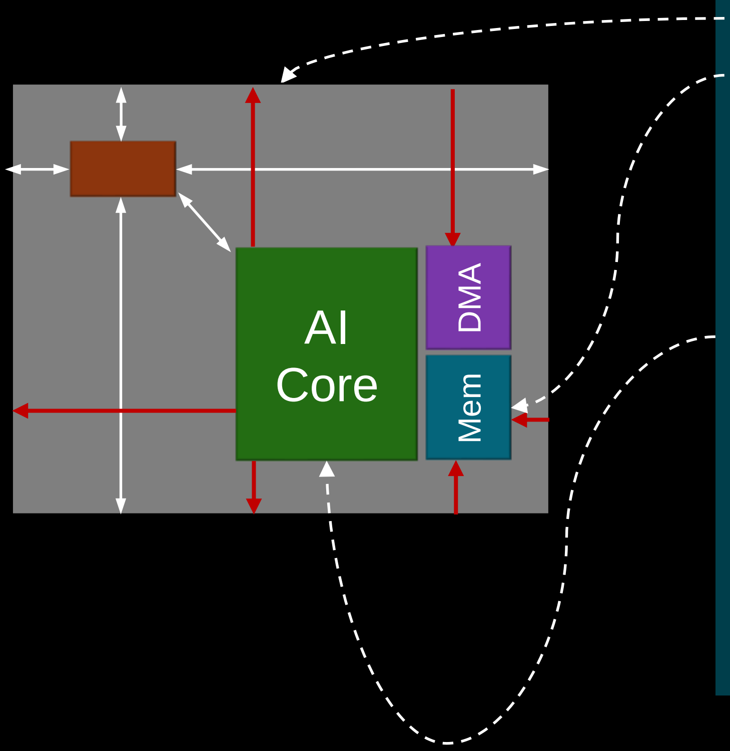
```
func.func @resnet50(%arg0: memref<1x32x32x64xf32>,
                        %arg1: memref<64x3x3x64xf32>,
                    %arg2: memref<64x3x3x64xf32>,
                    %arg3: memref<1x32x32x64xf32>) {
    ...
    %memref_2, %asyncToken_3 = gpu.alloc async [%3] () : memref<64x3x3x64xf32>
    %4 = gpu.memcpy async [%asyncToken_3] %memref_2, %arg1 : memref<64x3x3x64xf32>,
                                                              memref<64x3x3x64xf32>
    %5 = gpu.wait async [%2, %4]
    %6 = gpu.launch_func async [%5] @resnet50__part_0_module::@resnet50__part_0
                   blocks in (%c32, %c1, %c1) threads in (%c64, %c1, %c1)
                   args(%memref_0 : memref<1x32x32x64xf32>,
                    %memref_2 : memref<64x3x3x64xf32>,
                    %memref : memref<1x32x32x64xf32>)
    %7 = gpu.wait async
    ...
```



- Optimization at the network level can greatly reduce memory bandwidth
- Partitioning across devices (CPU, GPU, etc.) enables heterogenous execution
- Async annotations delegate execution dependency to devices

Device-Level Modeling

Running Code on a Core

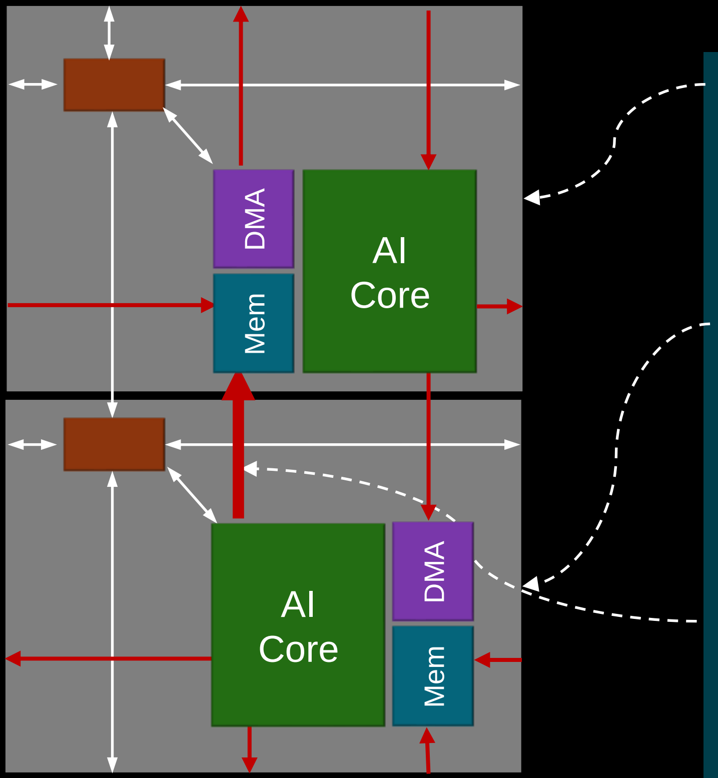


```
%tile13 = AIE.tile(1, 3)
%buf13_0 = AIE.buffer(%tile13)
       { sym_name = "a" } : memref<256xi32>

%core13 = AIE.core(%tile13) {
  %val1 = constant 7 : i32
  %idx1 = constant 3 : index
  %2 = addi %val1, %val1 : i32
  memref.store %2, %buf13_0[%idx1] : memref<256xi32>

  AIE.end
}
```

Moving buffers

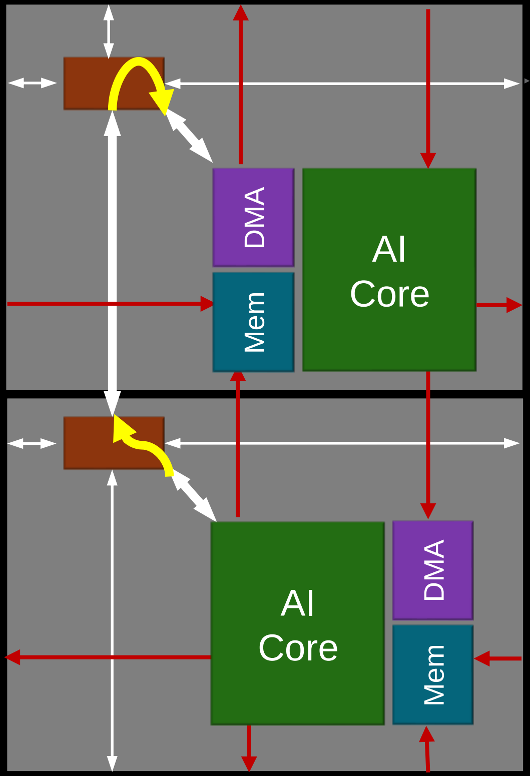


```
%tile14 = AIE.tile(1, 4)
%buf = AIE.buffer(%tile14)
       { sym_name = "a" } : memref<256xi32>

%tile13 = AIE.tile(1, 3)
%core13 = AIE.core(%tile13) {
  %val1 = constant 7 : i32
  %idx1 = constant 3 : index
  %2 = addi %val1, %val1 : i32
  AIE.useLock(%lock, "Acquire", 1)
  memref.store %2, %buf[%idx1] : memref<256xi32>
  AIE.useLock(%lock, "Release", 0)

  AIE.end
}
```

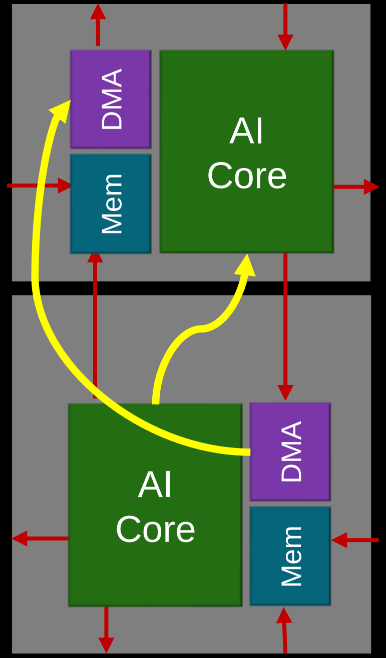
Connecting Streams



```
%tile14 = AIE.tile(1, 4)
AIE.switchbox(%tile14) {
  AIE.connect<"South" : 3, "Core" : 0>
  AIE.connect<"South" : 4, "DMA" : 0>
}

%tile13 = AIE.tile(1, 3)
AIE.switchbox(%tile13) {
  AIE.connect<"Core" : 0, "North" : 3>
  AIE.connect<"DMA" : 0 , "North" : 4>
}
```

Using Flows



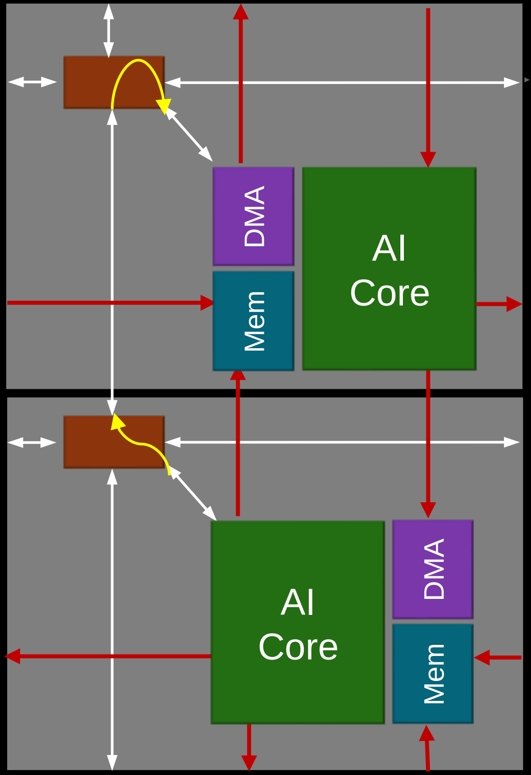
```
AIE.flow(%tile14, "DMA" : 0, %tile13, "DMA" : 0)
AIE.flow(%tile14, "Core" : 0, %tile13, "Core" : 0)
```

aie-opt --aie-create-flows

```
%tile14 = AIE.tile(1, 4)
AIE.switchbox(%tile14) {
  AIE.connect<"South" : 3, "Core" : 0>
  AIE.connect<"South" : 4, "DMA" : 0>
}

%tile13 = AIE.tile(1, 3)
AIE.switchbox(%tile13) {
  AIE.connect<"Core" : 0, "North" : 3>
  AIE.connect<"DMA" : 0 , "North" : 4>
}
```

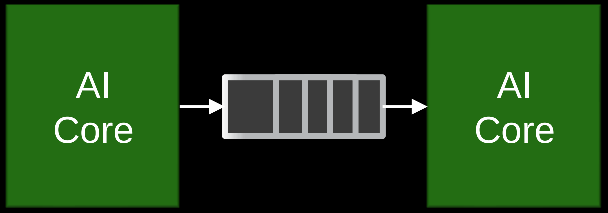
Using DMAs



```
%tile14 = AIE.tile(1, 4)
%mem14 = AIE.mem(%tile14) {
  %dma0 = AIE.dmaStart("S2MM", 0, ^bd0, ^end)
  ^bd0:
    AIE.useLock(%lock14_0, "Acquire", 1)
    AIE.dmaBd(<%buf: memref<512xi32>, 0, 512>, 0)
    AIE.useLock(%lock14_0, "Release", 0)
    br ^bd0
  ^end:
  AIE.end
}
%core14 = AIE.core(%tile14) {
  AIE.useLock(%lock, "Acquire", 0, 0)
  %data = memref.load %buf[%idx1] : memref<512xi32>
  AIE.useLock(%lock, "Release", 1, 0)
}
```

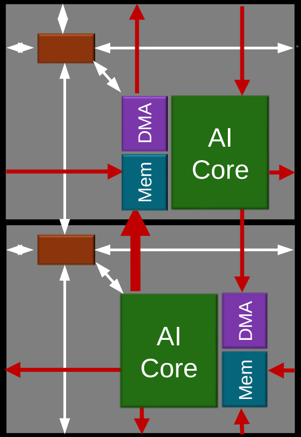
Abstract Communication with Object Fifo

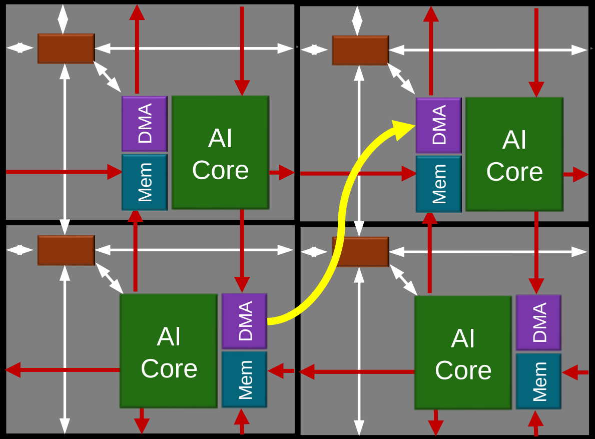
```
%f = AIE.objectFifo.createObjectFifo(%src, {%dst}, 2)
AIE.core(%src) {
    scf.for %indexInLine = %c0 to %c16 step %c1 {
        %v1 = AIE.objectFifo.acquire<Produce>(%objFifo, 1)
        ...
        AIE.objectFifo.release<Produce>(%objFifo, 1)
    }
    AIE.end
}
AIE.core(%dst) {
    scf.for %indexInLine = %c0 to %c16 step %c1 {
       %v1 = AIE.objectFifo.acquire<Consume>(%objFifo, 1)
       ...
       AIE.objectFifo.release<Consume>(%objFifo, 1)
    }
    AIE.end
}
```



ObjectFifo Implementation

aie-opt --aie-register-objectFifos --aie-objectFifo-stateful-transform --aie-lower-multicast





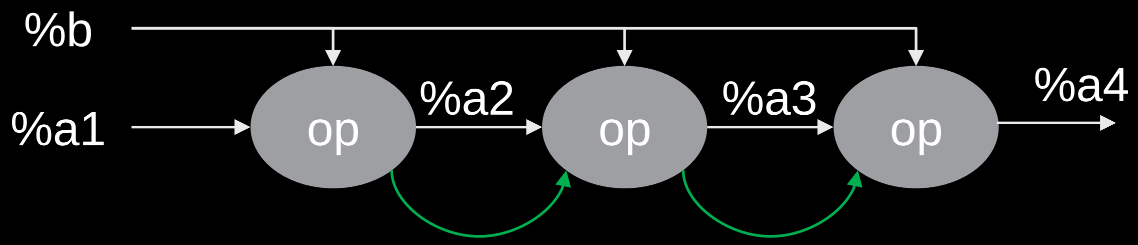
```
%4 = AIE.buffer(%0) : memref<16xi32>
%5 = AIE.lock(%0, 0)
%6 = AIE.buffer(%0) : memref<16xi32>
%7 = AIE.lock(%0, 1)
%8 = AIE.core(%0) { ... }
%9 = AIE.core(%1) { ... }
```

```
%6 = AIE.buffer(%0) : memref<16xi32>
%7 = AIE.lock(%0, 0)
%8 = AIE.buffer(%0) : memref<16xi32>
%9 = AIE.lock(%0, 1)
%10 = AIE.buffer(%2) : memref<16xi32>
%11 = AIE.lock(%2, 1)
%12 = AIE.buffer(%2) : memref<16xi32>
%13 = AIE.lock(%2, 2)
%14 = AIE.core(%0) { ... }
%15 = AIE.core(%2) { ... }
%16 = AIE.mem(%0) { ... }
%17 = AIE.mem(%2) { ... }
AIE.flow(%0, DMA : 0, %1, DMA : 0)
```

Programmable Logic and CIRCT

Basic MLIR Concepts

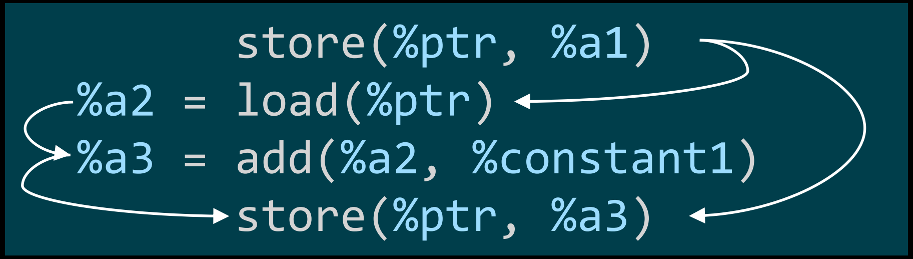
```
ssaRegion(%a1, %b) {
  %a2 = op(%a1, %b)
  %a3 = op(%a2, %b)
  %a4 = op(%a3, %b)
}
```



- At a fundamental level MLIR represents:
  - An ordered list of operations (nodes in a graph)
  - The connections between operations (edges in a graph)

- A list of operations represents a DAG (with multiple destination edges).
- Every DAG can be represented as a (not unique) list of operations.

This is great for representing sequential programs!



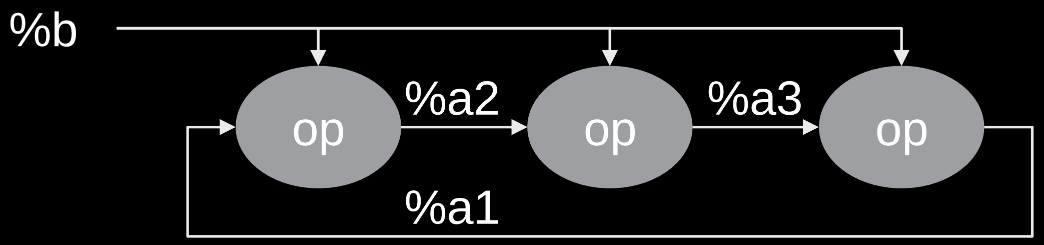
```
      store(%ptr, %a1)
%a2 = load(%ptr)
%a3 = add(%a2, %constant1)
      store(%ptr, %a3)
```

- Encodes data dependencies (load -> add -> store)
- Encodes sequential dependencies (store -> load, store -> store)

- With Basic Blocks can represent Single-static assignment control flow graph
- Arbitrary operations enables representing structured control flow

Graph Regions

```
NOTSSA(%b) {
  %a2 = op(%a1, %b)
  %a3 = op(%a2, %b)
  %a1 = op(%a3, %b)
}
```



- But can we represent an *arbitrary* graph *with cycles*?
  - **Yes! Graphs are directly supported in MLIR through *Graph Regions***
- Is this useful? What can it mean?
  - **Yes! Graphs are great for representing concurrent programs!**

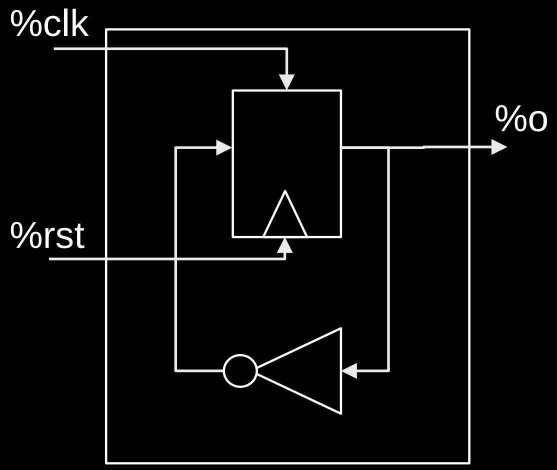
CIRCT: MLIR for Hardware

- Next generation open source synthesis infrastructure
  - LLVM incubator project
  - Industry: AMD, SiFive, Microsoft
  - Academia: ETH, EPFL, Cornell, TU Darmstadt, etc.
  - National Labs: PNNL
- Focus on RTL + HLS
  - Interfaces with SystemVerilog, Chisel, C++
  - Interest in FPGA+ASIC targets
  - Integrated simulation through LLVM backends
- Key concept: Graph Regions
  - Convenient representation of concurrent hardware

CIRCT RTL Dialects

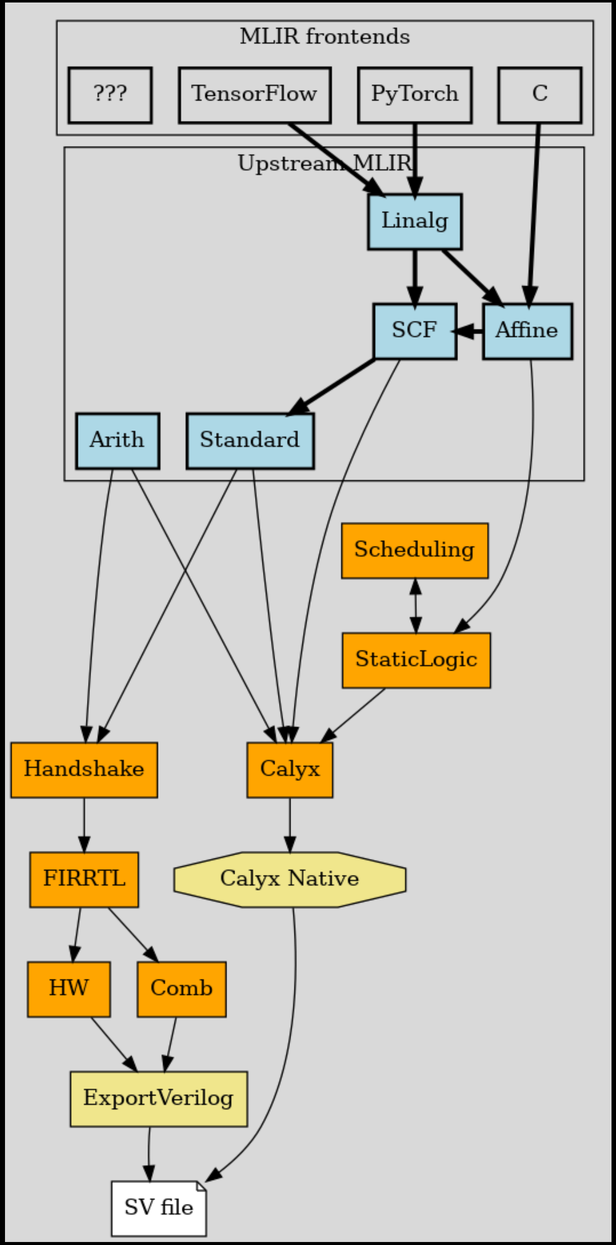
- `hw` Dialect: basic hierarchy for circuit descriptions
- `comb` Dialect: combinational logic gates
- `seq` Dialect: sequential (registered) components

```
hw.module @top(%clk: i1, %rst: i1) -> (o: i1) {
  %true = hw.constant true
  %false = hw.constant false
  %r0 = seq.compreg %0, %clk, %rst, %false  : i1
  %0 = comb.xor %r0, %true : i1
  hw.output %r0 : i1
}
```

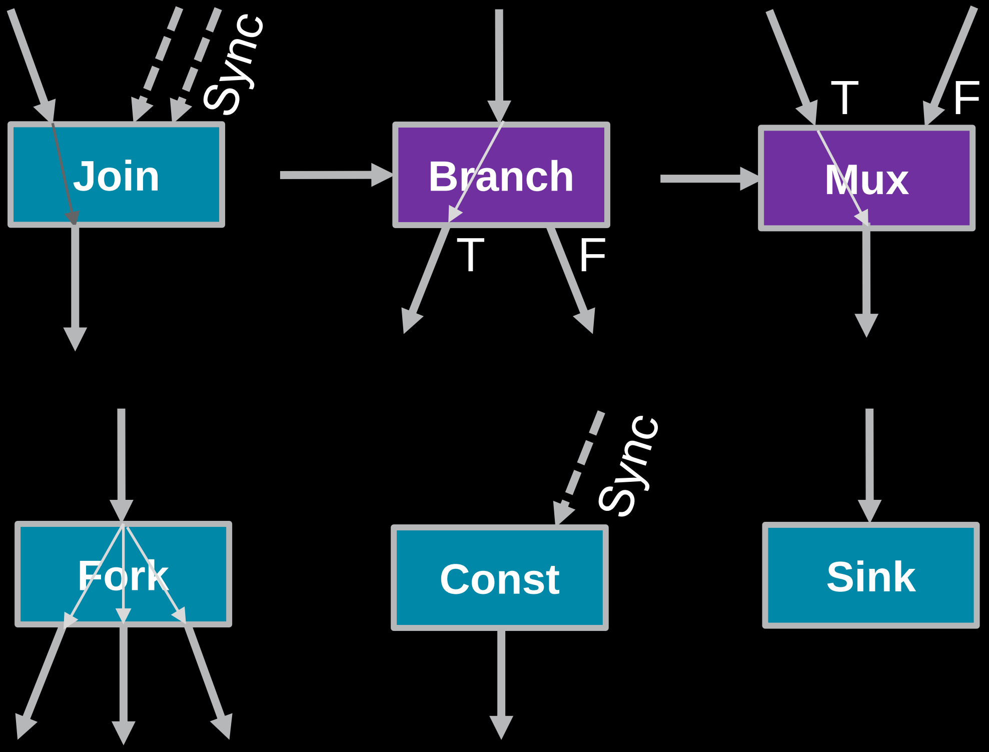


HLS

- Production Chisel/FIRRTL compiler
  - 5-10x faster than previous implementations
- 2 C to RTL Backend Flows
  - Statically Scheduled
  - Dynamically Scheduled
- Related work using MLIR + Vitis HLS
  - "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation" HPCA 2022



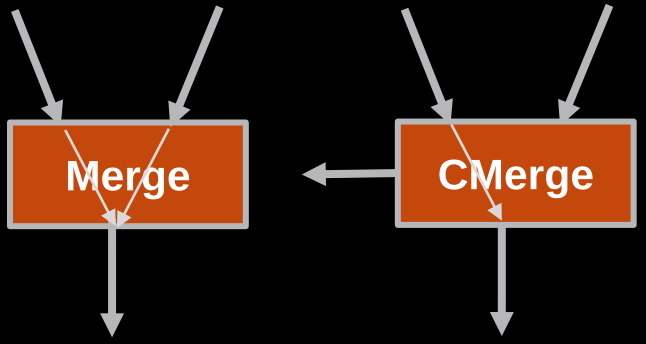
CIRCT Handshake Dialect



Deterministic operators

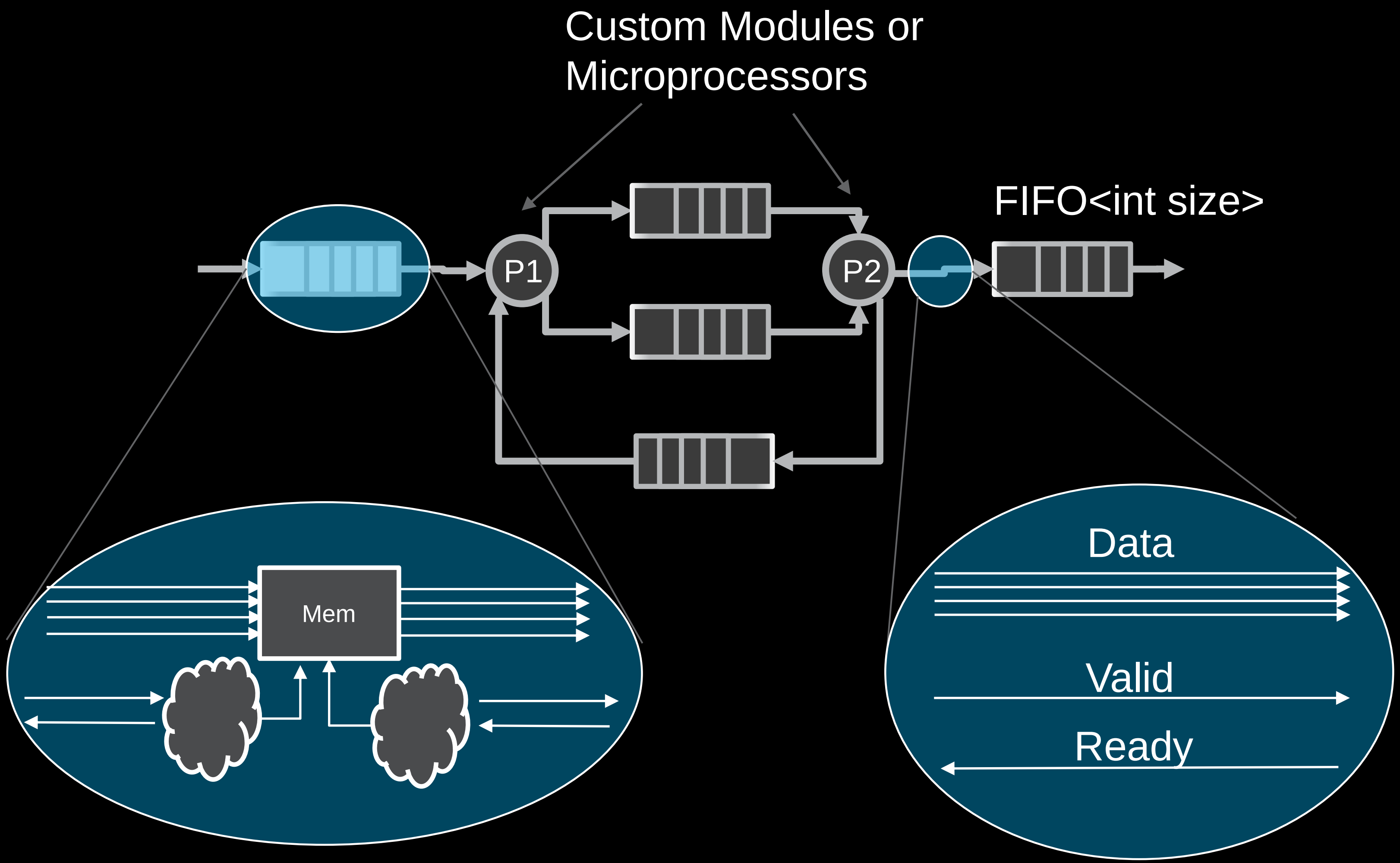
```
handshake.func @name (...) -> (...) {...}
```

Graph-Region container



Non-Deterministic operators

Simple Hardware Representation



AXI4-stream protocol
Data is transferred when Valid and Ready are both asserted in the same clock cycle

Valid can only be de-asserted when Ready is asserted and data is transferred

Asserting Valid cannot be dependent on waiting for Ready to be asserted

Latency Insensitive Elastic Circuits -> Simple Timing Closure

Conclusion

- Heterogenous devices -> Heterogeneous compiler
- Multi-Level IR good at representing different levels of abstraction in a system
- Multiple MLIR use models:
  - Sequential (SSA) semantics
  - Concurrent (Graph) semantics
  - Structural models
  - ML (DAG) models
- Broad Open Source substrate with more to come
  - Would love to hear your ideas

Copyright and disclaimer