T2: Leveraging MLIR to Design for AI Engines

@FPGA2023 Jack Lo, Andra Bisca, Samuel Bayliss, Kristof Denolf, Joseph Melber

@our_team Jeff Fifield, Phil James-Roxby, Alireza Khodamoradi, Eddie Richter, Mark Sears, Erwei Wang, Stephen Neuendorffer, Ralph Wittig

CTO office, Adaptive, Embedded and AI Group (AEAI)

AMD
together we advance_

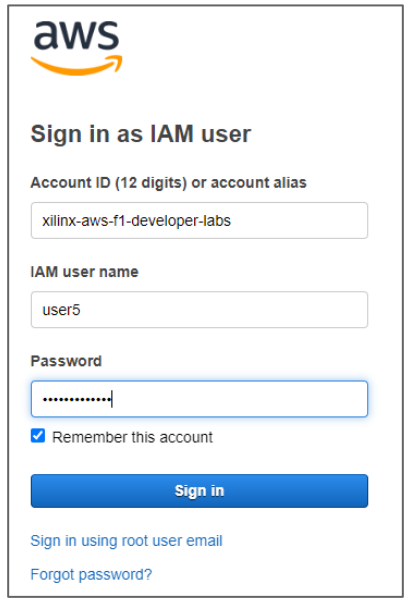
Agenda Leveraging MLIR to Design for AI Engines
Sunday February 12th, 2023, 1:30pm-5:30pm PST

Time	Topic	Presenter	HandsOn Online tutorial #
1:30 – 1:40	Welcome and Logistics	Kristof	
1:40 – 2:15	Introduction to MLIR and AIEngine	Joe	
2:15 – 3:00	Physical MLIR-AIE	Sam/Joe	Tutorial 1
3:00 – 3:30	Break + AWS Troubleshoot		
3:30 – 4:15	Host Code, Simulation and Performance Counters	Jack	Tutorial 2 (a,b,c)
4:15 – 4:45	Logical MLIR-AIE: Communication	Andra	Tutorials 3,4,5
4:45 – 5:15	Scale up to Many-AIE Designs and Hook to MLIR-AIR	Sam	Tutorial 11 or design examples
5:15 – 5:30	Wrap-Up	Kristof	

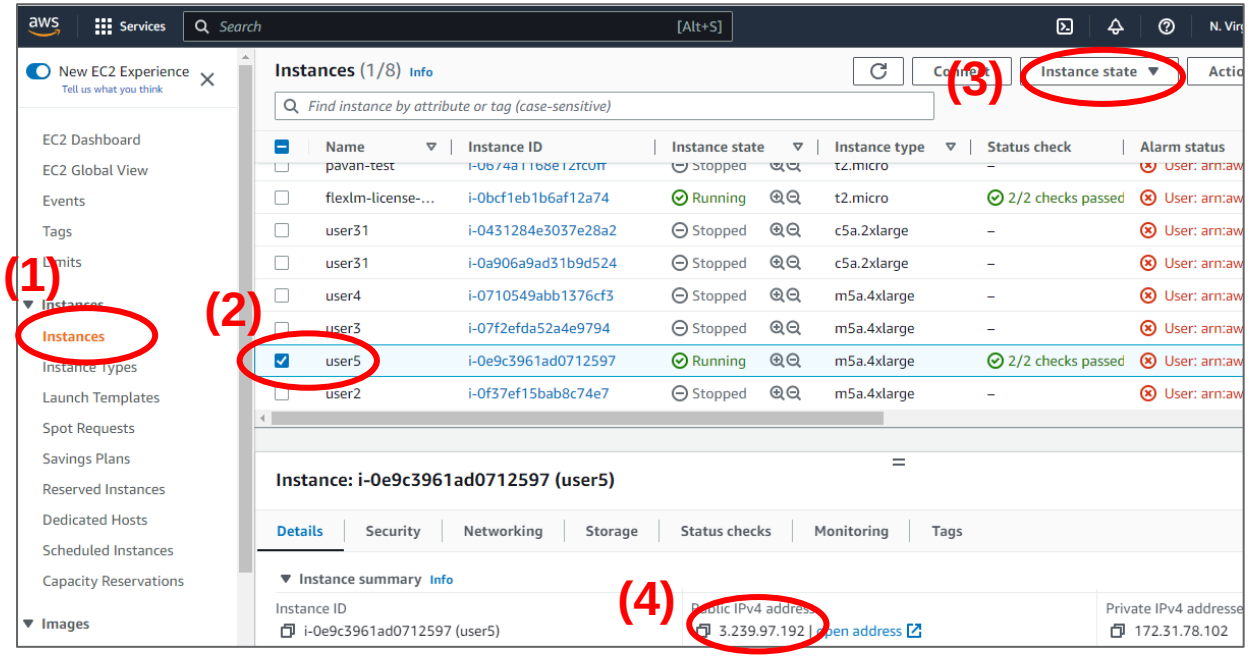
https://github.com/Xilinx/mlir-aie/tree/main/tutorials

AMD
together we advance_

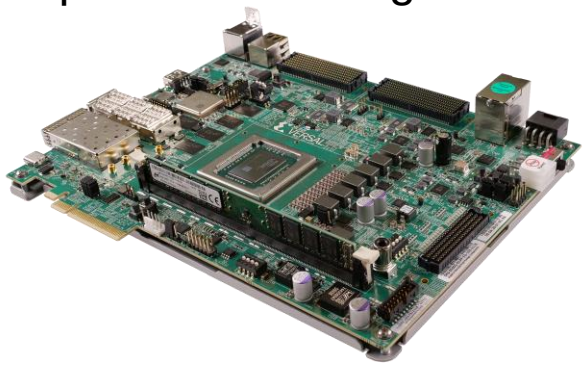
Hands on: AWS Login and VCK-190s in the Back



- AWS Login details (details for login is sent to your email)
  1. https://console.aws.amazon.com/ec2/
     - Account ID – xilinx-aws-f1-developer-labs
     - IAM user name – user# that was sent to your email (e.g. user5)
     - Password – xup_vitis2022
  2. Click on (1) “Instances”, select (2) “user#”, select (3) “Instance state” and “Start Instance”
  3. Download NICE Desktop Cloud Visualization Client
     - https://download.nice-dcv.com/
  4. Connect to (4) IP Address via NICE DCV
     - Username: ubuntu, Password: xup_vitis2022



- Optional: Experience running code on a VCK-190 board



AMD
together we advance_

Introduction to AIEngines

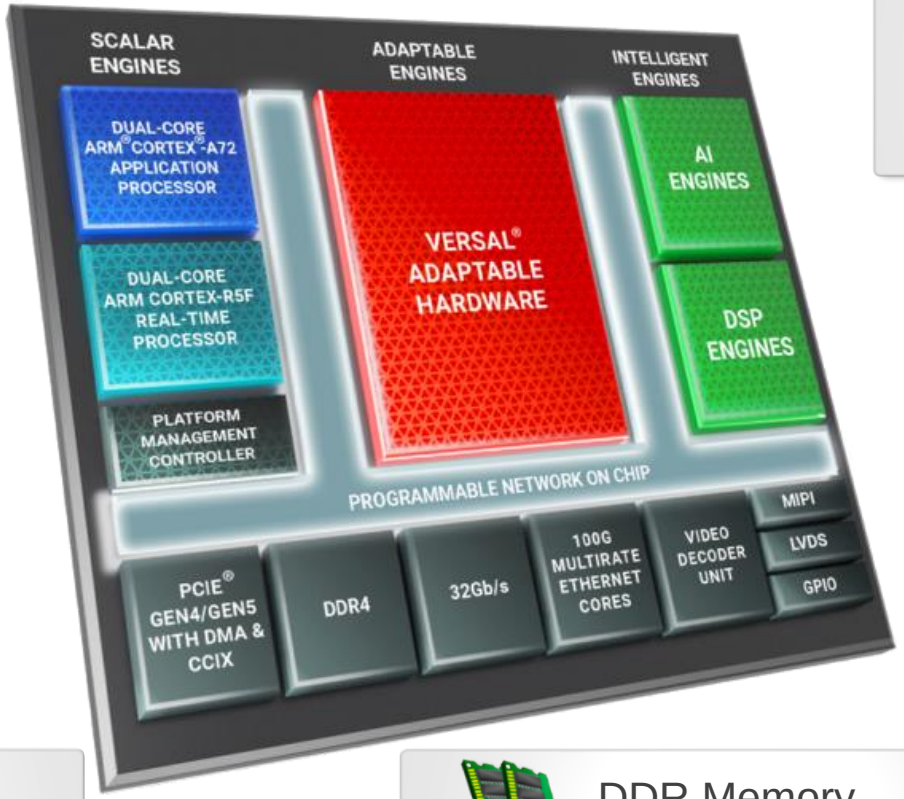
- Welcome & Logistics
- Introduction to AIEngines
- Physical MLIR-AIE
- Host Code, Simulation, Performance Counters
- Logical MLIR-AIE: Communication
- Scale up to Many-AIE Designs and Hook to MLIR-AIR
- Wrap Up

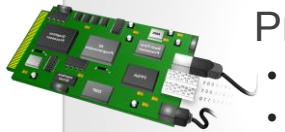
Versal ACAP Architecture

Adaptable Engines
2X compute density

Scalar Engines
- Platform Control
- Edge Compute

AI Engines
- AI Compute
- Diverse DSP workloads

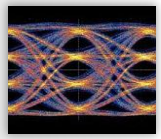




Protocol Engines
- Integrated 600G cores
- 4X encrypted bandwidth

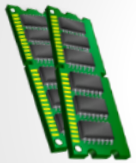
Network-on-Chip
- Guaranteed Bandwidth
- Enables SW Programmability

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



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

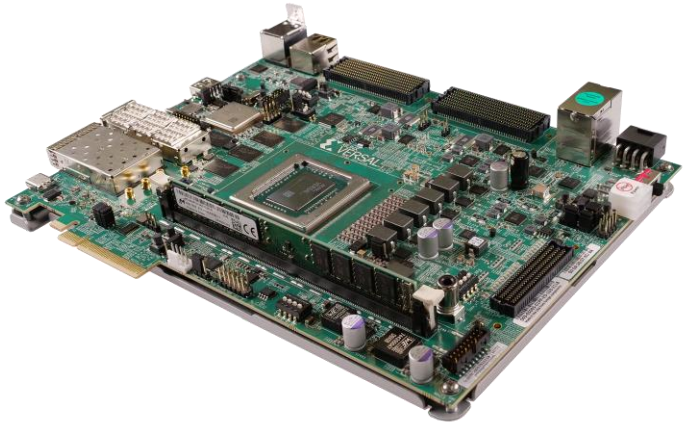
PCIe & CCIX
- 2X PCIe & DMA bandwidth
- Cache-coherent interface to accelerators



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

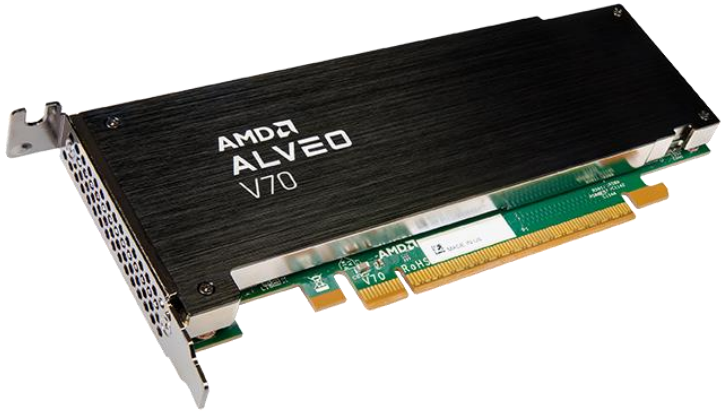
AMD
together we advance_

Yes, It’s Really Here



VCK190
1st gen AIE

VCK5000
1st gen AIE



V70
AMD XDNA
2nd gen AIE-ML

AMD
together we advance_

Heterogenous Accelerated Compute Clusters (HACC)

- Extend scope to conduct state-of-the-art research on heterogeneous accelerated computing by incorporating GPUs into CPU+FPGA/AIE systems
- The scope still encompasses systems, architecture, tools and applications
- Internal senior technical advisors now across the broader scope
- Heterogeneous reference systems
  - On-premise at the HACC sites (infrastructure research and innovation)

- Tightly integrated reference system
- Per node (SMC 4124GS)
  - 2 EPYC Milan/X CPUs
  - 2-4 MI210 GPUs
  - 2 Alveo U55C FPGA with HBM
  - 2 VCK-5000 ACAP/Versal with AIEs
  - Direct inter-board connections where appropriate
  - Run-time via ROCm, XRT
  - SW development via HIP, Vitis, frameworks

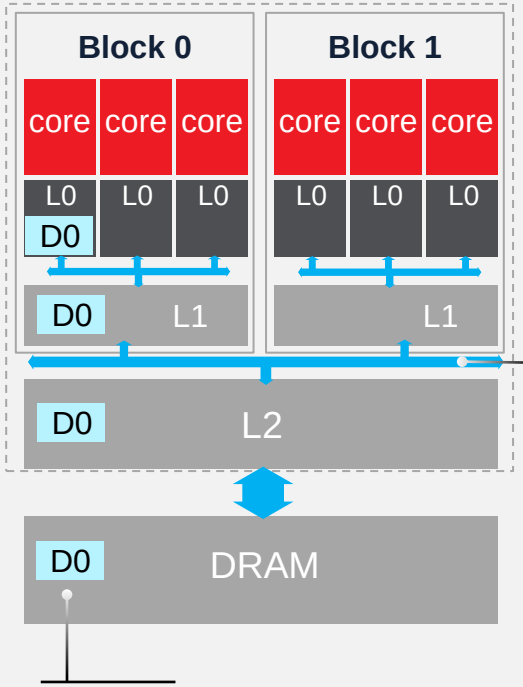
How can we target all of these heterogeneous devices?

AMD
together we advance_

AI Engine Architecture

AI Engine: Multi-Core Compute with Dedicated Memory

Traditional Multi-core
(cache-based architecture)

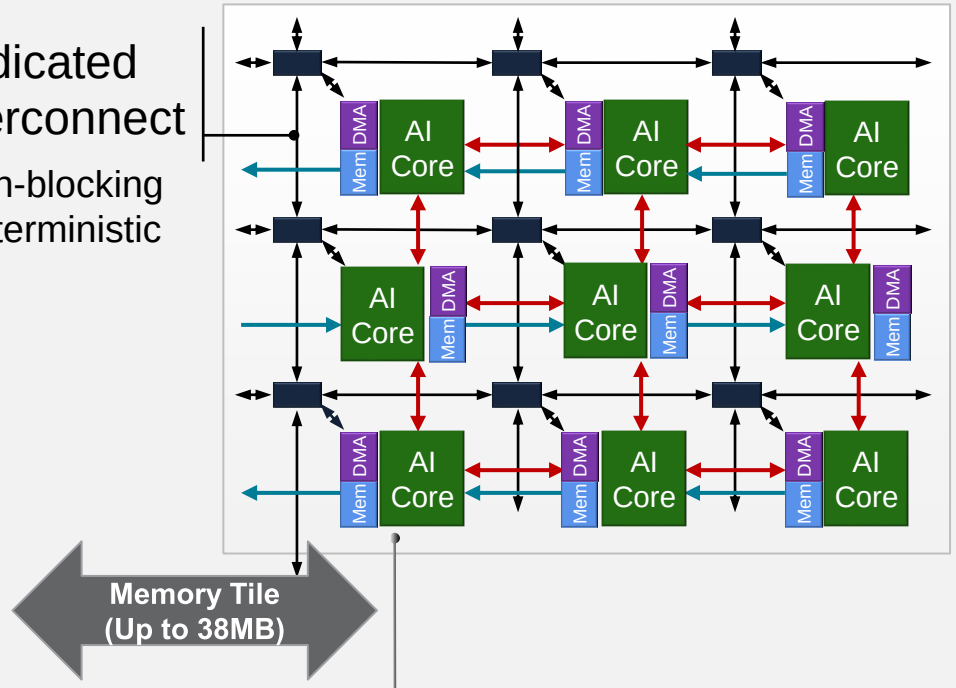


Fixed, Shared Interconnect
- Blocking limits compute
- Timing not deterministic

Data Replicated
- Robs bandwidth
- Reduces capacity

AI Engine Array
(adaptable distributed architecture)

Dedicated Interconnect
- Non-blocking
- Deterministic

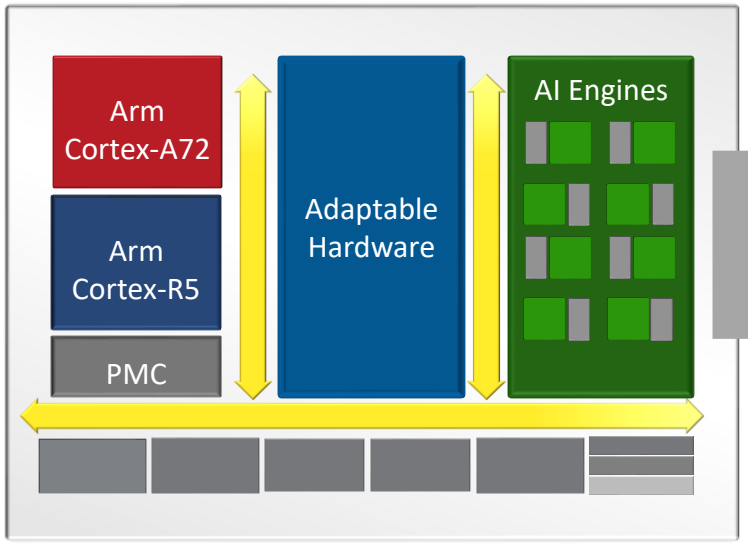


Local, Distributed Memory
- No cache misses
- Higher bandwidth
- Less capacity required

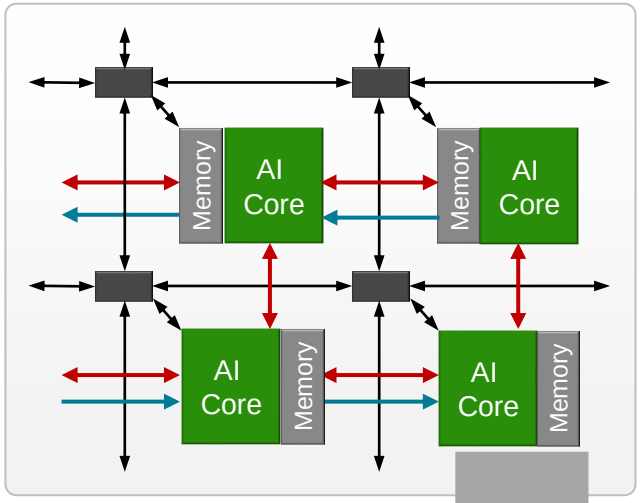
AMD
together we advance_

AI Engine: Terminology

Versal ACAP

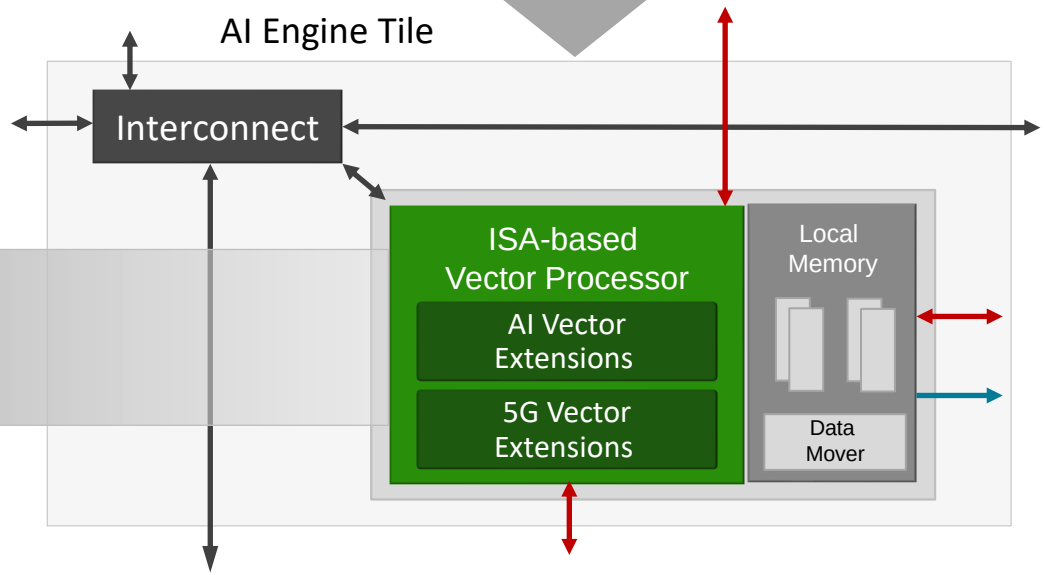


AI Engine Array

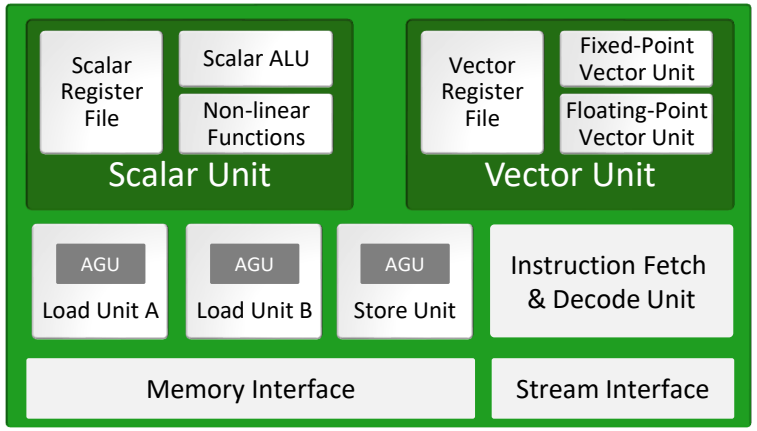


Memory Interface
Stream Interface
Cascade Interface

AI Engine Tile

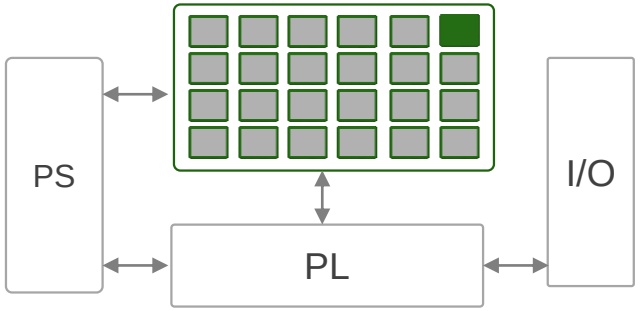


AI Engine Core



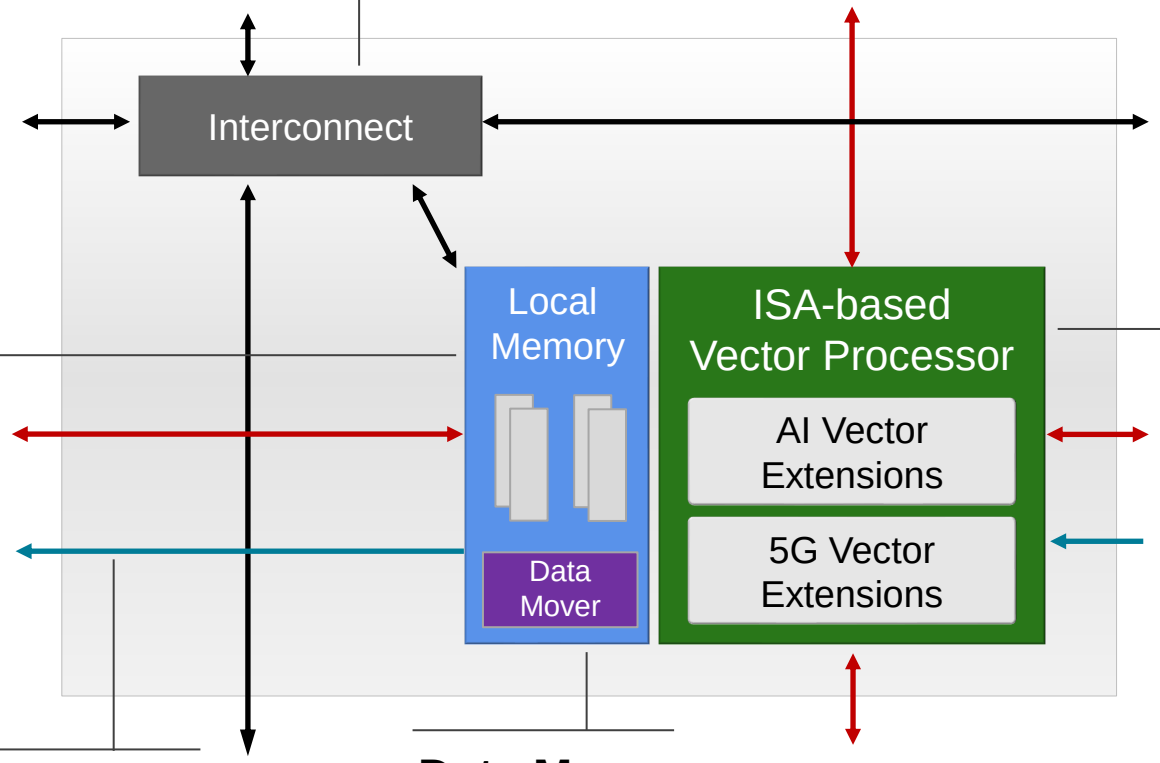
AMD
together we advance_

AI Engine: Tile-Based Architecture



Non-Blocking Interconnect
high GB/s bandwidth per tile

Memory Interface
Stream Interface
Cascade Interface



Local Memory
Multi-bank implementation
Shared across neighbor cores

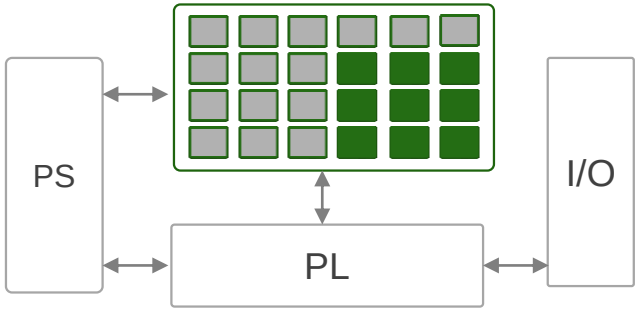
ISA-based Vector Processor
Software Programmable
(e.g., C/C++)

Cascade Interface
Partial results to next core

Data Mover
Non-neighbor data communication
Integrated synchronization primitives

AMD
together we advance_

AI Engine: Array Architecture



Array of AI Engines

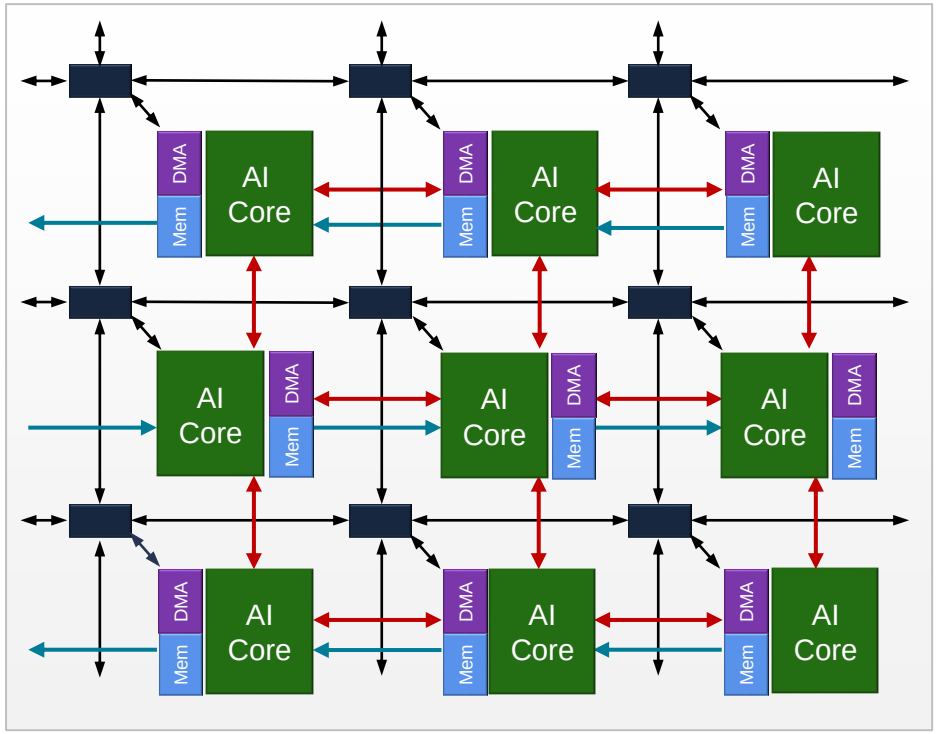
- Increase in compute, memory and communication bandwidth

Modular and scalable architecture

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

Distributed memory hierarchy

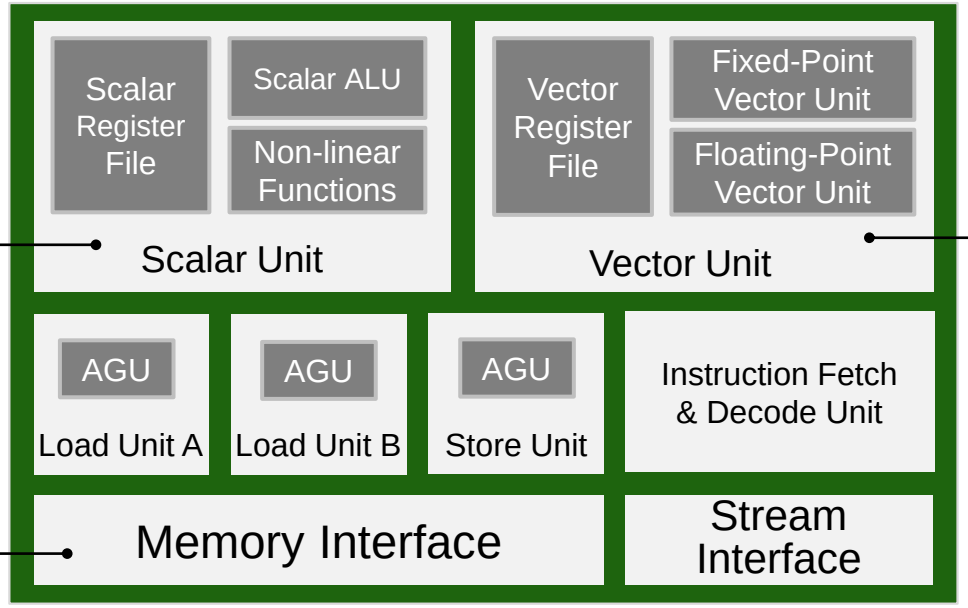
Maximize memory bandwidth



Deterministic Performance & Low Latency

AMD
together we advance_

AI Engine: Processor Core



32-bit Scalar RISC Processor

Vector Processor
512-bit SIMD Datapath

Local, Shareable Memory
- 32KB Local, 128KB Addressable

Instruction Parallelism: VLIW

7+ operations / clock cycle

- 2 Vector Loads / 1 Mult / 1 Store
- 2 Scalar Ops / Stream Access

Highly Parallel

Data Parallelism: SIMD

Multiple vector lanes

- Vector Datapath
- 8 / 16 / 32-bit & SPFP operands

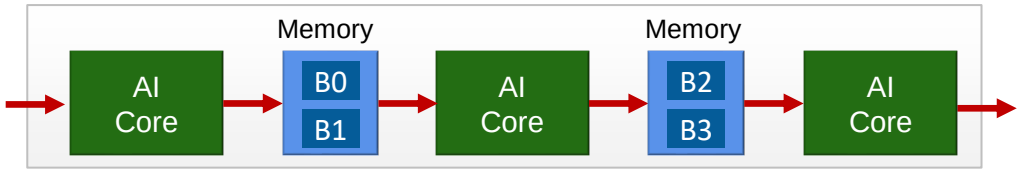
Up to 128 MACs / Clock Cycle per Core (INT 8)

AMD
together we advance_

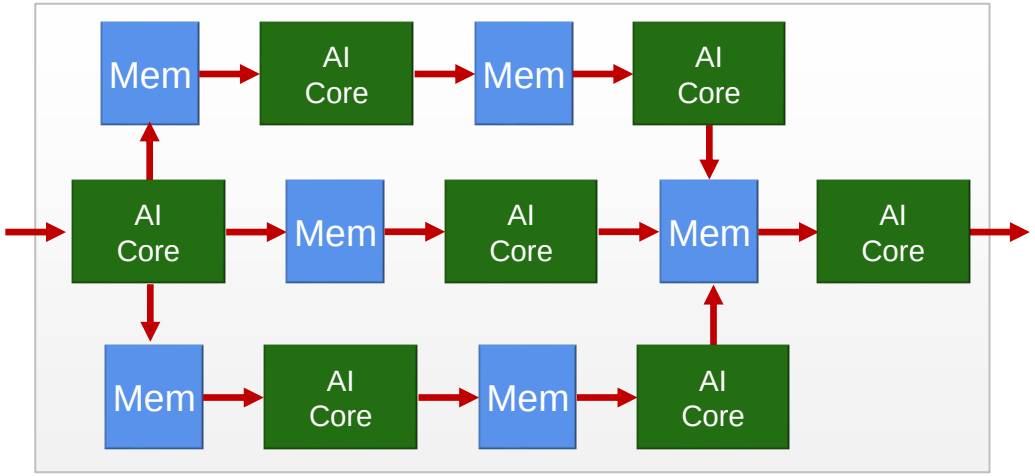
Data Movement Architecture

Memory Communication

Dataflow Pipeline



Dataflow Graph



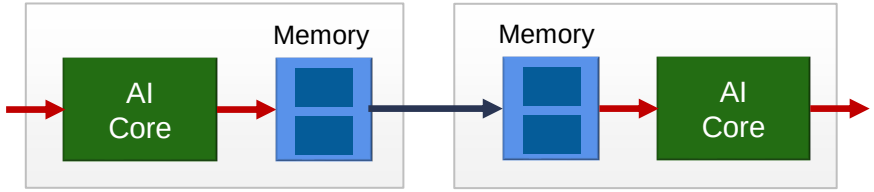
Memory Interface

Stream Interface

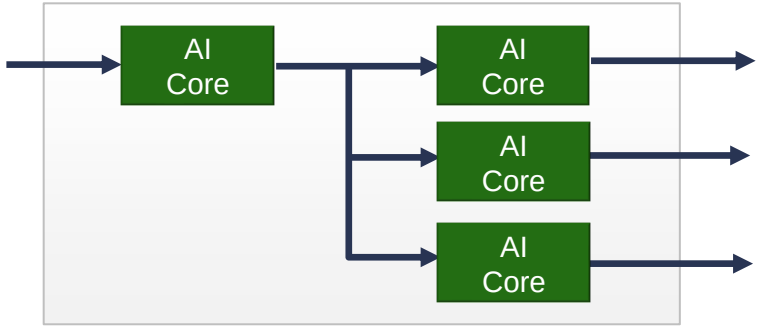
Cascade Interface

Streaming Communication

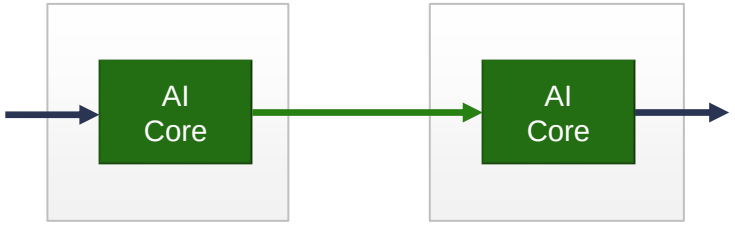
Non-Neighbor



Streaming Multicast

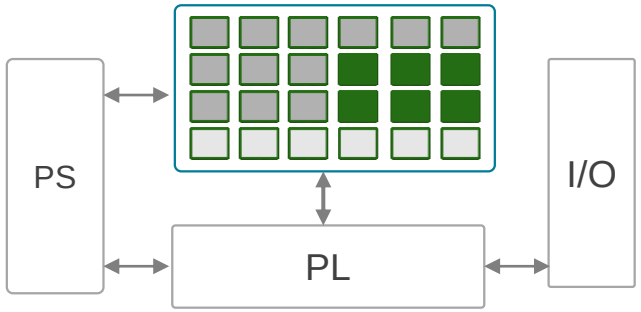


Cascade Streaming

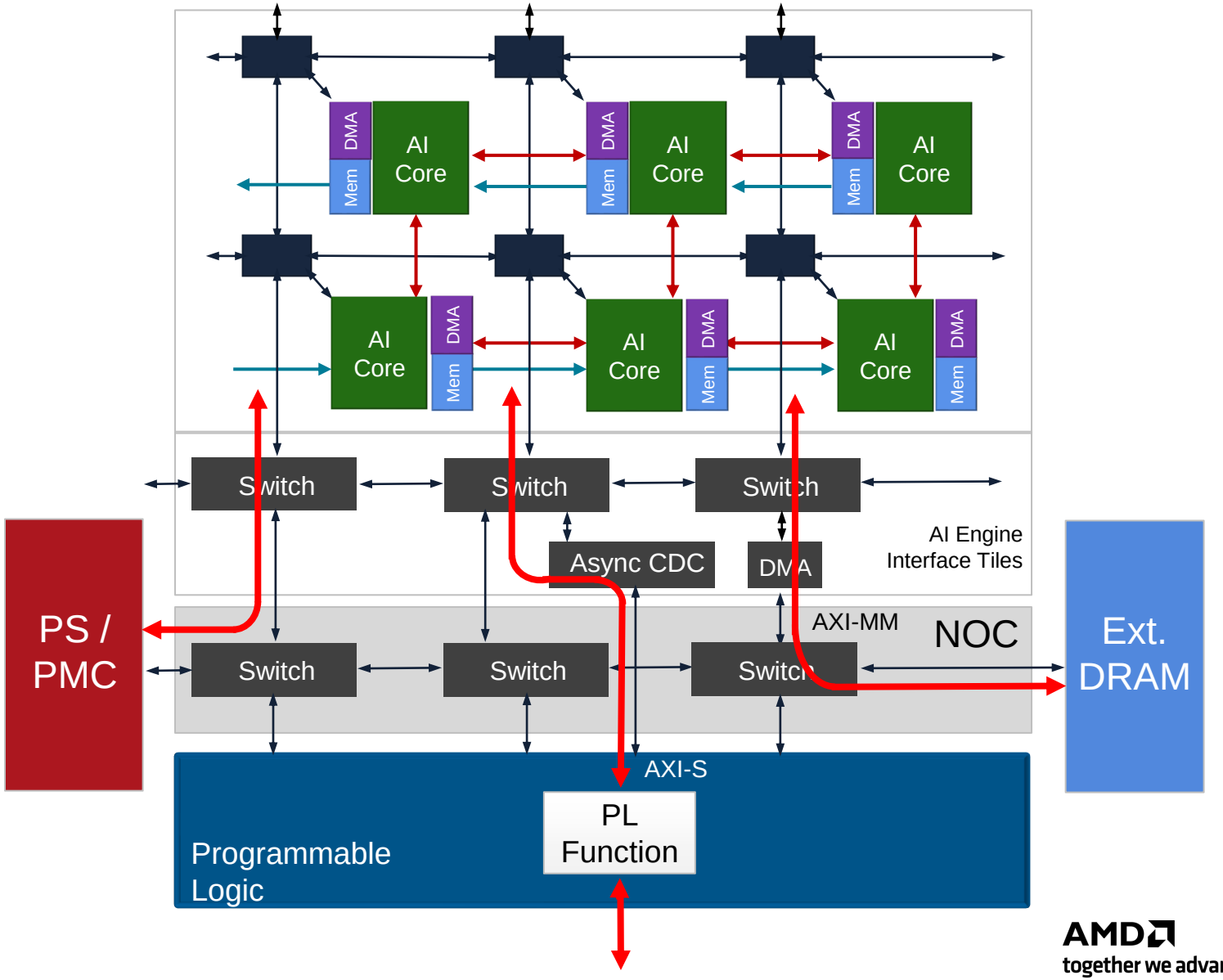


AMD
together we advance_

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)

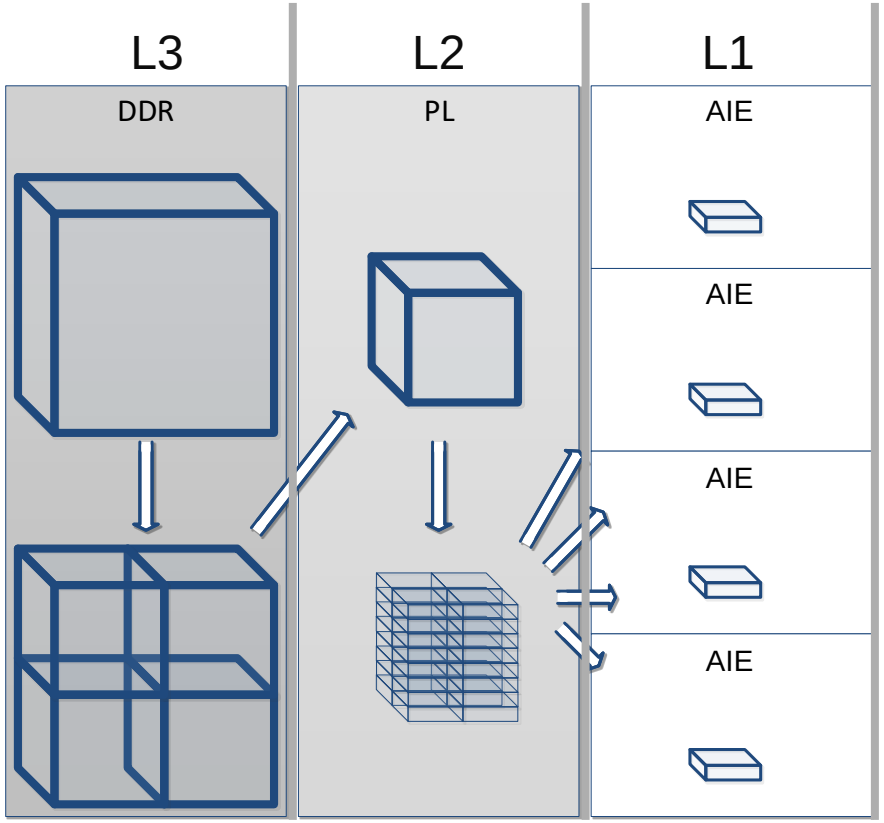


Memory Interface
Stream Interface
Cascade Interface

AMD
together we advance_

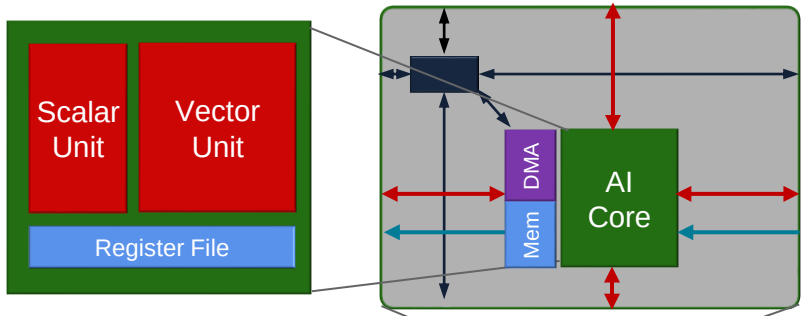
Challenge: Handling Large Data Sets

- Many applications can't easily be streamed
  - 3D Volumetric Data -> Weather Prediction
  - 2D Batched Data -> Image-based Machine Learning
  - 1D Time-Series Data -> 5G Beamforming/SAR
- Often large data sets (~100GB)
  - Doesn't fit on-chip
- Data must be tiled for processing on AIE
- Data must be prefetched and streamed
- Data reuse must be considered at multiple levels



AMD
together we advance_

Memory Reuse is Critical!

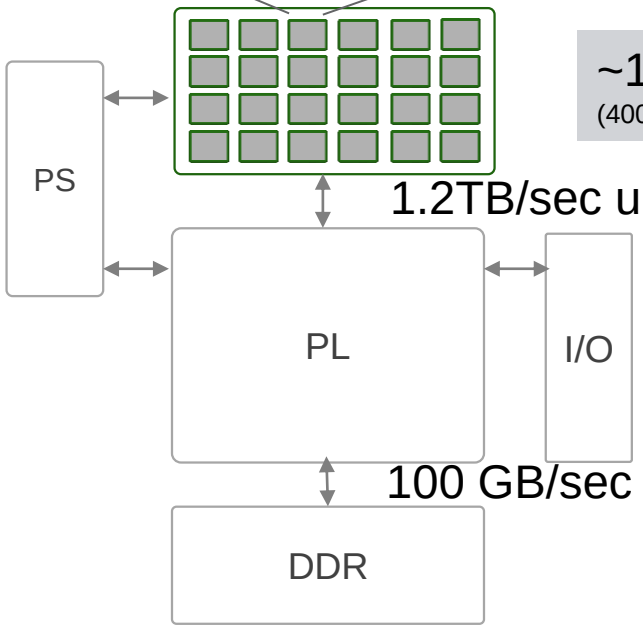


102TB/sec
(400 cores@2x128 bytes per instruction)

4x register file reuse

25.6TB/sec
(400 cores@2x 256-bit load per instruction)

21.3x core memory reuse (no broadcast)
2.7x core memory reuse (8x broadcast)



~13MByte
(400 cores@32kB)

1.2TB/sec up (300 streams @ 4GB/sec)

~20MByte
(432URAM + 967 BRAM)

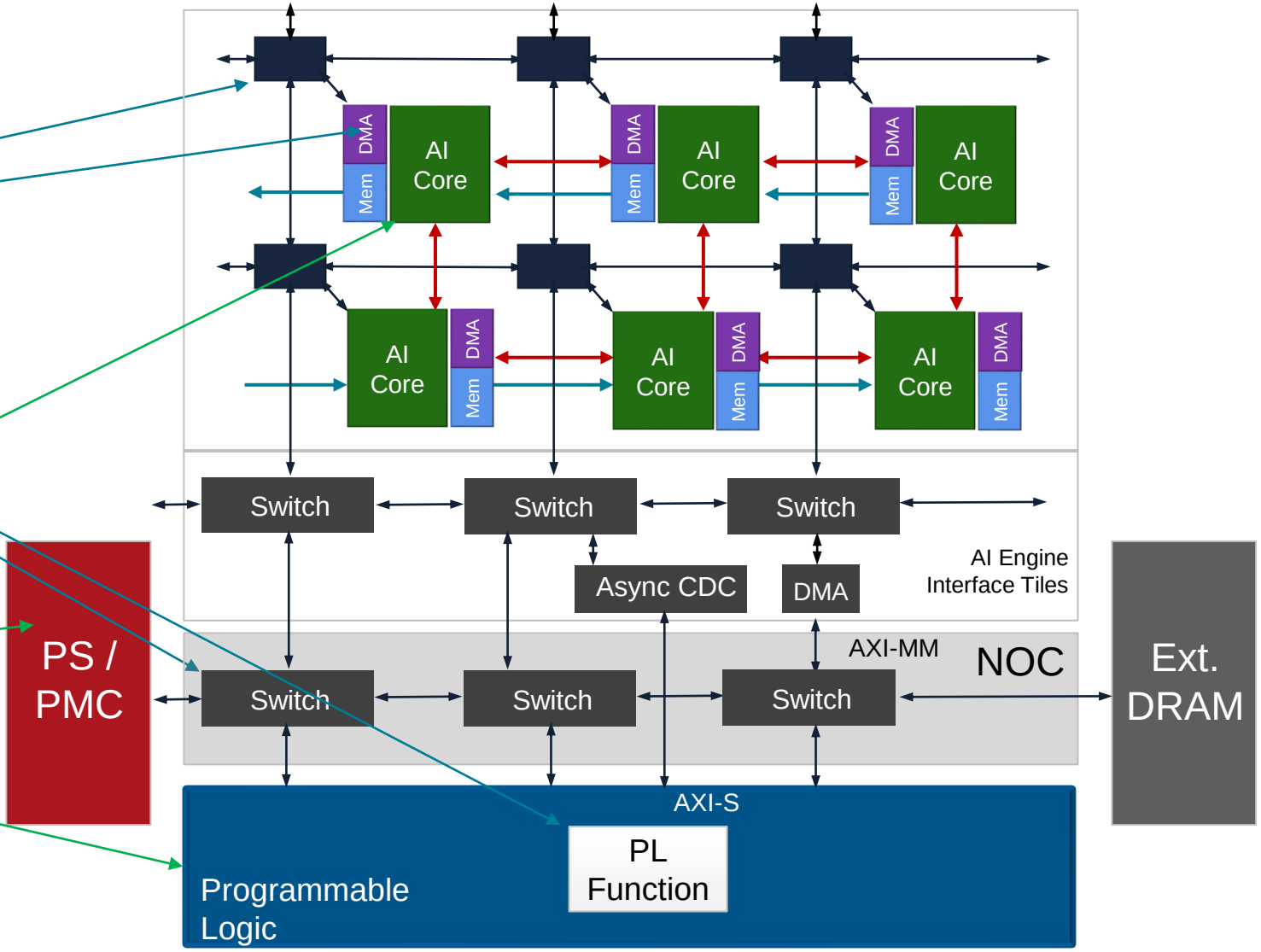
12x on-chip reuse

100 GB/sec (read/write)

AMD
together we advance_

Adaptability Where You Want It, Efficiency Where You Need It

- Accessible
- Adaptable
- Compute



Memory Interface
Stream Interface
Cascade Interface

AMD
together we advance_

MLIR-AIE

- Welcome & Logistics
- Introduction to AIEngines
- Physical MLIR-AIE
- Host Code, Simulation, Performance Counters
- Logical MLIR-AIE: Communication
- Scale up to Many-AIE Designs and Hook to MLIR-AIR
- Wrap Up

Various Toolflows

- Vitis AI
  - Audience: ML programmers
  - ML programming model
  - Details of architecture abstracted using an overlay
- Vitis
  - Audience: DSP programmers
  - Graph programming model
  - Some architecture concepts exposed

- **MLIR-AIE: Open-Source Multi-level IR toolflow**
  - Audience: Researchers and Tool developers
  - Enable multiple programming abstractions
  - Architectural details exposed and accessible

https://github.com/Xilinx/mlir-aie

MLIR-based AIEngine toolchain

build passing | open pull requests 4

This repository contains an MLIR-based toolchain for Xilinx Versal AIEngine-based devices. This can be used to generate low-level configuration for the AIEngine portion of the device, including processors, stream switches, TileDMA and ShimDMA blocks. Backend code generation is included, targetting the LibXAIE library. This project is primarily intended to support tool builders with convenient low-level access to devices and enable the development of a wide variety of programming models from higher level abstractions. As such, although it contains some examples, this project is not intended to represent end-to-end compilation flows or to be particularly easy to use for system design.

Full Documentation

You can:

Assemble AIEngine designs in a 'structural' way
Compile designs relatively quickly
Build your own higher-level tools
Contribute to ongoing open-source development

AMD
together we advance_

MLIR: Multi-Level Intermediate Representation

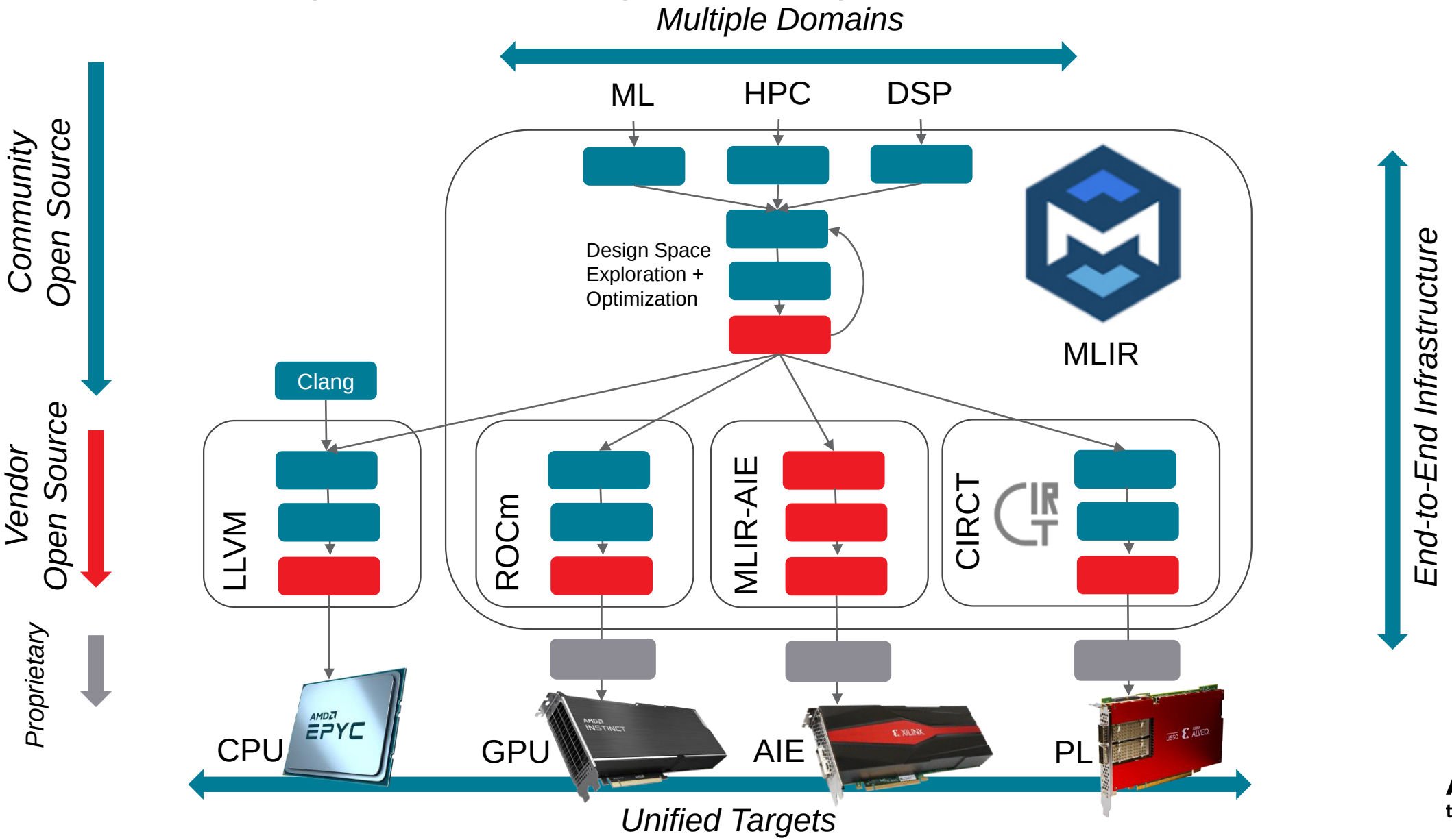
- Next generation **open source** compiler infrastructure
  - LLVM core project

- Key idea: 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
  }
}
```

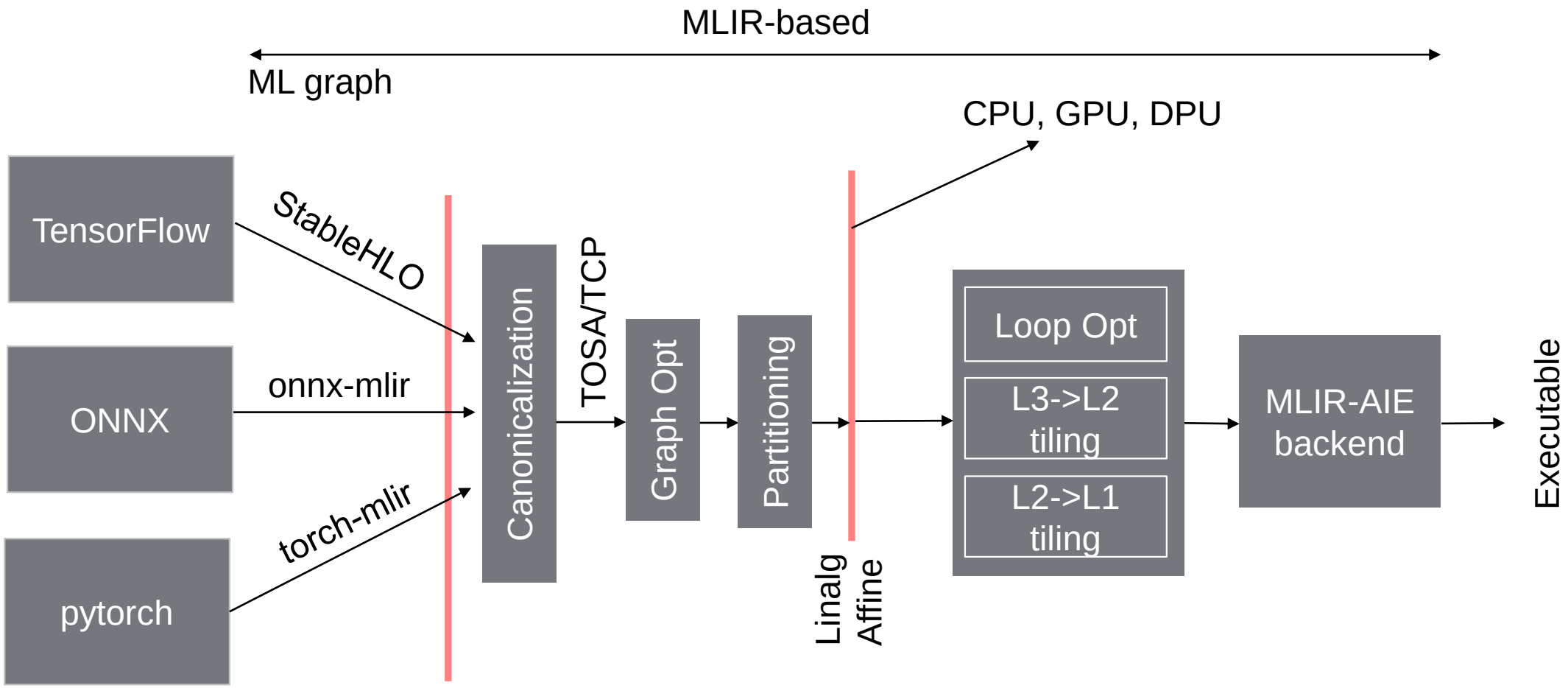

AMD
together we advance_

Future Heterogeneous Programming



AMD
together we advance_

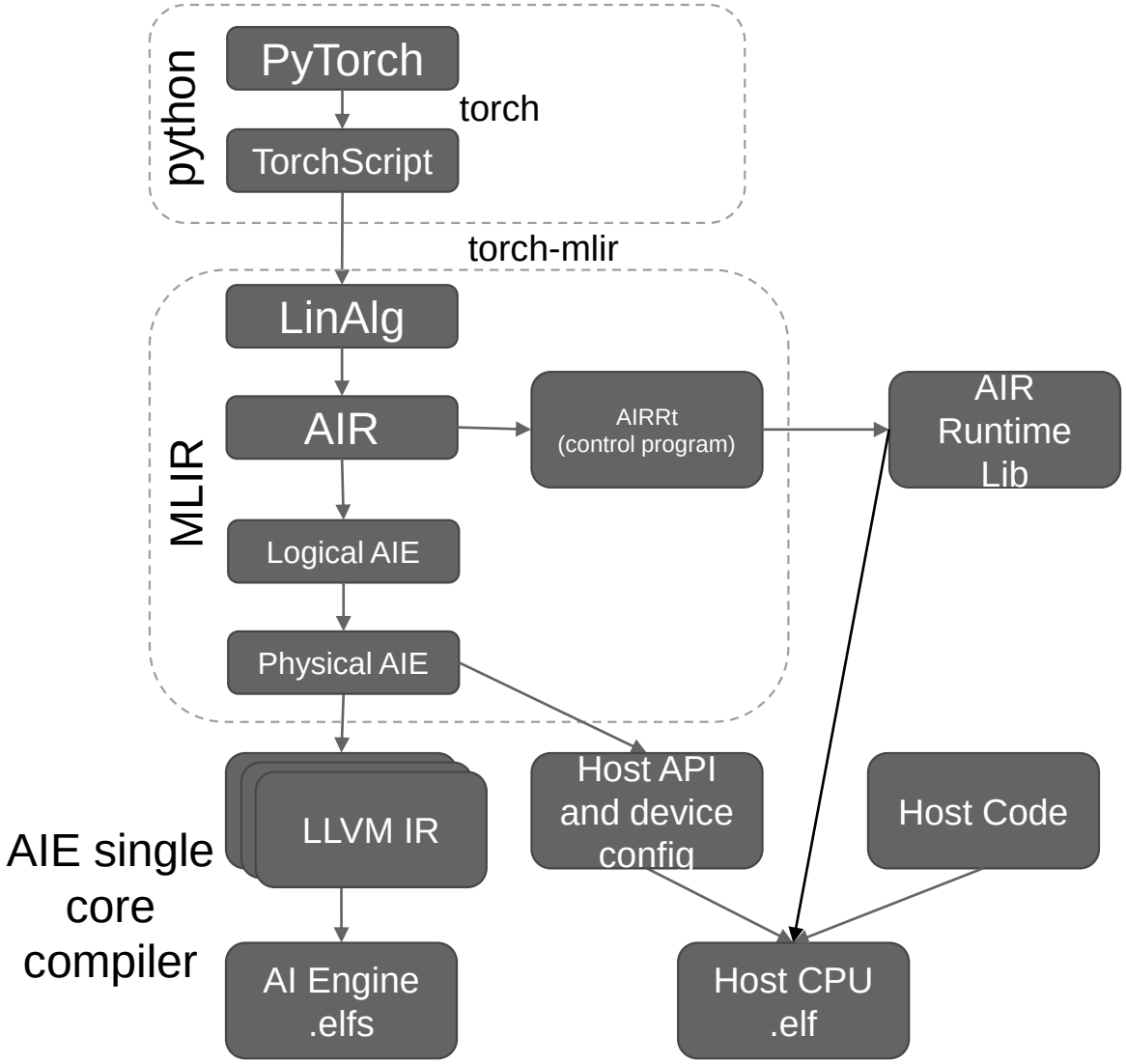
Machine Learning in MLIR



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

AMD
together we advance_

End-to-End Research Flow



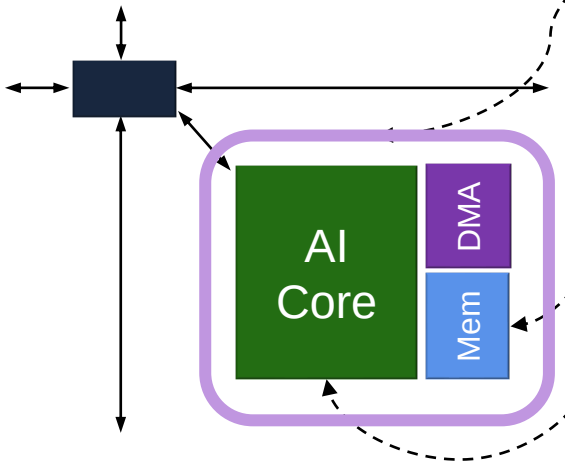
Most of this is already open source, except for the single-core compiler

This tutorial (mostly)

AMD
together we advance_

Physical MLIR-AIE

Running Code on a Core



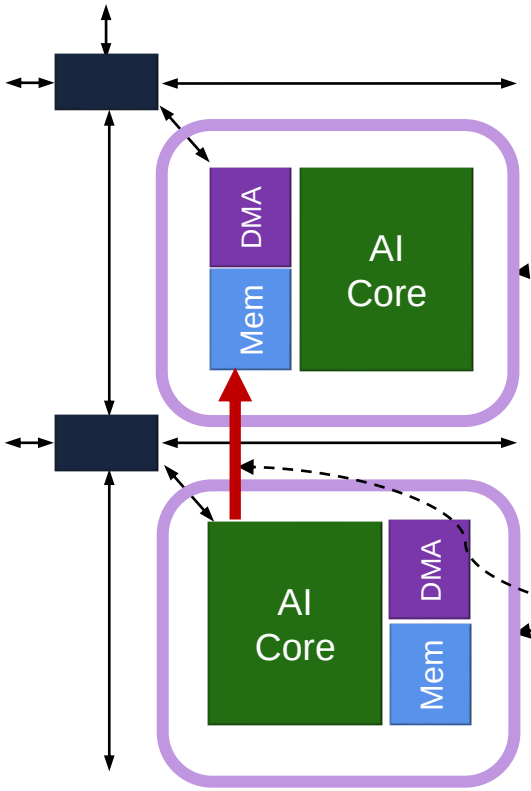

```
%tile13 = AIE.tile(1, 3)
%buf13_0 = AIE.buffer(%tile13)
        { sym_name = "a" } : memref<256xi32>

%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
}
```

Memory Interface

Stream Interface

AMD
together we advance_

Moving Buffers



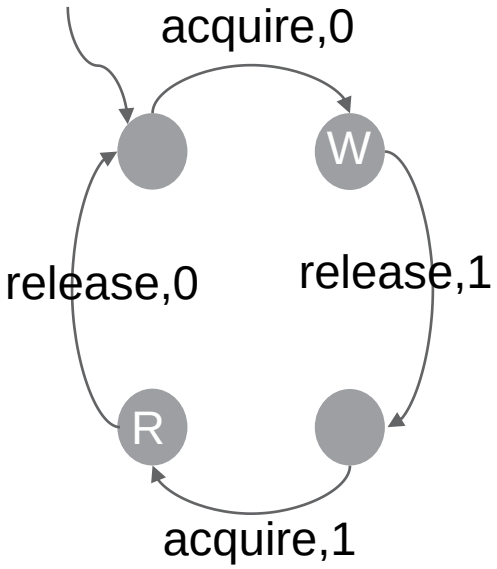

```
%tile14 = AIE.tile(1, 4)
%buf = AIE.buffer(%tile14)
        { sym_name = "a" } : memref<256xi32>

%tile13 = AIE.tile(1, 3)
%core13 = AIE.core(%tile13) {
  %val1 = constant 7 : i32
  %idx1 = constant 3 : index
  %2 = addi %val1, %val1 : i32

  memref.store %2, %buf[%idx1] : memref<256xi32>

  AIE.end
}
```

Memory Interface

Stream Interface

AMD
together we advance_

Synchronization with Locks




```
%tile14 = AIE.tile(1, 4)
%lock = AIE.lock(%tile14) { sym_name = "a_lock" }
%buf = AIE.buffer(%tile14)
       { sym_name = "a" } : memref<512xi32>

%tile13 = AIE.tile(1, 3)
%core13 = AIE.core(%tile13) {
  AIE.useLock(%lock, "Acquire", 0)
  memref.store %2, %buf[%idx1] : memref<512xi32>
  AIE.useLock(%lock, "Release", 1)
}

%core14 = AIE.core(%tile14) {
  AIE.useLock(%lock, "Acquire", 1)
  %data = memref.load %buf[%idx1] : memref<512xi32>
  AIE.useLock(%lock, "Release", 0)
}
```


AMD
together we advance_

Tutorial 1 Focus

- Tutorial 1 – Modules, tile, buffer, core, lock
  - aie.mlir
  - test.cpp
  - Makefile
  - README.md
  - answers/

https://github.com/Xilinx/mlir-aie/tree/main/tutorials

AMD
together we advance_

Host Code, Simulation and Performance Counters

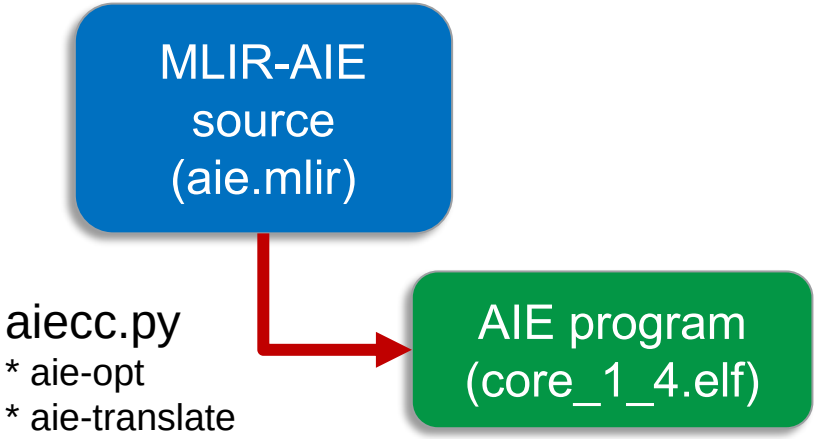
- Welcome & Logistics
- Introduction to AIEngines
- Physical MLIR-AIE
- Host Code, Simulation, Performance Counters
- Logical MLIR-AIE: Communication
- Scale up to Many-AIE Designs and Hook to MLIR-AIR
- Wrap Up

Lay of the Land (Tutorial – 1)

Pre-built libs

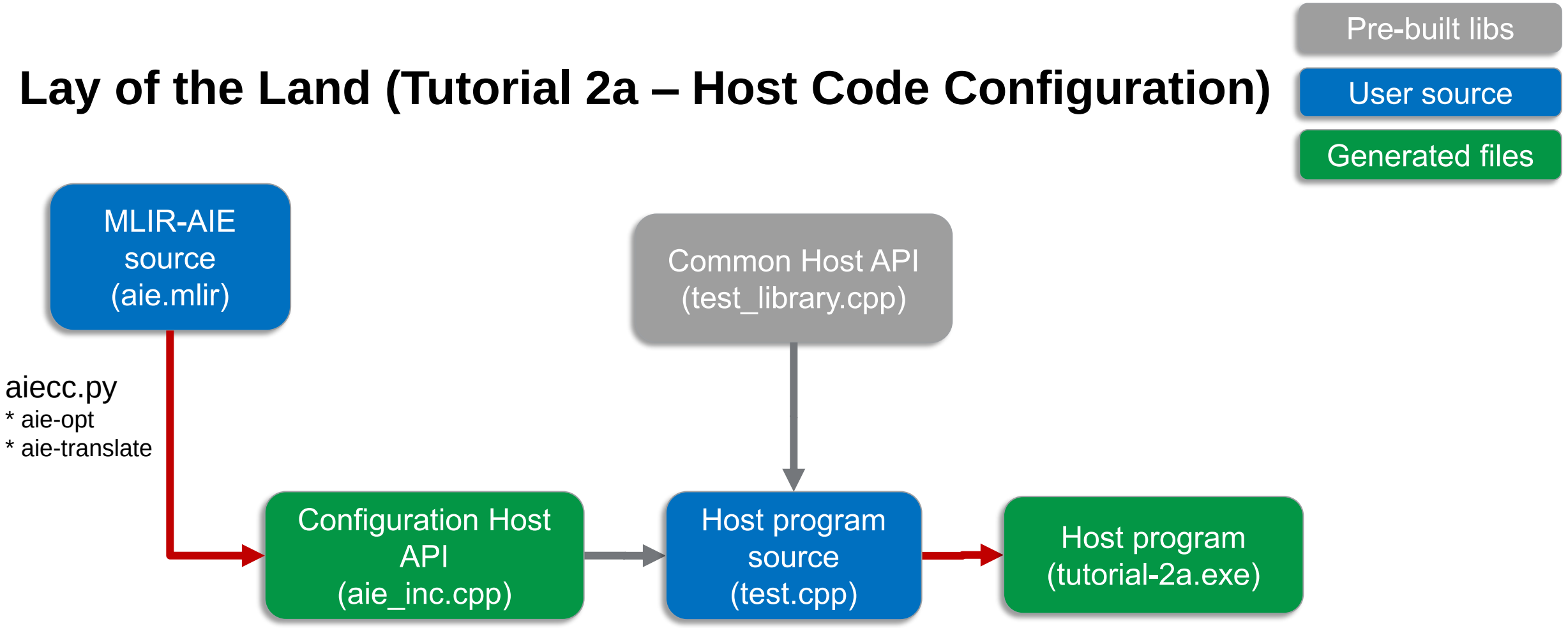
User source

Generated files



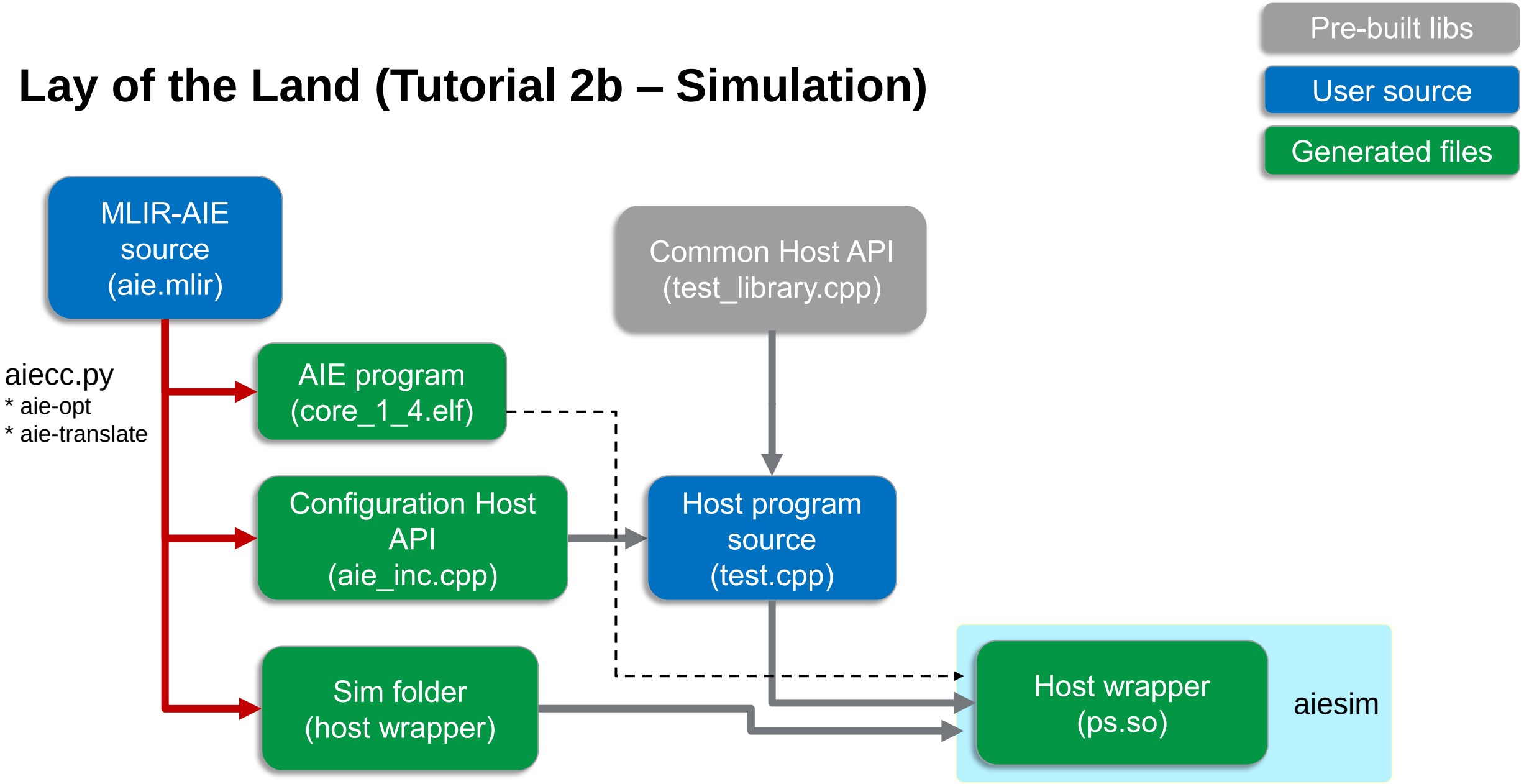
AMD
together we advance_

Lay of the Land (Tutorial 2a – Host Code Configuration)



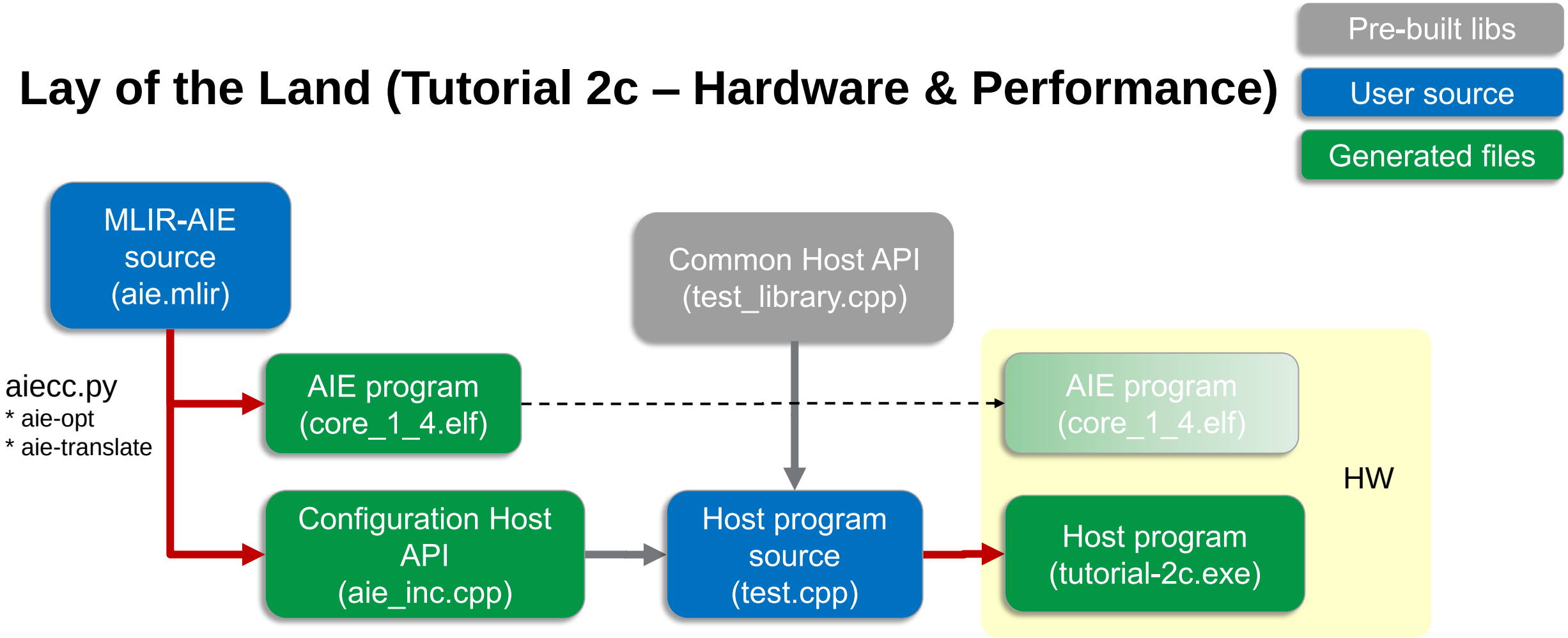
AMD
together we advance_

Lay of the Land (Tutorial 2b – Simulation)



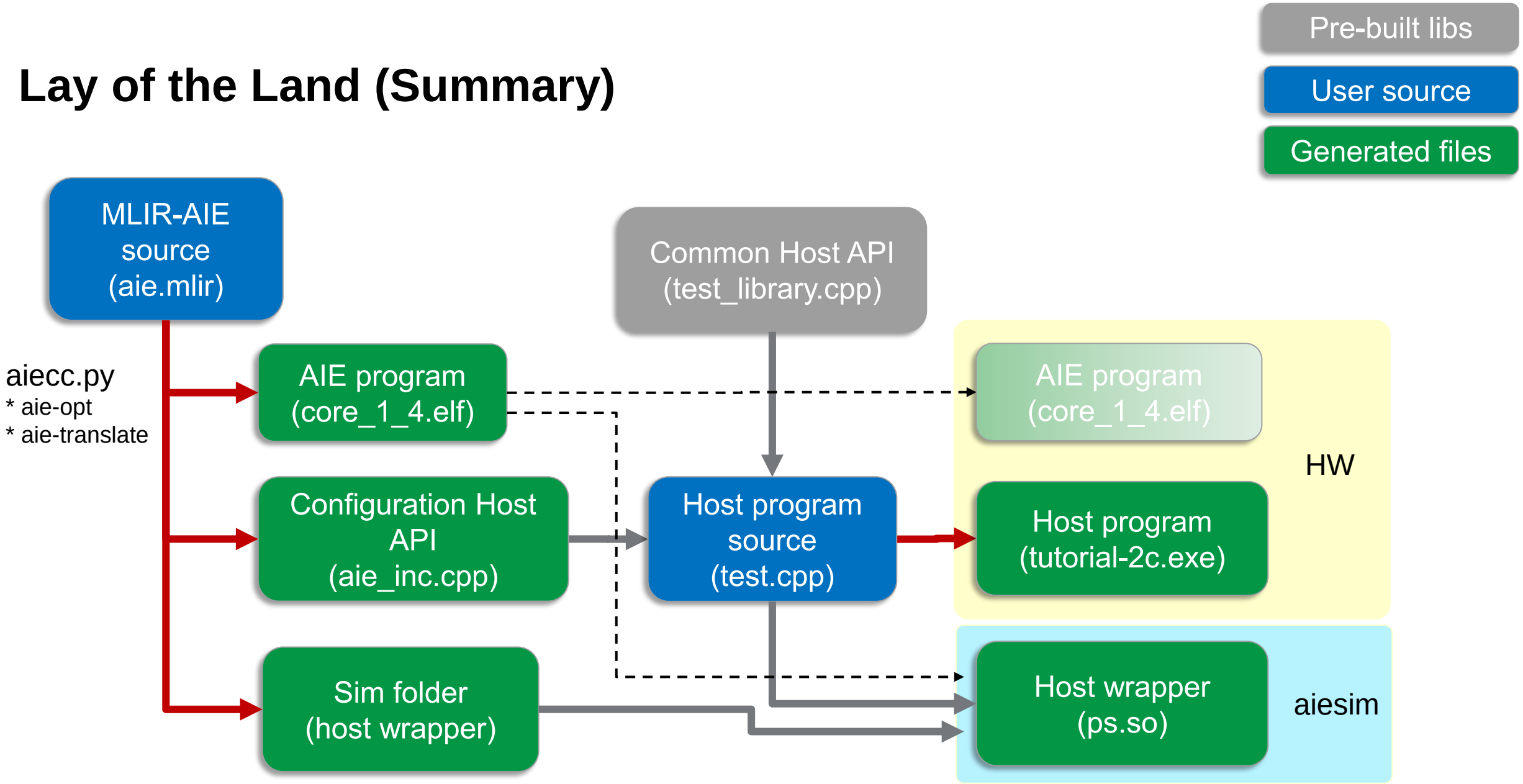
AMD
together we advance_

Lay of the Land (Tutorial 2c – Hardware & Performance)



AMD
together we advance_

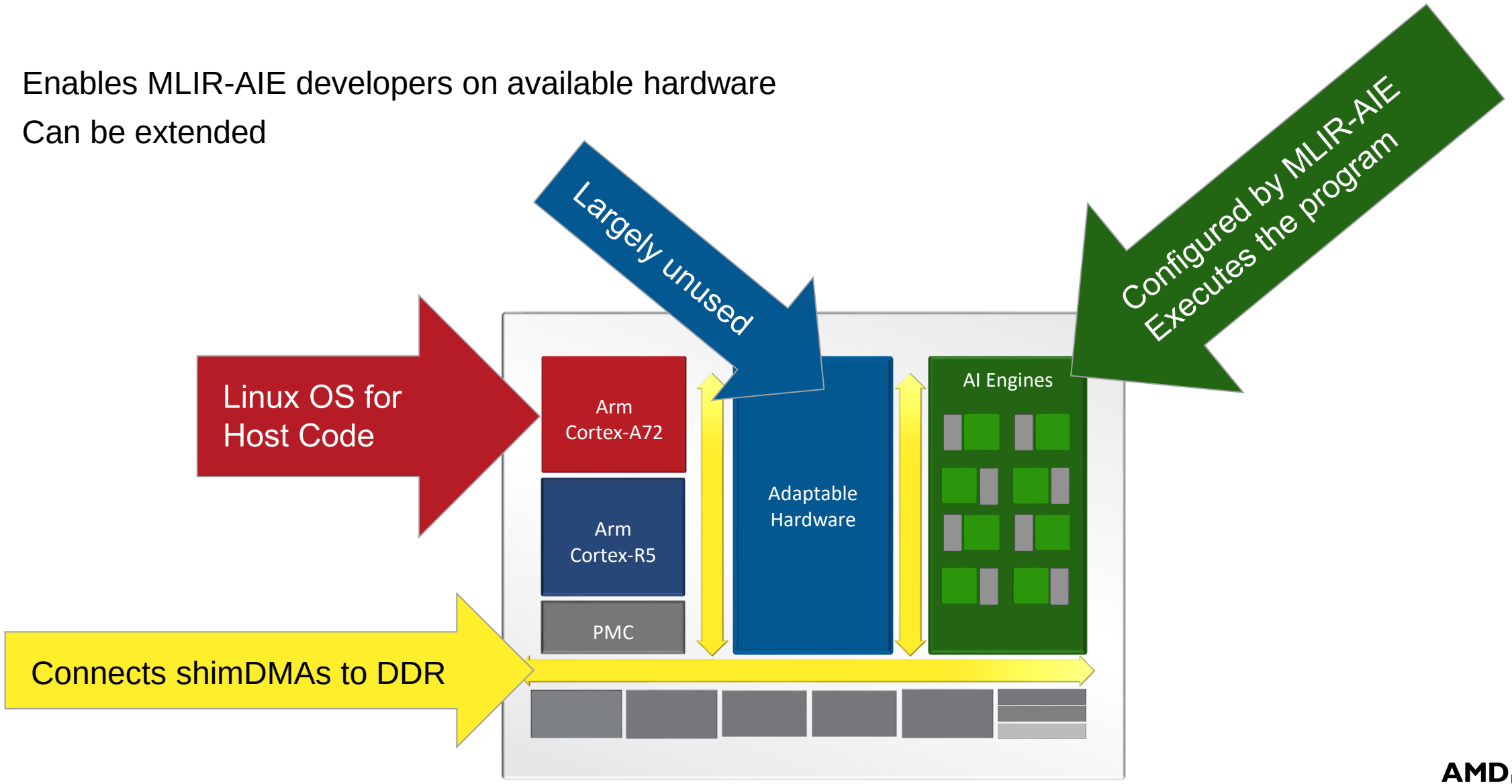
Lay of the Land (Summary)



AMD
together we advance_

MLIR-AIE VCK190 Platform Infrastructure

- Enables MLIR-AIE developers on available hardware
- Can be extended



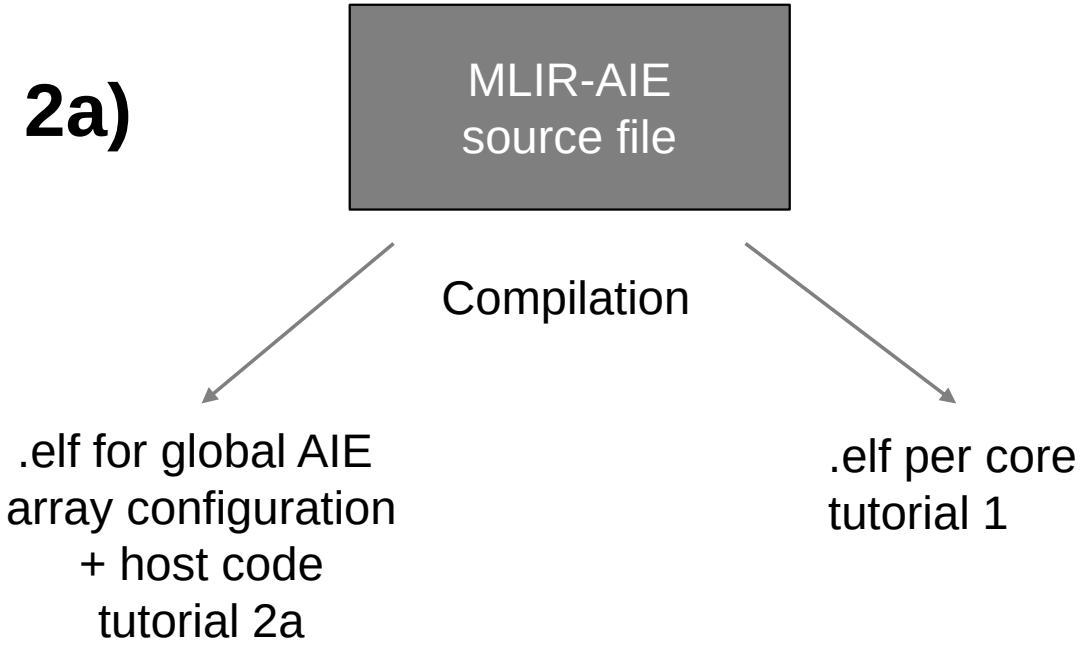
AMD
together we advance_

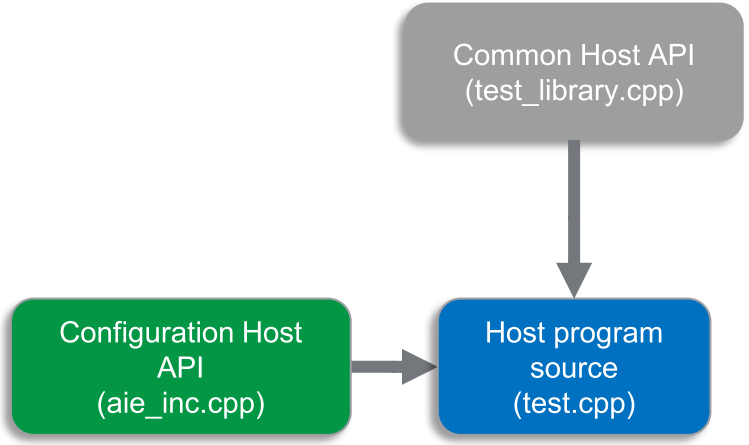
Host Code Configuration API (Tutorial 2a)

- AI Engine configuration and initialization – 2 ways
  1. Control processor running host code (e.g. ARM)
  2. Bootup (CDO files)
- Tutorial examples uses (1) with host code API
  - Host program (test.cpp) references host code API
  - Configuration host API (aie_inc.cpp)
    - Configure/ initialize AIE array design

```
aie_libxaie_ctx_t *_xaie = mlir_aie_init_libxaie();
mlir_aie_init_device(_xaie);
mlir_aie_configure_cores(_xaie);
mlir_aie_configure_switchboxes(_xaie);
mlir_aie_configure_dmas(_xaie);
mlir_aie_initialize_locks(_xaie);

mlir_aie_clear_tile_memory(_xaie, 1, 4); // clear local data memory for tile(1,4)
```

  - Common host code API (test_library.cpp)
    - Read/ write data memory
    - Acquire/ release locks
    - Check/ compare values





AMD
together we advance_

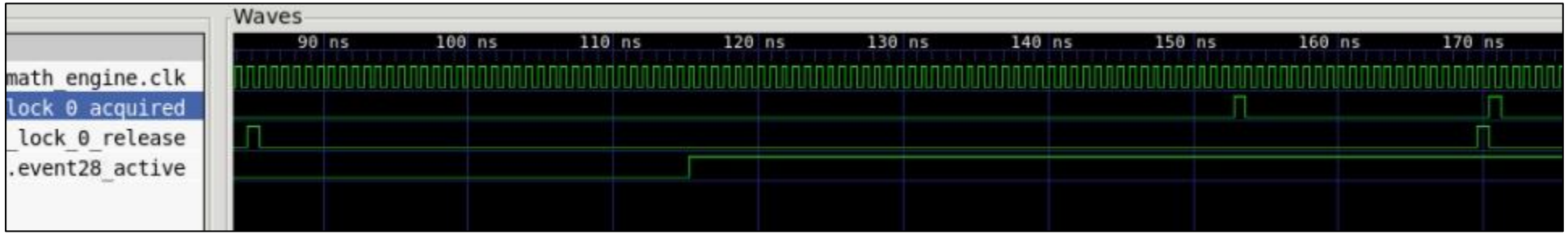
Simulation (Tutorial 2b)

make -C sim

(1) Recompile
test.cpp+aie_inc.cpp
test_library.cpp
host wrapper

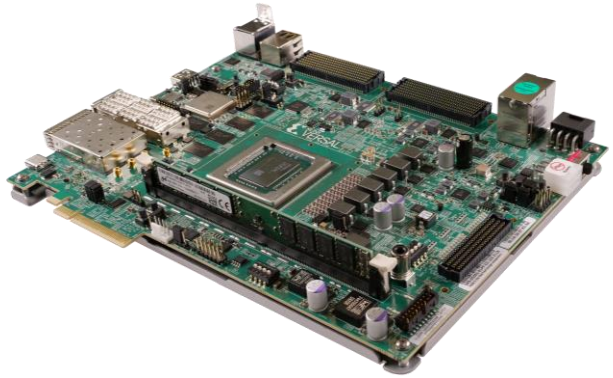
(2) Run aiesimulator

- We generate a sim folder every time aiecc.py is run
  - Contains stubs to work with command line AI Engine simulator (aiesimulator)
- aiesimulator
  - Cycle-accurate simulator per AI Engine core
  - Transaction level System-C simulator for array communication
  - Wrapper for host code programs
    - Models transfer between host side memory (DDR) and AI Engine Array data movers (shimDMA through NOC)
  - Useful for testing smaller designs to verify functionality
  - Larger designs can be run directly in hardware
- gtkwave
  - aiesimulator generates foo.vcd file which user can use gtkwave to view simulation waveforms



AMD
together we advance_

Running on Hardware (Tutorial 2c)

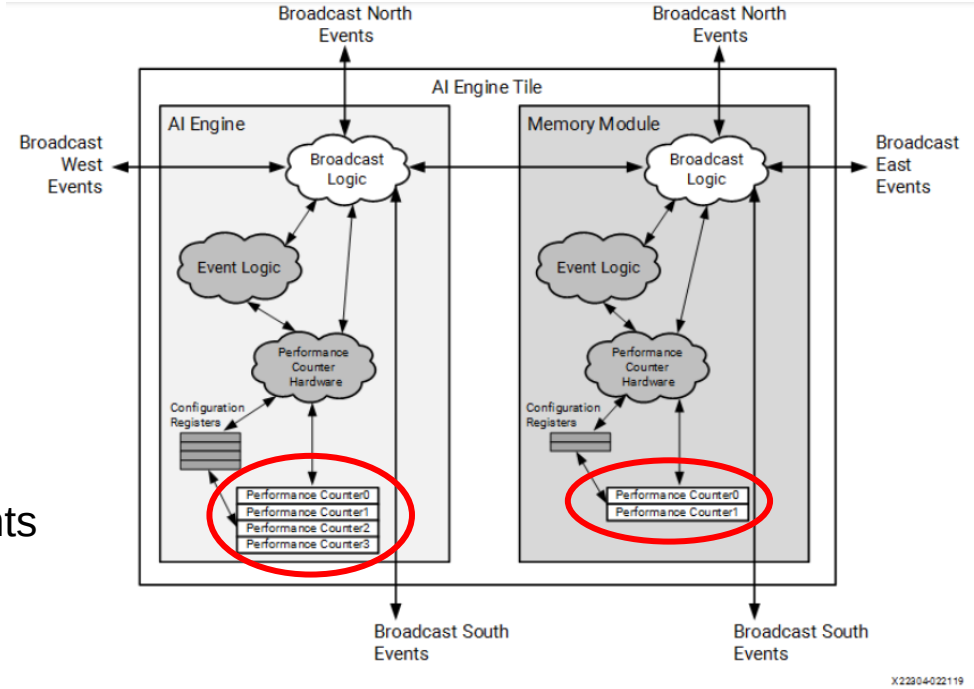


- Steps for running on vck190
  1. Copy .elf and .exe files to board (e.g. scp *elf xilinx@<board ip>/home/xilinx/.)
  2. ssh xilinx@<board ip>, pw: xilinx
  3. Run executable with sudo permissions (sudo ./tutorial-2c.exe)
- Time to verify designs on hardware is fast (no need to run synthesis, place & route)
  - Leverage a Vivado built platform design (PYNQ enabled) which configures the NOC to support all shim DMA connections between the AI Engine array and DDR memory (vck190)
    - Linux OS on Arm core for host code
  - Especially fast for very large designs (as compared to simulation)
- Simulation vs. Hardware
  - Timing relationship between host code and AIE programs changes between simulation vs. hardware
    - Simulation – Fast host code, “Slower” AIE programs (cycle accurate simulation)
    - Hardware – Fast AIE programs (1 GHz), “Slower” host code (100s of AIE cycles per host code API function call)
  - Examples
    - usleep can work for waiting for AIE programs to complete but does not work in simulation
    - Instead, use locks to synchronize between host and AI Engine simulation

AMD
together we advance_

Performance Counters (Tutorial 2c)

- Useful for counting cycles between events
  - Core (4 counters), Memory (2 counters)
  - Start, Stop, Reset – works like a stop watch
- Example events
  - https://docs.xilinx.com/r/en-US/am009-versal-ai-engine/AI-Engine-Events
  - Program addresses
  - Lock acquire, release (by ID but not value)
  - Broadcast (by ID)
- Tutorial-2c introduces performance counters with program address event triggers
- Tutorial-3 extends events triggers to locks
- Tutorial-4 extends event triggers to broadcast events



AMD
together we advance_

Tutorial 2 Focus

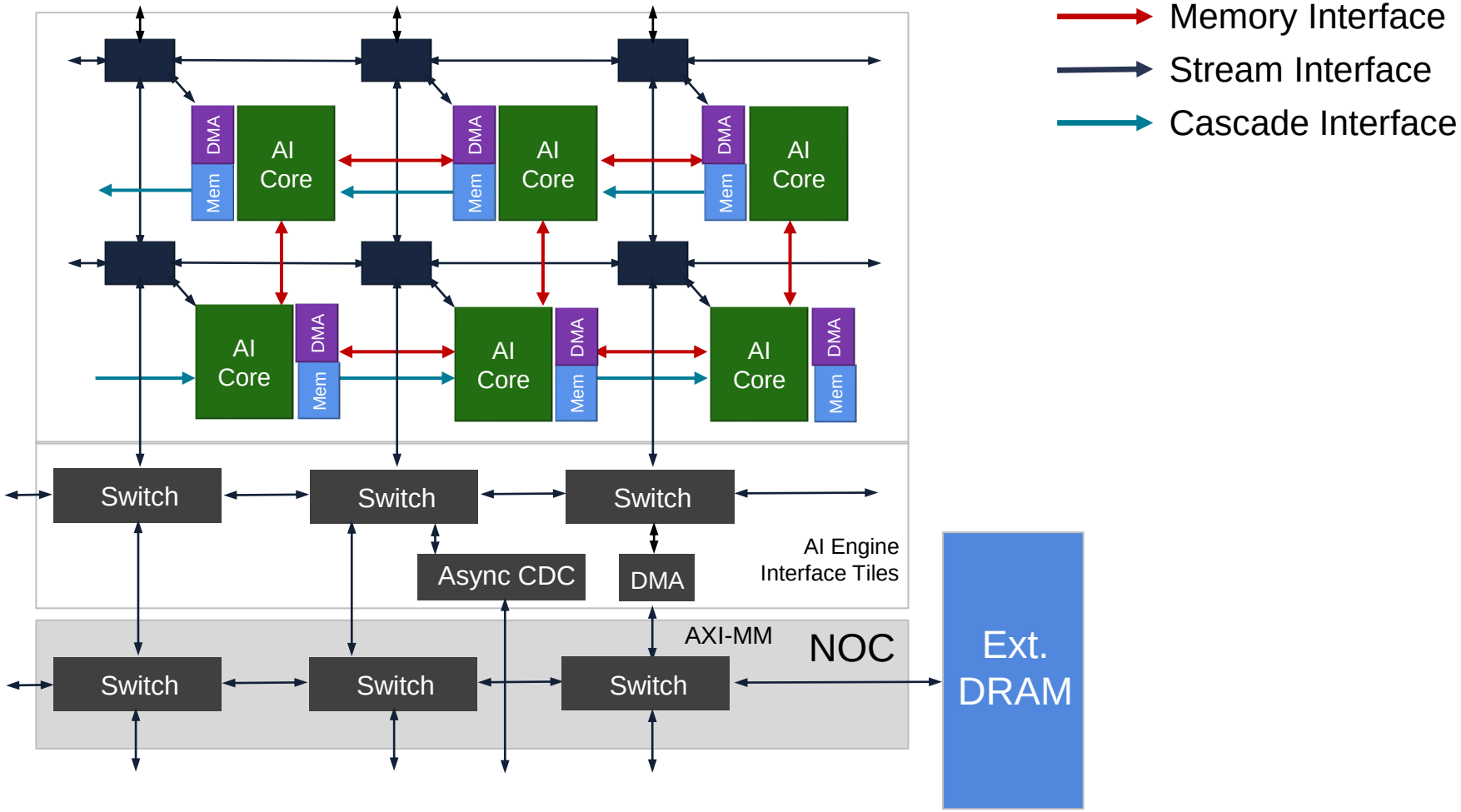
- Tutorial 2 – Host code configuration, simulation, hardware performance
  - Tutorial-2a - Host code configuration
    - aie.mlir
    - test.cpp
    - Makefile
    - README.md
  - Tutorial-2b - Simulation
    - aie.mlir
    - test.cpp
    - Makefile
    - README.md
  - Tutorial-2c - Running on hardware and measuring performance
    - aie.mlir
    - test.cpp
    - Makefile
    - README.md

https://github.com/Xilinx/mlir-aie/tree/main/tutorials

AMD
together we advance_

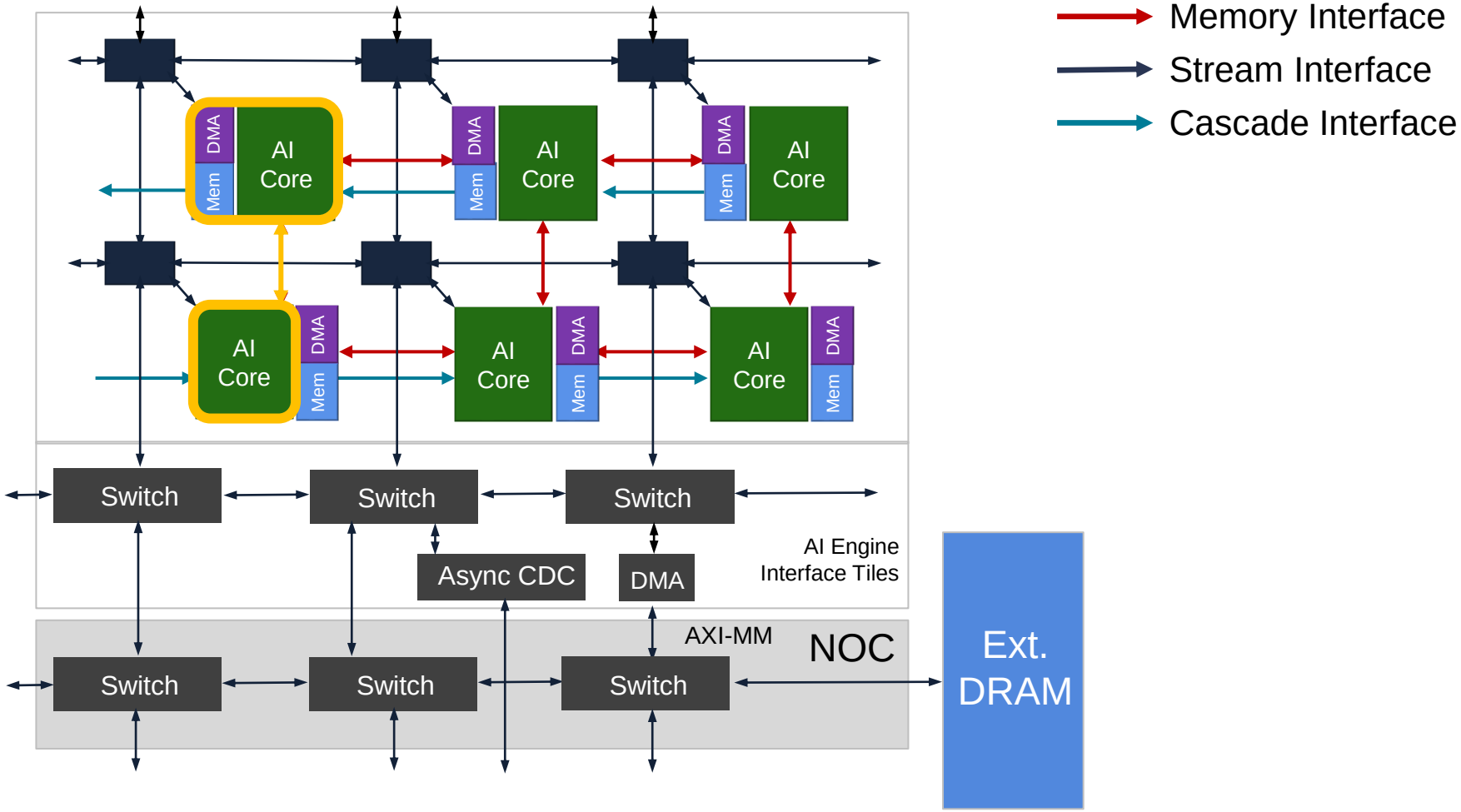
Logical MLIR-AIE: Communication

MLIR-AIE Communication Achieved by Configuring Different Components



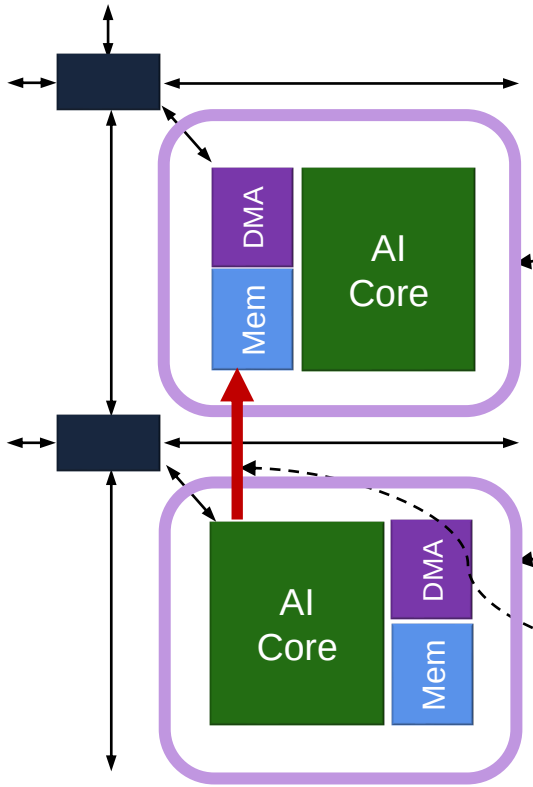
AMD
together we advance_

Tutorial 3 : MLIR-AIE Engine-to-Engine Communication Through Shared Memory



AMD
together we advance_

Buffers in Shared Memory Offer Straightforward Communication



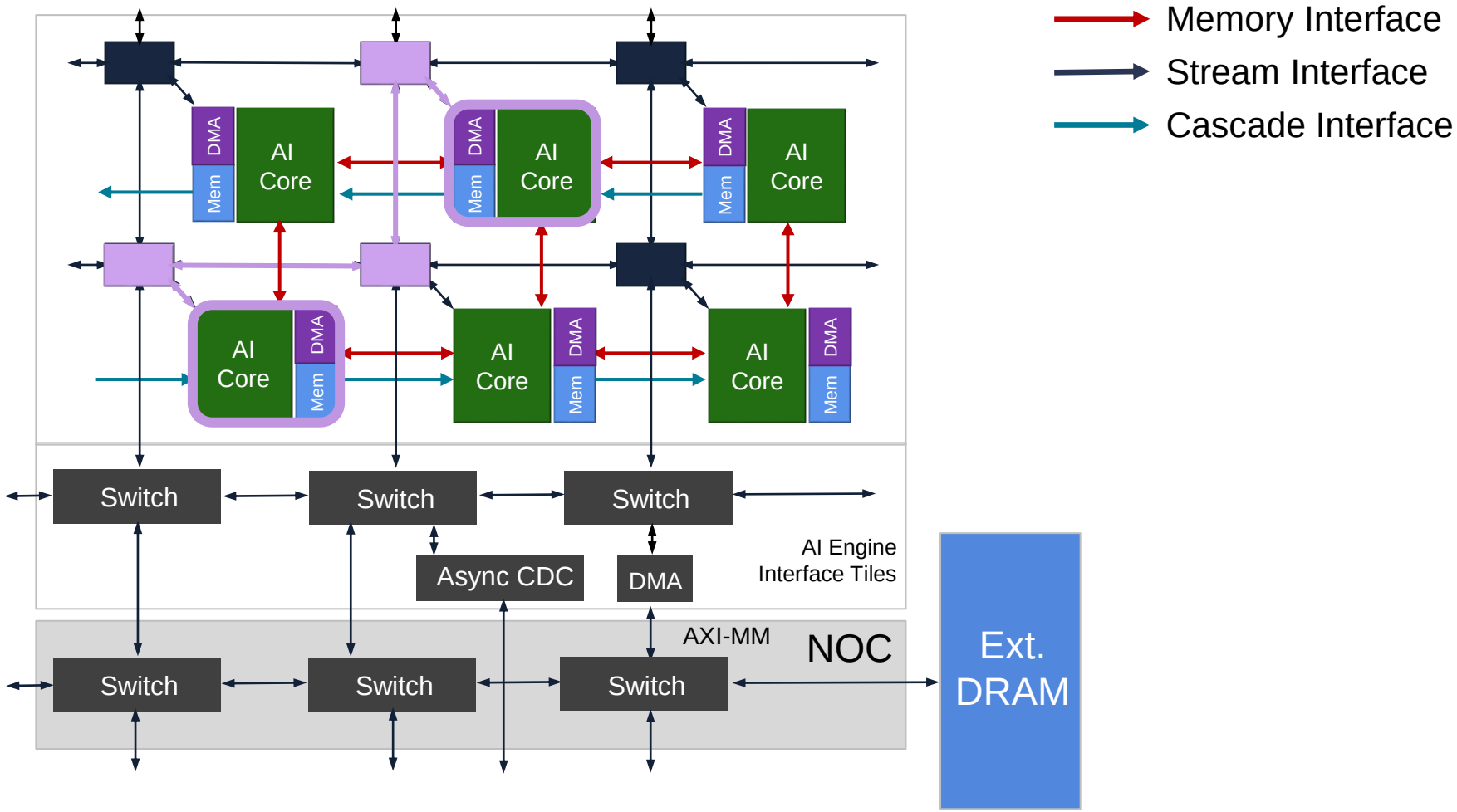

```
%tile14 = AIE.tile(1, 4)
%buf = AIE.buffer(%tile14)
        { sym_name = "a" } : memref<256xi32>

%tile13 = AIE.tile(1, 3)
%core13 = AIE.core(%tile13) {
  %val1 = constant 7 : i32
  %idx1 = constant 3 : index
  %2 = addi %val1, %val1 : i32

  memref.store %2, %buf[%idx1] : memref<256xi32>

  AIE.end
}
```

Memory Interface

Stream Interface

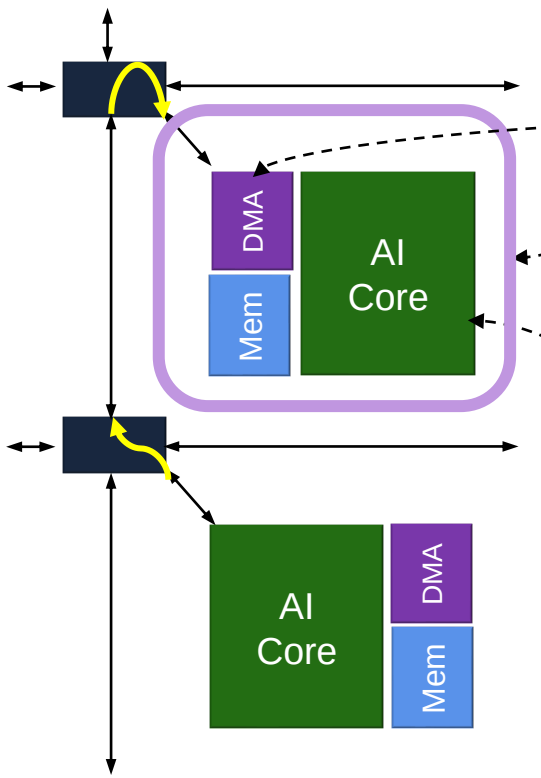
AMD
together we advance_

Tutorial 4 : MLIR-AIE Engine-to-Engine Communication Through Tile DMAs and Stream Switches



AMD
together we advance_

Configuring Tile DMAs is Tedious



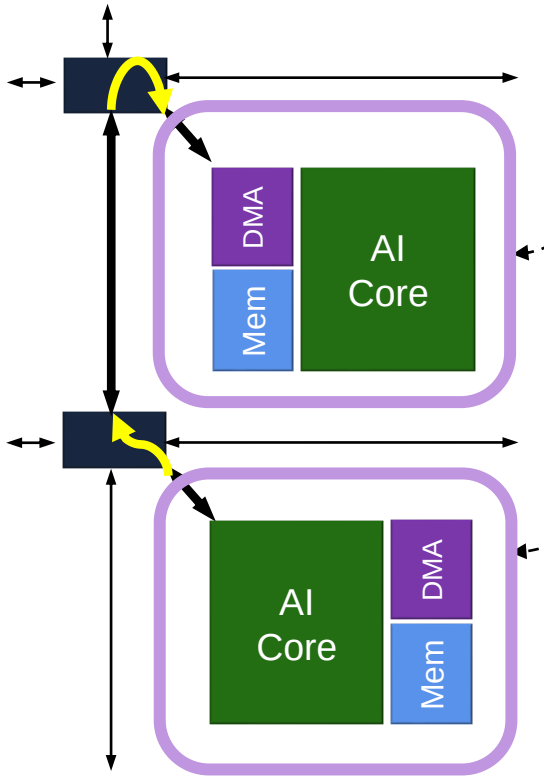

```
%tile14 = AIE.tile(1, 4)
%buf14 = AIE.buffer(%tile14) { sym_name = "a14" } : memref<256xi32>
%lock14_6 = AIE.lock(%tile14, 6) { sym_name = "lock_a14_6" }
%mem14 = AIE.mem(%tile14) {
        AIE.dmaStart("MM2S", 0, ^bd0, ^end)
    ^bd0:
        AIE.useLock(%lock14_6, "Acquire", 1)
        AIE.dmaBd(<%buf14 : memref<256xi32>, 0, 256>, 0)
        AIE.useLock(%lock14_6, "Release", 0)
        AIE.nextBd ^end
    ^end:
        AIE.end
}
%core14 = AIE.core(%tile14) {
    AIE.useLock(%lock14_6, "Acquire", 0)
    %val = arith.constant 14 : i32
    %idx = arith.constant 3 : index
    memref.store %val, %buf14[%idx] : memref<256xi32>
    AIE.useLock(%lock14_6, "Release", 1)
    AIE.end
}
```

Memory Interface

Stream Interface

AMD
together we advance_

Configuring Stream Switches is Tedious



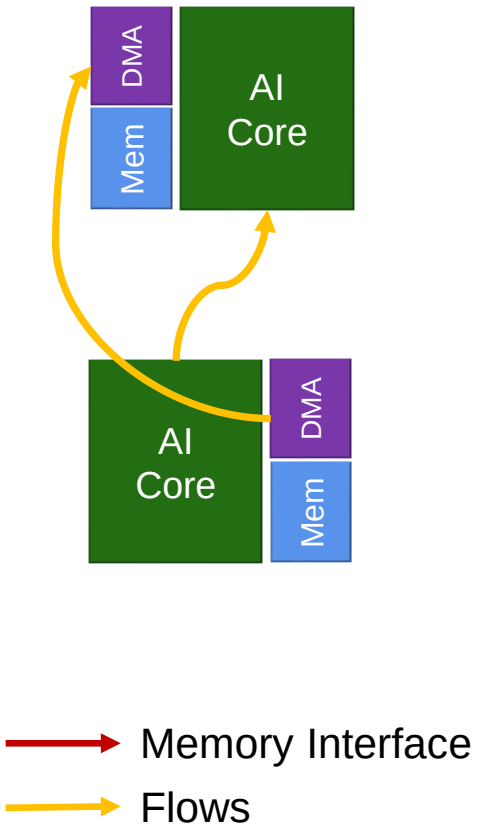

```
%tile14 = AIE.tile(1, 4)
AIE.switchbox(%tile14) {
  AIE.connect<"South" : 3, "Core" : 0>
  AIE.connect<"South" : 4, "DMA" : 0>
}

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

Memory Interface

Stream Interface

AMD
together we advance_

The Flow Higher Abstraction Absorbs Some of the Configuration...



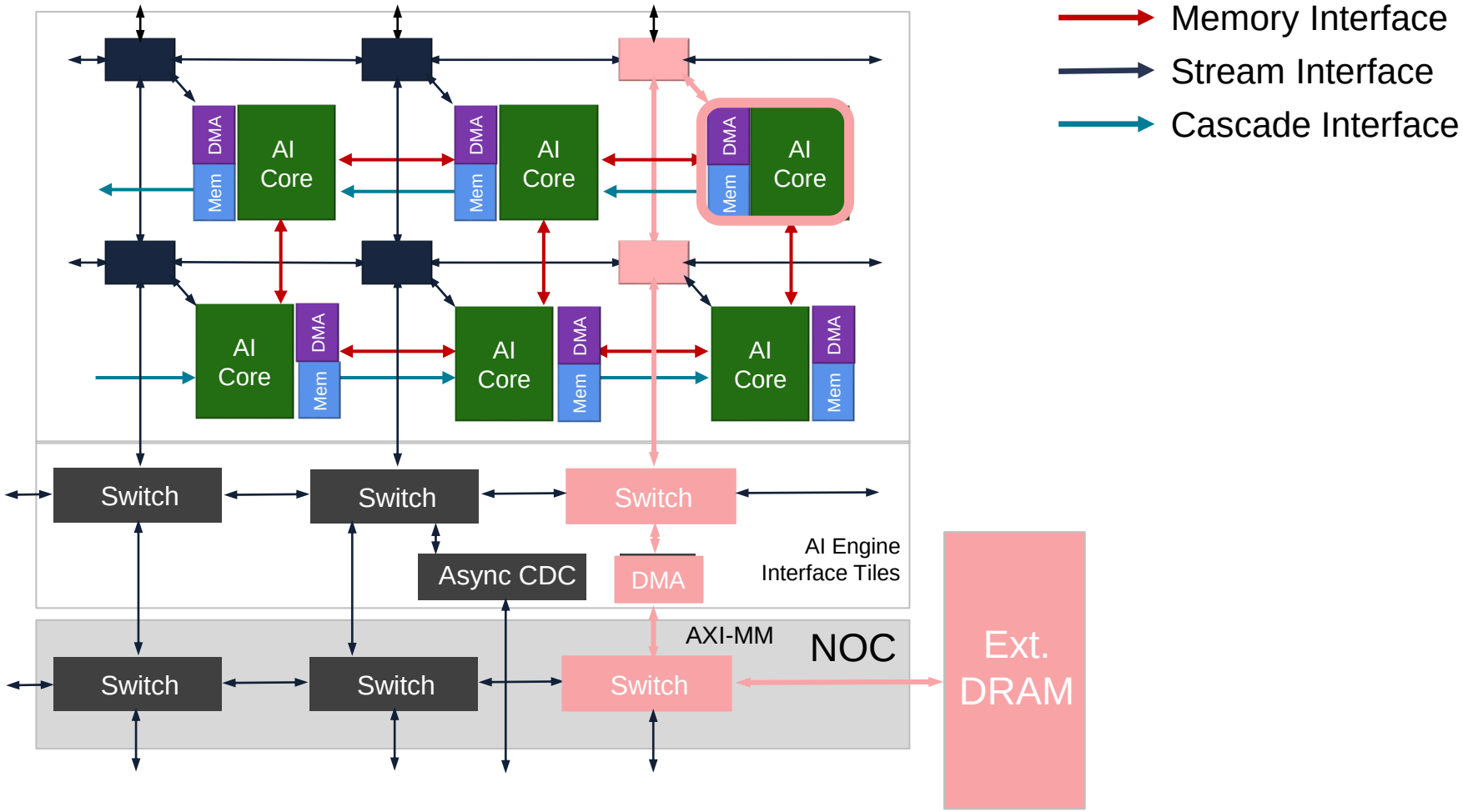

```
AIE.flow(%tile14, "DMA" : 0, %tile13, "DMA" : 0)
AIE.flow(%tile14, "Core" : 0, %tile13, "Core" : 0)
```

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>
}
```

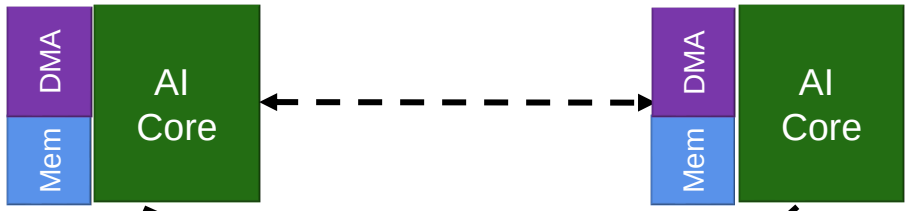

AMD
together we advance_

Tutorial 5 : MLIR-AIE Engine-to-External Memory Communication Through Tile DMAs, Stream Switches and NOC



AMD
together we advance_

The Object FIFO Abstracts Both the Data Allocation and the Data Movement



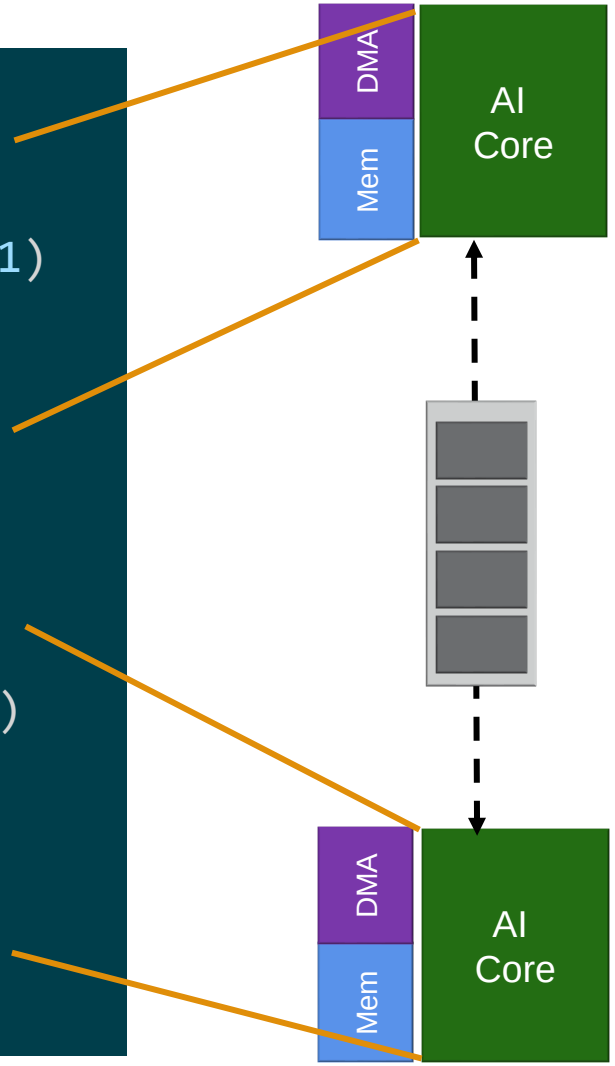

```
%objFifo = AIE.objectFifo.createObjectFifo(%tile14, {%tile24}, 2) : !AIE.objectFifo<memref<256xi32>>
```

```
%buff24_0 = AIE.buffer(%tile24) {sym_name = "of_0_buff_0"} : memref<256xi32>
%lock24_0 = AIE.lock(%tile24, 0) {sym_name = "of_0_lock_0"}
%buff24_1 = AIE.buffer(%tile24) {sym_name = "of_0_buff_1"} : memref<256xi32>
%lock24_1 = AIE.lock(%tile24, 1) {sym_name = "of_0_lock_1"}
```


AMD
together we advance_

Synchronized Accesses to the Object FIFO Ensure a Deadlock-Free Schedule

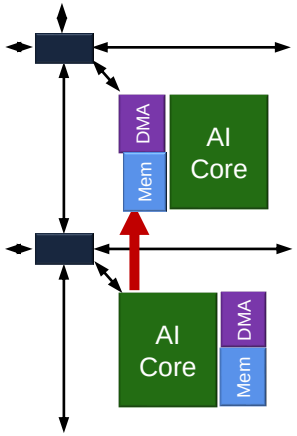
```
%f = AIE.objectFifo.createObjectFifo(%src, {%dst}, 2)
AIE.core(%src) {
    scf.for %indexInLine = %c0 to %c16 step %c1 {
        %v1 = AIE.objectFifo.acquire<Produce>(%objFifo, 1)
        ...
        AIE.objectFifo.release<Produce>(%objFifo, 1)
    }
    AIE.end
}
AIE.core(%dst) {
    scf.for %indexInLine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

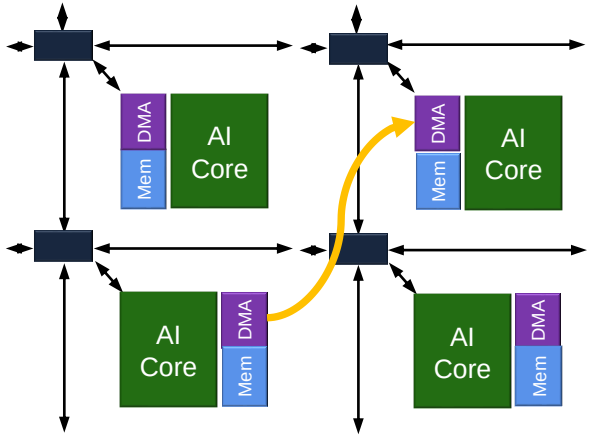


AMD
together we advance_

Object FIFO Lowers to Communication Components Based on the Endpoints

aie-opt --aie-objectFifo-stateful-transform






```
%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)
```


AMD
together we advance_

Tutorials 3, 4, 5 Focus

- Tutorial 3 – Communication (Local Memory)
  - aie.mlir
  - test.cpp
  - Makefile
  - README.md
  - answers/
  - objectFifo_ver/

- Tutorial 4 – Communication via objectFifo (tile DMA, logical routing)
  - aie.mlir
  - test.cpp
  - Makefile
  - README.md
  - flow/
  - switchbox/

- Tutorial 5 – Communication via objectFifo (shim DMA, external memory aka DDR)
  - aie.mlir
  - test.cpp
  - Makefile
  - README.md
  - flow/

https://github.com/Xilinx/mlir-aie/tree/main/tutorials

AMD
together we advance_

Scale up to Many-AIE and Hook to MLIR-AIR

Generating MLIR-AIE from Higher Level Tools

- Generate Textual MLIR-AIE ←
- Python Bindings for MLIR-AIE
- Higher Level Dialects - e.g. MLIR-AI**R**

reference_designs/horizontal_diffusion

- README.md
- HDIFF_single_AIE_objectFIFO_ping_pong
- HDIFF_dual_AIE_objectFIFO_ping_pong
- HDIFF_tri_AIE_objectFIFO_ping_pong
- HDIFF_tri_AIE_objectFIFO_ping_pong_scaled

AMD
together we advance_

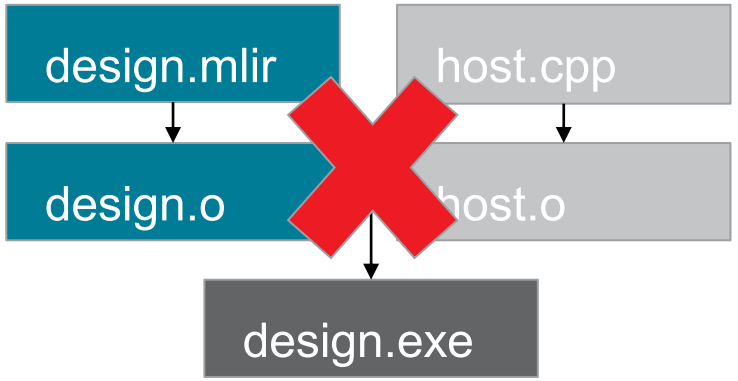
Recipes for scaling up…

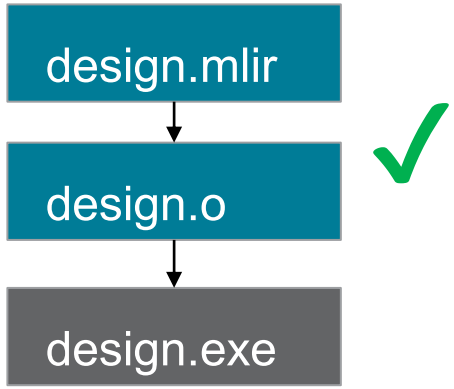
- *Increase Parallelism by stamping out design many times*
  - *Designs run concurrently without synchronization – coarse grained parallelism*
  - *Memory bandwidth scales with number of replications*
- *Build groups of cores that exchange and/or reuse data within inner memory hierarchy layers*
  - *Groups of cores run co-operatively with fine-grained synchronization*
  - *Data bought into local memories once and reused multiple times - increased operational intensity*
  - *Exploit special features of data fabric (e.g. broadcast) to further decrease off-chip memory bandwidth – scale further*

AMD
together we advance_

Single Source Language

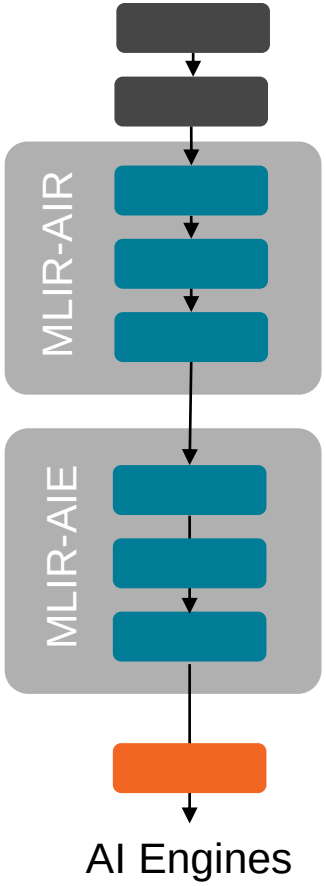
- MLIR-AIE describes accelerator behavior (e.g. *'kernel language'*)
  - But not the interaction of the accelerator with host code
    - for this we used a separate 'host.cpp' file
- User expectations set by languages like CUDA and SYCL
  - Accelerator specification and host integration are in one language
- Want an intermediate representation that captures both
  - Accelerator design(s)
  - Interaction with accelerator(s) from host, including
    - (Re)-configuration
    - Data transfer through a sequence of buffer descriptors





AMD
together we advance_

High Level Abstractions for Efficient Spatial Computing



Domain-specific Computing
Specialize for application space

Spatial Computing
Applicable to many fabrics

AI Engine
Exposes AI Engine-specific features

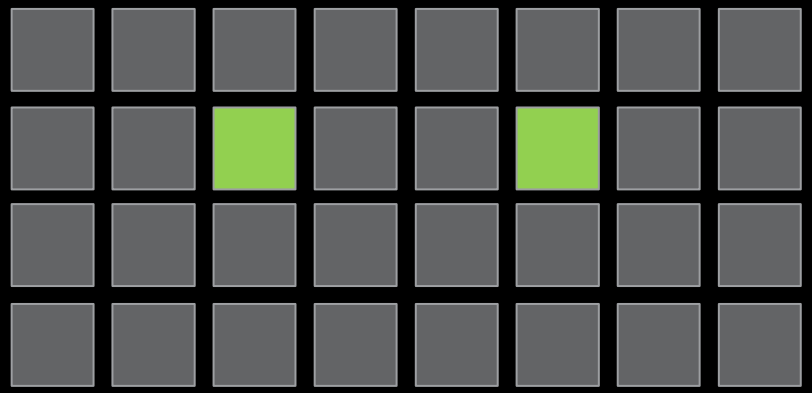
AMD
together we advance_

How do we program these Spatial Computing concepts using MLIR-AIR?

Resource Model

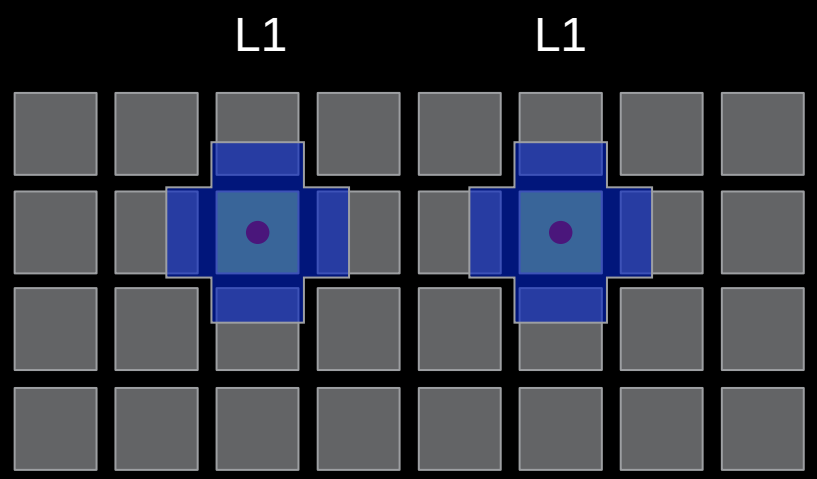
- Cores
- Memory Hierarchy

Spatial Compute Engines



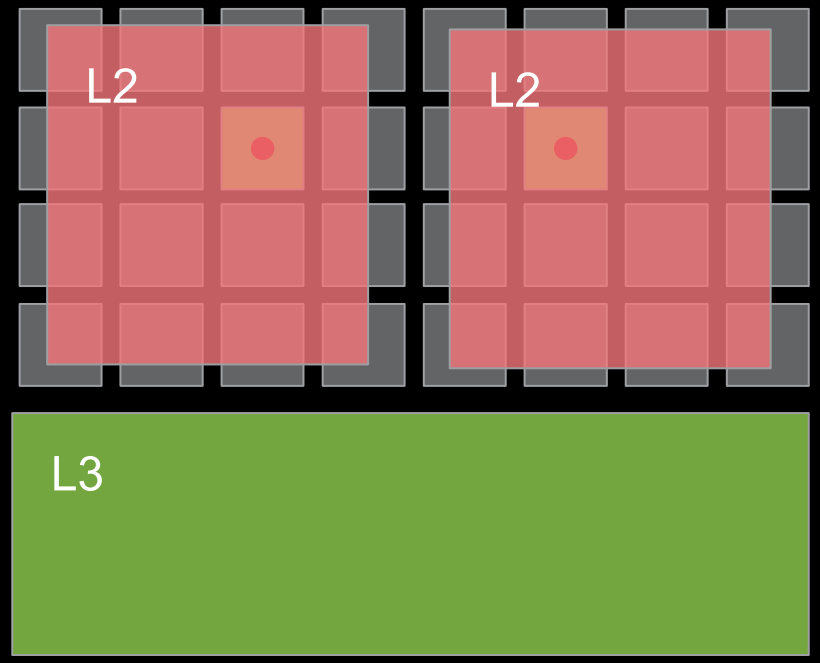
Resource Model

- Cores
- Memory Hierarchy
  - Local Memories (L1)



Resource Model

- Cores
- Memory Hierarchy
  - e.g. Local Memories (L1)
  - e.g. Shared Memories (L2)
  - e.g. Global Memory (L3)
  -



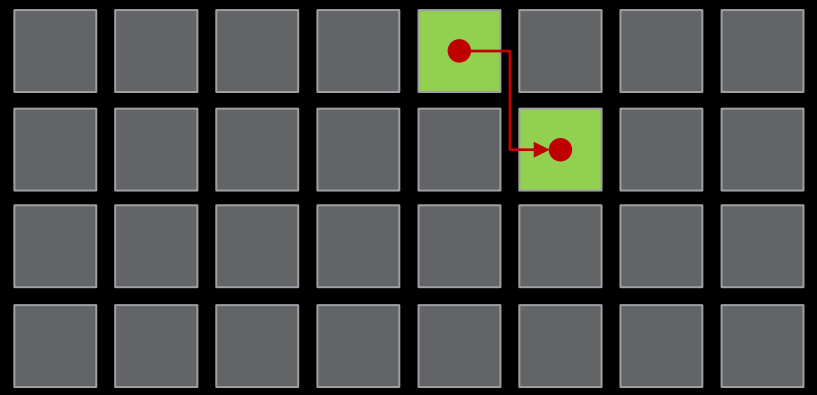
Basic Spatial Concepts

Software runtime determines placement

Lifetime of Accelerator

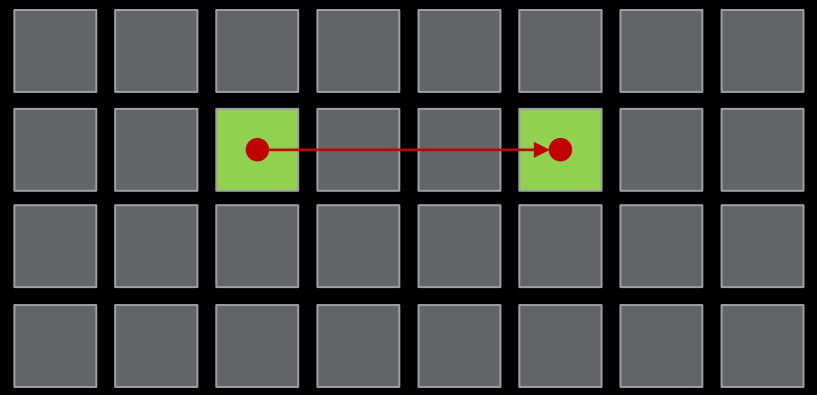
```
air.launch {
  air.channel @chan ();

  air.partition () {
   %buf0 = memref.alloc () :
      memref <32x32xbf16,1>
   air.put @chan ( %buf0 ):
      memref<32x32xbf16,1> -> ()
  }

  air.partition {
      %buf1 = memref.alloc () :
              memref <32x32xbf16,1>
     air.get @chan %buf1
  }
}
```

Hierarchy guarantees co-scheduling of partitions



Basic Spatial Concepts

```
air.launch {
   air.channel @chan ();

   air.partition () {
    %buf0 = memref.alloc () :
       memref <32x32xbf16,1>
    air.put @chan ( %buf0 ):
       memref<32x32xbf16,1> -> ()
   }

   air.partition {
       %buf1 = memref.alloc () :
              memref <32x32xbf16,1>
      air.get @chan %buf1
   }

}
```

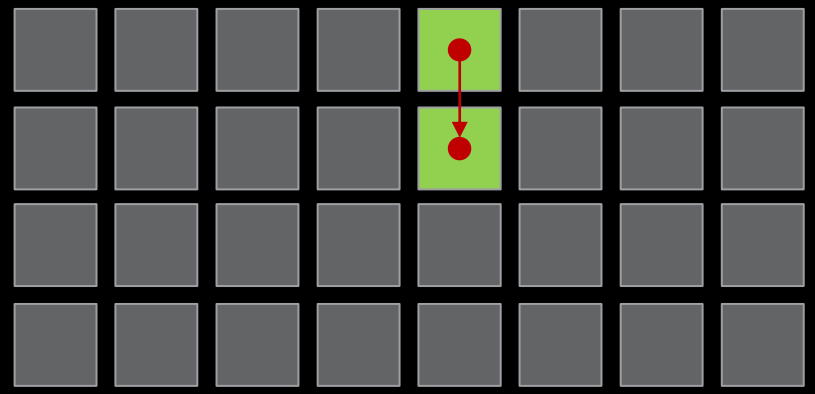


Basic Spatial Concepts

Software runtime determines placement

```
air.launch {
   air.channel @chan ();

   air.partition () {
    %buf0 = memref.alloc () :
       memref <32x32xbf16,1>
    air.put @chan ( %buf0 ):
       memref<32x32xbf16,1> -> ()
   }

   air.partition {
       %buf1 = memref.alloc () :
              memref <32x32xbf16,1>
      air.get @chan %buf1
   }

}
```

Hierarchy guarantees co-scheduling of partitions



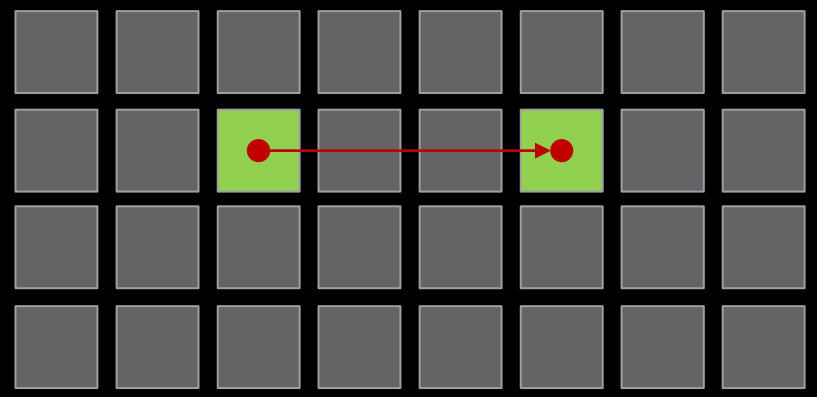
AMD

Basic Spatial Concepts

Software runtime determines placement

```
air.launch {
  air.channel @chan ();

  air.partition () {
   %buf0 = memref.alloc () :
      memref <32x32xbf16,1>
   air.put @chan ( %buf0 ):
      memref<32x32xbf16,1> -> ()
  }

  air.partition {
      %buf1 = memref.alloc () :
             memref <32x32xbf16,1>
     air.get @chan %buf1
  }

}
```

Hierarchy guarantees co-scheduling of partitions



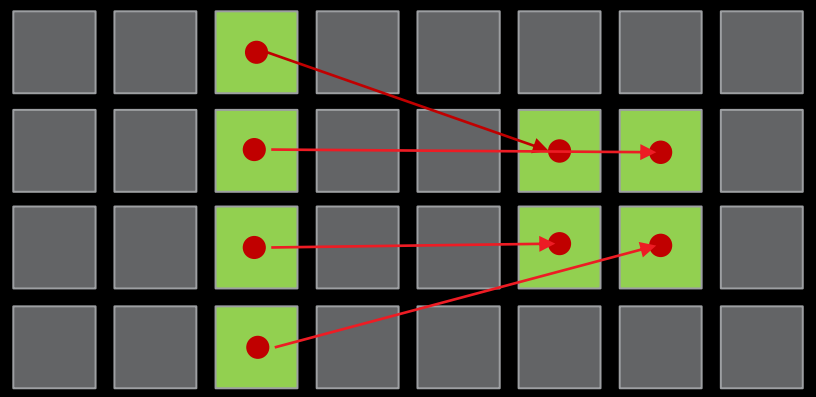
[Public]

Spatial Concepts to enable scaling

```
air.launch {
   air.channel [2][2] @chan ()
     %c4 = arith.constant 4 : index
     %c1 = arith.constant 1 : index
     %c2 = arith.constant 2 : index
```

Familiar “Index Space” program
Explicitly Spatial

```
   air.partition {
    air.herd (%x, %y) in (%a1 = c1,%a2 = %c4) {
      %i = arith.remsi (%x, %c4) : index
      %j = arith.floordivsi (%y, %c4) : index
       air.put @chan[%i, %j]( … )
     }
   }
   air.partition {
    air.herd (%x, %y) in (%a1 = %c2,%a2 = %c2) {
      air.get @chan[%x,%y] ( … )
   }
}
```



[Public]

Spatial Concepts to enable scaling

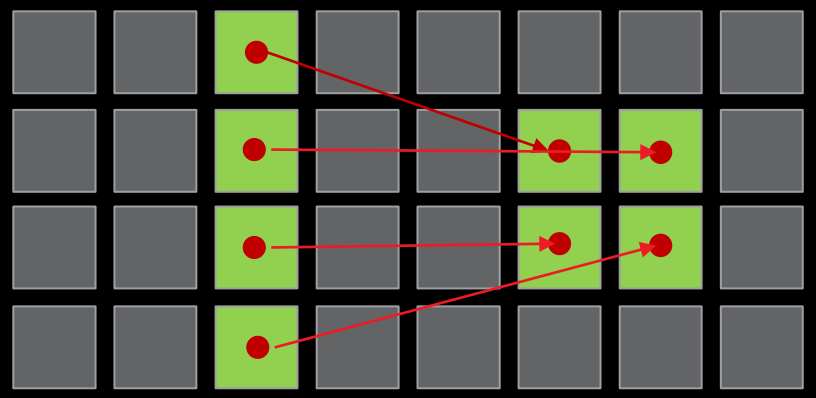
```
air.launch {
   air.channel [2][2] @chan ()
     %c4 = arith.constant 4 : index
     %c1 = arith.constant 1 : index
     %c2 = arith.constant 2 : index
```

Familiar “Index Space” program
Explicitly Spatial

```
   air.partition {
    air.herd (%x, %y) in (%a1 = c1,%a2 = %c4) {
      %i = arith.remsi (%x, %c4) : index
      %j = arith.floordivsi (%y, %c4) : index
       air.put @chan[%i, %j]( … )
     }
   }
   air.partition {
    air.herd (%x, %y) in (%a1 = %c2,%a2 = %c2) {
      air.get @chan[%x,%y] ( … )
   }
}
```

Fine-Grained Parallelism

Fine-Grained Parallelism

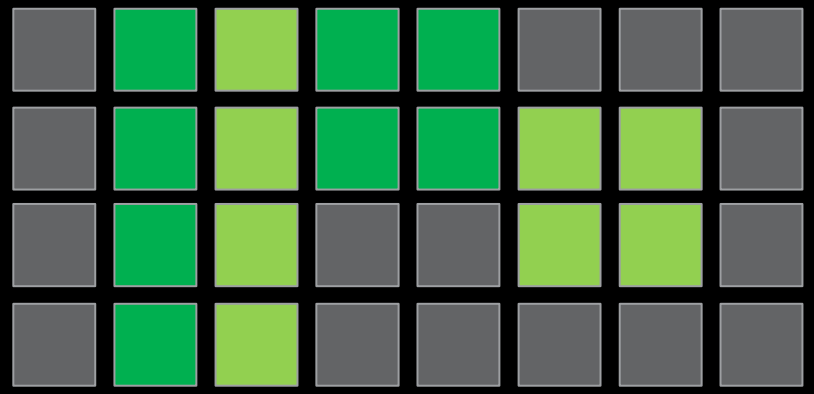


Basic Spatial Concepts

```
%lceil = arith.constant 2 : index
air.launch (%arg3) in (%arg5=%lceil) {
   air.channel [2][2] @chan ()
     %c4 = arith.constant 4 : index
     %c1 = arith.constant 1 : index
     %c2 = arith.constant 2 : index

   air.partition {
       air.herd (%x, %y) in (%a1 = c1,%a2 = %c4) {
      %i = arith.remsi (%x, %c2) : index
      %j = arith.floordivsi (%y, %c2) : index
       air.put @chan[%i, %j]( … )
     }
   }
   air.partition {
    air.herd (%x, %y) in (%a1 = %c2,%a2 = %c2) {
      air.get @chan[%x,%y] ( … )
   }
}
```

Runtime can determine concurrency - parallel operations

or schedule them in time



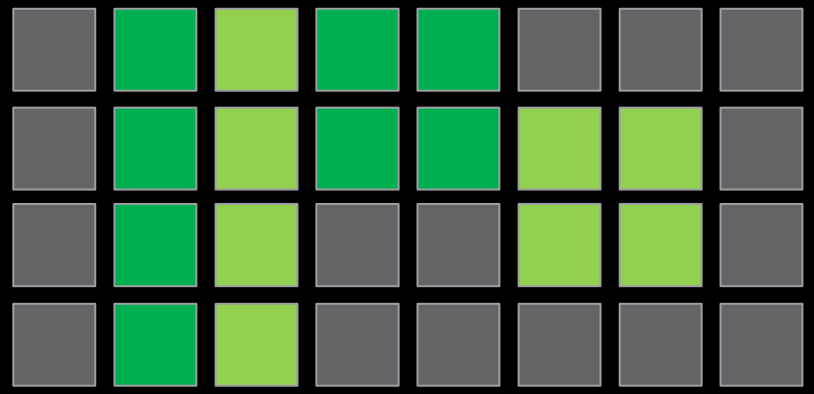
Basic Spatial Concepts

```
%lceil = arith.constant 2 : index
air.launch  (%arg3) in (%arg5=%lceil) {
  air.channel [2][2] @chan ()
    %c2 = arith.constant 2 : index
    %c1 = arith.constant 1 : index
    %c4 = arith.constant 4 : index

  air.partition {
     air.herd (%x, %y) in (%a1 = c1,%a2 = %c4) {
    %i = arith.remsi (%x, %c2) : index
    %j = arith.floordivsi (%y, %c2) : index
     air.put @chan[%i, %j]( … )
    }
  }
  air.partition {
   air.herd (%x, %y) in (%a1 = %c2,%a2 = %c2) {
     air.get @chan[%x,%y] ( … )
  }
}
```

Fine-Grained Parallelism

Coarse Grained Parallelism

Runtime can determine concurrency - parallel operations *or* schedule them in time



Coarse-Grained *and* Fine-Grained Parallelism

[Public]

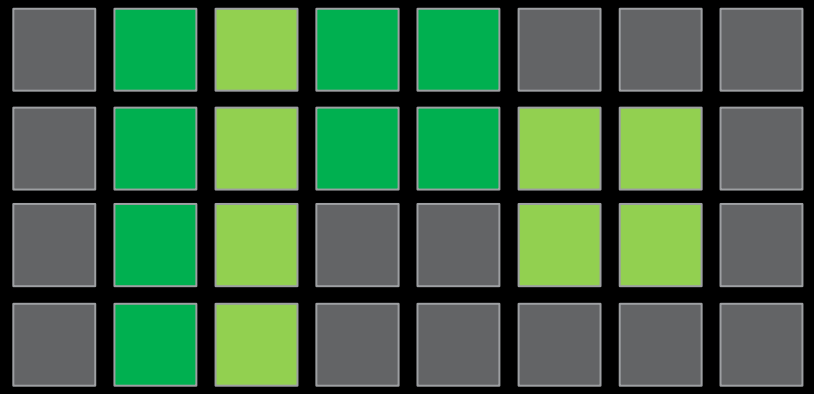
Basic Spatial Concepts

```
%lceil = arith.constant 2 : index
air.launch (%arg3) in (%arg5=%lceil) {
  air.channel [2][2] @chan ()
    %c4 = arith.constant 4 : index
    %c1 = arith.constant 1 : index
    %c2 = arith.constant 2 : index

  air.partition {
     air.herd (%x, %y) in (%a1 = c1,%a2 = %c4) {
    %i = arith.remsi (%x, %c2) : index
    %j = arith.floordivsi (%y, %c2) : index
     air.put @chan[%i, %j]( … )
    }
  }
  air.partition {
    air.herd (%x, %y) in (%a1 = %c2,%a2 = %c2) {
     air.get @chan[%i, %j]( … )
  }
}
```

Fine-Grained Parallelism

Fine-Grained Parallelism

Runtime can determine concurrency - parallel operations *or* schedule them in time



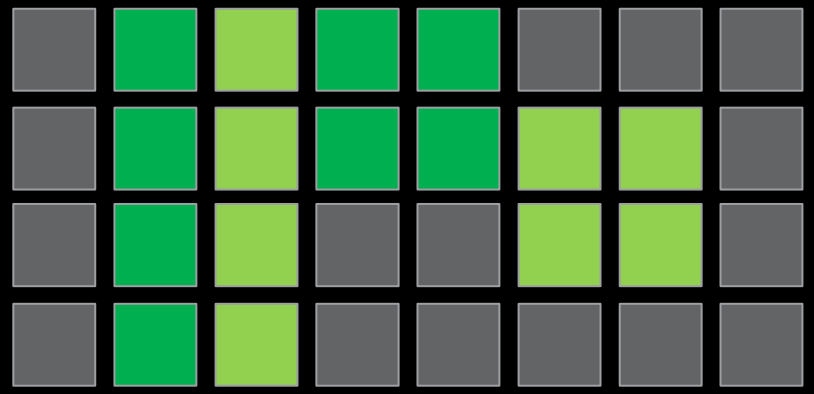
Coarse-Grained *and* Fine-Grained Parallelism

Basic Spatial Concepts

```
%lceil = arith.constant 2 : index
air.launch (%arg3) in (%arg5=%lceil) {
   air.channel [2][2] @chan ()
     %c4 = arith.constant 4 : index
     %c1 = arith.constant 1 : index
     %c2 = arith.constant 2 : index

   air.partition {
       air.herd (%x, %y) in (%a1 = c1,%a2 = %c4) {
      %i = arith.remsi (%x, %c2) : index
      %j = arith.floordivsi (%y, %c2) : index
       air.put @chan[%i, %j]( … )
     }
   }
   air.partition {
    air.herd (%x, %y) in (%a1 = %c2,%a2 = %c2) {
      air.get @chan[%x,%y] ( … )
   }
}
```

Runtime can determine concurrency - parallel operations

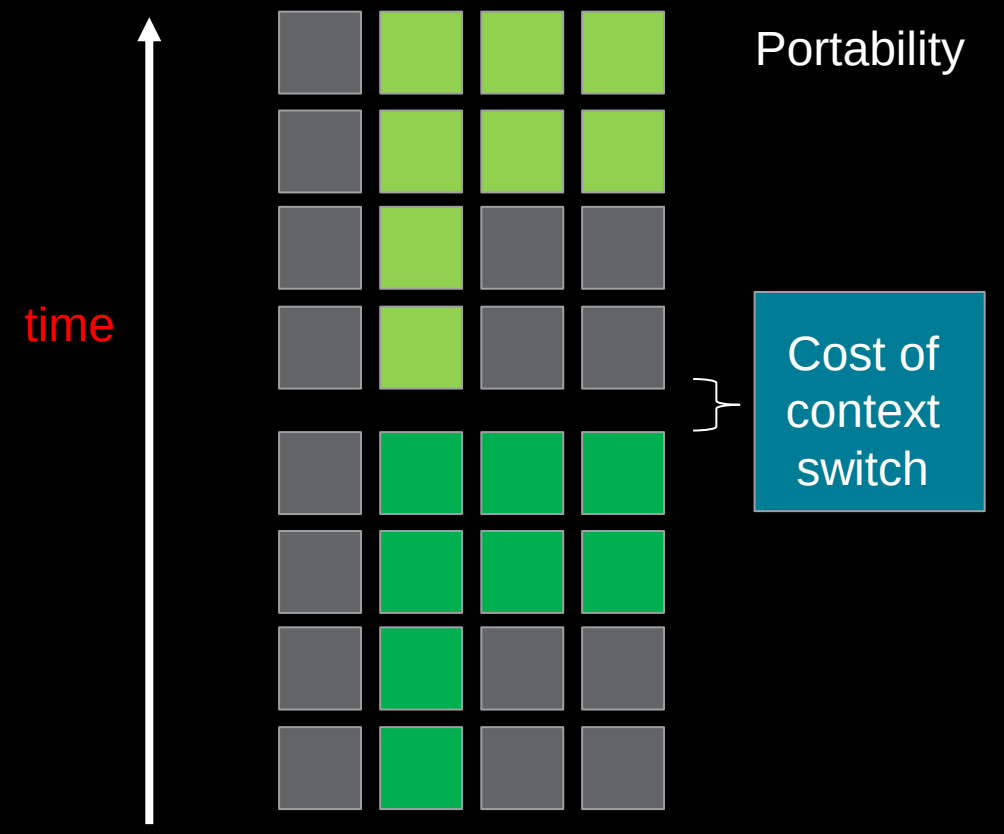
or schedule them in time



Basic Spatial Concepts

```
%lceil = arith.constant 2 : index
air.launch (%arg3) in (%arg5=%lceil) {
   air.channel [2][2] @chan ()
     %c4 = arith.constant 4 : index
     %c1 = arith.constant 1 : index
     %c2 = arith.constant 2 : index

   air.partition {
       air.herd (%x, %y) in (%a1 = c1,%a2 = %c4) {
      %i = arith.remsi (%x, %c2) : index
      %j = arith.floordivsi (%y, %c2) : index
       air.put @chan[%i, %j]( … )
     }
   }
   air.partition {
    air.herd (%x, %y) in (%a1 = %c2,%a2 = %c2) {
      air.get @chan[%x,%y] ( … )
   }
}
```

Runtime can determine concurrency - parallel operations
or schedule them in time



Explore More…

Other Concepts

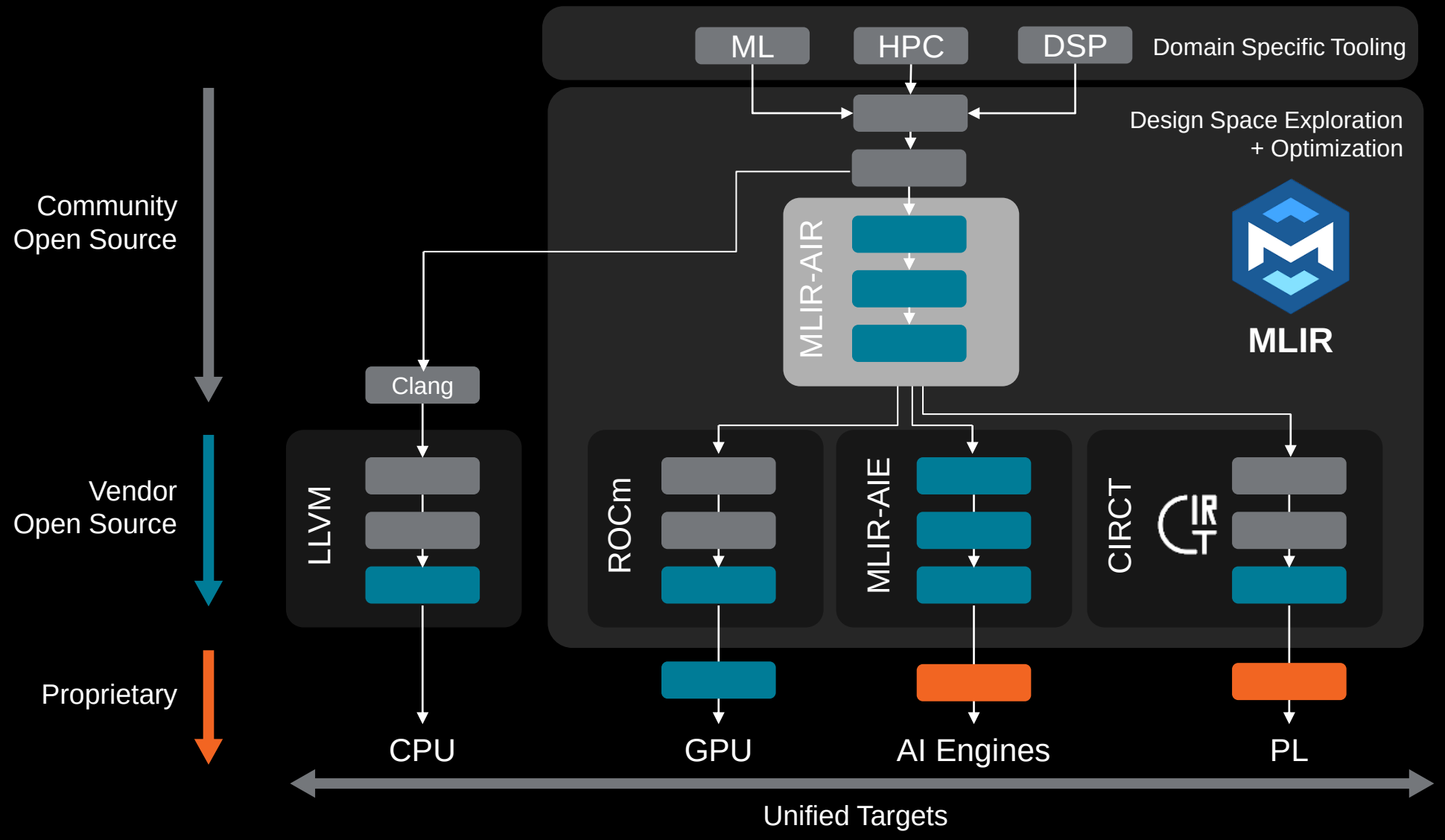
- Explicit multi-dimensional DMAs
- Asynchronous Concurrency using tokens

Transformation Passes

- Infer air.herd from scf.parallel operations
- Tiling from linalg.generic operations
- Form explicit data movement from memcopy
- Hoist Invariant Data Movement Operations
- Token lowering to native hardware sync primitives

[Public]

Tools Built around AIR dialect



Wrap Up

Thank You

- Thank you for joining!

- Want to experiment more:
  - Clone http://github.com/Xilinx/mlir-aie.git
  - Clone http://github.com/Xilinx/mlir-air.git
  - Tutorials are online: https://github.com/Xilinx/mlir-aie/tree/main/tutorials
    No board needed!
  - Get access to HACC: https://www.amd-haccs.io with pre-installed tools (and boards)
  - Contact: xup@amd.com
  - Internships
    - Interested in modelling new devices ? (https://careers.amd.com/careers-home/jobs/25264)
    - Interested in targeting new applications ?

- Feedback – QR code link for feedback form
  - Web link

AMD
together we advance_

AMD